



Quick answers to common problems

Microsoft Silverlight 5 Data and Services Cookbook

Over 100 practical recipes for creating rich, data-driven,
business applications in Silverlight 5

Gill Cleeren

Kevin Dockx

[PACKT] enterprise
professional expertise distilled

Microsoft Silverlight 5 Data and Services Cookbook

Over 100 practical recipes for creating rich, data-driven,
business applications in Silverlight 5

Gill Cleeren

Kevin Dockx



BIRMINGHAM - MUMBAI

Microsoft Silverlight 5 Data and Services Cookbook

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First Edition: April 2010

Second Edition: April 2012

Production Reference: 2130412

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-350-0

www.packtpub.com

Cover Image by David Gimenez (bilbaorocker@yahoo.co.uk)

Credits

Authors

Gill Cleeren

Kevin Dockx

Reviewers

Mario Van Hissenhoven

Evan Hutnick

Kris van der Mast

Dennis Miscoria

Project Coordinator

Vishal Bodwani

Proofreaders

Chris Smith

Josh Toth

Indexers

Tejal Daruwale

Monica Ajmera Mehta

Acquisition Editor

Kerry George

Graphics

Manu Joseph

Lead Technical Editor

Hyacintha D'Souza

Production Coordinator

Nilesh Mohite

Technical Editors

Ankita Shashi

Manasi Poonthottam

Sakina Kaydawala

Cover Work

Nilesh Mohite

About the Authors

Gill Cleeren is a Microsoft Regional Director, Silverlight MVP (former ASP.NET MVP), and Telerik MVP. He lives in Belgium where he works as .NET architect at Ordina. Passionate about .NET, he's always playing with the newest bits. In his role as Regional Director, Gill has given many sessions, webcasts, and trainings on new as well as existing technologies, such as Silverlight, ASP.NET, and WPF at conferences including TechEd Europe, TechDays Belgium—Switzerland—Sweden, DevDays NL, NDC Oslo Norway, Silverlight Roadshow in Sweden, Telerik RoadShow UK, and so on. He organizes the yearly Community Day event in Belgium and leads Visug, the largest .NET user group in Belgium. You can find his blog at www.snowball.be and on Twitter, you can follow him via @gillcleeren.

Gill published his first book, *Silverlight 4 Data and Services Cookbook*, with Packt Publishing. He also authored a chapter for *Real World .NET, C#, and Silverlight: Indispensable Experiences from 15 MVPs* and also authored numerous articles and eBooks for SilverlightShow.net.

After the publication of my first book with Packt, I was very happy with its success. It quickly got a lot of positive reviews on blogs and sites such as Amazon. This was for me the trigger to start writing again. What you're holding here is another year's work of two devoted people who love developing applications with Silverlight and want to share that love with you. Reading it will certainly help you figure out complex problems that you may encounter in your life as a Silverlight developer. Or maybe more generally as an XAML developer, since most of the content can be used in all places where XAML is used as the development language.

Of course, this book is not the work of only the authors. Without the team at Packt and the reviewers, it wouldn't even be possible to complete such a project. And of course, without the patience and love of my girlfriend (and wife-to-be from September 2012) and my mother, I wouldn't be able to walk into a bookstore and be able to hold my work in my hands!

Kevin Dockx lives in Belgium and works at RealDolmen, one of Belgium's biggest ICT companies, where he is a 30-year old technical specialist/project leader on .NET web applications, mainly Silverlight, and a solution manager for Rich Applications (Silverlight, Windows Phone 7, WPF, Surface, HTML5). His main focus lies on all things Silverlight, but he still keeps an eye on the new developments concerning other products from the Microsoft .NET (Web) Stack. As a Silverlight enthusiast, he's a regular speaker at various national and international events, like Microsoft DevDays in The Netherlands, Microsoft Techdays in Portugal, BESUG events (the Belgian Silverlight User Group), Simplicity Day, Community Day, and so on. Next to that, he also writes articles for various Silverlight-related sites. His blog, which contains various tidbits on Silverlight, .NET, and the occasional rambling, can be found at <http://blog.kevindockx.com/>, and you can contact him on Twitter via @KevinDockx.

He has worked on other books like *Silverlight 4 Data and Services Cookbook* (Packt Publishing). He has also worked on various articles and ebooks for SilverlightShow.net and other Silverlight-related sites.

I could come up with a long list of people I'd like to thank, and with a bunch of reasons to write this book. But I guess it all boils down to one thing: passion. Passion for technology. Passion to share knowledge. And passion for the next big thing. So I'm going to keep this short: one quote, that's all there is to it:

"Wandering along the lines of another next big thing, remember: there's always room for more ice cream."

About the Reviewers

Mario Van Hissenhoven is a certified Microsoft SQL Server Professional with more than 10 years of experience. His specialties are development in transact SQL.

Mario also has extended his knowledge to the .NET framework 2.0, 3.0, and 3.5 during the last 5 years. His interest has always been to be on top of the new Microsoft technologies such as Silverlight, WCF, and SQLCLR.

Nowadays, Mario is focusing on SQL Server 2008R2 Development and the Beta features of SQL Server 2012.

I would like to thank my wife and kids for the time spent on reviewing this book while I should have been spending time with them. And I promise I will make it up to them.

Evan Hutnick is a Developer Evangelist working for Telerik in the XAML space covering Silverlight, WPF, Windows Phone, and the up-and-coming WinRT platform. Evan is also a Silverlight MVP and recognized throughout the community for his contributions on best practices, designer/developer concepts, as well as general architecture guidance. While focusing on XAML technologies, Evan often specializes in enterprise development scenarios and best practices for solution structure and architecture.

I thank my wife Jennifer and my daughter Keira for tolerating my insane love of technology and for putting up with the long hours and late nights, I couldn't do it without them.

Kris van der Mast is a Microsoft MVP since 2007, Microsoft ASP Insider, and a respected member and moderator on the official ASP.NET forums where he ranks at the number one position.

After he became an engineer he followed an extensive path into the magical world of web. Besides his work, Kris plays a very active role in the community by delivering articles for magazines, being a board member of the Belgian Windows Azure user group (www.azug.be), presenting or teaching about the latest (web) technologies. More recently Kris became a part of MEET (Microsoft Extended Experts Team).

You can follow him on Twitter via @KvdM or his blog at <http://blog.krisvandermast.com> to find out about Windows Azure, ASP.NET (MVC), WebMatrix, jQuery, Orchard CMS, and so on.

Dennis Miscoria is an enthusiastic .NET developer living in Belgium. Over the past six years, he has built up an extensive knowledge in the mobile world starting from the Compact Framework up to the latest Windows Phone and Silverlight technology.

Dennis is currently working for the Belgian consultancy company Ordina (www.ordina.be) as senior .NET engineer. Due to his mobile background, he is also in charge of the .NET Mobile Competence Center.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Learning the Nuts and Bolts of Silverlight 5	7
Introduction	7
Getting our environment ready to start building Silverlight applications	8
Creating our first service-enabled and data-driven Silverlight 5 application using Visual Studio 2010	10
Using the workflow between Visual Studio 2010 and Blend 5	20
Using source control in Visual Studio 2010 and Blend 5	29
Deploying a Silverlight application on the server	32
Chapter 2: An Introduction to Data Binding	37
Introduction	37
Displaying data in Silverlight applications	40
Creating dynamic bindings	49
Binding data to another UI element	52
Binding collections to UI elements	56
Enabling a Silverlight application to automatically update its UI	60
Obtaining data from any UI element it is bound to	68
Using the different modes of data binding to allow persisting data	73
Debugging data binding expressions in Visual Studio	77
Data binding from Expression Blend 5	81
Using Expression Blend 5 for sample data generation	84
Chapter 3: Advanced Data Binding	87
Introduction	87
Hooking into the data binding process	88
Replacing converters with Silverlight 5 BindingBase properties	93
Validating data-bound input	97
Validating data input using attributes	101

Table of Contents

Validating using <code>IDataErrorInfo</code> and <code>INotifyDataErrorInfo</code>	104
Using templates to customize the way data is shown by controls	109
Using implicit data templates	115
Using the Ancestor RelativeSource binding	119
Creating custom markup extensions	123
Building a change-aware collection type	127
Combining converters, data binding, and <code>DataContext</code> into a custom <code>DataTemplate</code>	130
Chapter 4: The Data Grid	139
Introduction	139
Displaying data in a customized <code>DataGridView</code>	140
Inserting, updating, and deleting data in a <code>DataGridView</code>	146
Sorting and grouping data in a <code>DataGridView</code>	151
Filtering and paging data in a <code>DataGridView</code>	156
Using custom columns in the <code>DataGridView</code>	160
Implementing master-detail in the <code>DataGridView</code>	167
Validating the <code>DataGridView</code>	171
Chapter 5: Working with Local Data	175
Introduction	175
Reading data from and storing data in the isolated storage	176
Working with <code>IsolatedStorageSettings</code>	186
Caching data between different Silverlight applications using isolated storage	190
Using the Sterling database	192
Chapter 6: MVVM	205
Introduction	205
Creating a basic MVVM application	206
Using MVVM Light to enable MVVM applications	214
Connecting a View to a ViewModel using a <code>ViewModelLocator</code>	219
Connecting a View to a ViewModel using MEF	224
Using commands to pass your events to the ViewModel	229
Communicating between different ViewModels	235
Leveraging a messenger to wrap application-wide messages	239
Chapter 7: Working with Services	243
Introduction	243
Connecting and reading from a standardized service	244
Persisting data using a standardized service	250
Configuring cross-domain calls	254

Table of Contents

Working cross-domain from a trusted Silverlight application	262
Reading XML using <code>HttpWebRequest</code>	265
Reading out an RSS feed	271
Accessing a database in the cloud	274
Accessing a service in the cloud	279
Running a Silverlight application from the cloud	284
Using socket communication in Silverlight	288
Chapter 8: Talking to WCF and ASMX Services	301
Introduction	301
Invoking a service that exposes data	302
Invoking a service such as Bing.com	312
Optimizing performance using binary XML	315
Debugging a service in Silverlight	318
Using ASP.NET Authentication in Silverlight	325
Uploading files to a WCF service	332
Displaying images as a stream from a WCF service	338
Chapter 9: Talking to WCF and ASMX Services—One Step Beyond	345
Introduction	345
Using duplex communication over HTTP	346
Using duplex communication with the WCF net.tcp binding	355
Ensuring data is encrypted	363
Securing service communication using message-based security	369
Integrating Windows Identity Foundation in Silverlight	374
Calling a WCF service from Silverlight using ChannelFactory	383
Chapter 10: Talking to REST and WCF Data Services	387
Introduction	388
Reading data from a REST service	389
Parsing REST results with LINQ To XML	395
Persisting data using a REST service	399
Working with the ClientHttp stack	406
Communicating with a REST service using JSON	408
Using WCF Data Services with Silverlight	411
Reading data using WCF Data Services	416
Persisting data using WCF Data Services	421
Talking to Flickr	426
Talking to Twitter over REST	434
Passing credentials and cross-domain access to Twitter from a trusted Silverlight application	439

Table of Contents

Chapter 11: Using WCF RIA Services	451
Introduction	451
Setting up a data solution to work with WCF RIA Services	452
Using a WCF RIA Services class library	455
Getting data on the client	460
Using LoadBehavior to control what happens to your data once it's sent to the client	472
Controlling the server-side query from the client	476
Sorting and filtering data on the server	481
Paging through your data	485
Persisting a change set/unit of work	490
Working with concurrency and transactions	496
Chapter 12: Advanced WCF RIA Services	503
Introduction	504
Tracking a user's identity – default Windows authentication	504
Tracking a user's identity – a custom authentication service	507
Integrating Windows Identity Foundation with WCF RIA Services	513
Controlling a user's access to services and service methods	517
Validating data: using data annotations	521
Validating data: writing a custom validator	524
Validating data: server-side validation with client-side feedback	532
Validating data: triggering validation when needed	536
Validating data: using the ValidationContext	541
Handling errors on the server	545
Using SQL Azure with WCF RIA Services	548
Exposing WCF RIA Domain Services as OData endpoints	550
Exposing WCF RIA Domain Services for other technologies	553
Chapter 13: Windows Phone 7	557
Introduction	557
Getting our environment ready to start building Windows Phone 7 applications	562
Building your first data-driven Windows Phone 7 application	562
Getting data on your Windows Phone 7 using WCF	578
Accessing REST services from Windows Phone 7 using XML	587
Accessing REST services from Windows Phone 7 using JSON	591
Working with push notifications using the cloud	597
Storing data in a local SQL CE database	609
Using the background transfer service	616

Table of Contents

Appendix	623
Creating a REST service from WCF	623
Installing an SQL Server database	625
Working with Fiddler	625
Working with the Silverlight control toolkit	626
Working with WIF	627
Installing the WCF RIA Services Toolkit	627
Installing and using NuGet	627
Index	629

Preface

About 2 years ago, in the spring of 2010, Microsoft released Silverlight 4. Silverlight 4 proved to be a platform ready for Line-of-Business application development. Numerous developers learned how to build great apps with it, which can run both within the browser and as a stand-alone application on the user's machine. Silverlight 5 was the logical successor and extended the platform again with an extensive list of new features.

Soon after the launch of the Silverlight 4, the first edition of this book was released. Its success convinced us to write an updated version that focuses on Silverlight 5. While all existing content is updated to match the new version, a lot of new content is added. This can be found in recipes covering Silverlight 5-specific features as well as complete new chapters, covering other aspects of working with data such as MVVM (Model-View-ViewModel) or even from Windows Phone 7.

In this practical cookbook, you'll learn how to build data-rich business applications with Silverlight that draw on multiple sources of data. Although the book focuses on Silverlight 5, many of the recipes will work on Silverlight 4 projects as well. A large number will also work in Silverlight 3. This is indicated for each recipe.

Packed with reusable, real-world recipes, the book begins by introducing you to general principles for programming Silverlight. It then dives deep into the world of data services, covering all the options available to access data and communicate with services to make the most out of data in your Silverlight business applications, whilst at the same time providing a rich user experience. Chapters cover data binding, data controls, concepts of talking to services, communicating with WCF, ASMX, REST services, and much more. The chapter on accessing data and services from Windows Phone 7 applications discusses how to leverage your knowledge on the mobile platform.

By following the practical recipes in this book, which are of varying difficulty levels, you will learn concepts for creating data-rich business applications—from the creation of a Silverlight application, to displaying data in the Silverlight application and upgrading your existing applications to use Silverlight. Each recipe will cover a data services topic, starting from the description of the problem, covering a conceptual solution and a solution containing sample code.

What this book covers

Chapter 1, Learning the Nuts and Bolts of Silverlight 5, will get you up and running with Silverlight. While this book is aimed at developers who already have a basic knowledge of Silverlight, this chapter can act as a refresher. We'll also look at getting your environment correctly set up so that you enjoy developing Silverlight applications.

Chapter 2, An Introduction to Data Binding, will explore how data binding works. We'll start by building a small data-driven application that contains the most important data binding features, to get a grip of the general concepts. We'll also see that data binding isn't tied to just binding single objects to an interface; binding an entire collection of objects is supported as well. We'll also be looking at the binding modes. They allow us to specify how the data will flow (from source to target, target to source, or both). Visual Studio enables debugging data binding statements in version 5, which we'll dive into, and we'll finish this chapter by looking at the support that Blend 5 provides to build applications that use data binding features. In the next chapter, we'll be looking at the more advanced concepts of data binding.

Chapter 3, Advanced Data Binding, teaches you advanced data binding concepts that can be used for customization, validations, and applying templates to data bound controls. New Silverlight 5 features such as custom markup extensions, Ancestor Relative Source binding, and implicit data templates are discussed in this chapter as well. We also have a look at converters, which can be seen as hooks in the data binding process.

Chapter 4, The Data Grid, covers recipes on how to work with the DataGrid. This is an essential control for applications that rely on (collections of) data.

Chapter 5, Working with Local Data, covers storing data locally. The concept of local data is essential in many scenarios, varying from saving local user settings to entire blocks of data. Silverlight has always included the concept of Isolated Storage; we'll see how to use that.

Chapter 6, MVVM, explains all you need to get started with the Model-View-ViewModel design pattern, the de facto standard for XAML-based applications. Using this pattern to build Silverlight applications will result in better separation of concerns, code that's easier to test and maintain, and it ensures you leverage the true power of XAML.

Chapter 7, Working with Services, talks about the rich set of options that Silverlight provides to communicate with services. We'll see also how Silverlight and Azure can be used together for more powerful solutions.

Chapter 8, Talking to WCF and ASMX Services, discovers Silverlight's built-in support for communicating with Windows Communication Foundation (WCF) and classic ASMX web services. Integration with the ASP.NET Membership API as well as uploading and downloading files is covered in this chapter as well.

Chapter 9, Talking to WCF and ASMX Services—One Step Beyond, takes us on a tour of more complex WCF problems and their solutions. Perform unidirectional as well as bidirectional communication with much better performance using net.tcp binding in WCF using the recipes in this chapter. Security is vital when working with services and is explained as well through several recipes.

Chapter 10, Talking to REST and WCF Data Services, takes advantage of REST, which can be significant in the case of Silverlight. We will also look at how we can work with WCF Data Services. You will abstract away a lot of plumbing code with the use of the client-side library that is available for use with Silverlight.

Chapter 11, Using WCF RIA Services, is all about the framework built by Microsoft, to simplify and reduce development time for Line-of-Business RIA development. In this chapter, we look into the basics: how it works behind the scenes, how to fetch data, how to sort, filter, and page through your data, how to submit data, and how to structure your project.

Chapter 12, Advanced WCF RIA Services, tackles the more advanced techniques concerning WCF RIA Services: you'll learn all about authentication (Windows, Forms, and through WIF), various validation scenarios, error handling, and how you can expose your domain services for use with other technologies.

Chapter 13, Windows Phone 7, explains how Windows Phone 7 applications, which are by default built with Silverlight, can communicate with services and get access to server-side. In this chapter, recipes can be found which cover connecting to services that communicate with XML and JSON as well as SOAP over WCF. We'll also take a look at working with a local SQL CE database. Finally, a recipe on push notifications explains how a cloud service can connect with an application on a device, opening push-like scenarios.

The Appendix talks about creating a REST service from WCF, installing a SQL Server database, working with Fiddler and the Silverlight control toolkit, WCF RIA Services and WIF.

What you need for this book

To work with the recipes in this book, you should have Visual Studio installed. This book targets Silverlight 5, for which you need Visual Studio 2010 (or later). Many of the recipes in the book will also work in Silverlight 3 and 4, so for these recipes, you have the choice of Visual Studio 2008 (for Silverlight 3) or 2010 (Silverlight 3 and 4). We do recommend using Visual Studio 2010, as it features a lot of enhancements for developing with Silverlight. In both cases, you'll of course need to install the Silverlight Tools, which will update your Visual Studio instance to work with Silverlight. Some recipes also require Blend 5 to be installed on your machine (again, if working with Silverlight 3, Blend 3 will suffice here as well; Silverlight 4 applications can be built using Blend 4). For the Windows Phone recipes, you could use the specific phone-enabling Express edition of Visual Studio, which is bundled with the Windows Phone SDK & Tools.

The first recipe of Chapter 1, *Getting our environment ready to start developing Silverlight applications*, explains in detail how to get these tools and how to install them.

Who this book is for

If you are a .NET developer who wants to build professional data-driven applications with Silverlight, then this book is for you. Basic Silverlight experience and familiarity with accessing data using ADO.NET in regular .NET applications is required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Finally, the `DataReader` and connection are closed and the `StoreDTO` object is returned."

A block of code is set as follows:

```
<TextBlock x:Name="AmountTextBlock"
           Text="{Binding ElementName=AmountSlider, Path=Value}">
</TextBlock>
```

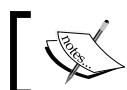
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<TextBlock x:Name="AmountTextBlock"
           Text="{Binding ElementName=AmountSlider, Path=Value}">
</TextBlock>
```

Any command-line input or output is written as follows:

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "Do this by right-clicking, selecting **Add New Item**, and then selecting **LINQ TO SQL Classes**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.



Downloading the example code

You can download the example code files for this book you have purchased from your account at http://www.packtpub.com/files/code/3500_Code.zip. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the errata submission form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Learning the Nuts and Bolts of Silverlight 5

In this chapter, we will cover the following topics:

- ▶ Getting our environment ready to start building Silverlight applications
- ▶ Creating our first service-enabled and data-driven Silverlight 5 application using Visual Studio 2010
- ▶ Using the workflow between Visual Studio 2010 and Blend 5
- ▶ Using source control in Visual Studio 2010 and Blend 5
- ▶ Deploying a Silverlight application on the server

Introduction

While we assume you have some basic knowledge of Silverlight, we also know that developers have very little time to grasp all the new technologies that keep coming out. Therefore, this first chapter contains all that we need to know to get going with Silverlight. We'll also guide you through the required tools and installations for a perfect Silverlight development environment.

Silverlight was released in the first half of 2007, and since then it has created a lot of buzz. While ASP.NET is a server-side development platform, with the arrival of Silverlight, the focus has shifted to the client side again. A Silverlight application runs in the browser of the client and on a specific version of the Common Language Runtime (CLR).

A big benefit for developers is that Silverlight uses .NET from version 2 onwards. It has a trimmed-down version of the Base Class Library (BCL), which is impressively extended, considering the size of the Silverlight plugin (about 6 MB). Because of the similarities, many skills achieved from developing applications in the full .NET framework can be leveraged for the creation of Silverlight applications.

Silverlight itself can be considered as a trimmed-down version of its desktop counterpart, **Windows Presentation Foundation (WPF)**. While there are some differences between the two platforms, it's not difficult to make the transition from the one to the other, since they share the same concepts. They both use **XAML** and patterns like **MVVM** (which is covered deeply in this book), which are applied the same way in both technologies.

With the release of Silverlight 2, Microsoft made it clear that Silverlight is aimed at both creating rich and interactive applications and next-generation enterprise applications in the browser. The latter can be easily seen with the addition of a rich control set, support for many types of services and platform features, such as data binding.

Due to its client-side characteristics, Silverlight applications need to perform particular tasks to get data. It doesn't support client-side databases—not even in version 5, the latest version. The way to retrieve data is through services. Silverlight 3 brought some interesting features to the platform in this area, such as support for binary XML, the WCF RIA services, and simplified duplex service communication. Silverlight 4 continued in the same manner, with improvements in data binding, support for net.tcp communication, cross-domain access to services by means of **Trusted Silverlight** applications, and much more. With Silverlight 5, we can see that Microsoft is continuing along the same path. Some really interesting features made it into this release, including data-binding debugging (which was perhaps the most anticipated feature), new data-binding feature support, such as **Ancestor RelativeSource**, deeper support for the MVVM pattern, **PInvoke** support, multiple windows, and so on.

In this chapter, we'll get you up and running with Silverlight. While this book is aimed at developers who already have a basic knowledge of Silverlight, this chapter can act as a refresher. We'll also look at getting your environment correctly set up, so that you can enjoy developing Silverlight applications.

Getting our environment ready to start building Silverlight applications

To start building Silverlight applications, we need more than just Notepad. In this recipe, we'll look at what we need to install to start developing Silverlight applications like a pro.

How to do it...

To start developing Silverlight applications, we'll need to install the necessary tools and SDKs. We will also need to carry out the following steps in order to get started:

1. We need to make sure that we install Visual Web Developer Express 2010 (available for free at <http://www.microsoft.com/express/downloads/>) or Visual Studio 2010 (trial version available at <http://www.microsoft.com/visualstudio/en-us/download>). If you already have any version of Visual Studio 2010 installed, you can skip this step. From a developer's perspective, the free Express edition will give you all you need to build applications with Silverlight, although some features are missing.
2. Go to <http://www.silverlight.net/getstarted/> to download and install the Silverlight 5 Tools for Visual Studio 2010. Visual Studio 2010 ships with Silverlight 3 templates installed out of the box; Silverlight 4 and 5 can be added to the IDE by installing the tools.
3. A trial of Expression Blend 5 can be downloaded from <http://www.microsoft.com/expression/>.

How it works...

For Silverlight development, the minimum that we need are the developer tools. If we want to develop Silverlight 4 or 5 applications, Visual Studio 2010 is required (if you are still using Silverlight 3, Visual Studio 2008 will do). In Visual Studio 2010, a visual designer is added for editing our XAML code. When installing the developer tools for Silverlight 5, the following components are automatically downloaded and installed:

- ▶ Silverlight 5 developer runtime
- ▶ Silverlight 5 software development kit and Visual Studio project support
- ▶ WCF RIA services SP2

We can write XAML code using Visual Studio. However, if you're serious about designing, you might want to consider using Microsoft Expression Blend. This tool, primarily aimed at designers, should be seen as an application that generates XAML for us by means of a rich number of options and an easy-to-use interface. It also integrates nicely with Visual Studio and source control software integration is available as well.

See also

After having installed all the necessary tools, it might be worth taking a look at the recipe *Creating our first service-enabled and data-driven Silverlight 5 application using Visual Studio 2010*, as well as the recipe *Using the workflow between Visual Studio 2010 and Blend 5*. In these recipes, we create an entire application in Visual Studio 2010 and Blend 5, respectively.

Creating our first service-enabled and data-driven Silverlight 5 application using Visual Studio 2010

Applies to Silverlight 4 and 5

In this recipe, we'll build a very simple Silverlight application that uses techniques that are explained in much more detail later on in the book. We'll be using data binding, which is a technique to easily connect data to the User Interface (UI), and connect to a Windows Communication Foundation (WCF) service.

However, the main goal is to get a grip on the basics of Silverlight by trying to answer questions, such as how a Silverlight application is built, what the project structure looks like, what files are created, and what their uses are.

Getting ready

To get started with Silverlight application development, make sure you have installed Visual Studio along with the necessary tools and libraries as outlined in the previous recipe.

We are building this application from the ground up. However, the finished product can be found in the Chapter01/SilverlightHotelBrowser folder in the code bundle that is available on the Packt website.

How to do it...

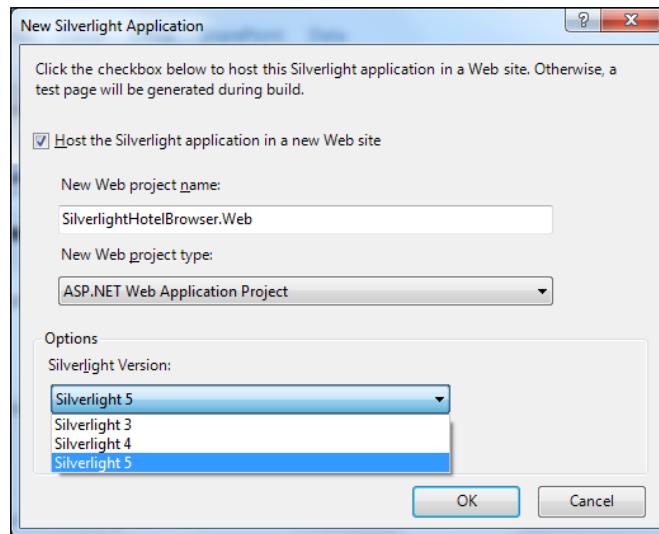
Our first Silverlight application allows the user to view the details of a hotel that is selected in a ComboBox control. The hotel information is retrieved over a service and is used for filling the ComboBox. The details are shown in several TextBlock controls, which are placed in a Grid.

The following screenshot shows the interface of the application:

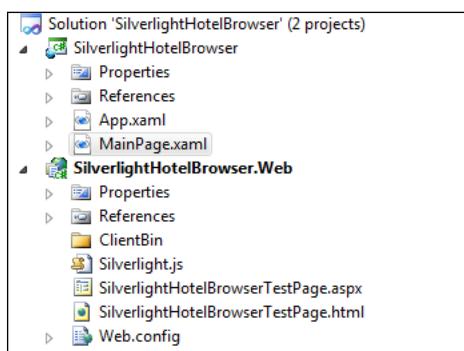


To start building any Silverlight application, we'll need to perform the following steps:

1. Open Visual Studio 2010 with the Silverlight 5 tools installed. Once inside the **Integrated Development Environment (IDE)**, go to **File | New | Project....** In the **New Project** dialog that appears, select the **Silverlight** node under **Visual C#**, and select **Silverlight Application**. Name the application **SilverlightHotelBrowser** and click the **OK** button. In the dialog that appears as shown in the following image select **ASP.NET Web Application Project** as the type of web project that will be used to host the Silverlight application. Also, make sure that **Silverlight 5** is selected as the target version of Silverlight:



2. After Visual Studio has finished executing the project template, two projects are created in the solution: the Silverlight project and a web application project that is responsible for hosting the Silverlight content. The created solution structure can be seen in the following screenshot:



3. Our service will return the hotel information. A hotel can be represented by an instance of the Hotel class. This class should be included in the web project—**SilverlightHotelBrowser.Web**. To add it, right-click on the project in the **Solution Explorer**, and select **Add | Class**. In the **Add new item** dialog, set the name of the class to Hotel.

There are a few things to note about this class that are as follows:

- ❑ This class has a `DataContract` attribute attached to it. This attribute is required to specify that this class can be serialized, when sent over the wire to the client application.
- ❑ Each property is attributed with the `DataMember` attribute. When adding this attribute to a property, we specify that this property is a part of the contract, and that it should be included in the serialized response that will be sent to the client.

The following code defines the Hotel class:

```
[DataContract]
public class Hotel
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public string Location { get; set; }
    [DataMember]
    public string Country { get; set; }
    [DataMember]
    public double Price { get; set; }
}
```

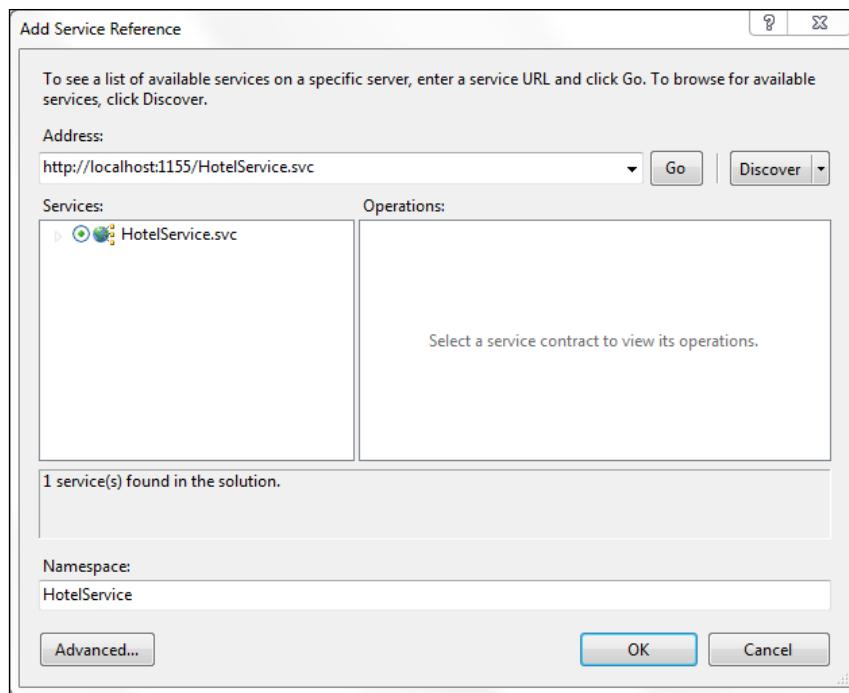
4. We'll now add a WCF service to the web project as well. Right-click on **SilverlightHotelBrowser.Web** and select **Add | New Item....** Add a Silverlight-enabled WCF Service by selecting the **Silverlight** node under **Visual C#**, naming it HotelService. Click the **Add** button. Two files are added, namely, `HotelService.svc` and `HotelService.svc.cs`.
5. In this service class, we can now add a method that returns a hard-coded list of hotels. Remove the `DoWork` method and replace it with the following code:

```
[ServiceContract(Namespace = "")]
[AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Allowed)]
public class HotelService
{
    [OperationContract]
    public List<Hotel> GetHotels()
```

```
{  
    return new List<Hotel>  
    {  
        new Hotel  
        {  
            Name = "Brussels Hotel",  
            Price = 100,  
            Location = "Brussels",  
            Country = "Belgium"  
        },  
        new Hotel  
        {  
            Name = "London Hotel",  
            Price = 200,  
            Location = "London",  
            Country = "United Kingdom"  
        },  
        new Hotel  
        {  
            Name = "Paris Hotel",  
            Price = 150,  
            Location = "Paris",  
            Country = "France"  
        },  
        new Hotel  
        {  
            Name = "New York Hotel",  
            Price = 230,  
            Location = "New York",  
            Country = "USA"  
        }  
    };  
}
```

Note the attributes that have been used in this class. The `ServiceContract` attribute specifies that the class contains a service contract that defines what functionality the service exposes. The `OperationContract` attribute is added to operations, which can be invoked by clients on the service. This effectively means that if you add methods to the service without this attribute, it can't be invoked from a client.

6. Now we'll build the solution, and if no errors are encountered, we're ready for writing Silverlight code. Should you get any errors, make sure to check that you entered the previous code correctly.
7. Let's first make the service known to the Silverlight application. Right-click on the Silverlight application, and select **Add Service Reference....** In the dialog box that appears, as shown in the following screenshot, click on **Discover** and select your service. As it is in the same solution, the service will appear. Enter HotelService in the **Namespace** field, shown as follows:



Click on the **OK** button to confirm. The service is now usable from the Silverlight application.

8. The UI was shown earlier and is quite easy. The XAML code for the Grid named `LayoutRoot` inside the `MainPage.xaml` file is as follows:

```
<Grid x:Name="LayoutRoot" Width="400" Height="300"
      Background="LightGray">
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <ComboBox x:Name="HotelComboBox" Width="250"
              SelectionChanged="HotelComboBox_SelectionChanged"
```

```
        DisplayMemberPath="Name"
        VerticalAlignment="Center">
    </ComboBox>
    <Grid x:Name="HotelDetailGrid" Grid.Row="1"
        VerticalAlignment="Top">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock x:Name="NameTextBlock"
            Grid.Row="0"
            Grid.Column="0"
            FontWeight="Bold"
            Text="Name: "
            HorizontalAlignment="Right">
        </TextBlock>
        <TextBlock x:Name="NameValueTextBlock"
            Grid.Row="0"
            Grid.Column="1"
            Text="{Binding Name}">
        </TextBlock>
        <TextBlock x:Name="LocationTextBlock"
            Grid.Row="1"
            Grid.Column="0"
            FontWeight="Bold"
            Text="Location: "
            HorizontalAlignment="Right">
        </TextBlock>
        <TextBlock x:Name="LocationValueTextBlock"
            Grid.Row="1"
            Grid.Column="1"
            Text="{Binding Location}">
        </TextBlock>
        <TextBlock x:Name="CountryTextBlock"
            Grid.Row="2"
            Grid.Column="0"
            FontWeight="Bold"
            Text="Country: "
```

```
        HorizontalAlignment="Right">
    </TextBlock>
    <TextBlock x:Name="CountryValueTextBlock"
        Grid.Row="2"
        Grid.Column="1"
        Text="{Binding Country}">
    </TextBlock>
    <TextBlock x:Name="PriceTextBlock"
        Grid.Row="3"
        Grid.Column="0"
        FontWeight="Bold"
        Text="Price: "
        HorizontalAlignment="Right">
    </TextBlock>
    <TextBlock x:Name="PriceValueTextBlock"
        Grid.Row="3"
        Grid.Column="1"
        Text="{Binding Price}">
    </TextBlock>
</Grid>
</Grid>
```

9. In the code-behind class, `MainPage.xaml.cs`, we connect to the service and the results are used in a data binding scenario. We won't focus here on what's actually happening with the service call and the data binding; all this is covered in detail later in this book. Add the following code in the code-behind of `MainPage.xaml`:

```
public MainPage()
{
    InitializeComponent();
    HotelService.HotelServiceClient proxy = new
        SilverlightHotelBrowser.HotelService.HotelServiceClient();
    proxy.GetHotelsCompleted += new EventHandler
        <SilverlightHotelBrowser.HotelService.
        GetHotelsCompletedEventArgs>
        (proxy_GetHotelsCompleted);
    proxy.GetHotelsAsync();
}
void proxy_GetHotelsCompleted(object sender,
    SilverlightHotelBrowser.HotelService.
    GetHotelsCompletedEventArgs e)
{
    HotelComboBox.ItemsSource = e.Result;
}
```

```
private void HotelComboBox_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    HotelDetailGrid.DataContext = (sender as ComboBox)
        .SelectedItem as HotelService.Hotel;
}
```

10. We will compile the solution again and then press the *F5* button. When running the application, select a hotel in the *ComboBox*. Each selection change will trigger an event that shows the details of the selected item using data binding.

How it works...

Silverlight applications always have to run in the context of the browser. That's the reason why Visual Studio prompts us initially by asking how we want to host the Silverlight content.

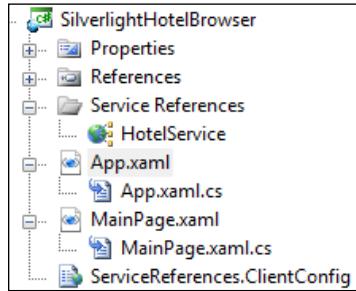


Starting with Silverlight 3, applications can run in the so-called out-of-browser mode, thereby allowing applications to run standalone. Silverlight 4 added the option to run a Silverlight application out of browser with elevated permissions, giving it more permissions on the local system. Silverlight 5 extended on this model even further. We'll be looking at out-of-browser applications later in this book.

The default option is the **ASP.NET Web Application Project**. This option gives us the maximum number of possibilities for the configuration of the host project. It's the option that we will be using the most throughout this book, because of the configuration options it offers for working with services. The second option is **ASP.NET Web Site** and is a file-based website known from ASP.NET 2.0. Finally, we can also uncheck the **Host the Silverlight application in a new Web site** checkbox. This will result in Visual Studio generating an empty HTML page containing the Silverlight application when we build our solution. This option is not well-suited for building Silverlight applications that work with services, as we have no control over the generation process.

The solution and project structure

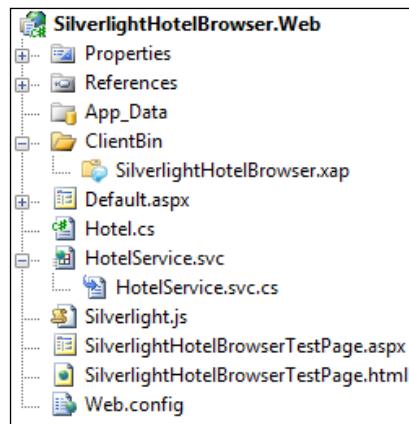
A new Silverlight solution contains normally two projects—a Silverlight project and a hosting application, which is nothing more than a website with a page that references the Silverlight application. Let's first take a look at the Silverlight project:



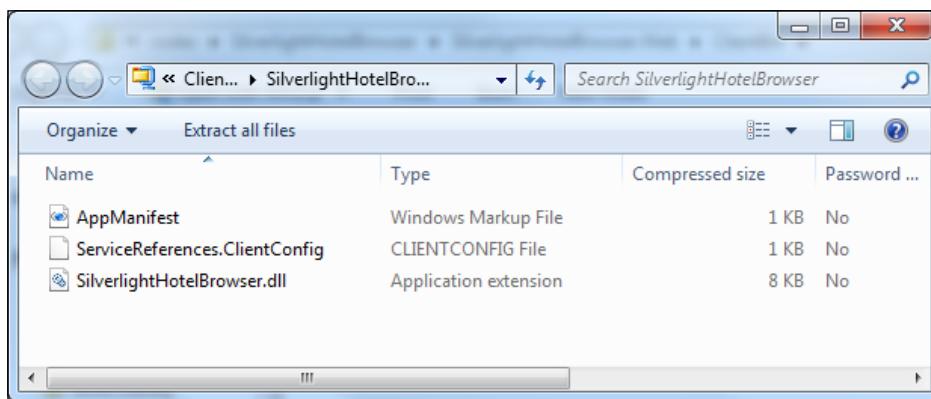
Silverlight applications contain XAML and C# (or VB.NET) files, among others. The XAML files contain the UI, and are linked at runtime to the partial classes that make up the code-behind. By default, one page is added for free—called **MainPage**. It's not really a page, but a user control that is hosted. We can add UI code (such as controls, grids, and so on) to this file. We add UI logic in the code-behind.

One special case is the `App.xaml` file. It's the entry point of an application and is responsible for loading an instance of `MainPage`, executing logic when an error occurs, and so on. Also, it can contain global resources, such as, styles that should be available over the entire application.

While building the solution, the Silverlight project is compiled into an assembly. In turn, this assembly—along with a manifest file that contains general information about the application and possible other resources—are wrapped into an XAP file. This XAP file is then copied into the hosting application. It shows up under the `ClientBin` directory in the web project, as shown in the following screenshot:



The XAP file is basically a ZIP (archive) file. When renaming the **SilverlightHotelBrowser.xap** file to **SilverlightHotelBrowser.zip**, we can see the original files (manifest and assembly). The following screenshot shows the contents of the ZIP file:



The generated ASPX page as well as the HTML page refer to the XAP file located in the ClientBin directory.

Services

Data is not readily available to a Silverlight application on the client side, so we need to retrieve it from the server. In Silverlight, this is done using services. Services need to be accessed asynchronously in Silverlight, hence the declaration of the callback method—`proxy_GetHotelsCompleted`. Silverlight has many options to communicate with services. These are covered in the following recipes of this book.

Data binding

We use the rich data binding features available in Silverlight to connect the data with the UI in this application. Data binding allows us to bind properties of objects (the data) to properties of controls. In this particular example, we bind a list of Hotel instances to the ComboBox, using the `ItemsSource` property. While changing the selection in the control, the `HotelComboBox_SelectionChanged` event handler fires and the selected item—a Hotel instance—is set as the `DataContext` for the HotelDetailGrid. This Grid contains the controls in which we want to show the details. Each of these controls uses a **Binding** markup extension in XAML to specify which property needs to be bound.

See also

Data binding was used in this application. It's also the topic of *Chapter 2, An Introduction to Data Binding* and *Chapter 3, Advanced Data Binding*, where we delve deep into what data binding has to offer. We also connected with a WCF service. Connecting and communicating with services is covered in *Chapter 7, Talking to Services*, through *Chapter 10, Talking to REST and WCF Data Services*.

Using the workflow between Visual Studio 2010 and Blend 5

Applies to Silverlight 4 and 5

Expression Blend (currently at version 5) is part of Microsoft's **Expression Suite**. It's a designer tool that allows designers to create compelling user experiences for use with WPF and Silverlight. While aimed at designers, it's a tool that should be in a Silverlight developer's toolbox as well. In some areas, it offers a richer designer experience than that of Visual Studio. One of the best examples of this is the timeline that makes it easy to create time-based animations.

In this recipe, we'll look at how Visual Studio and Blend integrate. When used together, they help us create our applications faster. In *Chapter 2, An Introduction to Data Binding*, we'll take another look at Blend—namely at its features that support data binding.

Getting ready

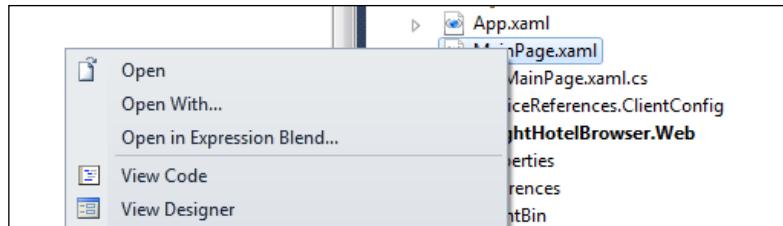
After having read the recipe *Getting our environment ready to start developing Silverlight applications*, you should have Expression Blend 5 installed.

In this recipe, we are creating the sample from scratch. The completed solution can be found in the Chapter01/Blend folder in the code bundle that is available on the Packt website.

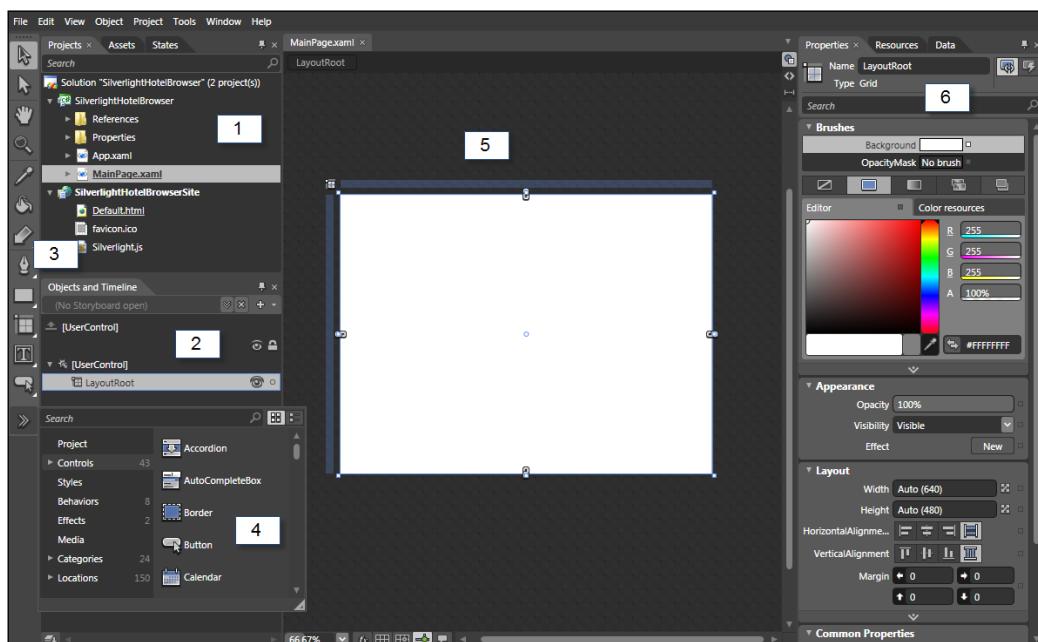
How to do it...

In this recipe, we'll recreate the Hotel Browser application. However, we'll do most of the work from Blend and switch back to Visual Studio when it is recommended. We'll need to carry out the following steps:

1. Although we can start a new solution from Blend, we'll let Visual Studio create the solution for us. The main reason is that we'll be using services later on in this sample, and working with services is easier if the hosting site is an ASP.NET web application. Adding an ASP.NET web application is possible from Visual Studio, but not from Blend. Therefore, open Visual Studio 2010 and create a new Silverlight solution. Name it **SilverlightHotelBrowser**, and make sure to select **ASP.NET Web Application Project** as the hosting website.
2. With the solution created in Visual Studio, right-click on one of the XAML files in the Silverlight project. In the context menu, select **Open in Expression Blend...** as shown in the following screenshot:



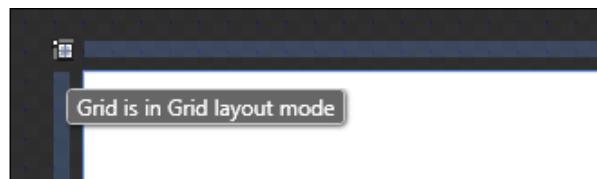
3. Expression Blend will open up and its workspace will be shown. The following is an image of the interface containing some numbered references:



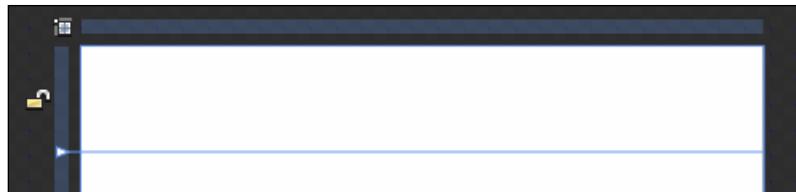
The following table describes some of the most important items in the Blend workspace:

Item	Name	Description
1	Projects window	Provides an overview of the loaded solution and its projects. It is comparable to the Solution Explorer in Visual Studio.
2	Objects and Timeline	By default, this window gives an overview of all the XAML objects in the currently loaded document. When we want to perform any action on an item (such as giving it a background color), we select it in the Objects and Timeline window. This opens the properties window for that item.
3	Toolbox	Comparable to what we know from Visual Studio, the toolbox contains all the tools available. Since Blend is a design tool, tools such as a Pen, Paint Bucket, and so on are available in the toolbox.
4	Assets window	The Assets window contains all controls (assets) that we can drag onto the design surface, such as buttons, ComboBoxes, and so on.
5	Design workspace	This is where all the action takes place! We can drag items from the Toolbox or the Assets window, rearrange them, and so on, to create a compelling user experience.
6	Properties window	The Properties window allows us to change the properties of the selected item. We can change the color, layout properties, transform properties, and so on.

4. Now that we know our way around the designer, we can get creative. We'll start with `MainPage.xaml` and split the main Grid (`LayoutRoot`) into two rows. Instead of writing the XAML code for this, we'll do this in the designer. Click on the icon on the top left of the user control in the designer, so that the Grid will be in the **Grid layout mode**. This can be seen in the following screenshot:



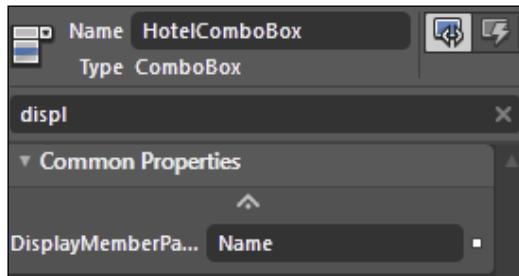
5. Now, click on the left bar next to the user control to add a row. It's possible to change the height of the created row by dragging the handle. The following screenshot shows a row added to the Grid:



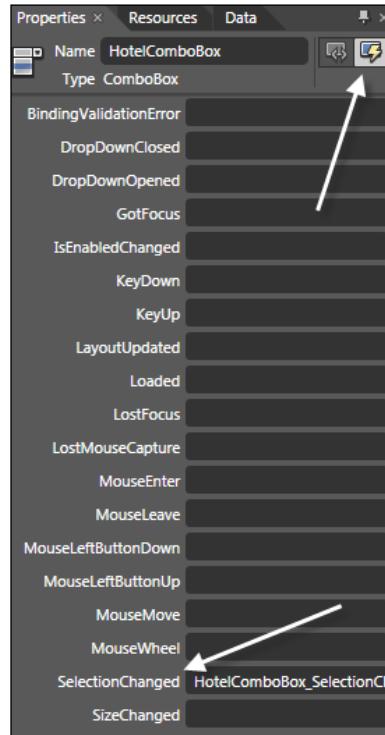
6. Select the **ComboBox** in the **Assets** window. Use the search function in this window to find it more quickly. On the designer, drag to create an instance of the **ComboBox** and place it on the top row that was just created. This can be seen in the following screenshot:



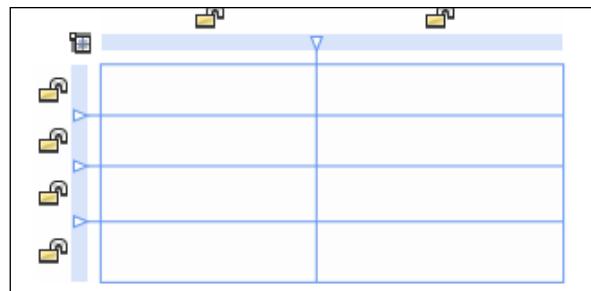
7. In the **Properties** window, give this **ComboBox** the name **HotelComboBox** and set the **DisplayMemberPath** property to **Name**. In the following screenshot, note that we are making use of the **Search** functionality within the **Properties** window. Simply enter part of the name of the property you are looking for (here **displ**), and Blend will filter the available properties:



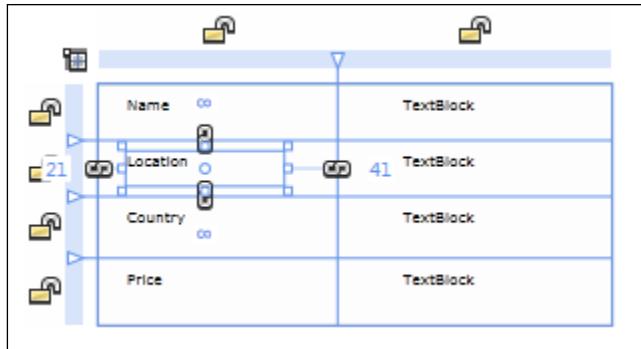
8. With the **ComboBox** still selected in the **Properties** window, change to the **Events** view (top arrow in the following image). In the list of events, double-click on the SelectionChanged event, so Blend will create an event handler (bottom arrow in the following image):



9. Let's now move back to the **Design** view of `MainPage.xaml`. Select the Grid item in the **Toolbox**. In the bottom cell of the LayoutRoot (the main Grid control), drag to create a nested Grid. Create four rows and two columns using the same technique as before. Columns are created quite logically by clicking on the top bar of the control. The result is shown in the following screenshot:



10. With this Grid still selected, change the name to **HotelDetailGrid** in the **Properties** window.
11. In each of the cells, drag a **TextBlock** from the **ToolBox**. For the **TextBlock** controls in the first column, change the **Text** property as shown in the following screenshot. Don't change the **Text** property of the controls in the second column; we'll look at these in the coming steps:



12. Let's now change the background color of the **LayoutRoot** grid. To do this, select the **LayoutRoot** node in the **Objects and Timeline** window, and in the **Properties** window, change the background by selecting a different color in the editor.
13. In *Chapter 2, An Introduction to Data Binding*, we'll look at how we can make data binding in Blend easier. As of now, we'll just type the XAML code from Blend. In the top-right corner of the **Design Surface**, select either the Split view or the XAML view. Blend shows us the XAML code that it created in the background. Search for the **TextBlock** controls in the second column of the **HotelDetailGrid** and change it as shown in the following code. Note that the generated code might not always be exactly the same, as values such as Margin could be different:

```

<TextBlock
    Margin="49,8,40,8"
    Grid.Column="1"
    Text="{Binding Name}"
    TextWrapping="Wrap" />
<TextBlock
    Margin="49,8,40,8"
    Grid.Column="1"
    Grid.Row="1"
    Text="{Binding Location}"
    TextWrapping="Wrap" />
<TextBlock
    Margin="49,8,40,8"
    Grid.Column="1"
    Grid.Row="2"
    Text="{Binding Country}"
    TextWrapping="Wrap" />
<TextBlock
    Margin="49,8,40,8"
    Grid.Column="1"
    Grid.Row="3"
    Text="{Binding Price}"
    TextWrapping="Wrap" />

```

```
    Grid.Row="2"
    Text="{Binding Country}"
    TextWrapping="Wrap"/>
<TextBlock
    Margin="49,8,40,8"
    Grid.Column="1"
    Grid.Row="3"
    Text="{Binding Price}"
    TextWrapping="Wrap"/>
```

14. The workflow between Blend and Visual Studio allows us to jump to Visual Studio for the tasks we can't achieve in Blend, for example, adding a WCF service and referencing it in the Silverlight project. In the **Projects** window, right-click on a file or a project and select **Edit** in Visual Studio. If Visual Studio is still open, it will reactivate. If not, a new instance will get launched with our solution.

15. In the website that was created with the project initialization (**SilverlightHotelBrowser.Web**), we need to have a service that will return the hotel information. A hotel is represented by an instance of the Hotel class. Include a using statement for the System.Runtime.Serialization namespace as well:

```
[DataContract]
public class Hotel
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public string Location { get; set; }
    [DataMember]
    public string Country { get; set; }
    [DataMember]
    public double Price { get; set; }
}
```

16. Of course, we need to add the service as well. To do this, add a Silverlight-enabled WCF service called HotelService. Replace the DoWork sample method with the following code:

```
[OperationContract]
public List<Hotel> GetHotels()
{
    return new List<Hotel>
    {
        new Hotel
        {
            Name = "Brussels Hotel",
            Price = 100,
```

```

        Location = "Brussels",
        Country = "Belgium"
    },
    new Hotel
    {
        Name = "London Hotel",
        Price = 200,
        Location = "London",
        Country = "United Kingdom"
    },
    new Hotel
    {
        Name = "Paris Hotel",
        Price = 150,
        Location = "Paris",
        Country = "France"
    },
    new Hotel
    {
        Name = "New York Hotel",
        Price = 230,
        Location = "New York",
        Country = "USA"
    }
);
}
}

```

17. Perform a build of the project, so that the service is built and is ready to be referenced by the Silverlight application.
18. In the Silverlight project, add a service reference by right-clicking on the project and selecting **Add Service Reference....** Click on the **Discover** button and the service should be found. Set the namespace to HotelService.
19. In the MainPage.xaml.cs, add the following code to load the hotels in the ComboBox control:

```

public MainPage()
{
    InitializeComponent();

    HotelService.HotelServiceClient proxy = new
        SilverlightHotelBrowser.HotelService.HotelServiceClient();
    proxy.GetHotelsCompleted += new
        EventHandler<SilverlightHotelBrowser.HotelService.
        GetHotelsCompletedEventArgs>(proxy_GetHotelsCompleted);
    proxy.GetHotelsAsync();
}

```

```
    }
    void proxy_GetHotelsCompleted(object sender,
        SilverlightHotelBrowser.HotelService.
        GetHotelsCompletedEventArgs e)
    {
        HotelComboBox.ItemsSource = e.Result;
    }
```

20. In the SelectionChanged event handler of the ComboBox, add the following code to load the details of a hotel, once the user selects a different option:

```
private void HotelComboBox_SelectionChanged(object sender,
    System.Windows.Controls.SelectionChangedEventArgs e)
{
    HotelDetailGrid.DataContext = (sender as ComboBox).SelectedItem
        as HotelService.Hotel;
}
```

21. Build and run the application in Visual Studio.

With these steps completed, we have created the application using both Blend and Visual Studio. For an application as easy as this one, there is less profit in switching between the two environments. However, with larger applications requiring large teams, both developers and designers, this strong integration can turn out to be very helpful.

How it works...

Visual Studio and Blend integrate nicely with each other. It's easy to jump from one application to the other. This allows a great workflow between designers and developers.

Designers can work in Blend and the changes made in this tool are automatically picked up by Visual Studio, and vice versa. This is achieved through the use of the same files (both code files and project files) by the two tools. A Silverlight solution created in Blend will open in Visual Studio. The same holds true for a Silverlight solution created in Visual Studio; Blend can work with it.

See also

In *Chapter 2, An Introduction to Data Binding*, we'll perform data binding directly from Blend.

Using source control in Visual Studio 2010 and Blend 5

Applies to Silverlight 4 and 5

When working on slightly larger projects in teams, source control is an absolute necessity. By far, the most popular source control system in the Microsoft world today is **Team Foundation Server (TFS)**. This recipe explains all that we need to get TFS to work with Silverlight applications in Visual Studio and Blend. However, it doesn't explain how to work with TFS itself.

Note that for this recipe, access to a working instance of TFS is required.

Getting ready

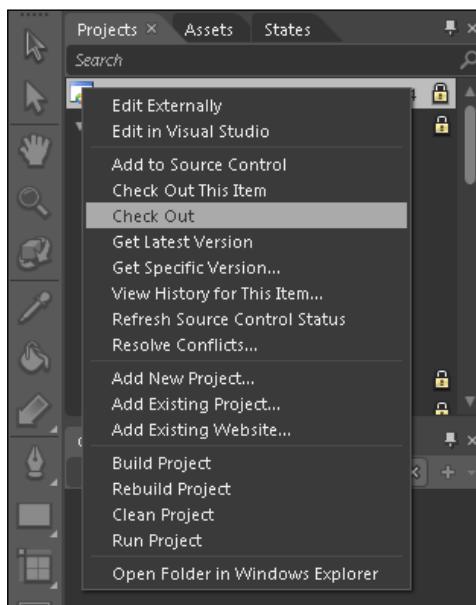
Before getting started, make sure that you have installed Blend 5 and the necessary developer tools as described in the recipe *Getting our environment ready to start developing Silverlight applications*.

How to do it...

To start using TFS as a versioning system with a Silverlight-enabled solution, we need to perform the following steps:

1. Using TFS source control with a Silverlight application in Visual Studio is exactly the same as using it with any other type of Visual Studio project. Team Explorer is used for this purpose, and this is automatically installed out of the box with Visual Studio 2010. (For Visual Studio 2008, it has to be separately installed from <http://www.microsoft.com/download/en/details.aspx?DisplayLang=en&id=16338>)
2. Blend 5 also supports source control out-of-the-box.

3. Once these are installed, we can use TFS source control from within Blend for any solution bound to the TFS source control. Right-click on any file in the **Solution Explorer** window to view the source control options, just as we would do in Visual Studio's Solution Explorer window. We can now check out, check in, and merge files from Blend. The following screenshot shows the checkout process:



How it works...

Team Foundation Server source control is the preferred way of enabling version control on our projects. Explaining in detail how to work with TFS is beyond the scope of this book, but the following are a few basic steps and references:

1. We must make sure that we have the correct permissions on our Team Foundation Server to handle the tasks we need to do. We need different permissions to (for example) create projects than to check out files. Have a look at this MSDN article to learn how to set these permissions: <http://msdn.microsoft.com/en-us/library/ms252587.aspx>
2. Next, we'll need to connect to TFS using Team Explorer. Have a look at this MSDN article to learn how to do that: <http://msdn.microsoft.com/en-us/library/ms181474.aspx>
3. We'll now need to create a workspace on our machine. We can look at the workspace as a local folder containing copies of the source-controlled files on the TFS. For more information, have a look at this MSDN article: <http://msdn.microsoft.com/en-us/library/ms181384.aspx>

4. After we've created a local workspace, we can download the files from TFS to that local folder. More information about this can be found at <http://msdn.microsoft.com/en-us/library/ms181385.aspx>
5. We're now able to open our source-controlled solution in Blend and/or Visual Studio, check out files, merge files, add projects, and so on, depending on the permissions.

There's more...

While working a lot with TFS, an interesting feature to download is the Team Foundation Server Power Tools. This is a set of extra features that is added to TFS and is mainly aimed at power users. It can be downloaded at <http://msdn.microsoft.com/en-us/teamsystem/bb980963.aspx>

Commonly used terms in TFS

A complete glossary of TFS terms can be found at <http://msdn.microsoft.com/en-us/library/ms242882.aspx>. As a reference, the following are a few of the more commonly used ones along with a brief explanation:

Term	Explanation
TFS workspace	A location on the Team Foundation Server (TFS), where a record of the changes between local files and corresponding repository files is stored. It can also be thought of as a copy of the client-side directory, a staging ground where local changes are persisted until they are checked into the server, or a collection of working folder mappings that can be viewed and edited.
Working folder	A client-side representation of the TFS workspace. Binding the TFS workspace to the client-side working folder is done through a TFS workspace mapping.
Check in	The task of committing a pending change/pending changes to a TFS repository. When you check in pending changes, a new changeset is created on the server.
Check out	The task of notifying the TFS server that you are changing the status of a resource from locked to writeable. When you check out for edit, TFS appends an edit to that resource.
Get latest	The task of retrieving the most recent version of a file from the TFS source control to your local working folder.

Deploying a Silverlight application on the server

Applies to Silverlight 4 and 5

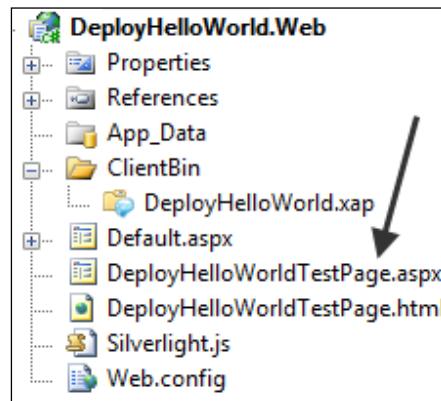
Once we have a Silverlight application ready, we will want to show it to the rest of the world. This means deploying it!

While Silverlight is a .NET technology, it doesn't require .NET to be installed on the server. Remember that it's a client-side technology. The Silverlight plugin on the client will download and run the application, using the version of the **Common Language Runtime (CLR)** embedded in the Silverlight plugin. In this recipe, we'll look at how we can deploy a Silverlight application.

How to do it...

Deploying a Silverlight application is easy; the Silverlight code is compiled and wrapped into a *.xap file. Getting this file on the client side and running it from there is our only concern. The following steps are to be carried out to deploy a Silverlight application:

1. We'll use the DeployHelloWorld application to demonstrate deployment, which is available with the code downloads in the Chapter01/DeployHelloWorld folder. Build the application and notice that Visual Studio has created a *.xap file in the ClientBin directory. This file, which is nothing more than a *.zip file but with another extension, contains the assembly (one or more) to which our Silverlight application was compiled, optional resources, and the AppManifest.xaml file.
2. While looking at the files created by default by Visual Studio in the web project, sample HTML (DeployHelloWorldTestPage.html) and ASPX (DeployHelloWorldTestPage.aspx) pages are created for us as shown in the following screenshot:



3. Both pages have an `object` tag included. One of the parameters is named `source` and it has a reference to the `*.xap` file in the `ClientBin`, as shown in the following code. If we want to deploy the `*.xap` file to another location, we need to update this reference. We'll use the default:

```
<object data="data:application/x-silverlight-2,"  
       type="application/x-silverlight-2"  
       width="100%"  
       height="100%">  
  <param name="source" value="ClientBin/DeployHelloWorld.xap"/>  
  <param name="onError" value="onSilverlightError" />  
  <param name="background" value="white" />  
  <param name="minRuntimeVersion" value="5.0.60401.0" />  
  <param name="autoUpgrade" value="true" />  
  <a href="http://go.microsoft.com/fwlink/?LinkId=149156&v=4.0.4  
      1108.0"  
      style="text-decoration:none">  
      
  </a>  
</object>
```



Note that the value of `minRuntimeVersion` may differ slightly because of different Silverlight version releases.

4. If using the HTML page, the following files need to be copied:

- `DeployHelloWorldTestPage.html`
- `Silverlight.js`
- `ClientBin/DeployHelloWorld.xap`

5. If using the ASPX page, we need to copy the following files:

- `DeployHelloWorldTestPage.aspx`
- `Silverlight.js`
- `ClientBin/DeployHelloWorld.xap`
- `bin` directory, if using code-behind for the ASPX page
- `web.config`

6. We'll need to test the page in a browser. If it fails to load, check the MIME types served by the web server software. There should be *.xap and *.xaml in there. (They are specified as the data type in the object tag).

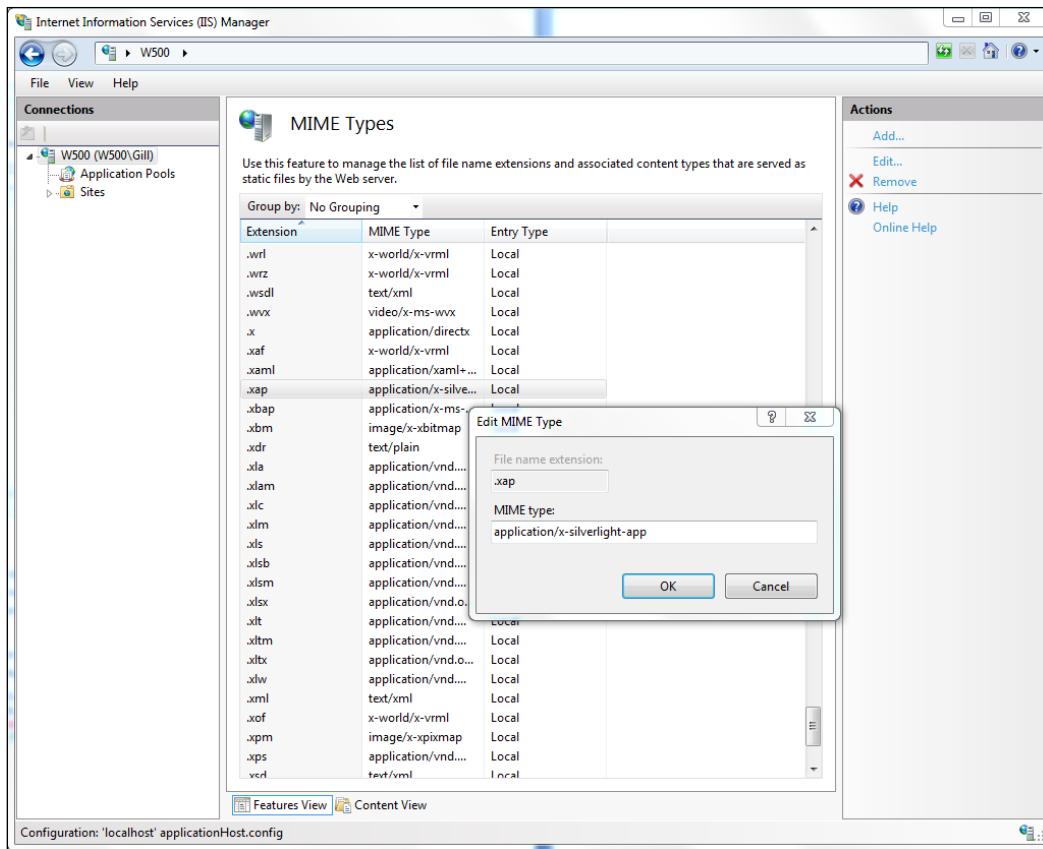
How it works...

One of the best things about Silverlight is that it can run from any type of server. If we're using ASP.NET, PHP, JSP, or plain-old HTML, Silverlight can still be embedded. Silverlight runs on the client side. The plugin has a CLR embedded, so that it hosts our application. On the server side, the only thing we need to do is to serve the files (most importantly, the *.xap file) that will be downloaded to the client side when requested.

Configuration changes on the server

If the Silverlight application isn't being shown, it might be that the server software (IIS or Apache) is not configured to serve the file types used by Silverlight (*.xap and *.xaml). Windows Vista SP1 and Windows Server ship including **Internet Information Services (IIS 7)**, while Windows 7 and Windows Server 2008 R2 include **IIS 7.5**. On these OS versions, both IIS 7 and IIS 7.5 are configured out-of-the-box to serve *.xap and *.xaml files. On Windows Vista without SP1, we need to add these to the known MIME types. We can do this by opening **Internet Information Services (IIS) Manager** and selecting **MIME Types**. Then, we simply click on **Add** and add the following two items:

- ▶ .xap in the **File name extension:** field, and application/x-silverlight-app in the **MIME type:** field
- ▶ .xaml in the **File name extension:** field, and application/xaml+xml in the **MIME type:** field



What if the server doesn't allow using XAP?

If the server environment doesn't allow adding MIME types (a shared hosting plan), there's no reason to panic. As a *.xap file is nothing more than a *.zip file, but with another extension, Silverlight supports the *.xap file being deployed as a *.zip file.

To get things working, start by renaming the *.xap file in the ClientBin to *.zip. Also, replace the reference of the *.xap file to the new name as shown in the following code:

```
<object data="data:application/x-silverlight-2,"  
       type="application/x-silverlight-2"  
       width="100%"  
       height="100%>  
  <param name="source" value="ClientBin/DeployHelloWorld.zip"/>  
</object>
```


2

An Introduction to Data Binding

In this chapter, we will cover:

- ▶ Displaying data in Silverlight applications
- ▶ Creating dynamic bindings
- ▶ Binding data to another UI element
- ▶ Binding collections to UI elements
- ▶ Enabling a Silverlight application to automatically update its UI
- ▶ Obtaining data from any UI element it is bound to
- ▶ Using the different modes of data binding to allow persisting data
- ▶ Debugging data binding expressions in Visual Studio
- ▶ Data binding from Expression Blend 5
- ▶ Using Expression Blend 5 for sample data generation

Introduction

Data binding allows us to build data-driven applications in Silverlight in a much easier and much faster way, compared to old-school methods of displaying and editing data. This chapter and the following one will take a look at how data binding works. We'll start by looking at the general concepts of data binding in Silverlight 5 in this chapter.

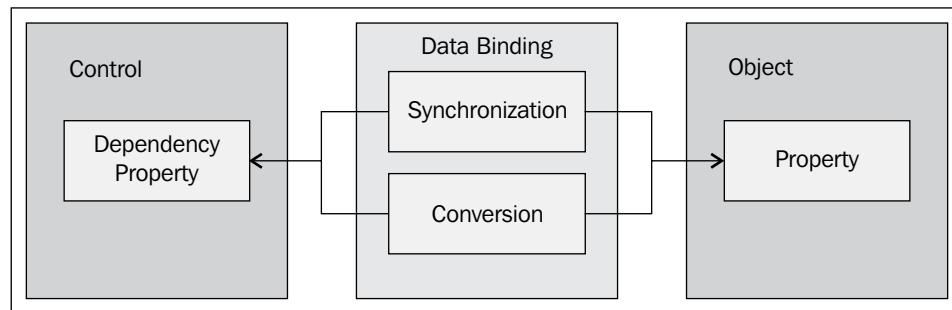
Analyzing the term data binding immediately reveals its intentions. It is a technique that allows us to bind properties of controls to objects or collections thereof.

The concept is, in fact, not new. Technologies such as ASP.NET, Windows Forms, and even older technologies, such as **Microsoft Foundation Classes (MFC)** include data binding features. However, WPF's data binding platform has changed the way we perform data binding; it allows loosely coupled bindings. The BindingSource control in Windows Forms has to know of the type we are binding to at design time. WPF's built-in data binding mechanism does not. We simply define which property of the source the target should bind. At runtime, the actual data—the object to which we are binding—is linked. Luckily for us, Silverlight inherits almost all data binding features from WPF, and thus has a rich way of displaying data.

A binding is defined by the following four items:

- ▶ **The source or source object:** This is the data we are binding to. The data that is used in data binding scenarios is in-memory data, that is, objects. Data binding itself has nothing to do with the actual data access. It works with the objects that are a result of reading from a database or communicating with a service. A typical example is a `Customer` object.
- ▶ **A property on the source object:** This can, for example, be the `Name` property of the `Customer` object.
- ▶ **The target control:** This is normally a visual control, such as a `TextBox` or a `ListBox` control. In general, the target can be a `DependencyObject`. In Silverlight 2 and Silverlight 3, the target had to derive from `FrameworkElement`; this left out some important types such as transformations.
- ▶ **A property on the target control:** This will, in some way—directly or after a conversion—display the data from the property on the source.

The data binding process can be summarized in the following image:



In the previous image, we can see that the data binding engine is also capable of synchronization. This means that data binding is capable of updating the display of data automatically. If the value of the source changes, Silverlight will change the value of the target as well, without us having to write a single line of code. Data binding isn't a complete black box either. There are hooks in the process, so we can perform custom actions on the data flowing from source to target, and vice versa. These hooks are the converters and we'll look at converters in the next chapter.

Our applications can still be created without data binding. However, the manual process—that is, getting data and setting all values manually on controls from code-behind—is error-prone and tedious to write. Using the data-binding features in Silverlight, we will be able to write more maintainable code faster.

In this chapter, we'll explore how data binding works. We'll start by building a small data-driven application, which contains the most important data binding features, to get a grasp of the general concepts. We'll also see that data binding isn't tied to only binding single objects to an interface; binding an entire collection of objects is also supported. We'll also be looking at the binding modes. They allow us to specify how the data will flow (from source to target, target to source, or both). With Silverlight 5, one of the most anticipated features was added: the ability to debug XAML data binding statements using breakpoints in XAML. This enables us at run-time to break into XAML code and look at the values within the data binding statement. We'll learn how we can add breakpoints and what values we can see.

We'll finish this chapter by looking at the support that Blend 5 provides to build applications that use data binding features. This feature has been available in Blend since version 3. In the next chapter, we'll be looking at the more advanced concepts of data binding.

In the recipes of this chapter and the following one, we'll assume that we are building a simple banking application using Silverlight. Each of the recipes in this chapter will highlight a part of this application where the specific feature comes into play. The following image shows the resulting Silverlight banking application:

The screenshot displays a Silverlight application window titled "Silverlight Bank - Account overview". At the top left, there are "Actions" and "Calculate loans..." buttons. Below them is a placeholder for a user profile picture. To the right, a section titled "Activities on the account:" lists recent transactions:

Date	Description	Amount	Action
01/09	Smith Woodworking Shop London Paid by credit card	-\$33.00	Details...
01/09	ABC Infrastructure Paycheck September 2009	\$1,000.00	Details...
02/09	Money Withdrawal ATM Oxford Street London	\$50.00	Details...
05/09	Jones Food Store	-\$123.56	Details...
06/09	Davy's Diner Paid by credit card	-\$12.23	Details...
08/09	A&B Clothing Store London Paid by Direct Debit card	-\$29.99	Details...
10/09	Davy's Diner Paid by credit card	-\$14.55	Details...
10/09	Money received from An Smith Royal Bank money deposit	\$50.00	Details...
15/09	Manny's Record Store Paid by Direct Debit card	-\$78.81	Details...
18/09	Davy's Diner Paid by credit card	-\$27.09	Details...
18/09	Money withdrawal ATM In Some Dark Alley	-\$13.00	Details...

On the left side of the main content area, there is a summary of account details:

- Owner ID: 1234567
- First name: John
- Last name: Smith
- Address: Oxford Street 24
- Zip code: W1A
- City: London
- State: NA
- Country: United Kingdom
- Birthdate: 09-Jun-1953
- Customer since: 20-Dec-1999
- Current balance: 1221.56
- Last activity on: 8/7/2009 9:43:34 PM
- Amount: -13

Below these details is a "Edit details..." button.

If you want to take a look at the complete application, run the solution found in the Chapter02/SilverlightBanking folder in the code bundle that is available on the Packt website.

Displaying data in Silverlight applications

Applies to Silverlight 3, 4 and 5

When building Silverlight applications, we often need to display data to the end user. Applications such as an online store with a catalogue and a shopping cart, an online banking application and so on, need to display data of some sort.

Silverlight contains a rich data binding platform that will help us write data-driven applications faster and with less code. In this recipe, we'll build a form that displays the data of a bank account's owner using data binding.

Getting ready

To follow along with this recipe, you can use the starter solution located in the Chapter02/SilverlightBanking_Displaying_Data_Starter folder in the code bundle available on the Packt website. The finished application for this recipe can be found in the Chapter02/SilverlightBanking_Displaying_Data_Completed folder.

How to do it...

Let's assume that we are building a form, part of an online banking application, in which we can view the details of the owner of the account. Instead of wiring up the fields of the owner manually, we'll use data binding. To get data binding up and running, carry out the following steps:

1. Open the starter solution, as outlined in the *Getting Ready* section.
2. The form we are building will bind to data. Data in data binding is in-memory data, not the data that lives in a database (it can originate from a database, though). The data to which we are binding is an instance of the `Owner` class. The following is the code for the class. Add this code in a new class file called `Owner` in the Silverlight project:

```
public class Owner
{
    public int OwnerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
```

```
    public string State { get; set; }
    public string Country { get; set; }
    public DateTime BirthDate { get; set; }
    public DateTime CustomerSince { get; set; }
    public string ImageName { get; set; }
    public DateTime LastActivityDate { get; set; }
    public double CurrentBalance { get; set; }
    public double LastActivityAmount { get; set; }
}
```

3. Now that we've created the class, we are able to create an instance of it in the `MainPage.xaml.cs` file, the code-behind class of `MainPage.xaml`. In the constructor, we call the `InitializeOwner` method, which creates an instance of the `Owner` class and populates its properties as follows:

```
private Owner owner;
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
}
private void InitializeOwner()
{
    owner = new Owner();
    owner.OwnerId = 1234567;
    owner.FirstName = "John";
    owner.LastName = "Smith";
    owner.Address = "Oxford Street 24";
    owner.ZipCode = "W1A";
    owner.City = "London";
    owner.Country = "United Kingdom";
    owner.State = "NA";
    owner.ImageName = "man.jpg";
    owner.LastActivityAmount = 100;
    owner.LastActivityDate = DateTime.Today;
    owner.CurrentBalance = 1234.56;
    owner.BirthDate = new DateTime(1953, 6, 9);
    owner.CustomerSince = new DateTime(1999, 12, 20);
}
```

4. Let's now focus on the form itself and build its UI. For this sample, we're not making the data editable. For every field of the Owner class, we'll use a `TextBlock`. To arrange the controls on the screen, we'll use a `Grid` called `OwnerDetailsGrid`. This `Grid` can be placed inside the `LayoutRoot` grid. We will want the `Text` property of each `TextBlock` to be bound to a specific property of the `Owner` instance. This can be done by specifying this binding using the **Binding markup extension** on this property. A markup extension can be recognized by the surrounding curly braces (`{Binding CurrentBalance}`) The following XAML code shows part of the UI, refer to the downloads of the book for the complete listing:

```
<Grid x:Name="OwnerDetailsGrid"
      VerticalAlignment="Stretch"
      HorizontalAlignment="Left"
      Background="LightGray"
      Margin="3 5 0 0"
      Width="300" >
  <Grid.RowDefinitions>
    <RowDefinition Height="100"></RowDefinition>
    <RowDefinition Height="30"></RowDefinition>
    ...
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <Image x:Name="OwnerImage"
         Grid.Row="0"
         Width="100"
         Height="100"
         Stretch="Uniform"
         HorizontalAlignment="Left"
         Margin="3"
         Source="/CustomerImages/man.jpg"
         Grid.ColumnSpan="2">
  </Image>
  <TextBlock x:Name="OwnerIdTextBlock"
            Grid.Row="1"
            FontWeight="Bold"
            Margin="2"
            Text="Owner ID:>
  </TextBlock>
  <TextBlock x:Name="FirstNameTextBlock"
            Grid.Row="2"
            FontWeight="Bold"
```

```
        Margin="2"
        Text="First name:>
    </TextBlock>
    <TextBlock x:Name="LastNameTextBlock"
        Grid.Row="3"
        FontWeight="Bold"
        Margin="2"
        Text="Last name:>
    </TextBlock>
    ...
    </TextBlock>
    <TextBlock x:Name="CustomerSinceValueTextBlock"
        Grid.Row="10"
        Grid.Column="1"
        Margin="2"
        Text="{Binding CustomerSince}">
    </TextBlock>
    <Button x:Name="OwnerDetailsEditButton"
        Grid.Row="11"
        Grid.ColumnSpan="2"
        Margin="3"
        Content="Edit details..."
        HorizontalAlignment="Right"
        VerticalAlignment="Top">
    </Button>
    <TextBlock x:Name="CurrentBalanceValueTextBlock"
        Grid.Row="12"
        Grid.Column="1"
        Margin="2"
        Text="{Binding CurrentBalance}" >
    </TextBlock>
    <TextBlock x:Name="LastActivityDateValueTextBlock"
        Grid.Row="13"
        Grid.Column="1"
        Margin="2"
        Text="{Binding LastActivityDate}" >
    </TextBlock>
    <TextBlock x:Name="LastActivityAmountValueTextBlock"
        Grid.Row="14"
        Grid.Column="1"
        Margin="2"
        Text="{Binding LastActivityAmount}" >
    </TextBlock>
</Grid>
```

5. At this point, all the controls know what property they need to bind to. However, we haven't specified the actual link. The controls don't know about the `Owner` instance we want them to bind to. Therefore, we can use `DataContext`. We specify the `DataContext` of the `OwnerDetailsGrid` to be the `Owner` instance. Each control within that container can then access the object and bind to its properties. Setting the `DataContext` is done using the following code:

```
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    OwnerDetailsGrid.DataContext = owner;
}
```

6. The result can be seen in the following image:



How it works...

Before we take a look at the specifics of data binding, let's see what code we would need to write if Silverlight did not support data binding. The following is the `ManualOwner` class and we will be binding an instance of this class manually:

```
public class ManualOwner
{
    public int OwnerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public DateTime BirthDate { get; set; }
    public DateTime CustomerSince { get; set; }
    public string ImageName { get; set; }
    public DateTime LastActivityDate { get; set; }
    public double CurrentBalance { get; set; }
    public double LastActivityAmount { get; set; }
}
```

The XAML code would look the same, apart from the binding markup extensions that are absent, as we aren't using the data binding functionality. The following is a part of the code that has no data binding markup extensions:

```
<TextBlock x:Name="OwnerIdValueTextBlock"
           Grid.Row="1"
           Grid.Column="1"
           Margin="2" >
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
           Grid.Row="2"
           Grid.Column="1"
           Margin="2" >
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
           Grid.Row="3"
           Grid.Column="1"
           Margin="2" >
</TextBlock>
<TextBlock x:Name="AddressValueTextBlock"
           Grid.Row="4"
           Grid.Column="1"
           Margin="2" >
</TextBlock>
```

Of course, the `DataContext` would also not be needed. Instead, we would manually have to link all the `TextBlock` controls with a property of the `ManualOwner` from code-behind, as shown in the following code. As you can see, this is not the most exciting code one can write:

```
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    SetOwnerValues();
}
private void SetOwnerValues()
{
    OwnerIdValueTextBlock.Text = owner.OwnerId.ToString();
    FirstNameValueTextBlock.Text = owner.FirstName;
    LastNameValueTextBlock.Text = owner.LastName;
    AddressValueTextBlock.Text = owner.Address;
    //other values go here
}
```

It is also easy to make errors this way. When a field gets added to the `ManualOwner`, we need to remember the places in which we have to update our code manually.

However, we can do better using data binding. Data binding enables us to write less code and have fewer opportunities to make errors.

Silverlight's data binding features allow us to bind the properties of the `Owner` instance to the `Text` property of the `TextBlock` controls using the `Binding` markup extension. A markup extension can be recognized by a pair of curly braces (`{}`). It's basically a signal for the XAML parser that more needs to be done than simple attribute parsing. In this case, an instance of the `System.Windows.Data.Binding` needs to be created for data binding to happen. The created `Binding` instance will bind the source object with the target control.

Looking back at the XAML code, we find that this binding is achieved for each `TextBlock` using the following code:

```
<TextBlock Text="{Binding CustomerSince}" />
```

This is, in fact, the shortened format. We could have written it as the following code:

```
<TextBlock Text="{Binding Path=CustomerSince}" />
```

The format for the binding is generally the following:

```
<TargetControl TargetProperty="{Binding SourceProperty,
    SomeBindingProperties}" />
```

Note that using `SomeBindingProperties`, more options can be specified when creating the binding. For example, we can specify that data should not only flow from source object to target control, but also vice versa. We'll explore a whole list of extra binding properties in the next recipes.

Are we missing something? Each control knows what it should bind to, but we haven't specified the actual source of the data. This is done using the `DataContext`. We set the `Owner` instance to be the `DataContext` of the `Grid` containing the controls. All controls within the `Grid` can access the data. We'll look at the `DataContext` in a later recipe.

Finally, there is one important point to note; we can't just bind everything. Basically, there are two rules we must follow:

1. The target object must be a `DependencyObject` (`System.Windows.DependencyObject`). In Silverlight 2 and Silverlight 3, the target could be a `FrameworkElement` instance only. `FrameworkElement` is lower in the class hierarchy than `DependencyObject`. Because of this, some important objects could not be used in data binding scenarios, such as Transformations. Since Silverlight 4, this problem has been solved.
2. The target property must be a dependency property. Again, don't panic, as almost all properties on UI controls (such as text, foreground, and so on) are dependency properties.

Dependency properties were introduced with WPF and can be considered as properties on steroids. They include a mechanism that at any point in time determines what the value of the property should be, based on several influences working on the property, such as data binding, styling, and so on. They can be considered the enabler for animations, data binding, styling, and so on.

More on dependency properties can be found at <http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx>.



There's more...

Instead of creating the `Owner` instance in code, we can create it from XAML as well. First, we need to map the CLR namespace to an XML namespace as follows:

```
xmlns:local="clr-namespace:SilverlightBanking"
```

In the Resources collection of the container (the UserControl), we instantiate the type like:

```
<UserControl.Resources>
    <local:Owner x:Key="localOwner"
        City="London"
        Country="United Kingdom"
        FirstName="John"
        LastName="Smith"
        OwnerId="1234567" ...>
    </local:Owner>
</UserControl.Resources>
```

The actual binding is almost the same, apart from specifying the source. We are not using the DataContext now, but we need to use the Source in each binding, referring to the item in the Resources as follows:

```
<TextBlock x:Name="OwnerIdValueTextBlock"
    Grid.Row="1"
    Grid.Column="1"
    Margin="2"
    Text="{Binding OwnerId,
        Source={StaticResource localOwner}}" >
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
    Grid.Row="2"
    Grid.Column="1"
    Margin="2"
    Text="{Binding FirstName,
        Source={StaticResource localOwner}}" >
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
    Grid.Row="3"
    Grid.Column="1"
    Margin="2"
    Text="{Binding LastName,
        Source={StaticResource localOwner}}" >
</TextBlock>
```

Whether binding from XAML is useful or not depends on the scenario. In most scenarios, we bind to objects that are created at runtime from the code-behind. Binding to objects that get created from XAML can be useful in scenarios where design-time data is required. This then enables us to view sample data during the design of the application, both from Visual Studio and Blend.

See also

The `DataContext` makes its first appearance in this recipe, but we'll look at it in more detail in the *Obtaining data from any UI element it is bound to* recipe in this chapter.

Creating dynamic bindings

Applies to Silverlight 3, 4 and 5

In the previous recipe, you learned how to use data binding in XAML. This is often useful because it allows you to show data easily to your user, for example, showing user information or a list of products. In this recipe, you'll learn how to do exactly the same in C# code instead of XAML. This can be useful in situations where you want to bind a dependency property to the property of an object that you'll know only at runtime.

Getting ready

For this recipe, we can continue from the solution that was completed in the previous recipe. Alternatively, you can find the starter solution in the `Chapter02/SilverlightBanking_Dynamic_Bindings_Starter` folder in the code bundle that is available on the Packt website. Also, the completed solution can be found in the `Chapter02/SilverlightBanking_Dynamic_Bindings_Completed` folder.

How to do it...

We're going to change the code from the previous recipe, so we can create the bindings in C#, instead of XAML. To do this, we'll carry out the following steps:

1. Open the solution created in the previous recipe, *Displaying data in Silverlight applications*, locate the grid named `OwnersDetailsGrid` in `MainPage.xaml`, and remove the `Binding` syntax from the XAML code for each `TextBlock` as shown in the following code:

```
<TextBlock x:Name="OwnerIdValueTextBlock"
          Grid.Row="1"
          Grid.Column="1"
          Margin="2">
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
          Grid.Row="2"
          Grid.Column="1"
          Margin="2">
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
```

```
        Grid.Row="3"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="AddressValueTextBlock"
        Grid.Row="4"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="ZipCodeValueTextBlock"
        Grid.Row="5"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="CityValueTextBlock"
        Grid.Row="6"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="StateValueTextBlock"
        Grid.Row="7"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="CountryValueTextBlock"
        Grid.Row="8"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="BirthDateValueTextBlock"
        Grid.Row="9"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
    <TextBlock x:Name="CustomerSinceValueTextBlock"
        Grid.Row="10"
        Grid.Column="1"
        Margin="2">
    </TextBlock>
```

2. Open the code-behind `MainPage.xaml.cs` file. Here, we're going to create the same bindings in the C# code. In the constructor, after the call to `InitializeComponent()`, add the following code:

```
OwnerIdValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("OwnerId"));
FirstNameValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("FirstName"));
LastNameValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("LastName"));
```

```
AddressValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("Address"));  
ZipCodeValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("ZipCode"));  
CityValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("City"));  
StateValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("State"));  
CountryValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("Country"));  
BirthDateValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("BirthDate"));  
CustomerSinceValueTextBlock.SetBinding(TextBlock.TextProperty,  
    new Binding("CustomerSince"));
```

A `using` statement to `System.Windows.Data` must be added to make this code compile.

3. We can now build and run the application, and you'll notice that the correct data is still displayed in the details form. The result can be seen in the following image:



How it works...

This recipe shows you how to set the binding using C# syntax. `Element.SetBinding` expects two parameters, a dependency property, and a binding object. The first parameter defines the `DependencyProperty` of the element you want to bind. The second parameter defines the binding by passing a string that refers to the property path of the object to which you are binding.

There's more...

In our example, we've used `new Binding("path")` as the syntax. The binding object, however, has different properties that you can set and which can be of interest. A few of these properties are `Converter`, `ConverterParameter`, `ElementName`, `Path`, `Mode`, and `ValidatesOnExceptions`.

To know when and how to use these properties, have a look at the other recipes in this chapter, and the next recipe will explain all the possibilities in detail. They are, however, already mentioned in this recipe to make it clear that you can do everything that is required as far as bindings are concerned in both C# and XAML.

Binding data to another UI element

Applies to Silverlight 3, 4 and 5

Sometimes the value of the property of an element is directly dependent on the value of the property of another element. In this case, you can create a binding in XAML called an `element binding` or `element-to-element binding`. This binding links both values. If needed, the data can flow bi-directionally.

In the banking application, we can add a loan calculator that allows the user to select an amount and the number of years in which they intend to pay the loan back to the bank, including (of course) interest.

Getting ready

To follow this recipe, you can either continue with your solution from the previous recipe or use the provided solution that can be found in the `Chapter02/SilverlightBanking_Element_Binding_Starter` folder in the code bundle that is available on the Packt website. The finished application for this recipe can be found in the `Chapter02/SilverlightBanking_Element_Binding_Completed` folder.

How to do it...

To build the loan calculator, we'll use `Slider` controls. Each `Slider` is bound to a `TextBlock` using an element-to-element binding to display the actual value. Let's take a look at the steps that we need to follow to create this binding:

1. We will build the loan calculator as a separate screen in the application. Add a new child window called `LoanCalculation.xaml`. To do so, right-click on the Silverlight project in the **Solution Explorer**, select **Add | New Item...**, and choose **Silverlight Child Window** under Visual C#.
2. Within `MainPage.xaml`, add a `Click` event on the `LoanCalculationButton`, as shown in the following code:

```
<Button x:Name="LoanCalculationButton"
        Click="LoanCalculationButton_Click" />
```

3. In the code-behind's event handler for this `Click` event, we can trigger the display of this new screen with the following code:

```
private void LoanCalculationButton_Click(object sender,
                                         RoutedEventArgs e)
{
    LoanCalculation loanCalculation = new LoanCalculation();
    loanCalculation.Show();
}
```

4. The UI of the `LoanCalculation.xaml` is quite simple—it contains two `Slider` controls. Each slider control has set values for its `Minimum` and `Maximum` values (not all UI code is included here; the complete listing can be found in the finished sample code), as shown in the following code:

```
<Slider x:Name="AmountSlider"
        Minimum="10000"
        Maximum="1000000"
        SmallChange="10000"
        LargeChange="10000"
        Width="300" >
</Slider>
<Slider x:Name="YearSlider"
        Minimum="5"
        Maximum="30"
        SmallChange="1"
        LargeChange="1"
        Width="300"
        UseLayoutRounding="True">
</Slider>
```

5. As dragging a `Slider` does not give us proper knowledge of where we are exactly between the two values, we add two `TextBlock` controls. We want the `TextBlock` controls to show the current value of the `Slider` control, even while dragging. This can be done by specifying an element-to-element binding, as shown in the following code:

```
<TextBlock x:Name="AmountTextBlock"
           Text="{Binding ElementName=AmountSlider, Path=Value}">
</TextBlock>
<TextBlock x:Name="MonthTextBlock"
           Text="{Binding ElementName=YearSlider, Path=Value}">
</TextBlock>
```

6. Add a `Button` that will perform the actual calculation called `CalculateButton`, and a `TextBlock` called `PaybackTextBlock` to show the results. This can be done using the following code:

```
<Button x:Name="CalculateButton"
        Content="Calculate"
        Click="CalculateButton_Click">
</Button>
<TextBlock x:Name="PaybackTextBlock"></TextBlock>
```

7. The code for the actual calculation that is executed when the `CalculateButton` is clicked uses the actual value for either the `Slider` or the `TextBlock`. This is shown in the following code:

```
private double percentage = 0.0345;
private void CalculateButton_Click(object sender,
                                   RoutedEventArgs e)
{
    double requestedAmount = AmountSlider.Value;
    int requestedYears = (int)YearSlider.Value;
    for (int i = 0; i < requestedYears; i++)
    {
        requestedAmount += requestedAmount * percentage;
    }
    double monthlyPayback =
        requestedAmount / (requestedYears * 12);
    PaybackTextBlock.Text =
        "€" + Math.Round(monthlyPayback, 2);
}
```

Having carried out the previous steps, we now have successfully linked the value of the Slider controls and the text of the TextBlock controls. The following screenshot shows the LoanCalculation.xaml screen, as it is included in the finished sample code containing some extra markup:



How it works...

An element binding links two properties of two controls directly from XAML. It allows us to create a Binding, where the source object is another control. For this to work, we need to create a Binding and specify the source control using the ElementName property. This is shown in the following code:

```
<TextBlock Text="{Binding ElementName=YearSlider, Path=Value}">  
</TextBlock>
```

Element bindings were added in Silverlight 3. Silverlight 2 did not support this type of binding.

There's more...

An element binding can also work in both directions, that is, from source to target and target to source. This can be achieved by specifying the Mode property on the Binding and setting it to TwoWay.

Take a look at the following code, where we replaced the TextBlock by a TextBox. When entering a value in the latter, the slider will adjust its position:

```
<TextBox x:Name="AmountTextBlock"  
        Text="{Binding ElementName=AmountSlider, Path=Value,  
                    Mode=TwoWay}">  
</TextBox>
```

Element bindings without bindings

Achieving the same effect in Silverlight 2—which does not support this feature—is also possible, but only through the use of an event handler, as shown in the following code. Element bindings eliminate this need:

```
private void AmountSlider_ValueChanged(object sender,
    RoutedPropertyChangedEventArgs<double> e)
{
    AmountSlider.Value = Math.Round(e.NewValue);
    AmountTextBlock.Text = AmountSlider.Value.ToString();
}
```

See also

Element-to-element bindings can be easily extended to use converters. For more information on TwoWay bindings, take a look at the recipe *Using the different modes of data binding to allow persisting data*, in this chapter.

Binding collections to UI elements

Applies to Silverlight 3, 4 and 5

Often, you'll want to display lists of data in your application, such as a list of shopping items, a list of users, a list of bank accounts, and so on. Such a list typically contains many items of a certain type that have the same properties and need to be displayed in the same fashion.

We can use data binding to easily bind a collection to a Silverlight control (such as a `ListBox` or `DataGrid`), and use the same data binding possibilities to define how every item in the collection should be bound. This recipe will show you how to achieve this.

Getting ready

For this recipe, you can find the starter solution in the `Chapter02/SilverlightBanking_Binding_Collections_Starter` folder and the completed solution in the `Chapter02/SilverlightBanking_Binding_Collections_Completed` folder in the code bundle that is available on the Packt website.

How to do it...

In this recipe, we'll create a `ListBox` bound to a collection of activities. To complete this task, carry out the following steps:

1. We'll need a collection of some kind. We'll create a new type, `AccountActivity`.

Add the `AccountActivity` class to your Silverlight project as shown in the following code:

```
public class AccountActivity
{
    public int ActivityId {get; set;}
    public double Amount { get; set; }
    public string Beneficiary { get; set; }
    public DateTime ActivityDate { get; set; }
    public string ActivityDescription { get; set; }
}
```

Add an `ObservableCollection` of `AccountActivity` to `MainPage.xaml.cs`, using the following code:

```
private ObservableCollection<AccountActivity>
    accountActivitiesCollection;
```

2. Now we'll instantiate `accountActivitiesCollection` and fill it with data. To do this, add the following code to `MainPage.xaml.cs`:

```
private void InitializeActivitiesCollection()
{
    accountActivitiesCollection = new
        ObservableCollection<AccountActivity>();
    AccountActivity accountActivity1 = new AccountActivity();
    accountActivity1.ActivityId = 1;
    accountActivity1.Amount = -33;
    accountActivity1.Beneficiary = "Smith Woodworking Shop London";
    accountActivity1.ActivityDescription = "Paid by credit card";
    accountActivity1.ActivityDate = new DateTime(2009, 9, 1);
    accountActivitiesCollection.Add(accountActivity1);
    AccountActivity accountActivity2 = new AccountActivity();
    accountActivity2.ActivityId = 2;
    accountActivity2.Amount = 1000;
    accountActivity2.Beneficiary = "ABC Infrastructure";
    accountActivity2.ActivityDescription = "Paycheck September
        2009";
    accountActivity2.ActivityDate = new DateTime(2009, 9, 1);
    accountActivitiesCollection.Add(accountActivity2);
}
```

This creates a collection with two items. You can add more if you want to.

3. Add the following code to the `MainPage` constructor to call the method you created in the previous step:

```
InitializeActivitiesCollection();
```

4. We're going to need a control to display these `AccountActivity` items. To do this, add a `ListBox` called `AccountActivityListBox`. This `ListBox` defines a `DataTemplate` that defines how each `AccountActivity` is displayed:

```
<Grid Background="DarkGray" Grid.Row="2" Grid.Column="1"
VerticalAlignment="Stretch">
    <ListBox x:Name="AccountActivityListBox"
        Width="600"
        Grid.Row="1">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Grid.RowDefinitions>
                        <RowDefinition></RowDefinition>
                        <RowDefinition></RowDefinition>
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="150" />
                        <ColumnDefinition Width="330" />
                        <ColumnDefinition Width="100" />
                    </Grid.ColumnDefinitions>
                    <TextBlock
                        Grid.Row="0"
                        Grid.Column="0"
                        Grid.RowSpan="2"
                        Text="{Binding ActivityDate}">
                    </TextBlock>
                    <TextBlock
                        Grid.Row="0"
                        Grid.Column="1"
                        Text="{Binding Beneficiary}"
                        FontWeight="Bold">
                    </TextBlock>
                    <TextBlock
                        Grid.Row="0"
                        Grid.Column="2"
                        HorizontalAlignment="Right"
                        Text="{Binding Amount}">
                    </TextBlock>
                    <TextBlock
                        Grid.Row="1">
```

```

        Grid.Column="1"
        Text="{Binding ActivityDescription}">
    </TextBlock>
</Grid>
</DataTemplate>
<ListBox.ItemTemplate>
</ListBox>

```

5. In the MainPage constructor, set the `ObservableCollection` of `AccountActivity` you created in step 2 as the `ItemsSource` of the `ListBox`, as shown in the following code:

```
AccountActivityListBox.ItemsSource = accountActivitiesCollection;
```

6. If we build and run the application now, we'll see that a list of `AccountActivity` items is displayed, as shown in the following image:

9/1/2009 12:00:00 AM	Smith Woodworking Shop London	-33
	Paid by credit card	
9/1/2009 12:00:00 AM	ABC Infrastructure	1000
	Paycheck September 2009	
9/2/2009 12:00:00 AM	Money Withdrawal	50
	ATM Oxford Street London	
9/5/2009 12:00:00 AM	Jones Food Store	-123.56
9/6/2009 12:00:00 AM	Davy's Diner	-12.23
	Paid by credit card	
9/8/2009 12:00:00 AM	A&B Clothing Store London	-29.99
	Paid by Direct Debit card	
9/10/2009 12:00:00 AM	Davy's Diner	-14.55
	Paid by credit card	

How it works...

The first three steps aren't important for people who have worked with collections before. A class is created to define the type of items that are held by the collection, which is initialized and then items are added to it. The default collection type to use in Silverlight is `ObservableCollection`. We're using this collection type here. (For more information about this, have a look at the *There's more...* section in this recipe.)

The real magic happens in step 4 and step 5. In step 4, we are creating a `ListBox`, which has an `ItemTemplate` property. This `ItemTemplate` property should contain a `DataTemplate`, and it's this `DataTemplate` that defines how each item of the collection should be visualized. The `DataTemplate` corresponds to one item of your collection that one `AccountActivity`. This means we can use the data binding syntax that binds to properties of an `AccountActivity` in this `DataTemplate`.

When the `ItemsSource` property of the `ListBox` gets set to the `ObservableCollection` of `AccountActivity`, each `AccountActivity` in the collection is evaluated and visualized as defined in the `DataTemplate`.

There's more...

An `ObservableCollection` is the default collection type you'll want to use in a Silverlight application, because it's a collection type that implements the `INotifyCollectionChanged` interface. This makes sure that the UI can automatically be updated when the collection is changed (by adding or deleting an item). More on this can be found in the recipe *Enabling a Silverlight application to automatically update its UI*, next in this chapter.

The same principle applies for the properties of classes that implement the `INotifyPropertyChanged` interface. More on this can be found in the same recipe, *Enabling a Silverlight application to automatically update its UI*.

In this recipe, we're using a `ListBox` to visualize our `observableCollection`. However, every control that inherits the `ItemsControl` class (directly or indirectly) can be used in this way, such as a `ComboBox`, `TreeView`, `DataGrid`, `WrapPanel`, and so on. For more information on what operations can be performed using `DataGrid`, have a look at Chapter 4, *The Data Grid*.

See also

To learn how an `ObservableCollection` enables a UI to be automatically updated, have a look at the recipe *Enabling a Silverlight application to automatically update its UI*.

Enabling a Silverlight application to automatically update its UI

Applies to Silverlight 3, 4 and 5

In the previous recipes, we looked at how we can display data more easily using data binding for both single objects as well as collections. However, there is another feature that data binding offers us for free, **automatic synchronization** between the target and the source. This synchronization will make sure that when the value of the source property changes, this change will be reflected in the target object as well (being a control on the user interface).

This also works in the opposite direction—when we change the value of a bound control, this change will be pushed to the data object as well. Silverlight's data binding engine allows us to opt-in to this synchronization process. We can specify if we want it to work—and if so, in which directions—using the mode of data binding.

The synchronization works for both single objects bound to the UI as well as entire collections. For it to work, an interface needs to be implemented in either case. This synchronization process is what we'll be looking at in this recipe.

Getting ready

If you want to follow along with this recipe, you can either use the code from the previous recipes or use the provided solution in the `Chapter02/SilverlightBanking_Update_UI_Starter` folder in the code bundle that is available on the Packt website. The finished solution for this recipe can be found in the `Chapter02/SilverlightBanking_Update_UI_Completed` folder.

How to do it...

In this recipe, we'll look at how Silverlight does automatic synchronization, both for a single object and for a collection of objects. To demonstrate both types of synchronization, we'll use a timer that adds another activity on the account, every 10 seconds. A single instance of the `Owner` class is bound to the UI. However, the newly-added activities will cause the `CurrentBalance`, `LastActivity`, and `LastActivityAmount` properties of the `Owner` class to get updated. Also, these activities on the account will be reflected in the list of activities. Following are the steps to achieve automatic synchronization:

1. For the data binding engine to notice changes on the source object, the source needs to send a notification that the value of one of its properties has changed. By default, the `Owner` class does not do so. The original `Owner` class is shown by the following code:

```
public class Owner
{
    public int OwnerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public DateTime BirthDate { get; set; }
    public DateTime CustomerSince { get; set; }
    public string ImageName { get; set; }
```

```
    public DateTime LastActivityDate { get; set; }
    public double CurrentBalance { get; set; }
    public double LastActivityAmount { get; set; }
}
```

2. To make this class support notifications, an interface has to be implemented, namely the `IPropertyChanged` interface. This interface defines one event, that is, the `PropertyChanged` event. Whenever one of the properties change, this event should be raised. The changed `Owner` class is shown in the following code (Only two properties are shown as they are all similar; the rest can be found in the finished solution in the book sample code):

```
public class Owner : IPropertyChanged
{
    private double currentBalance;
    private string firstName;
    public event PropertyChangedEventHandler PropertyChanged;
    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            firstName = value;
            if(PropertyChanged != null)
                PropertyChanged(this, new
                    PropertyChangedEventArgs("FirstName"));
        }
    }
    public double CurrentBalance
    {
        get
        {
            return currentBalance;
        }
        set
        {
            currentBalance = value;
            if(PropertyChanged != null)
                PropertyChanged(this, new
                    PropertyChangedEventArgs("CurrentBalance"));
        }
    }
}
```

}

3. To simulate updates, we'll use a `DispatcherTimer` in the `MainPage`. With every tick of this timer, a new activity on the account is created. We'll count the new value of the `CurrentBalance` with every tick, and update the value of the `LastActivityDate` and `LastActivityAmount`, as shown in the following code:

```
private DispatcherTimer timer;
private int currentActivityId = 11;
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    OwnerDetailsGrid.DataContext = owner;
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 10);
    timer.Tick += new EventHandler(timer_Tick);
    timer.Start();
}
void timer_Tick(object sender, EventArgs e)
{
    currentActivityId++;
    double amount = 0 - new Random().Next(100);
    AccountActivity newActivity = new AccountActivity();
    newActivity.ActivityId = currentActivityId;
    newActivity.Amount = amount;
    newActivity.Beneficiary = "Money withdrawal";
    newActivity.ActivityDescription = "ATM In Some Dark Alley";
    newActivity.ActivityDate = new DateTime(2009, 9, 18);
    owner.CurrentBalance += amount;
    owner.LastActivityDate = DateTime.Now;
    owner.LastActivityAmount = amount;
}
```

4. In XAML, the `TextBlock` controls are bound as mentioned before. If no `Mode` is specified, `OneWay` is assumed. This causes updates of the source to be reflected in the target, as shown in the following code:

```
<TextBlock x:Name="CountryValueTextBlock"
           Grid.Row="8"
           Grid.Column="1"
           Margin="2"
           Text="{Binding Country}" >
</TextBlock>
```

```
<TextBlock x:Name="BirthDateValueTextBlock"
           Grid.Row="9"
           Grid.Column="1"
           Margin="2"
           Text="{Binding BirthDate}" >
</TextBlock>
<TextBlock x:Name="CustomerSinceValueTextBlock"
           Grid.Row="10"
           Grid.Column="1"
           Margin="2"
           Text="{Binding CustomerSince}" >
</TextBlock>
```

5. If we run the application now, after 10 seconds, we'll see the values changing. The new values can be seen in the following image:

Current balance:	1183.56
Last activity on:	8/2/2009 5:48:55 PM
Amount:	-30

6. In the previous recipe, *Binding collections to UI elements*, we saw how to bind a list of AccountActivity items to a ListBox. If we want the UI to update automatically when changes occur in the list (when a new item is added or an existing item is removed), then the list to which we bind should implement the INotifyCollectionChanged interface. Silverlight has a built-in list that implements this interface, namely the ObservableCollection<T>. If we were binding to a List<T>, then these automatic updates wouldn't work. Working with an ObservableCollection<T> is no different than working with a List<T>. In the following code, we're creating the ObservableCollection<AccountActivity> and adding items to it:

```
private ObservableCollection<AccountActivity>
    accountActivitiesCollection;
private void InitializeActivitiesCollection()
{
    accountActivitiesCollection = new
        ObservableCollection<AccountActivity>();
    AccountActivity accountActivity1 = new AccountActivity();
    accountActivity1.ActivityId = 1;
    accountActivity1.Amount = -33;
    accountActivity1.Beneficiary = "Smith Woodworking Shop London";
    accountActivity1.ActivityDescription = "Paid by credit card";
    accountActivity1.ActivityDate = new DateTime(2009, 9, 1);
    accountActivitiesCollection.Add(accountActivity1);
}
```

7. Update the `Tick` event, so that each new `Activity` is added to the collection:

```
void timer_Tick(object sender, EventArgs e)
{
    ...
    AccountActivity newActivity = new AccountActivity();
    ...
    accountActivitiesCollection.Add(newActivity);
}
```

8. To bind this collection to the `ListBox`, we use the `ItemsSource` property. The following code can be added to the constructor to create the collection and perform the binding:

```
InitializeActivitiesCollection();
AccountActivityListBox.ItemsSource = accountActivitiesCollection;
```

When we run the application now, we see that all added activities appear in the `ListBox` control. With every tick of the `Timer`, a new activity is added and the UI refreshes automatically.

How it works...

In some scenarios, we might want to view changes to the source object in the user interface immediately. Silverlight's data binding engine can automatically synchronize the source and target for us, both for single objects and for collections.

Single objects

If we want the target controls on the UI to update automatically a property value of an instance changes, then the class to which we are binding should implement the `INotifyPropertyChanged` interface. This interface defines just one event—`PropertyChanged`. It is defined in the `System.ComponentModel` namespace, using the following code:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

This event should be raised whenever the value of a property changes. The name of the property that has changed, is passed as the parameter for the instance of `PropertyChangedEventArgs`.

A binding in XAML is set to `OneWay` by default. `OneWay` allows updates to be passed on to the target. If we had set the binding to `Mode=OneTime`, then only the initial values would have been loaded. For more information on binding modes, refer to the recipe *Using the different modes of data binding to allow persisting data*.

Now, what exactly happens when we bind to a class that implements this interface? Whenever we do so, Silverlight's data binding engine will notice this and will automatically start to check if the `PropertyChanged` event is raised by an instance of the class. It will react to this event, thereby resulting in an update of the target.

Collections

Whenever a collection changes, we might want to get updates of this collection as well. In this example, we want to view the direct information of all the activities on the account. Normally, we would have placed these in a `List<T>`. However, `List<T>` does not raise an event when items are being added or deleted. An interface similar to `INotifyPropertyChanged` exists, so that a list/collection should implement for data binding to pick up those changes. This interface is known as `INotifyCollectionChanged`.

We didn't directly create a class that implements this interface. However, we used an `ObservableCollection<T>`. This collection already implemented this interface for us.

Whenever items are being added, deleted, or the collection gets refreshed, an event will be raised to which the data binding engine will bind itself. As for single objects, changes will be reflected in the UI immediately.

Cleaning up the code

In the code for the `Owner` class, we have inputted all the properties as shown in the following code:

```
public double CurrentBalance
{
    get
    {
        return currentBalance;
    }
    set
    {
        currentBalance = value;
        if(currentBalance != null)
            PropertyChanged(this, new
                PropertyChangedEventArgs("CurrentBalance"));
    }
}
```

In the previous code, we are raising the `PropertyChanged` event from each property. It's a good idea to move this to a separate method as shown in the following code:

```
public void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new
            PropertyChangedEventArgs(propertyName));
    }
}
public double CurrentBalance
{
    get
    {
        return currentBalance;
    }
    set
    {
        if (currentBalance != value)
        {
            currentBalance = value;
            OnPropertyChanged("CurrentBalance");
        }
    }
}
```

It may also be a good idea to move this method to a base class and have the entities inherit from this class, as shown in the following code:

```
public class BaseEntity : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new
                PropertyChangedEventArgs(propertyName));
        }
    }
}
public class Owner : BaseEntity
{
}
```

While automatic synchronization is a nice feature that comes along with data binding for free, it's not always needed. Sometimes it's not even wanted. Therefore, implement the interfaces that are described here only when the application needs them. It's an opt-in model.

Obtaining data from any UI element it is bound to

Applies to Silverlight 3, 4 and 5

When a user who is working with your application performs a certain action, it's often essential to know on what object this action will be executed. For example, if a user clicks on a *Delete* button on an item, it's essential that you know which item is clicked, so that you can write the correct code to delete that item. Also, when a user wants to edit an item in a list, it's necessary that you—the programmer—know which item in the list the user wants to edit.

In Silverlight, there is a very easy mechanism called `DataContext` that helps us with this task. In this recipe, we're going to use the `DataContext` to get the data when we need it.

Getting ready

If you want to follow along with this recipe, you can either use the code from the previous recipes or use the provided solution in the `Chapter02/SilverlightBanking_ Obtaining_Data_Starter` folder in the code bundle that is available on the Packt website. The completed solution for this recipe can be found in the `Chapter02/ SilverlightBanking_Obtaining_Data_Completed` folder.

How to do it...

We're going to create a **Details...** button for each item in the `ListBox` containing `AccountActivities`. This **Details...** button will open a new `ChildWindow` that will display details about the selected `AccountActivity`. To achieve this, carry out the following steps:

1. We will start by opening the solution we have created by following all the steps of the recipe *Binding data to collections*. We add a new item to the Silverlight project—a `ChildWindow` named `ActivityDetailView`, and add the following code to the XAML defining this new control:

```
<Grid x:Name="LayoutRoot" Margin="2">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid x:Name="OwnerDetailsGrid">
        <Grid.RowDefinitions>
            <RowDefinition Height="30" />
```



```
        Text="{Binding ActivityId}" >
    </TextBlock>
    <TextBlock x:Name="BeneficiaryTextBlockValue"
        Grid.Row="1"
        Grid.Column="1"
        Margin="2"
        Text="{Binding Beneficiary}" >
    </TextBlock>
    <TextBlock x:Name="AmountTextBlockValue"
        Grid.Row="2"
        Grid.Column="1"
        Margin="2"
        Text="{Binding Amount}" >
    </TextBlock>
    <TextBlock x:Name="ActivityDateTextBlockValue"
        Grid.Row="3"
        Grid.Column="1"
        Margin="2"
        Text="{Binding ActivityDate}" >
    </TextBlock>
    <TextBlock x:Name="DescriptionTextBlockValue"
        Grid.Row="4"
        Grid.Column="1"
        Margin="2"
        Text="{Binding ActivityDescription}"
        TextWrapping="Wrap">
    </TextBlock>
</Grid>
<Button x:Name="btnOK"
    Content="OK"
    Click="btnOK_Click"
    Width="75"
    Height="23"
    HorizontalAlignment="Right"
    Margin="0,12,0,0"
    Grid.Row="1" />
</Grid>
```

2. Next, we open `ActivityDetailView.xaml.cs` and add the following code:

```
public ActivityDetailView(AccountActivity activity)
{
    InitializeComponent();
    this.DataContext = activity;
}
```

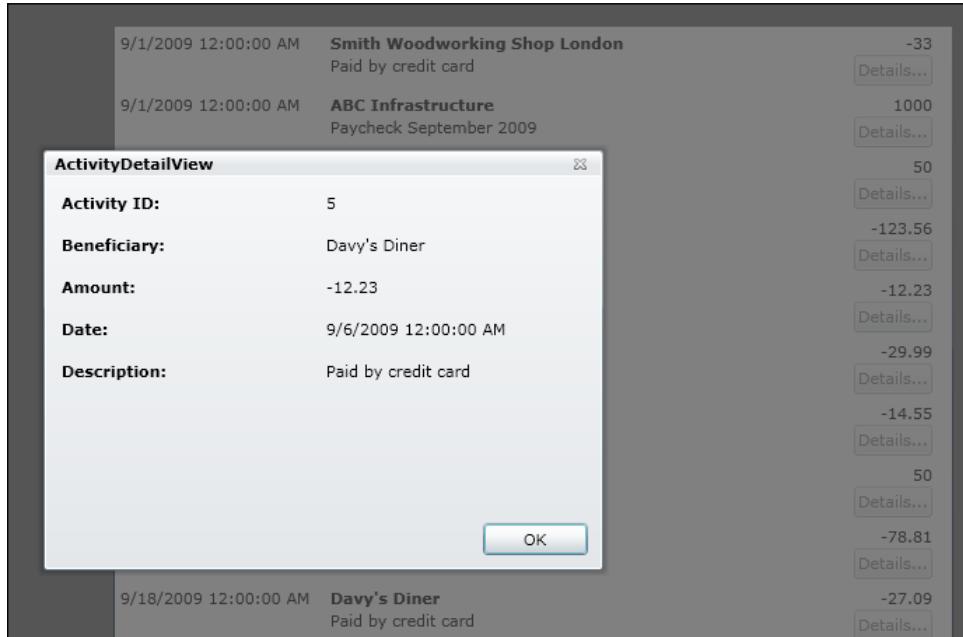
3. Now we open the `MainPage.xaml`, locate the `ListBox` named `AccountActivityListBox`, and add a button named `btnDetails` to the `DataTemplate` of that `ListBox`. This is shown in the following code:

```
<Button x:Name="btnDetails"
    Grid.Row="1"
    Grid.Column="2"
    HorizontalAlignment="Right"
    Content="Details..."
    Click="btnDetails_Click">
</Button>
```

4. Add the following C# code to `MainPage.xaml.cs` to handle the `Click` event of the button that we've added in the previous step:

```
private void btnDetails_Click(object sender, RoutedEventArgs e)
{
    ActivityDetailView activityDetailView = new ActivityDetailView
    ((AccountActivity)((Button)sender).DataContext);
    activityDetailView.Show();
}
```

5. We can now build and run the solution. When you click on the **Details...** button, you'll see the details of the selected `AccountActivity` in a `ChildWindow`. You can see the result in the following screenshot:



How it works...

Once the `DataContext` of a general control has been set (any CLR object can be used as `DataContext`), each child item of that control refers to the same `DataContext`.

For example, if we have a `UserControl` containing a `Grid` that has three columns, with a `TextBox` in the first two columns and a `Button` in the last column, and if the `DataContext` of the `UserControl` gets set to an object of the `Person` type, then the `Grid`, `TextBox`, and `Button` would have that same `Person` object as their `DataContext`. In general, when the `DataContext` of an item hasn't been set, Silverlight will find out if the parent of that item in the visual tree has its `DataContext` set to an object, and use that `DataContext` as the `DataContext` of the child item. Silverlight keeps on walking up to the visual tree right up to the uppermost level of the application.

If you use an `ItemsControl`, such as a `ListBox`, and give it a collection as an `ItemsSource`, then the `DataContext` of that `ListBox` is the collection you bound it to.

Following the same logic, the `DataContext` of one `ListBoxItem` is one item from the collection. In our example, one item is defined by a `DataTemplate` containing a `Grid`, various `TextBlock` instances, and a `Button`. Due to the fact that Silverlight keeps on trickling up to look for a valid `DataContext`, the `DataContext` of the `Grid`, all the `TextBlock` instances, and the `Button` are the same; they're one item from the `ItemsSource` collection of the `ListBox`.

With this in mind, we can now access the data that is bound to any UI element of our `ListBoxItem`. The data we need is the `DataContext` of the button that we're clicking.

The `Click` event of this button has a `sender` parameter—the `Button` itself. To access the `DataContext`, we cast the `sender` parameter to a `Button` object. As we know that the `ListBox` is bound to an `ObservableCollection` of `AccountActivity`, we can cast the `DataContext` to type `AccountActivity`. To show the details window, all we need to do now is pass this object to the constructor of the details `ChildWindow`.

See also

The `DataContext` is important when you're working with data binding, as it's the `DataContext` of an element that's looked at as the source of the binding properties. You can learn more about data binding and the various possibilities it offers by looking at almost any recipe in this chapter.

Using the different modes of data binding to allow persisting data

Applies to Silverlight 3, 4 and 5

Until now, the data has flowed from the source to the target (the UI controls). However, it can also flow in the opposite direction from the target towards the source. This way, not only can data binding help us in displaying data, but also in persisting data. The direction of the flow of data in a data binding scenario is controlled by the `Mode` property of the `Binding`. In this recipe, we'll look at an example that uses all the `Mode` options, and in one go we'll push the data that we enter ourselves to the source.

Getting ready

This recipe builds on the code that was created in the previous recipes, so if you're following along, you can keep using that codebase. You can also follow this recipe from the provided start solution. It can be found in the `Chapter02/SilverlightBanking_Binding_Modes_Starter` folder in the code bundle that is available on the Packt website. The `Chapter02/SilverlightBanking_Binding_Modes_Completed` folder contains the finished application of this recipe.

How to do it...

In this recipe, we'll build the `OwnerDetailsEdit` window of the `Owner` class. In this window, part of the data is editable, while some isn't. The editable data will be bound using a `TwoWay` binding, whereas the non-editable data is bound using a `OneTime` binding. The **Current balance** of the account is also shown—which uses the automatic synchronization—based on the `INotifyPropertyChanged` interface implementation. This is achieved using `OneWay` binding.

The following is an image of the details screen:



Let's go through the required steps to work with the different binding modes:

1. Add a new Silverlight child window called `OwnerDetailsEdit.xaml` to the Silverlight project.
2. In the code-behind of this window, change the default constructor so that it accepts an instance of the `Owner` class, as shown in the following code:

```
private Owner owner;
public OwnerDetailsEdit(Owner owner)
{
    InitializeComponent();
    this.owner = owner;
}
```

3. In the `MainPage.xaml`, add a `Click` event on the `OwnerDetailsEditButton` as follows:

```
<Button x:Name="OwnerDetailsEditButton"
        Click="OwnerDetailsEditButton_Click" >
```

4. In the event handler, add the following code, which will create a new instance of the `OwnerDetailsEdit` window, passing in the created `Owner` instance:

```
private void OwnerDetailsEditButton_Click(object sender,
    RoutedEventArgs e)
{
```

```

    OwnerDetailsEdit ownerDetailsEdit = new OwnerDetailsEdit(owner);
    ownerDetailsEdit.Show();
}

```

5. The XAML of the OwnerDetailsEdit is pretty simple. Take a look at the completed solution (Chapter02/ SilverlightBanking_Binding_Modes_Completed) for a complete listing. Don't forget to set the passed Owner instance as the DataContext for the OwnerDetailsGrid. This is shown in the following code:

```
OwnerDetailsGrid.DataContext = owner;
```

6. For the OneWay and TwoWay bindings to work, the object to which we are binding should be an instance of a class that implements the INotifyPropertyChanged interface. In our case, we are binding an Owner instance. This instance implements the interface correctly. The following code illustrates this:

```

public class Owner : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
}

```

7. Some of the data may not be updated on this screen and it will never change. For this type of binding, the Mode can be set to OneTime. This is the case for the OwnerId field. The users should neither be able to change their ID nor should the value of this field change in the background, thereby requiring an update in the UI. The following is the XAML code for this binding:

```

<TextBlock x:Name="OwnerIdValueTextBlock"
           Text="{Binding OwnerId, Mode=OneTime}" >
</TextBlock>

```

8. The CurrentBalance TextBlock at the bottom does not need to be edited by the user (allowing a user to change his or her account balance might not be beneficial for the bank), but it does need to change when the source changes. This is the automatic synchronization working for us and it is achieved by setting the Binding to Mode=OneWay. This is shown in the following code:

```

<TextBlock x:Name="CurrentBalanceValueTextBlock"
           Text="{Binding CurrentBalance, Mode=OneWay}" >
</TextBlock>

```

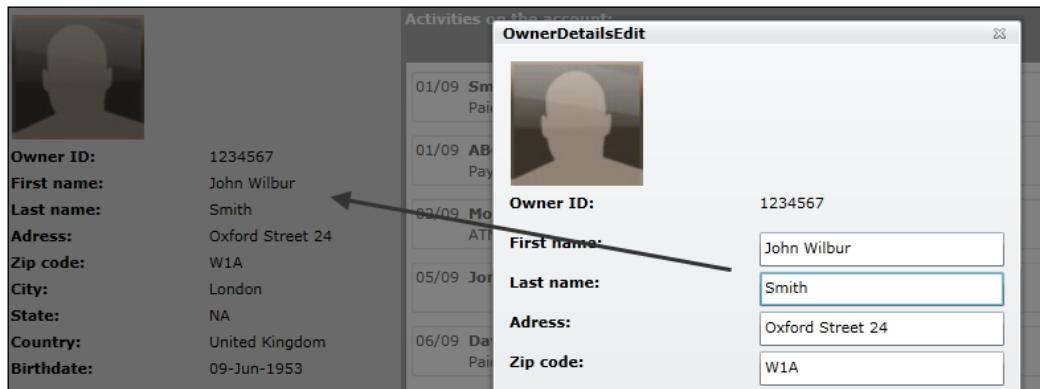
9. The final option for the Mode property is TwoWay. TwoWay bindings allow us to persist data by pushing data from the UI control to the source object. In this case, all other fields can be updated by the user. When we enter a new value, the bound Owner instance is changed. TwoWay bindings are illustrated using the following code:

```

<TextBox x:Name="FirstNameValueTextBlock"
         Text="{Binding FirstName, Mode=TwoWay}" >
</TextBox>

```

We've applied all the different binding modes at this point. Notice that when you change the values in the pop-up window, the details on the left of the screen are also updated. This is because all controls in the background are bound to the same source object as shown in the following image:



How it works...

When we looked at the basics of data binding, we saw that a binding always occurs between a source and a target. The first one is normally an in-memory object, but it can also be a UI control. The second one will always be a UI control.

Normally, data flows from source to target. However, using the `Mode` property, we have the option to control this.

A `OneTime` binding should be the default for data that does not change when displayed to the user. When using this mode, the data flows from source to target. The target receives the value initially during loading, and the data displayed in the target will never change. Quite logically, even if a `OneTime` binding is used for a `TextBox`, changes done to the data by the user will not flow back to the source. IDs are a good example of using `OneTime` bindings. Also, when building a catalogue application, `OneTime` bindings can be used, as we won't change the price of the items that are displayed to the user (or should we?).

We should use a `OneWay` binding for binding scenarios in which we want an up-to-date display of data. Data will flow from source to target here also, but every change in the values of the source properties will propagate to a change of the displayed values. Think of a stock market application where updates are happening every second. We need to push the updates to the UI of the application.

The `TwoWay` bindings can help in persisting data. The data can now flow from source to target and vice versa. Initially, the values of the source properties will be loaded in the properties of the controls. When we interact with these values (type in a textbox, drag a slider, and so on), these updates are pushed back to the source object. If needed, conversions can be done in both directions.

There is one important requirement for the `OneWay` and `TwoWay` bindings. If we want to display up-to-date values, then the `INotifyPropertyChanged` interface should be implemented. The `OneTime` and `OneWay` bindings would have the same effect, even if this interface is not implemented on the source. The `TwoWay` bindings would still send the updated values if the interface was not implemented; however, they wouldn't notify about the changed values. It can be considered as a good practice to implement the interface, unless there is no chance that the updates of the data would be displayed somewhere in the application. The overhead created by the implementation is minimal.

There's more...

Another option in a binding is the `UpdateSourceTrigger`. It allows us to specify when a `TwoWay` binding will push the data to the source. By default, this is determined by the control. For a `TextBox`, this is done on the `LostFocus` event; for most other controls, it's done on the `PropertyChanged` event.

The value can also be set to `Explicit`. This means that we can manually trigger the update of the source as follows:

```
BindingExpression expression = this.FirstNameValueTextBlock.  
    GetBindingExpression(TextBox.TextProperty);  
expression.UpdateSource();
```

See also

Changing the values that flow between source and target can be done using converters. We'll look at these in *Chapter 3, Advanced Data Binding*.

Debugging data binding expressions in Visual Studio

Applies to Silverlight 5 (after installation of Silverlight 5 tools, will also work on Silverlight 4 projects)

While data binding is one of, if not the most powerful features in Silverlight, data binding statements can be quite difficult to write. Since data binding is mostly written in XAML, we have limited **IntelliSense** support. A single typo will break the entire data binding statement, however we won't be notified about it at compile time. On top of that, most data binding errors aren't bubbling to the surface by default (the data binding engine is quite a forgiving piece of software), so finding errors is not straightforward. We're also limited in the options we have to fix errors, since we have no debugging support in Silverlight.

That has now changed: with Silverlight 5, it's possible to place breakpoints in XAML code that give us access to the values involved in the data binding as well as information about an error that may have occurred while Silverlight tried to apply the binding. In this recipe, we'll look at how these breakpoints can be used.

Getting ready

Any project with data binding statements can be used to test how the debugging process in Silverlight 5 works. For this recipe, you can use your finished solution from the previous recipe. Alternatively, you can use the solution located in the Chapter02/SilverlightBanking_Debugging folder.

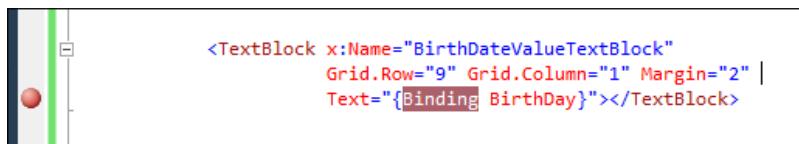
How to do it...

Since data binding statements in XAML have to be written manually without a lot of support from IntelliSense, it's a common place for errors. With Silverlight 5, placing breakpoints on data binding statements has become possible. Let's see how we can work with this new feature in the following steps:

1. With the solution open as outlined in the *Getting ready* section for this recipe, we are going to introduce an error for a change, so that the data binding fails. In `MainPage.xaml`, change the binding statement of the `BirthDate` property to `BirthDay`, as shown in the following code:

```
<TextBlock x:Name="BirthDateValueTextBlock"  
Text="{Binding BirthDay}"></TextBlock>
```

2. Like we are used to doing in C# code, place a breakpoint on the line in the XAML code of `MainPage.xaml`, containing the data binding expression, as shown in the following screenshot:



3. Run the application now and notice that Visual Studio will hit the breakpoint immediately.
4. With the breakpoint hit, open the **Locals** window in Visual Studio. If this window isn't open, you can find it under **Debug** | **Windows** | **Locals**.

5. In the **Locals** window, we get a lot of information about the binding. In this case, Silverlight explains what went wrong with the data binding. The following is the exact message copied from the **BindingState** in the **Locals** window:

```
BindingExpression path error: 'BirthDay' property not found
on 'SilverlightBanking.Owner' 'SilverlightBanking.Owner'
(HashCode=53517805). BindingExpression: Path='BirthDay'
DataItem='SilverlightBanking.Owner' (HashCode=53517805); target
element is 'System.Windows.Controls.TextBlock' (Name='BirthDateVal
ueTextBlock'); target property is 'Text' (type 'System.String')
```

The error message is clear in saying that the BirthDay property can't be found, which is indeed the cause of the error.

How it works

Apart from the error message itself, the **Locals** window has more interesting information that we can use to learn more about failing data binding expressions.

If we use a **BindingBase** property (which we cover in *Chapter 3, Advanced Data Binding*, in the recipe *Replacing converters with Silverlight 5 BindingBase properties*), we can see the values of these properties in the debugger window. The following screenshot shows that the **FallbackValue** for this binding was set to NA:

	BindingState	{Error: System.Exception: System.Wi
	Action	UpdatingTarget
	Binding	{System.Windows.Data.Binding}
	[System.Windows.Data.Binding]	{System.Windows.Data.Binding}
	base	{System.Windows.Data.Binding}
	FallbackValue	"NA"
	StringFormat	null
	TargetNullValue	null

An Introduction to Data Binding

Under the `FinalSource` property, we see the object that we are binding. In this case, an instance of the `Owner` class. This is shown in the following screenshot:

FinalSource	[SilverlightBanking.Owner]
Address	"Oxford Street 24"
address	"Oxford Street 24"
BirthDate	{9/06/1953 0:00:00}
birthDate	{9/06/1953 0:00:00}
city	"London"
City	"London"
Country	"United Kingdom"
country	"United Kingdom"
CurrentBalance	1234.56
currentBalance	1234.56
customerSince	{20/12/1999 0:00:00}
CustomerSince	{20/12/1999 0:00:00}
firstName	"John"
FirstName	"John"
ImageName	"man.jpg"
imageName	"man.jpg"
lastActivityAmount	100.0
LastActivityAmount	100.0
LastActivityDate	{29/07/2011 0:00:00}
lastActivityDate	{29/07/2011 0:00:00}

Finally, under the `System.Windows.Data.Debugging.UpdateTargetPipeline` node, some more values are available. As we'll see in the next chapter, we can hook into the data binding process using converters or properties, such as `FallbackValue`. This process is like a pipeline of operations that are acting on the original value. It's possible that something goes wrong along the way. Through this option in the **Locals** window, we can clearly see all the intermediate values. The following screenshot shows this node expanded:

[System.Windows.Data.Debugging.UpdateTargetPipeline]	[System.Windows.Data.Debugging.UpdateTargetPipeline]
base	[System.Windows.Data.Debugging.UpdateTargetPipeline]
InitialValue	null
AfterValueConversion	null
IsUsingTargetNullValue	null
AfterStringFormat	null
AfterFallbackValue	"NA"
AfterTypeConversion	null
ValidationErrors	null

See also

In this recipe, we touched on concepts that are covered in more detail in the next chapter. We talked about `BindingBase` properties, which are covered in the recipe *Replacing converters with Silverlight 5 BindingBase properties*. Converters are covered in the next chapter in the recipe *Hooking into the data binding process*.

Data binding from Expression Blend 5

Applies to Silverlight 5 (works similarly for Silverlight 4 together with Expression Blend 4)

While creating data bindings is probably a task mainly reserved for the developers in the team, Blend 5—the design tool for Silverlight applications—also has strong support for creating and using bindings.

In this recipe, we'll build a small, data-driven application that uses data binding. We won't manually create the data binding expressions; we'll use Blend 5 for this task.

How to do it...

For this recipe, we'll create a small application from scratch that allows us to edit the details of a bank account owner. In order to achieve this, carry out the following steps:

1. We'll need to open Blend 5 and go to **File | New Project....** In the **New Project** dialog box, select **Silverlight 4 Application + Website**. Name the project `SilverlightOwnerEdit` and click on the **OK** button. Blend will now create a Silverlight application and a hosting website.
2. We'll start by adding a new class called `Owner`. Right-click on the Silverlight project and select **Add New Item....** In the dialog box that appears, select the **Class** template and click on the **OK** button. The following is the code for the `Owner` class, and it can be edited inside Blend 5:

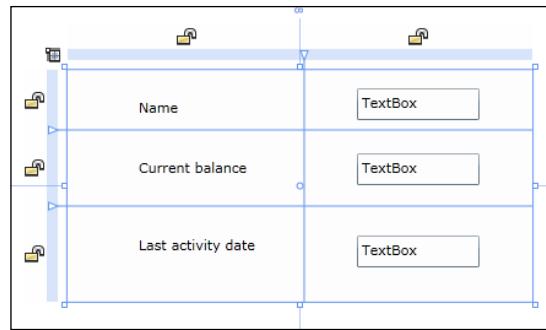
```
public class Owner
{
    public string Name {get; set;}
    public int CurrentBalance {get; set;}
    public DateTime LastActivityDate {get; set;}
}
```

3. In the code-behind of `MainPage.xaml`, create an instance of the `Owner` class and set it as the `DataContext` for the `LayoutRoot` of the page as follows:

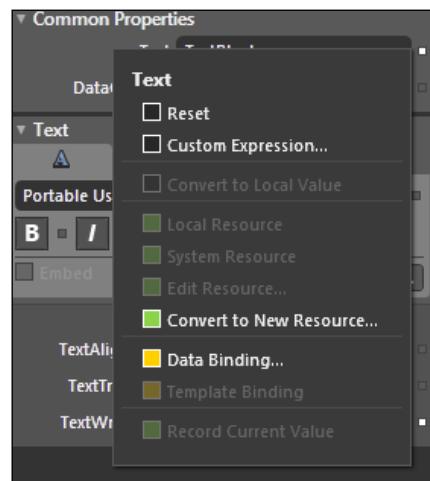
```
public partial class MainPage : UserControl
{
    public Owner owner;
    public MainPage()
    {
        // Required to initialize variables
        InitializeComponent();
        owner = new Owner()
        {
            Name="Gill Cleeren",
            CurrentBalance=300,
```

```
        LastActivityDate=DateTime.Now.Date  
    };  
    LayoutRoot.DataContext = owner;  
}  
}
```

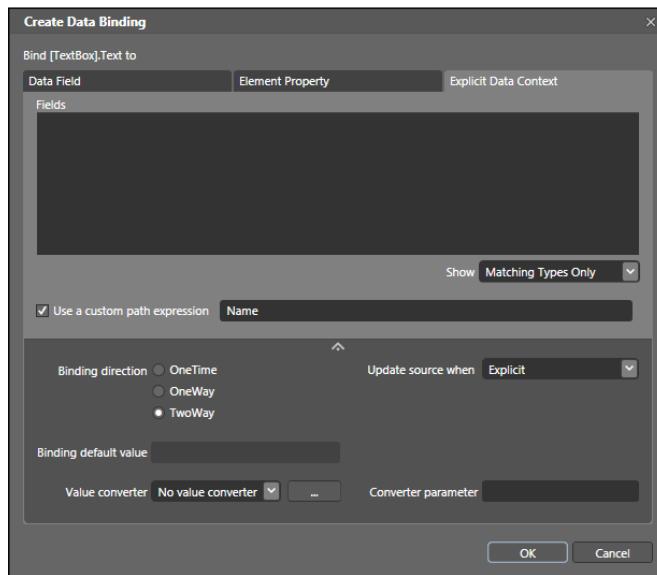
4. Build the solution so that the Owner class is known to Blend and it is able to use the class in its dialog boxes.
5. Now, in the designer, add a Grid containing three TextBlock instances and three TextBox controls, as shown in the following image:



6. We're now ready to add the data binding functionality. Select the first TextBox and in the **Properties** window, search for the **Text** property. Instead of typing a value, click on the small square for the **Advanced property** options next to the text field. Select **Data Binding...** in the menu. The following screenshot shows how to access this option:



7. In the dialog box that appears, we can now couple the `Name` property of the `Owner` type to the `Text` property of the `TextBox`. Under the **Explicit Data Context** tab, mark the **Use a custom path expression** checkbox and enter `Name` as the value. Click on the down arrow so that the advanced properties are expanded and mark `TwoWay` as the **Binding direction**. The other properties are similar, as shown in the following screenshot:



How it works...

Let's look at the resulting XAML code for a moment. Blend created the bindings for us automatically, taking into account the required options, such as `Mode=TwoWay`. This is shown in the following code:

```
<TextBox Grid.Column="1"
        Text="{Binding Name, Mode=TwoWay,
                  UpdateSourceTrigger=Default}"
        TextWrapping="Wrap"/>
<TextBox Grid.Column="1"
        Grid.Row="2"
        Text="{Binding LastActivityDate, Mode=TwoWay,
                  UpdateSourceTrigger=Default}"
        TextWrapping="Wrap"/>
<TextBox Grid.Column="1"
        Grid.Row="1"
        Text="{Binding CurrentBalance, Mode=TwoWay,
                  UpdateSourceTrigger=Default}"
        TextWrapping="Wrap"/>
```

When we have to create many bindings, it's often easier to do so through these dialog boxes than typing them manually in Visual Studio.

Using Expression Blend 5 for sample data generation

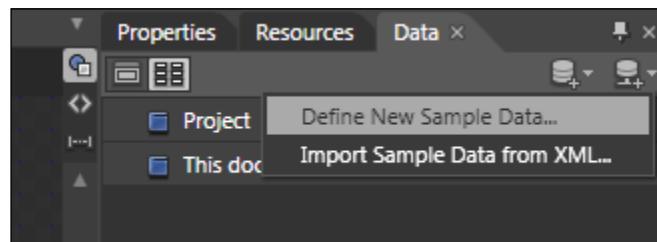
Applies to Silverlight 5 (works similarly for Silverlight 4 together with Expression Blend 4)

Expression Blend 5 contains a sample data generation feature, which can come in handy during the development of an application. It visualizes the data on which we are working, and provides us with an easier way to create an interface for a data-driven application. This feature was added to Blend in version 3.

How to do it...

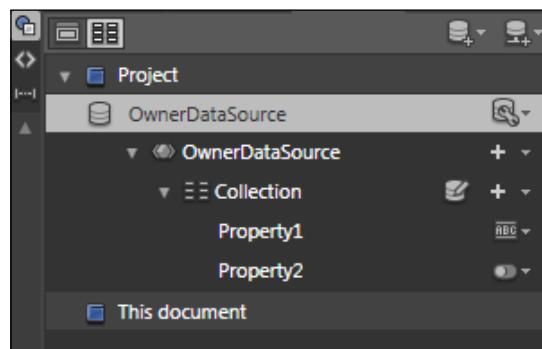
In this recipe, we'll build a small management screen for the usage of the bank employees. It will show an overview of the bank account owners. We wouldn't want to waste time with the creation of (sample) data, so we'll hand over this task to Blend. The following are the steps we need to follow for the creation of this data:

1. Open Blend 5 and go to **File | New Project....** In the dialog box that appears, select **Silverlight 5 Application + Website**. Name the project as **SilverlightBankingManagement** and click on the **OK** button. Blend will now create a Silverlight application and a hosting website.
2. With **MainPage.xaml** open in either the **Design View** or the **Split View**, go to the **Data** window. In this window, click on the **Add sample data source** icon and select **Define New Sample Data...**, as shown in the following screenshot:

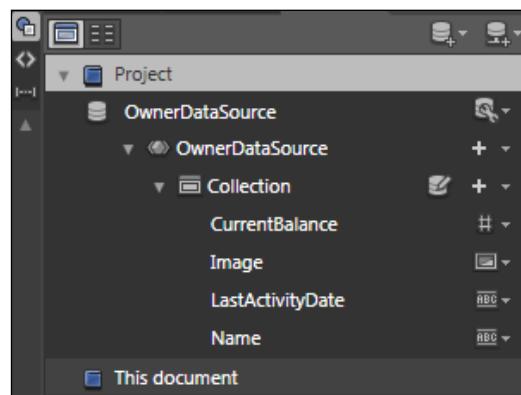


3. In the **Define New Sample Data** dialog box that appears, specify the Data source name as **OwnerDataSource**. We have the option to either embed this data source in the usercontrol (**This document**), or make it available for the entire project (**Project**). Select the latter option by selecting the **Project** radio button and clicking on the **OK** button.

- The last option in this window—**Enable sample data when application is running**—allows us to switch off the sample data while running the compiled application. If we leave the checkbox checked, then the sample data will be used for the design time as well as the runtime. We'll keep this option enabled. Blend will now generate the data source for us. The result is shown in the following screenshot:

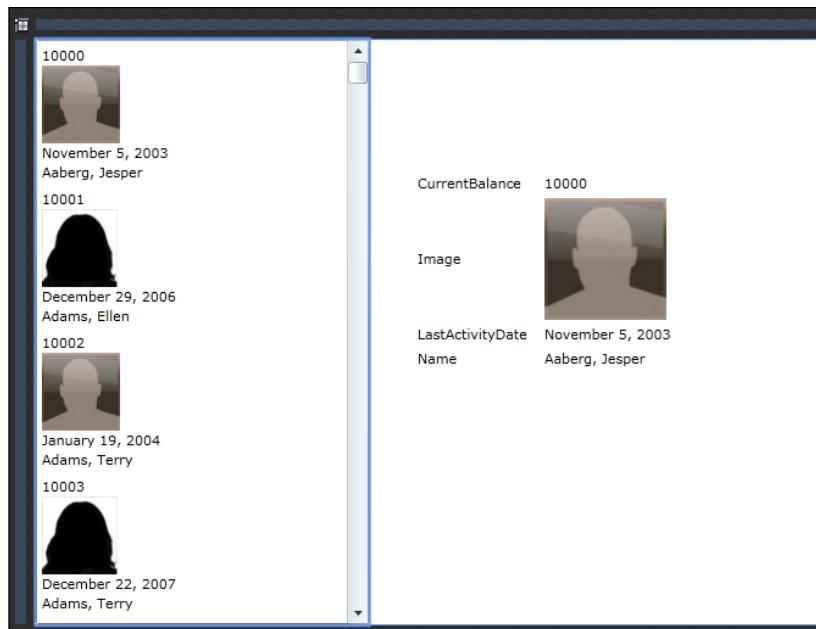


- By default, a **Collection** is created and it contains items with two properties. Each property has a type. Start by adding two more properties by clicking on the + sign next to the **Collection**, and select the **Add simple property** option. Rename **Property1** to **Name**. Now, change the type options by clicking on the **Change property type** icon, and selecting **Name** as the format. The other properties are similar and are shown in the following screenshot:



- For the **Image** type, we can select a folder that contains images. Blend will then copy these images to the **SampleData** subfolder inside the project.

7. We're now ready to use the sample data—for example—in a master-detail scenario. A **ListBox** will contain all the Owner data, from which we can select an instance. The details are shown in a **Grid**, using some **TextBlock** controls. Make sure that the **Data** window is set to **List Mode**, and drag the collection on to the design surface. This will trigger the creation of a **ListBox** in which the items are formatted, so we can see the details.
8. Now to view the details, we have to set the **Data** window to **Details Mode**. Then, instead of dragging the collection, we select the properties that we want to see in the detail view and drag those onto the design surface. The result should be similar to the following image:



Thus, Blend created all the data binding code in XAML as well as the sample data. For each different type, it generated different values.

3

Advanced Data Binding

In this chapter, we will cover:

- ▶ Hooking into the data binding process using converters
- ▶ Replacing converters with Silverlight 5 `BindingBase` properties
- ▶ Validating data bound input
- ▶ Validating data input using attributes
- ▶ Validating using `IDataErrorInfo` and `INotifyDataErrorInfo`
- ▶ Using templates to customize the way data is shown by controls
- ▶ Using implicit data templates
- ▶ Using the Ancestor RelativeSource binding
- ▶ Creating custom markup extensions
- ▶ Building a change-aware collection type
- ▶ Combining converters, data binding, and `DataContext` into a custom `DataTemplate`

Introduction

In the previous chapter, we explained in detail the concepts behind data binding and its implementation in Silverlight 5. Reflecting on what we learned in the previous chapter, we may start to think that data binding works as some kind of black box into which we insert the data and it displays it. However, this is not the case. The data binding engine gives us many points where we can extend or change this process.

The most obvious hooks we have in data binding are **Converters**. Converters allow us to grab a value when it's coming in from a source object, perform some action on it, and then pass it to the target control. The most obvious action that we can take is formatting, though many more are possible. We'll look at converters and their possibilities in this chapter.

With Silverlight 5, extra features were introduced in the platform in the data binding area which were already available in WPF but weren't in Silverlight. `Ancestor RelativeSource` and custom markup extensions are now available in Silverlight.

Data binding also allows us to perform **validations**. When entering data in data-bound controls such as a `TextBox`, it's important that we validate the data before it's sent back to the source. Silverlight has quite a few options to perform this validation. We'll look at these in this chapter as well.

We can also change the way our data is being displayed using **data templates**. Data templates allow us to override the default behavior of controls such as a `ListBox`. We will build some templates in this chapter to complete the look of the Silverlight Banking application. Silverlight 5 also introduces the notion of implicit data templates, which is also inherited from WPF. These allow us to define a template for a type which will be used by Silverlight to visualize an instance of that type. We'll use this in the Banking application as well to display particular types of data.

This chapter continues to use the same sample application—Silverlight Banking—which we have already used in the previous chapter. If you want to run the complete application, take a look at the code within the `Chapter02/SilverlightBanking` folder in the code bundle that is available on the Packt website.

Hooking into the data binding process

Applies to Silverlight 3, 4 and 5

We may want to perform some additional formatting for some types of data that we want to display using data binding. Think of a date. Normally, a date is stored in the database as a combination of a date and time. However, we may only want to display the date part—perhaps formatted according to a particular culture. Another example is a currency; the value is normally stored in the database as a double. In an application, we may want to format it by putting a dollar or a euro sign in front of it.

Silverlight's data binding engine offers us a hook in the data binding process, thereby allowing us to format, change, or do whatever we want to do with the data in both directions. This is achieved through the use of a converter.

Getting ready

This recipe builds on the code that was created in the recipes of the previous chapter. If you want to follow along, you can keep using your own code or use the provided starter solution that is located in the `Chapter03/SilverlightBanking_Converters_Starter` folder. The `Chapter03/SilverlightBanking_Converters_Completed` folder contains the complete solution for this recipe.

How to do it...

In this recipe, we'll build two converters. We'll start with a currency converter. This is quite basic. It will take a value and format it as a currency using the currency symbol based on the current culture. The second converter will be more advanced; it will convert from a numeric value to a color.

In the sample code of the book, some more converters have been added.

Carry out the following steps in order to get converters to work in a Silverlight application:

1. We'll start by creating the currency converter. A converter is nothing more than a class, in this sample called `CurrencyConverter`, which implements the `IValueConverter` interface. This interface defines two methods, `Convert` and `ConvertBack`. Place the `CurrencyConverter` class in a folder called `Converters` within the Silverlight project. The following is the code for this class:

```
public class CurrencyConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object
        parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
    public object ConvertBack(object value, Type targetType, object
        parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

2. The code in the Convert method will be applied to the data when it flows from the source to the target. Similarly, the ConvertBack method is called when the data flows from the target to the source, so when a TwoWay binding is active. The original value is passed in the value parameter. We have access to the current culture via the culture parameter. Also, we add a "minus" sign to the string value that is returned if the value is less than zero. This is shown in the following code:

```
public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
{
    double amount = double.Parse(value.ToString());
    if (amount < 0)
        return "- " + amount.ToString("c", culture);
    else
        return amount.ToString("c", culture);
}
```

3. Simply creating the converter doesn't do anything. An instance of the converter has to be created and passed along with the binding using the Converter property. This is to be done in the resources collection of the XAML file in which we will be using the converter or in App.xaml. The following code shows this instantiation in App.xaml. Note that we also need to add the namespace mapping. The latter is required to make the namespace in which the converter is declared, known from XAML code.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SilverlightBanking.App"
    xmlns:converters="clr-namespace:SilverlightBanking.Converters">
    <Application.Resources>
        <converters:CurrencyConverter x:Key="localCurrencyConverter">
        </converters:CurrencyConverter>
    </Application.Resources>
</Application>
```

4. After that, we specify this converter as the value for the Converter property in the Binding declaration. This is shown in the following code:

```
<TextBlock Text="{Binding CurrentBalance,
    Converter={StaticResource localCurrencyConverter}}"
    FontSize="12"
    FontWeight="Bold">
</TextBlock>
```

5. While this simple converter converts a double into a string, more advanced conversions can be performed. What if, for example, we want to display negative amounts in red and positive amounts in green? The Convert method looks quite similar, except that it now returns a SolidColorBrush. This is shown in the following code:

```

public object Convert(object value, Type targetType, object
    parameter, System.Globalization.CultureInfo culture)
{
    double amount = (double)value;
    if (amount >= 0)
        return new SolidColorBrush(Colors.Green);
    else
        return new SolidColorBrush(Colors.Red);
}

```

6. This type of converter can be applied in a Binding expression on a property that expects a SolidColorBrush, for example, the Foreground. This is shown in the following code:

```

<TextBlock Text="{Binding CurrentBalance,
    Converter={StaticResource localCurrencyConverter}}"
    Foreground="{Binding CurrentBalance,
    Converter={StaticResource
        localAmountToColorConverter}}">
</TextBlock>

```

The result of the conversion can be seen in the following image. The balance is positive, so the value is colored green.

Current balance:	1113.56
Last activity on:	8/6/2009 8:59:31 PM
Amount:	-58

How it works...

A **Converter** is a handy way of allowing us to get a hook in the data binding process. It allows us to change a value to another format or even another type (for example, a double value into a SolidColorBrush).

A converter is nothing more than a class that implements an interface called **IValueConverter**. This interface defines two methods: **Convert** and **ConvertBack**. When a binding specifies a converter, the **Convert** method is called automatically when the data flows from the source to the target. The same holds true for the **ConvertBack** method: this method is applied when the binding is happening, with data flowing from the target to the source. Thus the latter happens when the **Mode** of the binding is set to **TwoWay** and can be used to convert a value back into a format that is understood by the data store.

The ConverterParameter

The Convert as well as the ConvertBack methods of the `IValueConverter` interface also define an extra parameter that can be used to pass extra information into the converter to influence the conversion process. Take for example a `DateConverter`, which would require an extra parameter that defines the formatting of the date to be passed in. The following code shows the Convert method of such a converter:

```
public object Convert(object value, Type targetType, object
    parameter, System.Globalization.CultureInfo culture)
{
    DateTime dt = (DateTime)value;
    return dt.ToString(parameter.ToString(), culture);
}
```

The `ConverterParameter` is used in the Binding expression to pass the value to the parameter. This is shown in the following code:

```
<TextBlock x:Name="CustomerSinceValueTextBlock"
    Text="{Binding CustomerSince,
        Converter={StaticResource localDateConverter},
        ConverterParameter='dd-MMM-yyyy'}" >
</TextBlock>
```

Here, we are specifying to the converter that a date should be formatted as dd-MMM-yyyy.

Displaying images based on a URL with converters

Another nice way of using a converter is shown in the following code. Let's assume that in the database, we store the name of an image of the user. Of course, we want to display the image, not the name of the image. The `Source` property of an `Image` control is of type `ImageSource`. The class best suited for this is the `BitmapImage`. The converter that we need for this type of conversion is shown in the following code:

```
public class ImageConverter:IValueConverter
{
    private string baseUri = "http://localhost:1234/CustomerImages/";
    public object Convert(object value, Type targetType, object
        parameter, System.Globalization.CultureInfo culture)
    {
        if (value != null)
        {
            Uri imageUri = new Uri(baseUri + value);
            return new BitmapImage(imageUri);
        }
        else
```

```
    return "";
}
...
}
```

Using the converter in the XAML binding code is similar.

Replacing converters with Silverlight 5 BindingBase properties

Applies to Silverlight 4 and 5

In the previous recipe, we saw that using converters in data binding expressions can help us with a variety of things we want to do with the value that's being bound. It helps us in formatting the value as well as switching between colors. However, creating the converter can be a bit cumbersome for some tasks. To use it, we have to create the class that implements the `IValueConverter` interface, instantiate it, and change the binding expression. With Silverlight 4, some extra properties were added on the `BindingBase` class that can relieve us from writing a converter in some occasions. In Silverlight 5, no changes were made.

In this recipe, we'll look at how these three new properties, namely `TargetNullValue`, `StringFormat`, and `FallbackValue`, can be used instead of writing a converter.

Getting ready

This recipe builds on the code that was created in the previous recipe. If you want to follow along with this recipe, you can continue using your own code. Alternatively, you can use the start solution that can be found in the `Chapter03/SilverlightBanking_BindingBase_Properties_Starter` folder. The complete solution for this recipe can be found in the `Chapter03/SilverlightBanking_BindingBase_Properties_Completed` folder.

How to do it...

The `BindingBase` properties that are at our disposal since Silverlight 4 allow us to skip writing a converter during specific scenarios. We wrote quite a few in the previous example, some of which can be replaced by applying one or more of the new properties on the data binding expression. Let's take a look at how we can use these properties.

1. Let's first take a look at the `TargetNullValue` property. The value that we specify for `TargetNullValue` will be applied in the data binding expression if the value of the property is null. For the purpose of this example, let's say that a customer can also leave the bank. This `DateTime` value can be stored in the `NoMoreCustomerSince` property, which is a part of the `Owner` class. Add the following field and accompanying property to the `Owner` class:

```
private DateTime? noMoreCustomerSince;
public DateTime? NoMoreCustomerSince
{
    get
    {
        return noMoreCustomerSince;
    }
    set
    {
        if (noMoreCustomerSince != value)
        {
            noMoreCustomerSince = value;
            OnPropertyChanged("NoMoreCustomerSince");
        }
    }
}
```

2. For active customers, this value will be null. If we do not change anything in the initialization of the `Owner` instance in the `MainPage.xaml.cs`, then the value will be equal to null—that is, its default value. To display a value in the UI in any manner, we can use `TargetNullValue` and set it to `NA` (Not Available) using the following data binding expression:

```
<TextBlock x:Name="NoMoreCustomerSinceValueTextBlock"
    Text="{Binding NoMoreCustomerSince,
    TargetNullValue='NA'}" >
</TextBlock>
```

3. Very often, converters need to be written to format a value (as we did in the previous recipe). Formatting a currency or formatting a date is a task that we often encounter in business applications. Some of these can be replaced with another property of the `BindingBase`, that is, the `StringFormat` property. Instead of writing a converter to format all the dates, we use this property as shown in the following code. (We're showing `CustomerSince` here, but all others are similar.)

```
<TextBlock x:Name="CustomerSinceValueTextBlock"
    Text="{Binding CustomerSince, StringFormat='MM-dd-yyyy'}" >
</TextBlock>
```

4. `StringFormat` can also be used for currency formatting. The `LastActivityAmount` is formatted using this property as shown in the following code:

```
<TextBlock
    x:Name="LastActivityAmountValueTextBlock"
    Text="{Binding LastActivityAmount, StringFormat=C}" >
</TextBlock>
```

5. If we're binding to a property that does not exist, then the data binding engine will swallow the error and not display anything. This can be annoying in some situations. In such situations, the `FallbackValue` property can help. For example, assume that we have a class called `PreferredOwner` that inherits from `Owner` as shown in the following code:

```
public class PreferredOwner: Owner
{
    private DateTime preferredSince { get; set; }
    public DateTime PreferredSince
    {
        get
        {
            return preferredSince;
        }
        set
        {
            if (preferredSince != value)
            {
                preferredSince = value;
                OnPropertyChanged("PreferredSince");
            }
        }
    }
}
```

6. A situation may arise when an interface would bind to either an instance of `Owner` or `PreferredOwner`. The `PreferredSince` property is available only on `PreferredOwner`. If we are binding an `Owner` instance, no value would be displayed for this property. The `FallbackValue` can be used in this case to indicate that if the property is not found, a fallback value should be used. This can be seen in the following code:

```
<TextBlock x:Name="PreferredSinceValueTextBlock"
           Text="{Binding PreferredSince,
                         StringFormat='MM-dd-yyyy', FallbackValue='NA'}" >
</TextBlock>
```

With these three new properties in action, the UI looks like the following screenshot when an Owner instance is bound:



How it works...

Converters are a way of hooking into the data binding process. They allow operations to be executed on the data before it is displayed. While converters can be used for all kinds of operations, they require quite some code to be written.

Starting with Silverlight 4, the `BindingBase` class—the abstract base class for the `Binding` class—has been extended with some properties that can do some particular tasks for which we would have needed to write a converter.

The `TargetNullValue` property allows us to react to the value of the source property being null. If the value for the property is null, then the value specified for the `TargetNullValue` will be displayed.

`StringFormat` makes it possible to perform the formatting of the value of the source property. Formatting parameters such as percentage, currency and dates can be formatted without the need of writing a converter.

Finally, the `FallbackValue` allows us to display a value when the data binding fails. Assume that we are binding to a property that is not defined on the type. Data binding will fail, but it will not cause an exception. No value will be displayed, but the application will keep running. If we specify the `FallbackValue`, this value will be displayed.

See also

In the previous recipe, we looked at writing converters.

Validating data-bound input

Applies to Silverlight 3, 4 and 5

Validation of your data is a requirement for almost every application. By using validation, you make sure that no invalid data is (eventually) persisted in your data store. When you don't implement validation, there is a risk that a user will input wrongly-formatted or plain incorrect data on the screen and even persist this data in your data store. This is something you should definitely avoid.

In this recipe, we'll learn about implementing client-side validation on the bound fields in the UI.

Getting ready

To get ready for this recipe, you can either use the code from one of the previous recipes or use the provided starter solution in the Chapter03/SilverlightBanking_Validation_Starter folder in the code bundle available on the Packt website. The complete solution for this recipe can be found in the Chapter03/SilverlightBanking_Validation_Completed folder.

How to do it...

We're going to add validation logic to the **OwnerDetailsEdit** screen you created by following all the steps of the *Using the different modes of data binding to allow persisting data* recipe in the previous chapter (Alternatively, you can use the starter solution.). To achieve this, we'll carry out the following steps:

1. Open the solution that you created in the *Using the different modes of data binding to allow persisting data* recipe (or the starter solution) and locate the `OwnerDetailsEdit.xaml` file. In this XAML file, locate and change the `LastJsonValueTextBlock` and the `BirthDateValueTextBlock` by adding `NotifyOnValidationError=true` and `ValidatesOnExceptions=true` to the `Binding` syntax. This is shown in the following code:

```
<TextBox x:Name="LastNameValueTextBlock"
        Grid.Row="3"
        Grid.Column="1"
        Margin="2"
        Text="{Binding LastName, Mode=TwoWay,
            NotifyOnValidationError=true,
            ValidatesOnExceptions=true}" >
</TextBox>
```

```
<TextBox x:Name="BirthDateValueTextBlock"
    Grid.Row="9"
    Grid.Column="1"
    Margin="2"
    Text="{Binding BirthDate, Mode=TwoWay,
        NotifyOnValidationError=true,
        ValidatesOnExceptions=true}" >
</TextBox>
```

2. Add a handler to the surrounding `Grid`, that is, `OwnerDetailsGrid`. This is shown in the following code:

```
<Grid x:Name="OwnerDetailsGrid"
    BindingValidationError="OwnerDetailsGrid_
    BindingValidationError">
```

3. Add the following C# code to `OwnerDetailsEdit.xaml.cs`. This implements the handler we defined in the previous step.

```
private void OwnerDetailsGrid_BindingValidationError(object
    sender, ValidationErrorEventArgs e)
{
    if (e.Action == ValidationErrorEventAction.Added)
        OwnerDetailsGrid.Background = new
            SolidColorBrush(Color.FromArgb(25, 255, 0, 0));
    if (e.Action == ValidationErrorEventAction.Removed)
        OwnerDetailsGrid.Background = new
            SolidColorBrush(Color.FromArgb(0, 0, 0, 0));
}
```

4. Locate the `Owner.cs` file, which represents the type of `DataContext` of the **OwnerDetailsEdit** control. Add the following code to the `set` accessor of `LastName` to make sure that a validation error is thrown when needed.

```
set
{
    if (lastName != value)
    {
        if (value.Length > 20)
        {
            throw new Exception("Length must be <= 20");
        }
        else
        {
            lastName = value;
            OnPropertyChanged("LastName");
        }
    }
}
```

- We can now build and run the solution. When invalid data is inputted (a string that's too long for the **Last name** field or a value that isn't in a correct format for the **Birthdate** field), a validation error will occur.

The result can be observed in the following screenshot:

First name:	John
Last name:	Smith thisstringwillbe too long
Address:	Oxford Street 24
Zip code:	W1A
City:	London
State:	NA
Country:	United Kingdom
Birthdate:	invalid date
Customer since:	12/20/1999 12:00:00 AM
Current balance:	1087.56

How it works...

Silverlight automatically reports a validation error in a few cases. These include when type conversion fails on binding, when an exception is thrown in a property's set accessor, or when a value doesn't correspond to the applied validation attribute (more on this can be found in the next recipe, *Validating data input using attributes*).

In our example, we're throwing an exception in the property's set accessor. This means Silverlight will report the error. Next to that, Silverlight will also report an error when you try to input a value that doesn't correspond with the underlying type (you can try to input an invalid date value in the **Birthdate** field).

If you look at the Binding syntax in XAML, you'll see that we've added a few things, `ValidatesOnExceptions` and `NotifyOnValidationError` are set to true.

Setting `ValidatesOnExceptions` to true makes sure that Silverlight will provide visual feedback for the validation errors it reports. Setting `NotifyOnValidationError` to true makes sure that the binding engine raises the `BindingValidationError` event when a validation error occurs.

In the parent grid, this `BindingValidationError` event gets handled. We've written code that will change the background color of the complete box if an error occurs (this is optional).

Client-side validation is easily implemented by bringing these three principles together in the example you've just created.

There's more...

Along with reporting a validation error when type conversion fails on binding or when an exception is thrown in a property's set accessor, Silverlight also reports an error when a value doesn't correspond to the applied validation attribute. More on this can be found in the next recipe.

As you've noticed while running the solution we've created, Silverlight has a default style for showing the validation error. This can, of course, be customized by changing the control's default ControlTemplate. More information on customizing templates can be found in the [Using templates to customize the way data is shown by controls](#) recipe.

And last but not least, we can provide more detailed validation reporting by using the ValidationSummary control. This ValidationSummary control will automatically receive the BindingValidationError events of its parent container. On each BindingValidationError, the ValidationSummary receives a newly-created ValidationSummaryItem (added to ValidationSummary.Errors) with corresponding Message, MessageHeader, ItemType, and Context properties. Next to that, a new ValidationSummaryItemSource is created (and added to ValidationSummaryItem.Sources) with corresponding Control and PropertyName properties.

To use a ValidationSummary in the example created in this recipe, we have to add a reference to System.Windows.Controls.Data.Input in the Silverlight project and add the following code to the OwnerDetailsEdit control:

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

This will make sure that we can use ValidationSummary. Next, we'll have to locate the **OK** button and add a ValidationSummary control. This is shown in the following code:

```
<datainput:ValidationSummary Grid.Row="1"
                               Margin="2,5,2,5">
</datainput:ValidationSummary>
<Button x:Name="OKButton"
        Content="OK"
        Click="OKButton_Click"
        Width="75"
        Height="23"
        HorizontalAlignment="Right"
        Margin="0,12,0,0"
        Grid.Row="2" />
```

When we run our solution and input invalid data, a validation summary will be shown. This can be seen in the following screenshot:

The screenshot shows a form with the following fields and their values:

First name:	John
Last name:	Smith thisstringwillbe too long
Address:	Oxford Street 24
Zip code:	W1A
City:	London
State:	NA
Country:	United Kingdom
Birthdate:	ml
Customer since:	12/20/1999 12:00:00 AM
Current balance:	1087.56

A red bar at the bottom indicates **2 Errors**:

- LastName** Length must be <= 20
- BirthDate** The string was not recognized as a valid DateTime. Th

See also

If you want to learn more about validation, you might want to take a look at the next two recipes, *Validating data input using attributes* and *Validating using IDataErrorInfo and INotifyDataErrorInfo*. To learn more about two-way data binding, have a look at the *Using the different modes of data binding to allow persisting data* recipe in Chapter 2, *An Introduction to Data Binding*.

Validating data input using attributes

Applies to Silverlight 4 and 5

Validation of your data is a requirement for almost every application. By using validation, you make sure that no invalid data is (eventually) persisted in your data store. When you don't implement validation, there's a risk that a user will input wrongly-formatted or plain incorrect data on the screen and even persist this data in your data store. This is something you should definitely avoid.

In this recipe, we'll learn about implementing client-side validation on the bound fields in the UI using attributes (**Data Annotations**).

Getting ready

To get ready for this recipe, you can either use the code from the previous recipe or use the provided starter solution in the Chapter03/SilverlightBanking_Validation_Attributes_Starter folder in the code bundle available on the Packt website. The complete solution for this recipe can be found in the Chapter03/SilverlightBanking_Validation_Attributes_Completed folder.

How to do it...

In this recipe, we're going to replace the validation on `Lastname` by using attributes or, to be more specific, by using data annotations. To achieve this, we'll carry out the following steps:

1. We have to add a reference to `System.ComponentModel.DataAnnotations` in our Silverlight project.

2. Locate `Owner.cs` and add the following `using` statement:

```
using System.ComponentModel.DataAnnotations;
```

3. Next, we should locate the `Lastname` property and change it by adding a data annotation attribute to limit the maximum length. Add the following code to actually validate this property:

```
[StringLength(20, ErrorMessage="Length must be <= 20")]
public string LastName
{
    get
    {
        return lastName;
    }
    set
    {
        if (lastName != value)
        {
            Validator.ValidateProperty(value,
                new ValidationContext(this, null, null)
                { MemberName ="LastName" });
            lastName = value;
            OnPropertyChanged("LastName");
        }
    }
}
```

4. We can now build and run the solution. When a string having more than 20 characters in length is inputted in the **Last name** field, the correct error message will be shown. This can be seen in the following screenshot:

First name:	John
Last name:	Smith thisvalueis toolong
Address:	Oxford Street 24

How it works...

Silverlight automatically reports a validation error in a few cases such as when type conversion fails on binding, when an exception is thrown in a property's set accessor, or when a value doesn't correspond to the applied validation attribute. To learn about the first two cases, have a look at the previous recipe, *Validating data-bound input*.

In our example, we've added a `StringLength` attribute to the `Lastname` property, hereby passing in the length and the error message that should be shown. Next to that, we've added a `ValidateProperty` call. This will make sure that the property is validated. When you don't add this, no data validation using attributes occurs.

If you look at the Binding syntax in XAML, you'll see that we've added a few things, `ValidatesOnExceptions` and `NotifyOnValidationError` are set to true.

Setting `ValidatesOnExceptions` to true makes sure that Silverlight will provide visual feedback for the validation errors it reports. Setting `NotifyOnValidationError` to true makes sure that the binding engine raises the `BindingValidationError` event when a validation error occurs.

Client-side validation is easily implemented by bringing these principles together in the example we've just created.

There's more...

In this example, we've only used one data annotation attribute for validation—`StringLength`—to explain the principle. However, there are some more attributes you can use such as `CustomValidation`, `DataType`, `EnumDataType`, `Range`, `RegularExpression`, and `Required`.

Next to that, you'll notice we've inputted only one `NamedParameter` value, `ErrorMessage`. Most validation attributes accept more `NamedParameter` values that can be used to customize the way validation is handled such as `ErrorMessageResourceName`, `ErrorMessageResourceType`, and so on depending on the validation attribute you're using.

Other uses of data annotations

There are various other data annotations that can be used for validation, such as displaying attributes and modeling attributes. These are used to control how certain information should be displayed or how certain properties should relate to each other. Data annotations are heavily used by RIA Services and the DataForm control. You can learn more about this by looking at the corresponding chapters in this book.

See also

If you want to know more about validation, you might want to take a look at the previous recipe, *Validating data bound input* or the next recipe, *Validating using IDataErrorInfo and INotifyDataErrorInfo*. To learn more about two-way data binding, have a look at the *Using the different modes of data binding to allow persisting data* recipe in Chapter 2.

Validating using IDataErrorInfo and INotifyDataErrorInfo

Applies to Silverlight 4 and 5

Validation of your data is a requirement for almost every application. By using validation, you make sure that no invalid data is (eventually) persisted in your datastore. When you don't implement validation, there's a risk that a user will input wrongly formatted or plain incorrect data on the screen and even persist this data in your datastore. This is something you should definitely avoid.

Starting with Silverlight 4, validating your data is possible by using `IDataErrorInfo` or `INotifyDataErrorInfo`. This allows us to invalidate the properties without throwing exceptions and the validation code doesn't have to reside in the set accessor of the property. It can be called whenever it's needed.

In this recipe, we'll learn about implementing validation on the bound fields in the UI using `IDataErrorInfo` and `INotifyDataErrorInfo`.

Getting ready

To get ready for this recipe, you can either use the code from one of the previous recipes such as the *Using the different modes of data binding to allow persisting data* recipe in Chapter 2 or use the provided starter solution in the `Chapter03/SilverlightBanking_Validation_DataError_Starter` folder in the code bundle available on the Packt website. The complete solution for this recipe can be found in the `Chapter03/SilverlightBanking_Validation_DataError_Completed` folder.

How to do it...

We're going to add validation logic to the **OwnerDetailsEdit** screen, just as we did in the previous validation recipes. However, this time we're going to notify the UI through `INotifyDataErrorInfo`, instead of throwing exceptions. To achieve this, carry out the following steps:

1. Open the solution you created in the *Using the different modes of data binding to allow persisting data* recipe in Chapter 2 (or the starter solution) and locate the `OwnerDetailsEdit.xaml` file. In this XAML file, locate and change the `LastJsonValueTextBlock` by adding `NotifyOnValidationError=true` to the `Binding` syntax. This can be seen in the following code:

```
<TextBox x:Name="LastJsonValueTextBlock"
         Grid.Row="3"
         Grid.Column="1"
         Margin="2"
         Text="{Binding LastName, Mode=TwoWay,
               NotifyOnValidationError=true }" >
</TextBox>
```

2. Add a button named `ValidateButton` next to the `OKButton` as shown in the following code:

```
<StackPanel Orientation="Horizontal"
            Grid.Row="1">
    <Button x:Name="ValidateButton"
            Content="Validate"
            Click="ValidateButton_Click"
            Width="75"
            Height="23"
            HorizontalAlignment="Right"
            Margin="0,12,0,0" />
    <Button x:Name="OKButton"
            Content="OK"
            Click="OKButton_Click"
            Width="75"
            Height="23"
            HorizontalAlignment="Right"
            Margin="0,12,0,0"
            Grid.Row="1" />
</StackPanel>
```

3. Add the following C# code to `OwnerDetailsEdit.xaml.cs`. This implements the `ValidateButton` handler we defined in the previous step.

```
private void ValidateButton_Click(object sender,
    RoutedEventArgs e)
{
    owner.FireValidation();
}
```

4. Locate the `Owner.cs` file, which represents the type of `DataContext` of the `OwnerDetailsEdit` control, and let it implement the `INotifyDataErrorInfo` interface as shown in the following code:

```
public class Owner : INotifyPropertyChanged, INotifyDataErrorInfo
{
    private Dictionary<string, string> FailedRules
    { get; set; }
    public event EventHandler<DataErrorsChangedEventArgs>
        ErrorsChanged;
    public IEnumerable GetErrors(string propertyName)
    {
        if (FailedRules.ContainsKey(propertyName))
            return FailedRules[propertyName];
        else
            return FailedRules.Values;
    }
    public bool HasErrors
    {
        get { return FailedRules.Count > 0; }
    }
    private void NotifyErrorsChanged(string propertyName)
    {
        if (ErrorsChanged != null)
            ErrorsChanged(this, new
                DataErrorsChangedEventArgs(propertyName));
    }
}
```

5. Add a constructor to `Owner.cs` to initialize the `FailedRules` dictionary as shown in the following code:

```
public Owner()
{
    FailedRules = new Dictionary<string, string>();
```

6. Implement the `FireValidation` method, which is called in `OwnerDetailsEdit.xaml.cs`, as shown in the following code:

```
internal void FireValidation()
{
    if (lastName.Length > 20)
    {
        if (!FailedRules.ContainsKey("LastName"))
            FailedRules.Add("LastName",
                "Last name cannot have more than 20 characters");
    }
    else
    {
        if (FailedRules.ContainsKey("LastName"))
            FailedRules.Remove("LastName");
    }
    NotifyErrorsChanged("LastName");
}
```

7. We can now build and run the solution. When the **Validate** button is clicked and there are more than 20 characters entered in the **Last name** field, a validation error will be shown.

The result can be observed in the following screenshot:



How it works...

The Silverlight controls observe the `INotifyDataErrorInfo` interface automatically. This means the control will display the correct error (invalid state) when a validation rule is violated by an entity.

In this example, we're validating the `LastName` property when the **Validate** button is clicked. If the **Last name** field contains too many characters, then the validation rule will be violated and the UI will show the typical "invalid value" tooltip automatically.

We could have achieved the same result by throwing an exception in the `LastName` property's set accessor. However, the main difference is that we can now validate and have the UI react to it without having to write the validation code in the `LastName` property's set accessor.

We can call it from anywhere and the UI will still react to it. This is how we can use the `INotifyDataErrorInfo` to perform server-side validation and make our UI react to it. You could call a server-side method in your property's set accessor and notify the UI through the `INotifyDataErrorInfo` in the callback of that method.

Nevertheless, in this case, the property's set accessor is probably a better place to do the validation (through `INotifyDataErrorInfo`). But for demonstration purposes, it's done in the click handler of the button.

When we implement the `INotifyDataErrorInfo` interface, we get an `ErrorsChanged` event handler, a `GetErrors` method that must return the correct error message as `IEnumerable`, and a `HasErrors` method. We need to implement these methods. To do this, we create a Dictionary called `FailedRules`, which we initialize in the class constructor and which will contain a list of errors. The `GetErrors` method, which accepts a `propertyName` parameter, fetches the correct error (or errors, if you keep a list of errors) from the `FailedRules` dictionary, while the `HasErrors` method is implemented by returning whether or not there are errors in the dictionary.

On clicking the button, the `FireValidation` method is called. This method will check if the `LastName` has more than 20 characters and will add an error to the `FailedRules` dictionary if the validation fails (or remove the error if the validation succeeds). After this, the `NotifyErrorsChanged` method is called, which fires the `ErrorsChanged` event. This event will make sure that the UI is notified (if the `Binding` syntax states that `NotifyOnValidationError` should be `true`) and will let Silverlight display errors where applicable.

There's more...

In this recipe, we've implemented validation using the `INotifyDataErrorInfo` interface. However, another interface of the same family exists, the `IDataErrorInfo` interface. The `INotifyDataErrorInfo` interface is typically used for more complex scenarios such as the need for `async`(`ronymous`) server-side validation to which the UI has to react, or when multiple errors have to be represented with different messages. The `IDataErrorInfo` interface is used for simpler, client-side validation.

When you implement the `IDataErrorInfo` interface, you get an `Item` property (accessible through an indexer in C#) and an `Error` property. The first one is used to get a specific error message on a certain property of your entity, while the second one is typically used to get an error message related to the complete entity.

See also

If you want to learn more about validation, you might want to take a look at the previous two recipes, *Validating data bound input* and *Validating data input using attributes*. To learn more about two-way data binding, have a look at the *Using the different modes of data binding to allow persisting data* recipe in Chapter 2.

Using templates to customize the way data is shown by controls

Applies to Silverlight 3, 4 and 5

Normally when we ask Silverlight to visualize an object, for example a person or a customer, it will simply display the result of the `ToString()` method—which is, of course, a string. This can be seen when we're binding a collection of items to a `ListBox`. If we don't specify a value for the `DisplayMemberPath` property, we simply see the name of the type (unless we overloaded the `ToString()` method). However, it's possible to specify a **template** called a `DataTemplate`, which will be used to visualize an object. It's in fact nothing more than a block of XAML code that gets rendered when an item of a particular type is visualized.

In this recipe, we'll build a `DataTemplate` to render the activities in a `ListBox` occurring on an account.

Getting ready

To follow along with this recipe, you can either use your own code that has been created from the previous recipes or use the starter solution that is located in the Chapter03/SilverlightBanking_DataTemplates_Starter folder in the code bundle available on the Packt website. The complete solution for this recipe can be found in the Chapter03/SilverlightBanking_DataTemplates_Completed folder.

How to do it...

Instead of immediately building the template, we'll go through a few steps. We'll start from the simple text representation and finish with a complete DataTemplate. Let's get started!

1. The collection of AccountActivity items is displayed in a `ListBox`. The following is the XAML code for this control:

```
<ListBox x:Name="AccountActivityListBox"
        Width="600"
        Grid.Row="1">
</ListBox>
```

2. Getting the items in the `ListBox` is achieved through setting the `ItemsSource` property to the `ObservableCollection<AccountActivity>` called `accountActivitiesCollection`. This is shown in the following code:

```
AccountActivityListBox.ItemsSource = accountActivitiesCollection;
```

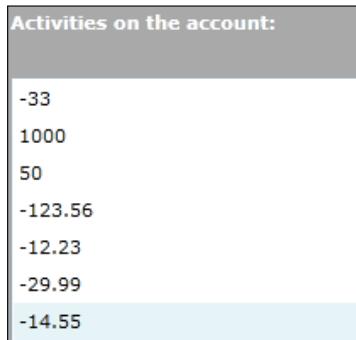
3. When populating this `ListBox` with `AccountActivity` objects and omitting any information that tells the `ListBox` what to display (as we did here), it will simply call the `ToString()` implementation of the object. This mostly results in displaying the string representation of the type of the object, as shown in the following screenshot:



4. We can tell the `ListBox` which property it should display through the `DisplayMemberPath`. For example, we can ask it to display the `Amount` property. This is shown in the following code:

```
<ListBox x:Name="AccountActivityListBox"
        DisplayMemberPath="Amount" >
</ListBox>
```

The `ListBox` now displays the value of the `Amount` property, as can be seen in the following screenshot:

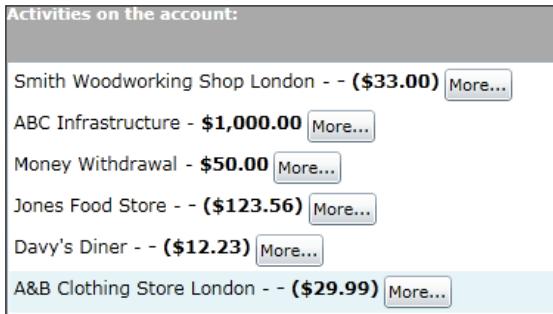


5. We can all agree that this is not the best way of displaying data! Now, let's start by creating an easy `DataTemplate`. Such a template is in fact nothing more than some XAML that contains some data binding statements. (Although it's not mandatory, it won't make sense to create a template without data binding.) Our first simple template contains a `StackPanel` with three `TextBlock` controls and a `Button`. We specify this template as a value for the `ItemTemplate`. This is shown in the following code. Note that the `DisplayMemberPath` property was removed.

```
<ListBox x:Name="AccountActivityListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Beneficiary}"
                           FontSize="12" >
                </TextBlock>
                <TextBlock Text=" - "
                           FontSize="12">
                </TextBlock>
                <TextBlock Text="{Binding Amount,
                               Converter={StaticResource
                               localCurrencyConverter}}"
                           FontSize="12"
                           FontWeight="Bold">
                </TextBlock>
            <Button x:Name="DetailButton"
                   Click="DetailButton_Click"
                   Content="More..." Margin="3 0 0 0">
```

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

The following screenshot shows the template in action:



6. If we want to reuse the template several times throughout the application, then it should be moved to the Resources collection of the App.xaml file. If we want to limit the scope, we can also place it in the Resources of a container such as a Grid or the UserControl. However, when placing the template in the Resources, we need to give it a name using the `x:Key` property. This key is then used for specifying which template is to be used. This can be seen in the following code:

```
<UserControl.Resources>
    <DataTemplate x:Key="SimpleTemplate">
        ...
    </DataTemplate>
</UserControl.Resources>
```

The following code shows how we should apply the template in a ListBox:

```
<ListBox x:Name="AccountActivityListBox"
    ItemTemplate="{StaticResource SimpleTemplate}">
</ListBox>
```

7. A template can also contain complex controls along with the simple controls placed in a StackPanel. The following code shows a more complex template named ComplexTemplate. It contains a Border with a LinearGradientBrush. Nested inside this border is a Grid, which contains some TextBlock controls, bound to a specific property. Note that we can also specify events such as the MouseLeftButtonDown inside the template.

```
<DataTemplate x:Key="ComplexTemplate">
    <Border BorderBrush="LightGray"
        BorderThickness="1"
        CornerRadius="2"
        Margin="0 3 0 1"
        Padding="2" >
        <Border.Background>
            <LinearGradientBrush EndPoint="1.207,0.457"
                StartPoint="-0.017,0.467">
                <GradientStop Color="#FF807777"/>
                <GradientStop Color="White" Offset="0.949"/>
            </LinearGradientBrush>
        </Border.Background>
        <Grid Width="580" >
            <Grid.RowDefinitions>
                <RowDefinition/><RowDefinition/>
                <RowDefinition/><RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="40"/></ColumnDefinition>
                <ColumnDefinition/></ColumnDefinition>
                <ColumnDefinition/></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <TextBlock Grid.Row="0"
                Grid.Column="0"
                Grid.RowSpan="2"
                Text="{Binding ActivityDate,
                    Converter={StaticResource
                        localShortDateConverter}}">
            </TextBlock>
            <TextBlock Grid.Row="0"
                Grid.Column="1"
                Text="{Binding Beneficiary}"
                FontWeight="Bold">
            </TextBlock>
            <TextBlock Grid.Row="0"
                Grid.Column="2"
                HorizontalAlignment="Right"
                Text="{Binding Amount,
                    Converter={StaticResource
                        localCurrencyConverter}}"
                Foreground="{Binding Amount,
                    Converter={StaticResource
                        localAmountToColorConverter}}">
            </TextBlock>
        </Grid>
    </Border>
</DataTemplate>
```

```
<TextBlock Grid.Row="1"
           Grid.Column="1"
           Text="{Binding ActivityDescription}">
</TextBlock>
<TextBlock x:Name="DetailsTextBlock"
           Grid.Row="1"
           Grid.Column="2"
           HorizontalAlignment="Right"
           Text="Details..."
           Tag="{Binding ActivityId}"
           MouseLeftButtonDown=
               "DetailsTextBlock_MouseLeftButtonDown"
           TextDecorations="Underline"
           Foreground="Blue" >
</TextBlock>
</Grid>
</Border>
</DataTemplate>
```

The result of this template is shown in the following screenshot:

01/09 Smith Woodworking Shop London Paid by credit card	- (\$33.00) Details...
01/09 ABC Infrastructure Paycheck September 2009	\$1,000.00 Details...
02/09 Money Withdrawal ATM Oxford Street London	\$50.00 Details...
05/09 Jones Food Store	- (\$123.56) Details...

How it works...

A DataTemplate allows us to define how a data object should be visualized. They work really well when binding data to an `ItemsControl`, such as a `ListBox`. By default, while binding the items to this control, it will render the items as a string, coming from the `ToString()` method. When specifying a `DataTemplate`, for each item bound to the `ListBox`, Silverlight will render the XAML code specified in the template by taking into account the data binding expressions contained in the template.

A `DataTemplate` can contain all types of controls, varying from grids to buttons. Events such as a click on a `Button` or a `MouseLeftButtonDown` on a `TextBlock` from within a template are supported as well. To find out which item was clicked, we can use the `DataContext`. The `DataContext` for each item in the list is an `AccountActivity`. The following line of code displays a detail window based on the selected item:

```
ActivityDetailView activityDetailView = new  
    ActivityDetailView(accountActivitiesCollection.  
        Where<AccountActivity>(a => a.ActivityId ==  
            ((AccountActivity)((TextBlock)sender).DataContext).  
                ActivityId).First<AccountActivity>());
```

A `DataTemplate` can be specified on the control itself. For a `ListBox`, this is done by specifying the template as a value for the `ItemTemplate`. However, it's more often useful to specify the template at a higher level in the XAML hierarchy, such as the `UserControl` or (even better) the `App.xaml` file. While using the latter, the template will be available throughout the entire application. One thing to note here is that the template should then be given a name that is specified through the `x:Key` property. This value is then used for retrieving the correct template in the resources collection.

Using implicit data templates

Applies to Silverlight 5

In the previous recipe, we defined a data template in resources and referred to it from the `ListBox` control that makes use of this template. With Silverlight 5, another type of data template is added to Silverlight: the **implicit data template**. Using this type, we define a template for a **type** and when Silverlight needs to visualize an instance of that type, it uses the template. This functionality has been available in WPF and is now available in Silverlight as well. In this recipe, we'll introduce implicit data templates to visualize data instances.

Getting ready

If you are following along with the recipes, you can continue using the code from the previous recipe. Alternatively, a starter solution is provided in the `Chapter03/SilverlightBanking_Implicit_DataTemplates_Starter` folder. The complete solution can be found in the `Chapter03/SilverlightBanking_Implicit_DataTemplates_Completed` folder.

How to do it...

Instead of using a regular data template, we are going to define an implicit data template for the `AccountActivity` type. Also, we are going to create an inheriting type to show that implicit data templates can give us more freedom when we have more than one type being displayed in a `ListBox`. The following steps will show you how to do this:

1. Open the solution as outlined in the *Getting ready* section. First, let's refactor the data template so that it becomes an implicit data template. Open `MainPage.xaml` and look for the `DataTemplate` (in the resources block of the `UserControl`) named `ComplexTemplate`. As we can see, it has a key, which is used to refer to it from within the XAML code. Delete the `key` attribute and introduce the `DataType` element as shown next:

```
<DataTemplate DataType="local:AccountActivity">  
    ...  
</DataTemplate>
```

2. The `DataType` refers to the `AccountActivity` type. Whenever we need to visualize an instance of this type, Silverlight will pick this template automatically.
3. Still in the `MainPage.xaml`, in the `AccountActivityListBox` declaration near the end of the page, remove the `ItemTemplate` attribute, so that you have the following code:

```
<ListBox x:Name="AccountActivityListBox"  
        Width="600" Grid.Row="1"></ListBox>
```

4. Run the application now. We are getting the same result as before: the `ListBox` is filled with `AccountActivity` instances and Silverlight uses the implicit data template to visualize these.
5. We are now going to introduce an inheriting type called `RecurringAccountActivity`, which inherits from `AccountActivity`. Add a new class and add the following code:

```
public class RecurringAccountActivity : AccountActivity  
{  
    public string RecurringActivityDescription { get; set; }  
}
```

6. In the `MainPage.xaml`, we are now going to introduce a second implicit data template for this type. Add the following template to the resources of the `UserControl`:

```
<DataTemplate DataType="local:RecurringAccountActivity">  
    <Border BorderBrush="LightGray"  
            BorderThickness="1"  
            CornerRadius="2"  
            Background="LightGreen"
```

```
Margin="0 3 0 1"
Padding="2" >
<Grid Width="580" >
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="40"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="0"
        Grid.RowSpan="2"
        Text="{Binding ActivityDate,
            Converter={StaticResource
                localShortDateConverter}}}>
    </TextBlock>
    <TextBlock Grid.Row="0" Grid.Column="1"
        Text="{Binding Beneficiary}"
        FontWeight="Bold">
    </TextBlock>
    <TextBlock Grid.Row="0" Grid.Column="2"
        HorizontalAlignment="Right"
        Text="{Binding Amount, Converter={StaticResource
            localCurrencyConverter}}}"
        Foreground="{Binding Amount,
            Converter={StaticResource
                localAmountToColorConverter}}}>
    </TextBlock>
    <StackPanel Grid.Row="1"
        Grid.Column="1"
        Orientation="Horizontal">
        <TextBlock Text="{Binding ActivityDescription}">
        </TextBlock>
        <TextBlock Text=" - "></TextBlock>
        <TextBlock Text="{Binding
            RecurringActivityDescription}">
        </TextBlock>
    </StackPanel>
    <TextBlock x:Name="DetailsTextBlock"
        Grid.Row="1"
        Grid.Column="2"
        HorizontalAlignment="Right"
```

```
        Text="Details..."  
        Tag="{Binding ActivityId}"  
        MouseLeftButtonDown="DetailsTextBlock_  
        MouseLeftButtonDown"  
        TextDecorations="Underline"  
        Foreground="Blue" >  
    </TextBlock>  
  </Grid>  
</Border>  
</DataTemplate>
```

7. As it can be seen, this template is based on the template for AccountActivity. It has a green background instead of the gradient and it displays the RecurringActivityDescription field as well.
8. In the Mainpage.xaml.cs, in the InitializeActivitiesCollection() method, let's add an instance of this new type to the collection. Add the following code after adding the first element to the collection:

```
RecurringAccountActivity recurringAccountActivity =  
    new RecurringAccountActivity();  
recurringAccountActivity.ActivityId = 101;  
recurringAccountActivity.Amount = -120;  
recurringAccountActivity.Beneficiary = "Silverlight Bank";  
recurringAccountActivity.ActivityDescription = "Loan payment";  
recurringAccountActivity.ActivityDate =  
    new DateTime(2009, 9, 1);  
recurringAccountActivity.RecurringActivityDescription = "Month  
September";  
accountActivitiesCollection.Add(recurringAccountActivity);
```

Run the application now. As seen in the following screenshot, the correct template is being loaded based on the type of the instance.

01/09 Smith Woodworking Shop London Paid by credit card	- (\$33.00) Details...
01/09 Silverlight Bank Loan payment - Month September	- (\$120.00) Details...
01/09 ABC Infrastructure Paycheck September 2009	\$1,000.00 Details...

How it works

When we want to display an instance in a control such as a `ListBox` and we don't want to settle for the default way of displaying that instance, being a single line of data, we can use data templates. When using regular data templates, we always have to refer to the key of the template.

Also, within a single container such as the `ListBox` we are limited in that we don't have the ability to visualize inheriting types in a different manner. Only one single template can be used.

In WPF, this issue could be solved using implicit data templates. With Silverlight 5, this feature is now available as well. An implicit data template is associated with a type. When the type needs to be visualized, Silverlight picks the correct template. Just like with other templates, we can define these templates within a specific scope. This scope can be the `UserControl`, a container such as a `Grid` or even the application itself, through the `App.xaml`.

If we have a collection of types and deriving types and we created templates for each of these, Silverlight will pick the template for the inheriting type when needed. This gives us the flexibility to use more than one template within a single container.

Not everything though that's possible in WPF in this area is available in Silverlight 5. WPF has the `DataTemplateSelector`, which allows selecting a different template based on the value of a specific property.

See also

In the previous recipe, *Using templates to customize the way data is shown by controls*, we looked at regular data templates.

Using the Ancestor RelativeSource binding

Applies to Silverlight 5

In Silverlight 5, a new feature was introduced called the **Ancestor RelativeSource** binding. This type was already available to WPF developers but with the introduction of version 5, also made its appearance in Silverlight. Using this binding, we can bind to relative source elements easily. This feature is useful in data templates. In this recipe, we'll extend the Silverlight banking sample once again to including this binding type.

Getting ready

For this recipe, a starter solution is provided in the `Chapter03/SilverlightBanking_Relative_Source_Starter` folder. The complete solution can be found in the `Chapter03/SilverlightBanking_Relative_Source_Completed` folder.

How to do it...

When we create a data template, for example for a `ListBox` control, we may want to bind to a value of the `ListBoxItem`, for example the `IsSelected` property. Before Silverlight 5, accessing that value was not easy. With the addition of the Ancestor RelativeSource binding, this has become possible. The following steps will show how we can bind to the `IsSelected` property from the data template.

1. Open the solution as outlined in the *Getting ready* section. In `MainPage.xaml`, in the `UserControl` resources, we have defined a template named `ComplexTemplate`. We'd like to change this template so that it displays a rectangle in the first column of the `Grid` within the template. First, let's change the XAML to include this rectangle, as shown next:

```
<DataTemplate x:Key="ComplexTemplate">
    <Border BorderBrush="LightGray"
            BorderThickness="1"
            CornerRadius="2"
            Margin="0 3 0 1"
            Padding="2">
        <Border.Background>
            <LinearGradientBrush EndPoint="1.207,0.457"
                                StartPoint="-0.017,0.467">
                <GradientStop Color="#FF807777"/>
                <GradientStop Color="White" Offset="0.949"/>
            </LinearGradientBrush>
        </Border.Background>
        <Grid Width="580" >
            <Grid.RowDefinitions>
                <RowDefinition/></RowDefinition>
                <RowDefinition/></RowDefinition>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="20"/></ColumnDefinition>
                <ColumnDefinition Width="40"/></ColumnDefinition>
                <ColumnDefinition/></ColumnDefinition>
                <ColumnDefinition/></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <Rectangle Grid.RowSpan="2"
                      Grid.Row="0" Grid.Column="0"
                      Stroke="Green"
                      StrokeThickness="1"
                      Width="10"
                      Height="10"
                      Fill="Red"
```

```

        RadiusX="3"
        RadiusY="3" />
    <TextBlock Grid.Row="0" Grid.Column="1"
        Grid.RowSpan="2"
        Text="{Binding ActivityDate,
        Converter={StaticResource localShortDateConverter}}">
    </TextBlock>
    <TextBlock Grid.Row="0" Grid.Column="2"
        Text="{Binding Beneficiary}"
        FontWeight="Bold">
    </TextBlock>
    <TextBlock Grid.Row="0" Grid.Column="3"
        HorizontalAlignment="Right"
        Text="{Binding Amount,
        Converter={StaticResource localCurrencyConverter}}"
        Foreground="{Binding Amount, Converter={StaticResource
        localAmountToColorConverter}}">
    </TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="2"
        Text="{Binding ActivityDescription}">
    </TextBlock>
    <TextBlock x:Name="DetailsTextBlock" Grid.Row="1"
        Grid.Column="3" HorizontalAlignment="Right"
        Text="Details...">
        Tag="{Binding ActivityId}"
        MouseLeftButtonDown="DetailsTextBlock_
        MouseLeftButtonDown"
        TextDecorations="Underline"
        Foreground="Blue" >
    </TextBlock>
</Grid>
</Border>
</DataTemplate>

```

2. Note that the Grid has an extra column where the Rectangle was added.
3. We need to introduce a converter that will convert the value of `IsSelected`, which is a Boolean, into a value of the `Visibility` enumeration. Add a class named `VisibilityConverter` to the `Converters` folder and add the following code:

```

public class VisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object
    parameter, CultureInfo culture)
    {
        bool visible = (bool)value;
        if (visible)

```

```
        return System.Windows.Visibility.Visible;
    else
        return System.Windows.Visibility.Collapsed;
}
public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
{
    throw new NotImplementedException();
}
```

4. Instantiate the converter in the resources of the `MainPage.xaml` as follows:

```
<localconverters:VisibilityConverter
x:Key="localVisibilityConverter">
</localconverters:VisibilityConverter>
```

5. We now want to show the rectangle only when the `ListBoxItem` is selected. We can do this using the following code:

```
<Rectangle Grid.RowSpan="2" Grid.Row="0" Grid.Column="0"
Visibility="{Binding RelativeSource={RelativeSource
AncestorType=ListBoxItem}, Path=IsSelected,
Converter={StaticResource localVisibilityConverter}}"
Stroke="Green"
StrokeThickness="1"
Width="10"
Height="10"
Fill="Red"
RadiusX="3"
RadiusY="3" />
```

We are binding the `Visibility` of the `Rectangle` to the `Visibility` of the `ListBoxItem`.

In the following screenshot, we can see that when an item gets selected, a Retangle appears.

01/09 Smith Woodworking Shop London Paid by credit card	- (\$33.00) Details...
01/09 ABC Infrastructure Paycheck September 2009	\$1,000.00 Details...
02/09 Money Withdrawal ATM Oxford Street London	\$50.00 Details...

How it works

Before Silverlight 5, it was difficult from within a data template to reach a value of a property in the container. We could, in the case of a `ListBoxItem`, start changing the `ItemContainerStyle` and then build a new template for the `ListBoxItem` from there. It's easy to understand that this approach is quite cumbersome and not often used.

Version 5 introduced a solution in the form of the `AncestorType` on the `RelativeSource`. The following code shows the usage of this:

```
<Rectangle  
    Visibility="{Binding RelativeSource={RelativeSource  
        AncestorType=ListBoxItem},  
        Path=IsSelected,  
        Converter={StaticResource localVisibilityConverter}}">  
</Rectangle>
```

When specifying the value of the `AncestorType` to a specific type, Silverlight will search up the visual tree for the first occurrence of an instance of that type (here, the `ListBoxItem`). This `ListBoxItem` then becomes the source for the rest of the binding: we are effectively binding the `Visibility` of the `Rectangle` to the `IsSelected` property of the `ListBoxItem`. The converter needs to be added since `IsSelected` is of type `Boolean`, whereas the `Visibility` expects a value of the `Visibility` enumeration.

See also

In the *Using templates to customize the way data is shown by controls* recipe of this chapter, we looked at working with data templates. Converters are covered in the *Hooking into the data binding process using converters* recipe, also from this chapter.

Creating custom markup extensions

Applies to Silverlight 5

The binding statement we have been using up until now is a **markup extension**. Next to the binding markup extension, `StaticResource` is a second one that's built-in in Silverlight. Every time we see a notation that starts and ends with a curly brace (`{ }`), we're working with a markup extension.

Up until Silverlight 4, only the built-in markup extensions could be used. Starting with Silverlight 5, it's now possible to create our own. In this recipe, we'll look at how we can create a markup extension that is capable of returning a translated text.

Getting ready

If you have been following along with the recipes of this chapter, you can continue using the recipe from the previous one. Alternatively, a starter solution is available from the Chapter03/ SilverlightBanking_Custom_Markup_Extensions_Starter folder. The complete solution can be found in the Chapter03/SilverlightBanking_Custom_Markup_Extensions_Completed folder.

How to do it...

To show how we can create and use our own markup extension, we'll build one that gets a translated value from a dictionary class. Let's look at the steps we need to perform:

1. With the solution open, create a new folder called Dictionaries in the Silverlight project.
2. In this folder, create a class called DutchDictionary. This class contains a Dictionary<string, string> which contains a list of translated text values. It also contains a GetTranslation() method, which interrogates the Dictionary for the translated text. If none is found, the default is returned. The class is shown next:

```
public static class DutchDictionary
{
    public static Dictionary<string, string> Translations
    { get; set; }
    static DutchDictionary()
    {
        InitializeTranslations();
    }
    private static void InitializeTranslations()
    {
        Translations = new Dictionary<string, string>();
        Translations.Add
            ("ApplicationName", "Welkom bij Silverlight Bank");
    }
    public static string GetTranslation
        (string key, string defaultTranslation)
    {
        if (Translations.ContainsKey(key))
            return Translations[key];
        return defaultTranslation;
    }
}
```

3. This code requires that you add a using statement to the `System.Collections.Generic` namespace.
4. Create the `FrenchDictionary` and `EnglishDictionary` classes as well. The code can be found in the finished solution.
5. Now we'll create the markup extension that will return a translated value. Start by creating a folder in the solution named `Extensions`.
6. In this folder, create a class called `TranslationExtension`. This class should inherit from `MarkupExtension`, a class added to Silverlight with version 5. The class so far is shown next:

```
public class TranslatorExtension : MarkupExtension
{
}
```

7. If the `MarkupExtension` class can't be found, add a using statement for `System.Windows.Markup`.
8. In this class, add two properties, `Key` and `Default`. These will be used as parameters for our markup extension when used from XAML. Add the following code:

```
public string Key { get; set; }
public string Default { get; set; }
```

9. The most important part of the class is the override of the `ProvideValue()` method. This method will return an object that is returned as the value of the target property for where the markup extension is used. In this method, we are checking which is the current culture. Based on this value, a translated text value is returned from one of the available dictionaries. Add the following code to the `TranslatorExtension` class:

```
public override object ProvideValue
    (IServiceProvider serviceProvider)
{
    string language =
        Thread.CurrentThread.CurrentCulture.ToString();
    if (language == "nl-BE")
    {
        return DutchDictionary.GetTranslation(Key, Default);
    }
    else if (language == "fr-BE")
    {
        return FrenchDictionary.GetTranslation(Key, Default);
    }
    else
    {
```

```
        return EnglishDictionary.GetTranslation(Key, Default);
    }
}
```

10. We can now use our extension. First, in the `MainPage.xaml`, add the following line, which makes the namespace known inside XAML code:

```
xmlns:extensions="clr-namespace:SilverlightBanking.Extensions"
```

11. Search for the `TitleTextBlock` control and change its `Text` property as follows:

```
<TextBlock x:Name="TitleTextBlock"
    Text="{extensions:TranslatorExtension
        Key=ApplicationName, Default=Welcome}" >
</TextBlock>
```

If we run the application now, we'll get a translated value. On my personal machine, my culture setting is *nl-BE*, so the Dutch translation is retrieved, as shown in the following screenshot:



How it works

We've been using markup extensions throughout this and the previous chapter all along. Every data binding we have created using the `Binding` between { and } is a markup extension. A markup extension, as the word says, basically extends XAML. Using a markup extension, we can pass information which can be used by the XAML parser when creating the object.

Up until Silverlight 4, we were limited to using the markup extensions that came with the framework. However, WPF already had the ability to allow developers to create their own extensions. With Silverlight 5, the platform has been extended with the required base classes and interfaces to support custom markup extensions.

Custom markup extensions are useful in many scenarios. For example, we can wrap functionality inside an extension, for which we would normally have to expose extra properties from our data context. Also, functionality that we would normally add using a converter can be wrapped in a markup extension. Finally, we also get access to static classes from XAML, which isn't possible otherwise. This can be useful to access classes such as service locators which are often static.

Creating a custom markup extension is possible by inheriting from the `MarkupExtension` class. In our extension, we then have to override the `ProvideValue()` method. This method, as the name implies, returns an object for the value of the property where we are using the extension in XAML.

A custom markup extension can have parameters. We can create parameters by creating one or more public properties on the extension class. From XAML, these are then available within the markup extension. In the following code, both `Key` and `Default` are parameters.

```
Text="{extensions:TranslatorExtension Key=ApplicationName,  
Default=Welcome}"
```

One important note for WPF developers is that positional parameters are not available in Silverlight's implementation of custom markup extensions. This means that we always have to specify the name of the parameter; we can't fall back on a default property.

Building a change-aware collection type

Applies to Silverlight 3, 4 and 5

We may not always have the option of binding to a collection that implements the `INotifyCollectionChanged` interface. For example, what if we have a service that returns `IList<T>?` Can't we use the automatic synchronization features that Silverlight's data binding engine offers us?

The good news is that we can. For that, we need to build a wrapper class around the `IList<T>`. This class will implement the necessary interface and will allow data binding to work in the manner in which we are accustomed.

Getting ready

The complete solution for this recipe can be found in the `Chapter03/CustomCollections` folder in the code bundle available on the Packt website.

How to do it...

For this recipe, we'll assume that we need to work with an external assembly called `UnchangeableCode` in the sample code, which we simply can't change. Inside the assembly, a class returns a list of `Owner` instances as `IList<Owner>`. However, in our Silverlight application, we would still like to use the synchronization that data binding offers us. We'll implement this by building a wrapper class. We need to perform the following steps in order to achieve this:

1. The `UnchangeableCode` project contains a class called `OwnerService`. This class contains a `List<Owner>` as shown in the following code:

```
public class OwnerService
{
    private List<Owner> owners;
    public List<Owner> Owners
    {
        get { return owners; }
        set { owners = value; }
    }
    public OwnerService()
    {
        owners = new List<Owner>();
        Owner o1 = new Owner()
        {
            Name = "Gill Cleeren",
            CurrentBalance = 100
        };
        Owner o2 = new Owner()
        {
            Name = "Kevin Dockx",
            CurrentBalance = 200
        };
        Owner o3 = new Owner()
        {
            Name = "Marina Smith",
            CurrentBalance = 300
        };
        Owner o4 = new Owner()
        {
            Name = "Lindsey Smith",
            CurrentBalance = 400
        };
        owners.Add(o1);
        owners.Add(o2);
        owners.Add(o3);
        owners.Add(o4);
    }
}
```

2. In our Silverlight application, we would like to bind to the list of `Owner` instances not only for displaying the data, but also for viewing any changes done to the list immediately. We'll create a class that wraps around the `List<Owner>`. This class will also implement the `INotifyCollectionChanged` interface as shown in the following code:

```
public class CustomOwnerList : IList<Owner>,
    INotifyCollectionChanged
{
    private IList<Owner> owners;
    public CustomOwnerList(IList<Owner> owner)
    {
        this.owners = owner;
    }
}
```

3. We can now start implementing all the methods that are defined by both interfaces. The `INotifyCollectionChanged` interface defines only one event, which is called the `CollectionChanged` event. This is shown in the following line of code:

```
public event NotifyCollectionChangedEventHandler
    CollectionChanged;
```

4. The `IList` interface contains a lot of methods that we need to implement. The following is the code for the `Insert` method. Notice that we're manually calling the `CollectionChanged` event when something changes in the list. We wrap the call of the `CollectionChanged` event in the `OnCollectionChanged` method. This method includes checking that the event isn't null. The other methods are similar and the code for these methods can be found in the code bundle available on the Packt website.

```
public void Insert(int index, Owner item)
{
    owners.Insert(index, item);
    OnCollectionChanged(new NotifyCollectionChangedEventArgs
        (NotifyCollectionChangedAction.Add, item, index));
}
private void OnCollectionChanged(NotifyCollectionChangedEventArgs
    notifyCollectionChangedEventArgs)
{
    if (CollectionChanged != null)
        CollectionChanged(this, notifyCollectionChangedEventArgs);
}
```

5. Now that we have the wrapper, we can work with the list as if it's a regular `ObservableCollection`. Whenever we add, remove, or change items in the list, we'll see those changes directly in the UI. The following code shows the instantiation of the new collection and sets it as the `DataContext` for a `ListBox` control:

```
OwnerService someOldClass = new OwnerService();
CustomOwnerList list = new CustomOwnerList(someOldClass.Owners);
OwnerListBox.ItemsSource = list;
```

How it works...

If we want to make use of the automatic synchronization offered by Silverlight's data binding for a collection, then this collection should implement the `INotifyCollectionChanged` interface. If it doesn't do this, we can still bind and show the items in the collection. However, changes to the collection won't be propagated into the UI. Although using the `ObservableCollection` is advised, sometimes we need to work with a service or an assembly from a third party that returns, for example, a generic list.

If we want the data of the generic list to be bound to the UI and the changes to the list to be visualized, then we need to build a class that wraps around the list. This class needs to implement the `IList<T>` interface. As a result, while implementing the methods, we work with the original list itself. For example, while implementing the `Insert` method, we insert an item in a specific location in the underlying list.

Also, the class needs to implement the `INotifyCollectionChanged` interface. For every change that is done in the list (such as adding an item), our wrapper class will raise the `CollectionChanged` event.

Now, whenever we want to bind, we bind to an instance of our wrapper class. Silverlight notices that this class implements the `INotifyCollectionChanged` interface, so it will register for the events that are raised by an instance of the wrapper class.

See also

Binding to regular collections is explained in the *Binding collections to UI elements* recipe of the previous chapter.

Combining converters, data binding, and DataContext into a custom DataTemplate

Applies to Silverlight 3, 4 and 5

A lot of things we've talked about in this chapter are great features on their own, but they really shine when you combine them and let them work together. This recipe will show you how to bring some of the most powerful and built-in features of the Silverlight SDK together or, to put it differently, how to program "The Silverlight Way". We're going to create an editable `ComboBox` of people using a custom `DataTemplate`, the `DataContext`, an `ObservableCollection` with two-way data binding, and `Converters` to make the UI fluid, interactive, and responsive.

Getting ready

We're starting off with a completely new, blank Silverlight solution for this recipe. So, to get started, make sure you have one of those. To create an empty Silverlight solution, start a new Silverlight project in Visual Studio by selecting **File | New | Project...** and let it create an accompanying web application automatically for hosting the Silverlight application. Name the project `Editables_Combobox`.

You can find the complete solution for this recipe in the `Chapter03/Combining_Converters_Databinding_And_Context_Completed` folder in the code bundle available on the Packt website.

How to do it...

We want to end up with a `ComboBox` that displays the names of a few people. Each person's name should be editable from inside the list of items in the `ComboBox`. To achieve this, we'll need to carry out the following steps:

1. We're going to start by adding a new class to our Silverlight project. This class is named `Person` and it has three properties: an `ID`, a `Name`, and a field that represents the current edit state of the person—`InEditMode`. This class implements the `INotifyPropertyChanged` interface as shown in the following code:

```
using system.ComponentModel;

public class Person : INotifyPropertyChanged
{
    public int PersonID { get; set; }
    private bool pInEditMode;
    public bool InEditMode
    {
        get
        {
            return pInEditMode;
        }
        set
        {
            pInEditMode = value;
            NotifyPropertyChanged("InEditMode");
        }
    }
    private string pName;
    public string Name
    {
        get
```

```
{  
    return pName;  
}  
set  
{  
    pName = value;  
    NotifyPropertyChanged("Name");  
}  
}  
#region INotifyPropertyChanged Members  
public event PropertyChangedEventHandler PropertyChanged;  
public void NotifyPropertyChanged(string propertyName)  
{  
    if (PropertyChanged != null)  
    {  
        PropertyChanged(this, new  
            PropertyChangedEventArgs(propertyName));  
    }  
}  
#endregion  
}
```

2. Next, we're going to add another class to our Silverlight project. This class is named `BoolToVisibilityConverter`. It will implement the `IValueConverter` interface and convert a Boolean value to a Visibility value. This is shown in the following code:

```
public class BoolToVisibilityConverter : IValueConverter  
{  
    #region IValueConverter Members  
    public object Convert(object value, Type targetType,  
        object parameter, System.Globalization.CultureInfo culture)  
    {  
        bool normalDirection = true;  
        if (parameter != null)  
        {  
            if (parameter.ToString().Trim().ToLower() ==  
                "trueiscollapsed")  
                normalDirection = false;  
        }  
        if (value is bool)  
        {  
            if ((bool)value)  
            {  
                return normalDirection ?  
                    Visibility.Visible : Visibility.Collapsed;  
            }  
        }  
    }  
}
```

```
        }
    else
    {
        return normalDirection ?
            Visibility.Collapsed : Visibility.Visible;
    }
}
else
{
    return Visibility.Visible;
}
}

public object ConvertBack(object value, Type targetType,
    object parameter, System.Globalization.CultureInfo culture)
{
    bool normalDirection = true;
    if (parameter.ToString().Trim().ToLower() == "trueiscollapsed")
        normalDirection = false;
    if (value is Visibility)
    {
        if ((Visibility)value == Visibility.Visible)
        {
            return normalDirection ? true : false;
        }
        else
        {
            return normalDirection ? false : true;
        }
    }
    else
    {
        return true;
    }
}
#endregion
}
```

3. Add the following using statements to that class:

```
using System;
using System.Windows;
using System.Windows.Data;
```

4. Open the MainPage.xaml file. We'll add the following code to represent our UI. It includes the Binding syntax for the person objects visible in our ComboBox, the necessary Converter syntax, and an event handler for the Click events of our **Edit** and **Save** buttons.

```
<UserControl x:Class="Editable_Combobox.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"
    mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480"
    xmlns:local="clr-namespace:Editable_Combobox">
    <UserControl.Resources>
        <local:BoolToVisibilityConverter
            x:Name="BoolToVisibilityConverter" />
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot" Margin="10" >
        <Grid.RowDefinitions>
            <RowDefinition Height="30" /></RowDefinition>
            <RowDefinition /></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock Text="An editable ComboBox"
            HorizontalAlignment="Left"
            VerticalAlignment="Top" >
        </TextBlock>
        <ComboBox x:Name="cmbPersons" Grid.Row="1"
            Width="220" Height="30"
            HorizontalAlignment="Left"
            VerticalAlignment="Top" >
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <Grid Width="280" Height="30" >
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="200" /></ColumnDefinition>
                            <ColumnDefinition /></ColumnDefinition>
                        </Grid.ColumnDefinitions>
                        <TextBlock Text="{Binding Name, Mode=TwoWay}"
                            HorizontalAlignment="Left"
                            VerticalAlignment="Center"
                            IsHitTestVisible="False"
                            Width="180"
                            Visibility="{Binding InEditMode,
                                Converter={StaticResource
                                    BoolToVisibilityConverter},
                                ConverterParameter=trueiscollapsed}" />
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
    </Grid>
</UserControl>
```

```

<TextBox Text="{Binding Name, Mode=TwoWay}"
Width="180" HorizontalAlignment="Left"
VerticalAlignment="Center"
Visibility="{Binding InEditMode,
Converter={StaticResource
BoolToVisibilityConverter},
ConverterParameter=trueisVisible}"/>
<Button x:Name="btnEdit" Width="70" Height="20"
Click="btnEditSave_Click"
Content="Edit" Grid.Column="1"
Visibility="{Binding InEditMode,
Converter={StaticResource
BoolToVisibilityConverter},
ConverterParameter=trueiscollapsed} " />
<Button x:Name="btnSave" Width="70" Height="20"
Click="btnEditSave_Click"
Content="Save" Grid.Column="1"
Visibility="{Binding InEditMode,
Converter={StaticResource
BoolToVisibilityConverter}}"/>
</Grid>
</DataTemplate>
</ComboBox.ItemTemplate>
</ComboBox>
</Grid>
</UserControl>

```

5. Open the `MainPage.xaml.cs` file. This is our code-behind file in which we'll write the following code to handle the `Click` events of our buttons as well as to initialize an `ObservableCollection` of the `Person` type:

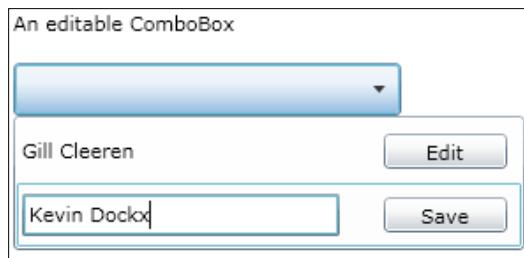
```

public partial class MainPage : UserControl
{
    public ObservableCollection<Person> Persons
    { get; set; }
    public MainPage()
    {
        InitializeComponent();
        InitializeCollection();
    }
    private void InitializeCollection()
    {
        Persons = new ObservableCollection<Person>()
        {
            new Person()
            {

```

```
        PersonID=1, Name="Gill Cleeren", InEditMode = false
    },
    new Person()
    {
        PersonID=2, Name="Kevin Dockx", InEditMode = false
    }
);
cmbPersons.ItemsSource = Persons;
}
private void btnEditSave_Click(object sender, RoutedEventArgs e)
{
    Person p = (Person)((Button)sender).DataContext;
    p.InEditMode = !p.InEditMode;
}
```

We can now build and run this project. The result can be observed in the following screenshot:



How it works...

This recipe brings together quite a few Silverlight principles into one project. Let's start with the `Person` class. This class represents the people shown in our editable ComboBox. It implements the `IPropertyChanged` interface, which makes sure that the UI is notified when one of the properties changes.

Our converter converts a Boolean value to a `Visibility` value. We bind the `Visibility` property of our `TextBlock`, `TextBox`, and `Buttons` to the `InEditMode` property of the `Person` class. This is done by using the converter to convert the Boolean value to a `Visibility` value and by using the `ConverterParameter` to decide how the value should be converted. As a result of this, the `TextBlock` and the **Edit** button will be `Visible` when the `InEditMode` property is `false`, and `Collapsed` when it's `true`. On the other hand, the `TextBox` and the **Save** button will be `Collapsed` when the `InEditMode` property is `false` and `Visible` when it's `true`.

Next, we've got the `Click` event handler on our buttons. In this handler, we can get the `DataContext` of the sender. Due to the fact that the `ItemsSource` in a `ComboBox` is a collection of persons, the `DataContext` of this `Button` is always exactly one person. We can then cast this `DataContext` in the `Person` and change its `InEditMode` property.

Bringing it all together, the `ObservableCollection` of the `Person` represents the data shown in the `ComboBox`. The `Converter` makes sure that the correct pieces of the UI are shown. Due to the `DataContext`, we can easily access our `Person` object on the click of a button. Also, as the `INotifyPropertyChanged` interface is implemented on the `InEditMode` property, the UI is updated when we change this property. Finally, the `TwoWay` data binding makes sure that the changes we make to a person's name are automatically persisted in the underlying object.

See also

This recipe brought together most of the principles that are covered in this book. To learn more about data binding, have a look at the following recipes in *Chapter 2*:

- ▶ *Displaying data in Silverlight applications*
- ▶ *Creating dynamic bindings*
- ▶ *Binding data to another UI element*
- ▶ *Binding collections to UI elements*
- ▶ *Enabling a Silverlight application to automatically update its UI*

To learn more about the `DataContext`, you can refer to the *Obtaining data from any UI element it is bound to* recipe in *Chapter 2*. Additionally, converters are covered in the *Hooking into the data binding process* recipe in this chapter. For more information on the `ObservableCollection`, have a look at the *Binding collections to UI elements* recipe in *Chapter 2*.

4

The Data Grid

This chapter takes an in-depth look at working with the `DataGridView` using the following recipes:

- ▶ Displaying data in a customized `DataGridView`
- ▶ Inserting, updating, and deleting data in a `DataGridView`
- ▶ Sorting and grouping data in a `DataGridView`
- ▶ Filtering and paging data in a `DataGridView`
- ▶ Using custom columns in the `DataGridView`
- ▶ Implementing master-detail in the `DataGridView`
- ▶ Validating the `DataGridView`

Introduction

If we want to build applications that deal with large amounts of data, then a control such as a data grid is vital. This control shows the data in a tabular format and allows for adding, editing, and deleting the data inline. It allows the sorting of data into columns by clicking on a column header. Finally, a data grid should support grouping, so that we can create levels in the data.

Silverlight included a data grid from version 2 onwards, even before Windows Presentation Foundation (WPF) had one. It's very powerful, supports all the features outlined previously, and is thus a good solution to work with large amounts of data in the browser. It lives in the `System.Windows.Controls` namespace. However, it's not included in the default assemblies that are installed with the Silverlight core. When using it in our application, Visual Studio will embed several assemblies into the XAP file.

In order to maintain its performance, Silverlight's `DataGrid` control features UI virtualization. This feature means that Silverlight will only create the items that are currently visible. As a result of this, even if we are displaying thousands or even millions of rows, the `DataGrid` control will still keep running smoothly.

In the recipes of this chapter, we'll look at how to work with the `DataGrid`. This is an essential control for applications that rely on (collections of) data.

Displaying data in a customized `DataGrid`

Applies to Silverlight 3, 4 and 5

Displaying data is probably the most straightforward task we can ask the `DataGrid` to do for us. In this recipe, we'll create a collection of data and hand it over to the `DataGrid` for display. While the `DataGrid` may seem to have a rather fixed layout, there are many options available on this control that we can use to customize it.

In this recipe, we'll focus on getting the data to show up in the `DataGrid` and customize it to our likings.

Getting ready

In this recipe, we'll start from an empty Silverlight application. The finished solution for this recipe can be found in the `Chapter04/Datagrid_Displaying_Data_Completed` folder in the code bundle that is available on the Packt website.

How to do it...

We'll create a collection of `Book` objects and display this collection in a `DataGrid`. However, we want to customize the `DataGrid`. More specifically, we want to make the `DataGrid` fixed. In other words, we don't want the user to make any changes to the bound data or move the columns around. Also, we want to change the visual representation of the `DataGrid` by changing the background color of the rows. We also want the vertical column separators hidden and the horizontal ones a different color. Finally, we'll hook into the `LoadingRow` event, which will give us access to the values that are bound to a row and based on that value, the `LoadingRow` event will allow us to make changes to the visual appearance of the row.

To create this `datagrid`, you'll need to carry out the following steps:

1. Start a new Silverlight solution called **DatagridDisplayingData** in Visual Studio. We'll start by creating the `Book` class. Add a new class to the Silverlight project in the solution and name this class `Book`. Note that this class uses two enumerations—one for the `Category` and the other for the `Language`. These can be found in the sample code. The following is the code for the `Book` class:

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public DateTime PurchaseDate { get; set; }
    public Category Category { get; set; }
    public string Publisher { get; set; }
    public Languages Language { get; set; }
    public string ImageName { get; set; }
    public bool AlreadyRead { get; set; }
}
```

2. In the code-behind of the generated `MainPage.xaml` file, we need to create a generic list of `Book` instances (`List<Book>`) and load data into this collection. This is shown in the following code:

```
private List<Book> bookCollection;
public MainPage()
{
    InitializeComponent();
    LoadBooks();
}
private void LoadBooks()
{
    bookCollection = new List<Book>();
    Book b1 = new Book();
    b1.Title = "Book AAA";
    b1.Author = "Author AAA";
    b1.Language = Languages.English;
    b1.PageCount = 350;
    b1.Publisher = "Publisher BBB";
    b1.PurchaseDate = new DateTime(2009, 3, 10);
    b1.ImageName = "AAA.png";
    b1.AlreadyRead = true;
    b1.Category = Category.Computing;
    bookCollection.Add(b1);
    ...
}
```

3. Next, we'll add a `DataGrid` to the `MainPage.xaml` file. For now, we won't add any extra properties on the `DataGrid`. It's advisable to add it to the page by dragging it from the toolbox, so that Visual Studio adds the correct references to the required assemblies in the project, as well as adds the namespace mapping in the XAML code. The following line of code shows a default `DataGrid` with its name set to `BookDataGrid`:

```
<sdk:DataGrid x:Name="BookDataGrid"></sdk:DataGrid>
```

4. Currently, no data is bound to the `DataGrid`. To make the `DataGrid` show the book collection, we set the `ItemsSource` property from the code-behind in the constructor. This is shown in the following code:

```
public MainPage()  
{  
    InitializeComponent();  
    LoadBooks();  
    BookDataGrid.ItemsSource = bookCollection;  
}
```

5. Running the code now shows a default `DataGrid` that generates a column for each public property of the `Book` type. This happens because the `AutoGenerateColumns` property is `True` by default.

6. Let's continue by making the `DataGrid` look the way we want it to look. By default, the `DataGrid` is user-editable, so we may want to change this feature. Setting the `IsReadOnly` property to `True` will make it impossible for a user to edit the data in the control. We can lock the display even further by setting both the `CanUserResizeColumns` and the `CanUserReorderColumns` properties to `False`. This will prohibit the user from resizing and reordering the columns inside the `DataGrid`, which are enabled by default. This is shown in the following code:

```
<sdk:DataGrid x:Name="BookDataGrid"  
    AutoGenerateColumns="True"  
    CanUserReorderColumns="False"  
    CanUserResizeColumns="False"  
    IsReadOnly="True">  
</sdk:DataGrid>
```

7. The `DataGrid` also offers quite an impressive list of properties that we can use to change its appearance. By adding the following code, we specify alternating the background colors (the `RowBackground` and `AlternatingRowBackground` properties), column widths (the `ColumnWidth` property), and row heights (the `RowHeight` property). We also specify how the gridlines should be displayed (the `GridLinesVisibility` and `HorizontalGridLinesBrush` properties). Finally, we specify that we also want a row header to be added (the `HeadersVisibility` property):

```
<sdk:DataGrid x:Name="BookDataGrid"
    AutoGenerateColumns="True"
    CanUserReorderColumns="False"
    CanUserResizeColumns="False"
    RowBackground="#999999"
    AlternatingRowBackground="#CCCCCC"
    ColumnWidth="90"
    RowHeight="30"
    GridLinesVisibility="Horizontal"
    HeadersVisibility="All"
    HorizontalGridLinesBrush="Blue">
</sdk:DataGrid>
```

8. We can also get a hook into the loading of the rows. For this, the `LoadingRow` event has to be used. This event is triggered when each row gets loaded. Using this event, we can get access to a row and change its properties based on custom code. In the following code, we are specifying that if the book is a thriller, we want the row to have a red background:

```
private void BookDataGrid_LoadingRow(object sender,
    DataGridRowEventArgs e)
{
    Book loadedBook = e.Row.DataContext as Book;
    if (loadedBook.Category == Category.Thriller)
    {
        e.Row.Background = new SolidColorBrush(Colors.Red);
        //It's a thriller!
        e.Row.Height = 40;
    }
    else
    {
        e.Row.Background = null;
    }
}
```

- 9 To link the event handler with the event from XAML, make sure the `DataGrid` instance in XAML contains the following code:

```
>LoadingRow="BookDataGrid_LoadingRow"
```

The Data Grid

After completing these steps, we have the `DataGridView` that we wanted. It displays the data (including headers), fixes the columns, and makes it impossible for the user to edit the data. Also, the color of the rows and alternating rows is changed, the vertical grid lines are hidden, and a different color is applied to the horizontal grid lines. Using the `LoadingRow` event, we have checked whether the book being added is of the `Thriller` category, and if so, a red color is applied as the background color for the row. The result can be seen in the following image:

Actions									
Title	Author	PageCount	PurchaseDate	Category	Publisher	Language	ImageName	AlreadyRead	
Book AAA	Author AAA	350	3/10/2009 12:00:00 AM	Computing	Publisher BBB	English	AAA.png	<input checked="" type="checkbox"/>	
Book BBB	Author AAA	667	4/11/2009 12:00:00 AM	Thriller	Publisher AAA	Dutch	BBB.png	<input type="checkbox"/>	
Book CCC	Author AAA	289	12/10/2009 12:00:00 AM	Fiction	Publisher AAA	French	CCC.png	<input checked="" type="checkbox"/>	
Book DDD	Author BBB	200	1/20/2009 12:00:00 AM	Thriller	Publisher DDD	German	DDD.png	<input type="checkbox"/>	
Book EEE	Author BBB	403	3/1/2007 12:00:00 AM	Biography	Publisher DDD	German	EEE.png	<input checked="" type="checkbox"/>	
Book FFF	Author AAA	296	9/4/2009 12:00:00 AM	Comics	Publisher AAA	English	FFF.png	<input checked="" type="checkbox"/>	
Book HHH	Author CCC	675	1/31/2007 12:00:00 AM	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>	
Book HHH	Author CCC	675	1/31/2007 12:00:00 AM	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>	
Book III	Author DDD	1300	7/1/2008 12:00:00 AM	Computing	Publisher DDD	French	III.png	<input checked="" type="checkbox"/>	
Book KKK	Author BBB	200	1/2/2009 12:00:00 AM	Thriller	Publisher BBB	French	KKK.png	<input checked="" type="checkbox"/>	
Book KKK	Author BBB	200	1/2/2009 12:00:00 AM	Thriller	Publisher BBB	French	KKK.png	<input checked="" type="checkbox"/>	
Book LLL	Author DDD	400	10/23/2009 12:00:00 AM	Computing	Publisher CCC	English	LLL.png	<input checked="" type="checkbox"/>	

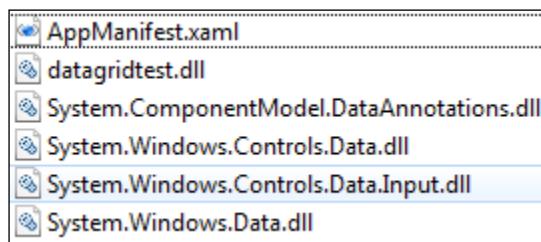
How it works...

The `DataGridView` allows us to display the data easily, while still offering us many customization options to format the control as needed.

The `DataGridView` is defined in the `System.Windows.Controls` namespace, which is located in the `System.Windows.Controls.Data` assembly. By default, this assembly is not referenced while creating a new Silverlight application. Therefore, the following extra references are added while dragging the control from the toolbox for the first time:

- ▶ `System.ComponentModel.DataAnnotations`
- ▶ `System.Windows.Controls.Data`
- ▶ `System.Windows.Controls.Data.Input`
- ▶ `System.Windows.Data`

While compiling the application, the corresponding assemblies are added to the XAP file (as can be seen in the following image, which shows the contents of the XAP file). These assemblies need to be added because while installing the Silverlight plugin, they aren't installed as a part of the CLR. This is done in order to keep the plugin size small. However, when we use them in our application, they are embedded as part of the application. This results in an increase of the download size of the XAP file (around 200 KB of compressed DLLs are added to XAP). In most circumstances, this is not a problem. However, if the file size is an important requirement, then it is essential to keep an eye on this.



Also, Visual Studio will include the following namespace mapping into the XAML file:

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

From then on, we can use the control as shown in the following line of code:

```
<sdk:DataGrid x:Name="BookDataGrid"> </sdk:DataGrid>
```

Once the control is added on the page, we can use it in a data binding scenario. To do so, we can point the `ItemsSource` property to any `IEnumerable` implementation. Each row in the `DataGrid` will correspond to an object in the collection.

When `AutoGenerateColumns` is set to `True` (the default), the `DataGrid` uses a reflection on the type of objects bound to it. For each public property it encounters, it generates a corresponding column. Out of the box, the `DataGrid` includes a text column, a checkbox column, and a template column. For all the types that can't be displayed, it uses the `ToString` method and a text column.

If we want the `DataGrid` to feature automatic synchronization, the collection should implement the `INotifyCollectionChanged` interface. If changes to the objects are to be reflected in the `DataGrid`, then the objects in the collection should themselves implement the `INotifyPropertyChanged` interface.

There's more...

While loading large amounts of data into the `DataGridView`, the performance will still be very good. This is the result of the `DataGridView` implementing UI virtualization, which is enabled by default.

Let's assume that the `DataGridView` is bound to a collection of 1,000,000 items (whether or not this is useful is another question). Loading all of these items into memory would be a time-consuming task as well as a big performance hit. Due to UI virtualization, the control loads only the rows it's currently displaying. (It will actually load a few more to improve the scrolling experience.) While scrolling, a small lag appears when the control is loading the new items. Starting with Silverlight 3, the `ListBox` also features UI virtualization.

See also...

In the recipe *Using custom columns in the DataGridView* of this chapter, we'll look at how we can specify which columns should be included in the `DataGridView`.

Inserting, updating, and deleting data in a `DataGridView`

Applies to Silverlight 3, 4 and 5

The `DataGridView` is an outstanding control to use while working with large amounts of data at the same time. Through its Excel-like interface, not only can we easily view the data, but also add new records or update and delete existing ones.

In this recipe, we'll take a look at how to build a `DataGridView` that supports all of these actions on a collection of items.

Getting ready

This recipe builds on the code that was created in the previous recipe. To follow along with this recipe, you can keep using your code or use the starter solution located in the `Chapter04/Datagrid_Editing_Data_Starter` folder in the code bundle available on the Packt website. The complete solution for this recipe can be found in the `Chapter04/Datagrid_Editing_Data_Completed` folder.

How to do it...

In this recipe, we'll work with the same Book class as in the previous recipe. Through the use of a `DataGridView`, we'll manage an `ObservableCollection<Book>`. We'll make it possible to add, update, and delete the items in the collection through the `DataGridView`. An `ObservableCollection` raises an event when items are added, removed, and so on, and Silverlight will listen for this event. The existing data will be edited by doing inline edits to the rows, which will be pushed back to the underlying collection. We'll allow the user to add or delete an item in the `DataGridView` by clicking on a button. Behind the scenes, an item is added to or removed from the underlying collection. We'll also include a detail panel where the user can view more properties on the selected item in the `DataGridView`.

The following are the steps we need to perform:

1. In the `MainPage.xaml.cs` file, we bind to a generic list of `Book` instances (`List<Book>`). For the `DataGridView` to react to the changes in the bound collection, the collection itself should implement the `INotifyCollectionChanged` interface. Thus, instead of a `List<Book>`, we'll use an `ObservableCollection<Book>` as shown in the following line of code:

```
ObservableCollection<Book> bookCollection =
    new ObservableCollection<Book>();
```

2. Let's first look at deleting the items. We may want to link the hitting of the `Delete` key on the keyboard with the removal of a row in the `DataGridView`. In fact, we're asking to remove the currently selected item from the bound collection. For this, we register for the `KeyDown` event on the `DataGridView` as shown in the following code:

```
<sdk:DataGridView x:Name="BookDataGrid"
    KeyDown="BookDataGrid_KeyDown" ...>
```

3. In the event handler, we'll need to check whether the key was the `Delete` key. Also, the required code for inserting the data—triggered by hitting the `Insert` key—is included. This is shown in the following code:

```
private bool cellEditing = false;
private void BookDataGrid_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Delete && !cellEditing)
    {
        RemoveBook();
    }
    else if (e.Key == Key.Insert && !cellEditing)
    {
        AddEmptyBook();
    }
}
```

4. Note the `!cellEditing` in the previous code. It's a Boolean field that we are using to check whether we are currently editing a value that is in a cell or we simply have a row selected. In order to carry out this check, we should add both the `BeginningEdit` and the `CellEditEnded` events in the `DataGrid` as shown in the following code. These will be triggered when the cell enters or leaves the edit mode respectively.

```
<sdk:DataGrid x:Name="BookDataGrid"
    BeginningEdit="BookDataGrid_BeginningEdit"
    CellEditEnded="BookDataGrid_CellEditEnded" ...>
```

5. In the event handlers, we change the value of the `cellEditing` variable as shown in the following code:

```
private void BookDataGrid_BeginningEdit(object sender,
    DataGridBeginningEditEventArgs e)
{
    cellEditing = true;
}
private void BookDataGrid_CellEditEnded(object sender,
    DataGridCellEditEndedEventArgs e)
{
    cellEditing = false;
}
```

6. Next, we need to write the code either to add an empty `Book` object or to remove an existing one. Here, we're actually working with the `ObservableCollection<Book>`. We're adding items to the collection or removing them from it. If you've used the starter solution for this recipe, two `Button` controls are already included. If you're using your own code from the previous recipe, simply add two `Button` instances to the XAML file. We can add two `Click` event handlers that will trigger adding or removing an item using the following code. Note that while deleting, we are checking whether an item is selected.

```
private void AddButton_Click(object sender, RoutedEventArgs e)
{
    AddEmptyBook();
}
private void DeleteButton_Click(object sender, RoutedEventArgs e)
{
    RemoveBook();
}
private void AddEmptyBook()
```

```
{  
    Book b = new Book();  
    bookCollection.Add(b);  
}  
private void RemoveBook()  
{  
    if (BookDataGrid.SelectedItem != null)  
    {  
        Book deleteBook = BookDataGrid.SelectedItem as Book;  
        bookCollection.Remove(deleteBook);  
    }  
}
```

7. Finally, let's take a look at updating the items. In fact, simply typing in new values for the existing items in the DataGrid will push the updates back to the bound collection. Add a Grid containing the TextBlock controls in order to see this. The entire Grid should be bound to a selected row of the DataGrid. This is done by means of an element data binding. The following code is a part of this code. The remaining code can be found in the completed solution in the code bundle.

```
<Grid DataContext="{Binding ElementName=BookDataGrid,  
    Path=SelectedItem}">  
    <TextBlock Text="Title:"  
        FontWeight="Bold"  
        Grid.Row="1"  
        Grid.Column="0">  
    </TextBlock>  
    <TextBlock Text="{Binding Title}"  
        Grid.Row="1"  
        Grid.Column="1">  
    </TextBlock>  
</Grid>
```

The Data Grid

We now have a fully-working application to manage the data of the Book collection. We have a data-entry application that allows us to perform **CRUD (create, read, update, and delete)** operations on the data using the DataGrid. The final application is shown in the following screenshot:

Book Library

Actions Add book Delete book

	Title	Author	PageCount	PurchaseDate	Category	Publisher	Language	ImageName	AlreadyRead	
	Book AAA	Author AAA	350	3/10/2009 12:00:00 AM	Computing	Publisher BBB	English	AAA.png	<input checked="" type="checkbox"/>	
	Book BBB	Author AAA	667	4/11/2009 12:00:00 AM	Thriller	Publisher AAA	Dutch	BBB.png	<input type="checkbox"/>	
	Book CCC	Author AAA	289	12/10/2009 12:00:00 AM	Fiction	Publisher AAA	French	CCC.png	<input checked="" type="checkbox"/>	
	Book DDD	Author BBB	200	1/20/2009 12:00:00 AM	Thriller	Publisher DDD	German	DDD.png	<input type="checkbox"/>	
▶	Book EEE	Author BBB	403	3/1/2007 12:00:00 AM	Biography	Publisher DDD	German	EEE.png	<input checked="" type="checkbox"/>	
	Book FFF	Author AAA	296	9/4/2009 12:00:00 AM	Comics	Publisher AAA	English	FFF.png	<input checked="" type="checkbox"/>	
	Book HHH	Author CCC	675	1/31/2007 12:00:00 AM	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>	
	Book HHH	Author CCC	675	1/31/2007 12:00:00 AM	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>	
	Book III	Author DDD	1300	7/1/2008 12:00:00 AM	Computing	Publisher DDD	French	III.png	<input checked="" type="checkbox"/>	

Book details

Title: Book EEE

Author: Author BBB

Pagecount: 403

Publisher: Publisher DDD

How it works...

The DataGrid is bound to an `ObservableCollection<Book>`. This way, changes to the collection are reflected in the control immediately because of the automatic synchronization that data binding offers on collections that implement the `INotifyCollectionChanged` interface. If the class (in our case, the `Book` class) itself implements the `INotifyPropertyChanged` interface, then the changes to the individual items are also reflected. Implicitly, a DataGrid implements a `TwoWay` binding. We don't have to specify this anywhere.

To remove an item by hitting the *Delete* key, we first need to check that we're not editing the value of the cell. If we are, then the row shouldn't be deleted. This is done using the `BeginningEdit` and `CellEditEnded` events. The former one is called before the user can edit the value. It can also be used to perform some action on the value in the cell such as formatting. The latter event is called when the focus moves away from the cell.

In the end, managing (inserting, deleting, and so on) the data in the `DataGridView` comes down to managing the items in the collection. We leverage this here. We aren't adding any items to the `DataGridView` itself, but we are either adding items to the bound collection or removing items from the bound collection.

See also...

For more information on the data binding features, take a look at *Chapter 2, An Introduction to Data Binding* and *Chapter 3, Advanced Data Binding*, where we look in much detail at all the features offered by Silverlight in this area.

Sorting and grouping data in a DataGridView

Applies to Silverlight 3, 4 and 5

Sorting the values within a column in a control such as a `DataGridView` is something that we take for granted. Silverlight's implementation has some very strong sorting options working out of the box for us. It allows us to sort by clicking on the header of a column, amongst other things.

Along with sorting, the `DataGridView` enables the grouping of values. Items possessing a particular property (that is, in the same column) and having equal values can be visually grouped within the `DataGridView`.

All of this is possible by using a view on top of the bound collection. In this recipe, we'll look at how we can leverage this view to customize the sorting and grouping of data within the `DataGridView`.

Getting ready

This sample continues with the same code that was created in the previous recipes of this chapter. If you want to follow along with this recipe, you can continue using your code or use the provided start solution located in the `Chapter04/Datagrid_Sorting_And_Grouping_Starter` folder in the code bundle that is available on the Packt website. The finished code for this recipe can be found in the `Chapter04/Datagrid_Sorting_And_Grouping_Completed` folder.

How to do it...

We'll be using the familiar list of Book items again in this recipe. This list, which is implemented as an `ObservableCollection<Book>`, will not be directly bound to the `DataGrid` in this case. Instead, we'll use a `PagedCollectionView` that acts as a view on top of the collection. We'll change the way the `DataGrid` is sorted by default as well as introduce grouping within the control. The following are the steps to achieve all of this:

1. Instead of using the `AutoGenerateColumns` feature, we'll define the columns that we want to see manually. We'll make use of several `DataGridTextColumn`s, a `DataGridCheckBoxColumn`, and a `DataGridTemplateColumn`. The following is the code for the `DataGrid`:

```
<sdk:DataGrid x:Name="CopyBookDataGrid"
               AutoGenerateColumns="False" ... >
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn x:Name="CopyTitleColumn"
                           Binding="{Binding Title}"
                           Header="Title">
    </sdk:DataGridTextColumn>
    <sdk:DataGridTextColumn x:Name="CopyAuthorColumn"
                           Binding="{Binding Author}"
                           Header="Author">
    </sdk:DataGridTextColumn>
    <sdk:DataGridTextColumn x:Name="CopyPublisherColumn"
                           Binding="{Binding Publisher}"
                           Header="Publisher">
    </sdk:DataGridTextColumn>
    <sdk:DataGridTextColumn x:Name="CopyLanguageColumn"
                           Binding="{Binding Language}"
                           Header="Language">
    </data:DataGridTextColumn>
    <data:DataGridTextColumn x:Name="CopyCategoryColumn"
                           Binding="{Binding Category}"
                           Header="Category">
    </sdk:DataGridTextColumn>
    <sdk:DataGridCheckBoxColumn x:Name="CopyAlreadyReadColumn"
                               Binding="{Binding AlreadyRead,
                               Mode=TwoWay}"
                               Header="Already read">
    </sdk:DataGridCheckBoxColumn>
    <sdk:DataGridTemplateColumn Header="Purchase date"
                               x:Name="CopyPurchaseDateColumn">
      <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <controls:DatePicker SelectedDate="{Binding
          PurchaseDate}">

```

```

        </controls:DatePicker>
    </DataTemplate>
</sdk:DataGridTemplateColumn.CellTemplate>
</sdk:DataGridTemplateColumn>
</sdk:DataGrid.Columns>
</sdk:DataGrid>
```

In this code, we are using the `DatePicker` control. It's prefixed with the `controls` prefix, which maps to the `System.Windows.Controls` namespace in the `System.Windows.Controls` assembly. Therefore, if not yet added in your XAML, add the following `xmlns` namespace mapping at the `UserControl` level:

```
<UserControl
    xmlns:controls="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls"
    ... />
```

2. In order to implement both sorting and grouping, we'll use the `PagedCollectionView`. It offers us a view on top of our data and allows the data to be sorted, grouped, and filtered without changing the underlying collection. The `PagedCollectionView` is instantiated using the following code. We pass in the collection (in this case, the `bookCollection`) on which we want to put the view.

```
private PagedCollectionView view;

public MainPage()
{
    InitializeComponent();

    LoadBooks();
    view = new PagedCollectionView(bookCollection);
}
```

3. In order to change the manner of sorting from the code, we need to add a new `SortDescription` to the `SortDescriptions` collection of the view. In the following code, we are specifying that we want the sorting to occur on the `Title` property of the books in a descending order:

```
view.SortDescriptions.Add(new SortDescription("Title",
    ListSortDirection.Descending));
```

4. If we want our data to appear in groups, we can make it so by adding a new `PropertyGroupDescription` to the `GroupDescriptions` collection of the view. In this case, we want the grouping to be based on the value of the `Language` property. This is shown in the following code:

```
view.GroupDescriptions.Add(new
    PropertyGroupDescription("Language"));
```

The Data Grid

5. The `DataGridView` will not bind to the collection, but to the view. We specify this by setting the `ItemsSource` property to the instance of `PagedCollectionView`. The following code should be placed in the constructor as well:

```
public MainPage()
{
    InitializeComponent();
    LoadBooks();
    view = new PagedCollectionView(bookCollection);
    view.SortDescriptions.Add(new SortDescription("Title",
        ListSortDirection.Descending));
    view.GroupDescriptions.Add(new
        PropertyGroupDescription("Language"));
    BookDataGrid.ItemsSource = view;
}
```

We have now created a `DataGridView` that allows the user to sort the values in a column as well as group the values based on a value in the column. The resulting `DataGridView` is shown in the following image:

Book Library							
Actions		Group by		Language	Expand all		
	Title	Author	Publisher	Language	Category	Already read	Purchase date
▲ Dutch (4 items)							
	Book PPP	Author CCC	Publisher AAA	Dutch	Computing	<input type="checkbox"/>	9/09/2009 <input type="button" value="15"/>
	Book HHH	Author CCC	Publisher CCC	Dutch	Fiction	<input type="checkbox"/>	31/01/2007 <input type="button" value="15"/>
	Book HHH	Author CCC	Publisher CCC	Dutch	Fiction	<input type="checkbox"/>	31/01/2007 <input type="button" value="15"/>
	Book BBB	Author AAA	Publisher AAA	Dutch	Thriller	<input type="checkbox"/>	11/04/2009 <input type="button" value="15"/>
▲ English (5 items)							
	Book OOO	Author BBB	Publisher DDD	English	Thriller	<input checked="" type="checkbox"/>	2/02/2009 <input type="button" value="15"/>
	Book NNN	Author DDD	Publisher CCC	English	Fiction	<input type="checkbox"/>	24/12/2008 <input type="button" value="15"/>
	Book LLL	Author DDD	Publisher CCC	English	Computing	<input checked="" type="checkbox"/>	23/10/2009 <input type="button" value="15"/>
	Book FFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	4/09/2009 <input type="button" value="15"/>
▶	Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	10/03/2009 <input type="button" value="15"/>
▲ German (3 items)							
	Book MMM	Author AAA	Publisher DDD	German	Fiction	<input checked="" type="checkbox"/>	5/04/2009 <input type="button" value="15"/>
	Book EEE	Author BBB	Publisher DDD	German	Biography	<input checked="" type="checkbox"/>	1/03/2007 <input type="button" value="15"/>
	Book DDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	20/01/2009 <input type="button" value="15"/>
▲ French (4 items)							
	Book KKK	Author BBB	Publisher BBB	French	Thriller	<input checked="" type="checkbox"/>	2/01/2009 <input type="button" value="15"/>
	Book KKK	Author BBB	Publisher BBB	French	Thriller	<input checked="" type="checkbox"/>	2/01/2009 <input type="button" value="15"/>
	Book III	Author DDD	Publisher DDD	French	Computing	<input checked="" type="checkbox"/>	1/07/2008 <input type="button" value="15"/>
	Book CCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	10/12/2009 <input type="button" value="15"/>

How it works...

The actions such as sorting, grouping, filtering, and so on don't work on an actual collection of data. They are applied on a view that sits on top of the collection (either a `List<T>` or an `ObservableCollection<T>`). This way, the original data is not changed. Due to this, we can show the same collection more than once in a different format on the same screen. Different views are applied on the same source data (for example, sorted in one `DataGrid` by `Title` and in another one by `Author`). This view is implemented through the `PagedCollectionView` class.

To change the sorting, we can add a new `SortDescription` to the `SortDescriptions` collection that the view encapsulates. Note that `SortDescriptions` is a collection in which we can add more than one sort field. The second `SortDescription` value will be used only when equal values are encountered for the first `SortDescription` value.

Grouping (using the `PropertyGroupDescription`) allows us to split the grid into different levels. Each section will contain items that have the same value for a particular property. Similar to sorting, we can add more than one `PropertyGroupDescription`, which results in nested groups.

There's more...

From the code, we can control all the groups to expand or collapse. The following code shows us how to do so:

```
private void CollapseGroupsButton_Click(object sender,
    RoutedEventArgs e)
{
    foreach (CollectionViewGroup group in view.Groups)
    {
        BookDataGrid.CollapseRowGroup(group, true);
    }
}
private void ExpandGroupsButton_Click(object sender,
    RoutedEventArgs e)
{
    foreach (CollectionViewGroup group in view.Groups)
    {
        BookDataGrid.ExpandRowGroup(group, true);
    }
}
```

Sorting a template column

If we want to sort a template column, we have to specify which value needs to be taken into account for the sorting to be executed. Otherwise, Silverlight has no clue which field it should take.

This is done by setting the `SortMemberPath` property as shown in the following code:

```
<sdk:DataGridTemplateColumn x:Name="PurchaseDateColumn"
    SortMemberPath="PurchaseDate">
```

We'll look at the `DataGridTemplateColumn` in more detail in the recipe *Using custom columns in the DataGrid* of this chapter.

See also

In the next recipe, we'll use the `PagedCollectionView` once more.

Filtering and paging data in a DataGrid

Applies to Silverlight 3, 4, and 5

Along with offering us support for the sorting and filtering of data, the `PagedCollectionView` has more tricks up its sleeve. It is also the enabler for filtering rows in a `DataGrid` and in combination with the `DataPager` control (a control added with Silverlight 3); it allows us to spread the data over several pages within the `DataGrid`.

In this recipe, we'll look at how we can filter based on a value specified by the user and we'll page the results based on the number of returned results, if needed.

Getting ready

This recipe builds on the code that was created in the previous recipes. You can continue using your code to follow this recipe. Alternatively, you can use the start solution located in the `Chapter04/Datagrid_Filtering_And_Paging_Starter` folder in the code bundle that is available on the Packt website. The complete code for this recipe can be found in the `Chapter04/Datagrid_Filtering_And_Paging_Completed` folder.

How to do it...

For this recipe, we'll again work with the `Book` class for which an `ObservableCollection<Book>` is created. This collection is then used as input for the `PagedCollectionView`, which offers a view on the collection. We'll add a search functionality on the collection of books using a filter and a paging functionality using the `DataPager`. Following are the steps to get this up and running:

1. We'll add some XAML controls to the filter. These include a `TextBlock` for indicating the purpose of a field, a `TextBox` in which the user can enter a value, and a `Button`. This is shown in the following code:

```
<TextBlock x:Name="FilterTextBlock"
           Text="Search book titles"
           Margin="3"
           VerticalAlignment="Center"
           Foreground="White">
</TextBlock>
<TextBox x:Name="FilterTextBox"
         Width="200"
         VerticalAlignment="Center"
         HorizontalAlignment="Center"
         Margin="3">
</TextBox>
<Button x:Name="FilterButton"
         Content="Search"
         Margin="3"
         HorizontalAlignment="Center"
         VerticalAlignment="Center"
         Click="FilterButton_Click">
</Button>
```

2. The `DataGridView` is bound to the `PagedCollectionView`, which is a view over the items of a used collection. This is shown in the following code:

```
PagedCollectionView view = new
    PagedCollectionView(bookCollection);
BookDataGridView.ItemsSource = view;
```

3. Upon clicking on the `Button`, we need to search the collection. Searching means looping over the collection and checking whether or not each item satisfies the query. This sounds like a perfect job for a predicate and that's exactly how it's implemented. In the predicate, we'll check whether a book title contains the value entered by the user. This is shown in the following code:

```
private void FilterButton_Click(object sender, RoutedEventArgs e)
{
    view.Filter = null;
    view.Filter = new Predicate<object>(Search);
}
private bool Search(object b)
{
    Book book = b as Book;
    bool foundSearchHit = false;
    if (book != null)
    {
        if (book.Title.Contains(FilterTextBox.Text))
```

```
        foundSearchHit = true;
    }
    return foundSearchHit;
}
```

4. Finally, let's add paging support to the `DataGridView`. Paging is the job of the `DataPager`. This control adds paging support to controls such as the `ListBox` and the `DataGridView`. We simply add a `DataPager` on the XAML page and specify the `PageSize` property as 5, shown in the following code:

```
<sdk:DataPager x:Name="BookPager"
    PageSize="5"
    DisplayMode="PreviousNextNumeric">
</sdk:DataPager>
```

5. To make the `DataPager` control display the pages, we need to set its `Source` to the same `PagedCollectionView` as the `DataGridView`. The following code shows how to do this:

```
public MainPage()
{
    InitializeComponent();
    LoadBooks();
    view = new PagedCollectionView(bookCollection);
    BookDataGridView.ItemsSource = view;
    BookPager.Source = view;
}
```

We have now implemented a filter on the `DataGridView` using the `PagedCollectionView`. A user can search for a value and filtering of the in-memory data will be done. The resulting `DataGridView` is paged using a `DataPager`. The following image shows the result:

The screenshot shows a Windows application window titled "Book Library". At the top, there is a toolbar with buttons for "Actions", "Search book titles", and a search input field containing "AAA" with a "Search" button next to it. Below the toolbar is a `DataPager` control with page numbers 1 and 2. The main area contains a `DataGridView` with the following data:

Title	Author	Publisher	Language	Category	Already read	Date Read	Page Count
Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	10/03/2009	<input type="button" value="15"/>
Book AAACCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	10/12/2009	<input type="button" value="15"/>
Book AAADDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	20/01/2009	<input type="button" value="15"/>
Book AAAEEE	Author BBB	Publisher DDD	German	Biography	<input checked="" type="checkbox"/>	1/03/2007	<input type="button" value="15"/>
Book AAAFFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	4/09/2009	<input type="button" value="15"/>

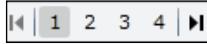
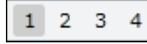
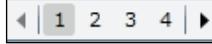
How it works...

Just like sorting and grouping, **filtering** is not done on the collection itself, but it's done using a view on top of the collection. This way, the original collection remains intact and can be used on the same screen with different filter values more than once.

Filtering is done using a predicate. Silverlight will loop over all the items of the view and execute the method (in this case the `Search` method) being passed in as the parameter for each item. This method contains the logic that will check whether or not the item should be included in the result set.

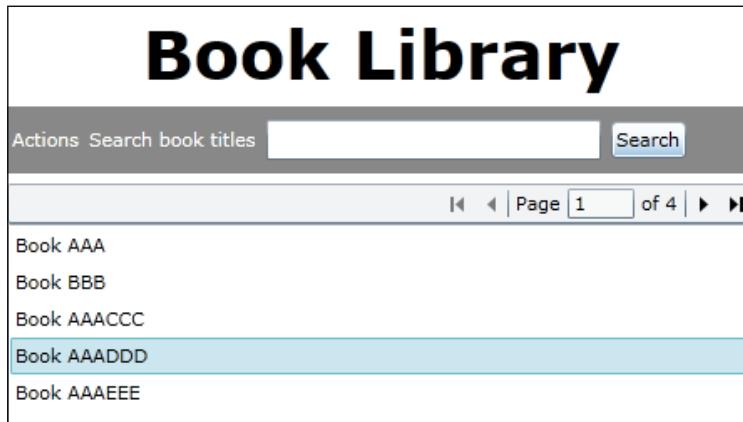
Paging is not directly done by the `DataGridView`. A second control, the `DataPager`, comes to the rescue. This control does not have a direct link to the `DataGridView`. Both the `DataGridView.ItemsSource` and the `DataPager.Source` properties point to the same instance of the `PagedCollectionView`. This way, the `DataPager` knows on which control it should add the paging functionality.

The `DataPager` control has a `DisplayStyle` property that requires a value of the `PagerDisplayStyle` enumeration. The following table shows the different options in action:

PagerDisplayStyle value	Visualization
FirstLastNumeric	
FirstLastPreviousNext	
FirstLastPreviousNextNumeric	
Numeric	
PreviousNext	
PreviousNextNumeric	

There's more...

The DataPager is not exclusively tied to the DataGrid; it can also be used with the ListBox. The following image shows a DataPager working together with a ListBox:



There is no difference code-wise. Both the ListBox and the DataPager refer to the same PagedCollectionView instance.

See also...

In the previous recipe, *Sorting and grouping the data in a DataGrid*, we used the PagedCollectionView for sorting and grouping the data in a DataGrid. In the next recipe, we'll explain more about defining the columns that we want to appear in the DataGrid.

Using custom columns in the DataGrid

Applies to Silverlight 3, 4 and 5

By default, the DataGrid will generate columns for us based on the type of objects that we pass to the control. We looked at this in *Displaying the data in a customized DataGrid*. However, we'll want more control over what is being displayed most of the time. We'll want to make decisions such as which columns should be shown, in what order and so on. In addition to that, we may want to allow the user to select a value from a ComboBox for a particular column or entirely reformat a value.

In this recipe, we'll take full control over what will be displayed by the DataGrid by creating a number of columns ourselves.

Getting ready

To follow along with this recipe, you can continue using the code that was created in the previous recipes of this chapter. You can also use the start solution located in the Chapter04/Datagrid_Custom_Columns_Starter folder in the code bundle that is available on the Packt website. The completed solution for this recipe can be found in the Chapter04/Datagrid_Custom_Columns_Completed folder.

How to do it...

There are three types of columns from which we can choose—the `DataGridViewTextColumn`, the `DataGridViewCheckBoxColumn`, and the `DataGridViewTemplateColumn`. We can either declare columns from XAML by adding them to the `Columns` collection of the `DataGridView` or add them from the code-behind. We'll again work with the `Book` class. We'll create an `ObservableCollection<Book>` in the code-behind and bind this to the `DataGridView`. We'll create a few custom columns in the following list of steps:

1. The `AutoGenerateColumns` property defaults to `True`. Therefore, in the declaration of the `DataGridView`, we set the property to `False`. The custom-created columns will be added to the `Columns` collection. This is shown in the following code:

```
<sdk:DataGrid x:Name="BookDataGrid"
              AutoGenerateColumns="False">
    <sdk:DataGrid.Columns>
    </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

2. In order to display plain textual values such as the `Title`, the `Author`, and the `Publisher`, we can use the `DataGridViewTextColumn` as shown in the following code. We need to specify the `Binding` for each column. Note that we now need to set the `Mode` property to `TwoWay`. If we omit this, the value will not be pushed back to the underlying collection.

```
<sdk:DataGridViewTextColumn x:Name="TitleColumn"
                            Binding="{Binding Title}"
                            Header="Title">
</sdk:DataGridViewTextColumn>
<sdk:DataGridViewTextColumn x:Name="AuthorColumn"
                            Binding="{Binding Author}"
                            Header="Author">
</sdk:DataGridViewTextColumn>
<sdk:DataGridViewTextColumn x:Name="PublisherColumn"
                            Binding="{Binding Publisher,
                            Mode=TwoWay}"
                            Header="Publisher">
</sdk:DataGridViewTextColumn>
```

3. The `AlreadyRead` property of our `Book` class is of the `bool` type. We can bind such a value to a `CheckBoxColumn` as shown in the following code:

```
<sdk:DataGridCheckBoxColumn x:Name="AlreadyReadColumn"
    Binding="{Binding AlreadyRead,
    Mode=TwoWay}"
    Header="Already read">
</sdk:DataGridCheckBoxColumn>
```

4. The `TemplateColumn` is the most powerful column type. Using this type, we can specify the template for the column manually. The following is the code for the `ImageName` property. We specify a converter, which is used to convert the `ImageName` property of type string into a `BitmapImage`. This `BitmapImage` can then be used for setting the `Source` property of the `Image` control. The code for the converter can be found in the code bundle that is available on the Packt website.

```
<sdk:DataGridTemplateColumn x:Name="ImageColumn">
    <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <Image Source="{Binding ImageName,
            Converter={StaticResource localImageConverter}}"
            Margin="2">
            </Image>
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
</sdk:DataGridTemplateColumn>
```

5. A `CellTemplate` was defined in the previous template. However, we can also define a `CellEditingTemplate`. The cell will switch to the editing template when the user starts editing inside the cell. For the `Language` property in edit mode, we want to offer the user a `ComboBox` containing the available languages. First, we need to make it possible to retrieve the different languages. We can do so by creating a helper class called `LanguageHelper`, which defines a property. The return value of this property is a list of `Language` instances. This is shown in the following code:

```
public class LanguageHelper
{
    public List<string> LanguageList
    {
        get
        {
            List<string> languages = new List<string>();
            Type languageType = typeof(Languages);
            var fields = from c in languageType.GetFields()
                where c.IsLiteral
                select c;

            foreach (var f in fields)
```

```
        {
            var value = f.GetValue(languageType);
            languages.Add(value.ToString());
        }
        return languages;
    }
}
```

6. We can instantiate this class in `MainPage.xaml` as shown in the following code:

```
<UserControl.Resources>
    <local:LanguageHelper x:Key="localLanguageHelper">
        </local:LanguageHelper>
</UserControl.Resources>
```

7. We can now use this instance to fill the `ComboBox`. The following is the code for the `Language` column. The normal, non-editing template shows a `TextBlock` and the editing template shows a `ComboBox`. The `ItemsSource` property defines the data binding between the `ComboBox` and the `LanguageList` property on the instance of the `LanguageHelper` class:

```
<sdk:DataGridTemplateColumn x:Name="LanguageColumn"
    Header="Language">
    <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Language}"
                VerticalAlignment="Center">
            </TextBlock>
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
    <sdk:DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
            <ComboBox VerticalAlignment="Center"
                SelectedItem="{Binding Language,
                    Converter={StaticResource localEnumConverter},
                    Mode=TwoWay}"
                ItemsSource="{Binding LanguageList,
                    Source={StaticResource localLanguageHelper}}" >
            </ComboBox>
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellEditingTemplate>
</sdk:DataGridTemplateColumn>
```

The Data Grid

Not all columns are shown here, but they are all similar to the previous samples. The completed sample code contains the remaining ones. All the columns have been added to the DataGrid as shown in the following image:

Book Library									
Actions									
Title	Author	Publisher	Language	Category	Already read	Date	Image		
Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	3/10/2009 12:00:00 AM			
Book BBB	Author AAA	Publisher AAA	Dutch	Thriller	<input type="checkbox"/>	4/11/2009 12:00:00 AM			
Book CCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	12/10/2009 12:00:00 AM			
▶ Book DDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	1/20/2009 12:00:00 AM			
Book EEE	Author BBB	Publisher DDD	German	Fiction Comics Computing Biography	<input checked="" type="checkbox"/>	3/1/2007 12:00:00 AM			
Book FFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	9/4/2009 12:00:00 AM			
Book HHH	Author CCC	Publisher CCC	Dutch	Fiction	<input type="checkbox"/>	1/31/2007 12:00:00 AM			

How it works...

In most cases, we will not use the auto-generate function of the DataGrid. We can specify the columns ourselves by adding them to the Columns collection. Three types are available, of which the DataGridTemplateColumn is the most powerful.

If we need to display plain text, then we can use the DataGridTextColumn. However, we only have limited control over the formatting of the text. For example, we can change the ForeGround, the FontSize, and the FontWeight properties. However, if we want the text to wrap, we need to use the ElementStyle property as shown in the following code:

```
<sdk:DataGridTextColumn x:Name="PublisherColumn"  
    Binding="{Binding Publisher, Mode=TwoWay}"  
    Header="Publisher">  
    <sdk:DataGridTextColumn.ElementStyle>  
        <Style TargetType="TextBlock">  
            <Setter Property="TextWrapping"  
                Value="Wrap">  
            </Setter>
```

```
</Style>
</sdk:DataGridTextColumn.ElementStyle>
</sdk:DataGridTextColumn>
```

While displaying a boolean property, we can use a `DataGridCheckboxColumn`, which will render a checkbox per item.

As mentioned before, the real power lies in the `DataGridTemplateColumn` because we can specify how a column will render its contents. We specify a `DataTemplate` containing the data binding statements for the `CellTemplate` property. In this template, we can use whichever control we want (for example, a `DateTimePicker`, an `Image`, or a `ComboBox`).

Each column can have a `CellTemplate` as well as a `CellEditingTemplate`. When both are specified, the column renders the editing template when the user starts editing its content.

In this editing template, we can offer the user a way to make a selection from several options. We have allowed this using a `ComboBox`. However, we need some way to bind the list of possible options to this `ComboBox`. To do so, we can create a helper class that exposes a property that returns a `List<T>`. We can then instantiate this helper class in XAML and perform a data binding with this instance as the source.

There's more...

Starting with Silverlight 4, more sizing options were added for the columns of a `GridView`. Silverlight 2 and Silverlight 3 offered us two options. Under these options, we either needed to specify a width for a column, or else had to leave this task for Silverlight. In the latter case, Silverlight would basically do an auto-sizing (sizing a column according to its contents).

With Silverlight 4, three new options were added, bringing the total to five options to size the columns. No changes were made in this area with the release of Silverlight 5. The following table shows an overview of these size options:

Size option	Function
Auto	Sized to content and header
Pixel (Fixed)	Fixed width in pixels
SizeToCells	Sized to fit content of cells
SizeToHeader	Sized to fit header
Star	Size is a weighted proportion of the available space

The Data Grid

The most interesting option is the `star` option, which works similarly to the star in a regular Grid. Using this option, we can now, for example, specify a cell to either take all of the remaining space or become twice as wide as another cell. The following image shows how the `TitleColumn` is set to take all the remaining space, the `PurchaseDateColumn` and the `ImageColumn` are set to a size according to their cells, and the `AlreadyReadColumn` is set to a size according to its header.

Book Library							
Actions							
Title	Author	Publisher	Language	Category	Already read	Purchase Date	Image
Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	3/10/2009 12:00:00 AM	
Book BBB	Author AAA	Publisher AAA	Dutch	Thriller	<input type="checkbox"/>	4/11/2009 12:00:00 AM	
Book CCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	12/10/2009 12:00:00 AM	
Book DDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	1/20/2009 12:00:00 AM	
Book EEE	Author BBB	Publisher DDD	German	Biography	<input checked="" type="checkbox"/>	3/1/2007 12:00:00 AM	
Book FFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	9/4/2009 12:00:00 AM	

The following code shows how the cells are sized using these sizing options (only the relevant part of the code is posted here):

```
<sdk:DataGrid>
    <sdk:DataGrid.Columns>
        <sdk:DataGridTextColumn x:Name="TitleColumn"
            Width="*">
        </sdk:DataGridTextColumn>
        <sdk:DataGridTextColumn x:Name="AuthorColumn"
            Width="100">
        </sdk:DataGridTextColumn>
        <sdk:DataGridTextColumn x:Name="PublisherColumn"
            Width="150">
        </sdk:DataGridTextColumn>
        <sdk:DataGridTemplateColumn x:Name="LanguageColumn"
            Width="100">
        </sdk:DataGridTemplateColumn>
        <sdk:DataGridTemplateColumn x:Name="CategoryColumn"
            Width="100">
        </sdk:DataGridTemplateColumn>
        <sdk:DataGridCheckBoxColumn x:Name="AlreadyReadColumn"
            Width="SizeToHeader">
        </sdk:DataGridTemplateColumn>
```

```

<sdk:DataGridCheckBoxColumn x:Name="PurchaseDateColumn"
    Width="SizeToCells">
</sdk:DataGridTemplateColumn>
<sdk:DataGridTemplateColumn x:Name="ImageColumn"
    Width="SizeToCells">
</sdk:DataGridTemplateColumn>
</sdk:DataGrid.Columns>
</sdk:DataGrid>

```

Implementing master-detail in the DataGrid

Applies to Silverlight 3, 4 and 5

In order to save screen space, not creating too many columns in a DataGrid may be a good idea. A better solution in this case is to create a **master-detail implementation**. The master, being the original row in the DataGrid, would then contain a few columns only. When clicking on any row, the details of that row are shown. In the Silverlight DataGrid, this is possible due to the RowDetailsTemplate.

Getting ready

To follow along with this recipe, you can continue using the code that was created in the previous recipes. Alternatively, you can use the starter solution located in the Chapter04/Datagrid_Master_Detail_Starter folder in the code bundle that is available on the Packt website. The finished solution for this recipe can be found in the Chapter04/Datagrid_Master_Detail_Completed folder.

How to do it...

For this recipe, we'll again use an `ObservableCollection<Book>`, which is bound to a `DataGrid`. However, we'll display only the `Title` and the `Author` in the default view. When clicking on an item, the details would be shown using a `RowDetailsTemplate`. The following are the steps we need to follow in order to implement this:

1. We want the `DataGrid` to contain only two columns. One of the columns is needed for the `Title` property and the other one for the `Author` property. In the following code, both of these columns are declared as a `DataGridTextColumn` and they contain a `Binding` to the respective properties of the `Book` class:

```

<sdk:DataGrid x:Name="BookDataGrid"
    AutoGenerateColumns="False">
<sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn x:Name="TitleColumn"
        Binding="{Binding Title}"
        Header="Title">

```

```
</sdk:DataGridTextColumn>
<sdk:DataGridTextColumn x:Name="AuthorColumn"
    Binding="{Binding Author}"
    Header="Author">
</sdk:DataGridTextColumn>
</sdk:DataGrid.Columns>
</sdk:DataGrid>
```

2. A detail template is defined on the `DataGridView` itself as shown in the following code:

```
<sdk:DataGrid>
    <sdk:DataGrid.RowDetailsTemplate>
        </sdk:DataGrid.RowDetailsTemplate>
    </sdk:DataGrid>
```

3. Similar to the `CellTemplate`, a `RowDetailsTemplate` is a `DataTemplate` that we can define ourselves. The following code defines a `DataTemplate` containing a `Border`. Inside this `Border`, a `Grid` is nested containing an `Image` control, several `TextBlock` controls, and a `DatePicker`. All of these are data bound to display the value of the selected Book.

```
<DataTemplate>
    <Border Background="AntiqueWhite"
        BorderThickness="2"
        BorderBrush="Blue"
        CornerRadius="5">
        <Grid>
            <Grid.RowDefinitions>
                ...
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                ...
            </Grid.ColumnDefinitions>
            <Image Grid.Row="0"
                Grid.Column="0"
                Grid.RowSpan="2"
                Source="{Binding ImageName,
                    Converter={StaticResource localImageConverter}}"
                Margin="2">
            </Image>
            <StackPanel Grid.Row="0"
                Grid.Column="1"
                Orientation="Horizontal">
                <TextBlock Text="Publisher:"
                    FontWeight="Bold"
                    HorizontalAlignment="Left">
                </TextBlock>
                <TextBlock Text="{Binding Publisher}">
            </StackPanel>
        </Grid>
    </Border>
</DataTemplate>
```

```
        HorizontalAlignment="Left"
        Margin="1">
    </TextBlock>
</StackPanel>
<StackPanel Grid.Row="1"
            Grid.Column="1"
            Orientation="Horizontal">
    <TextBlock Text="Language:"
              FontWeight="Bold"
              HorizontalAlignment="Left">
    </TextBlock>
    <TextBlock Text="{Binding Language}"
              HorizontalAlignment="Left"
              Margin="1">
    </TextBlock>
</StackPanel>
<StackPanel Grid.Row="0"
            Grid.Column="2"
            Orientation="Horizontal">
    <TextBlock Text="Category:"
              FontWeight="Bold"
              HorizontalAlignment="Left">
    </TextBlock>
    <TextBlock Text="{Binding Category}"
              HorizontalAlignment="Left"
              Margin="1">
    </TextBlock>
</StackPanel>
<StackPanel Grid.Row="1"
            Grid.Column="2"
            Orientation="Horizontal">
    <TextBlock Text="Purchase date:"
              FontWeight="Bold"
              HorizontalAlignment="Left">
    </TextBlock>
    <controls:DatePicker SelectedDate="{Binding PurchaseDate}"
                           VerticalAlignment="Top"
                           Margin="1">
    </controls:DatePicker>
</StackPanel>
<StackPanel Grid.Row="0"
            Grid.Column="3"
            Orientation="Horizontal">
    <TextBlock Text="Already read:"
              FontWeight="Bold"
              HorizontalAlignment="Left">
    </TextBlock>
```

The Data Grid

```
<CheckBox IsChecked="{Binding AlreadyRead}">
</CheckBox>
</StackPanel>
</Grid>
</Border>
</DataTemplate>
```

We have now created a master-detail scenario. This can be seen in the following image:

The screenshot shows a Windows application window titled "Book Library". At the top left, there are buttons for "Actions" and "Expand all details". The main area contains a DataGrid displaying a list of books. The first book in the list, "Book BBB", is selected, and its details are shown in a RowDetailsTemplate. The template includes a thumbnail image of the book cover, publisher information ("Publisher: Publisher AAA"), category ("Category: Thriller"), purchase date ("Purchase date: 11/04/2009 15"), and a checkbox labeled "Already read: ". The DataGrid has 12 rows, each containing a book title and author name.

Title	Author
Book AAA	Author AAA
Book BBB	Author AAA
Book CCC	Author AAA
Book DDD	Author BBB
Book EEE	Author BBB
Book FFF	Author AAA
Book HHH	Author CCC
Book HHH	Author CCC
Book III	Author DDD
Book KKK	Author BBB
Book KKK	Author BBB

How it works...

For an easy way of implementing a master-detail scenario, the `RowDetailsTemplate` of the `DataGrid` is a perfect fit. It allows the user to view more details of a record when clicking on it.

The template is defined as a `DataTemplate` on the `RowDetailsTemplate` of the `DataGrid` control. Inside this template—just like other implementations of the `DataTemplate`—we can place whatever controls we want. We can use data binding to get the values inside the controls. Each detail template gets the object to which the selected row is bound as the input for this data binding. Inside the data template, the selected row serves as a data source for the data binding expressions within the template.

There's more...

What if we want to add a `Button` in the template and based on the selected item, want to perform a custom action such as navigating to an edit screen where we can edit the selected item?

This can be solved by binding the `Tag` property of the `Button` as shown in the following code:

```
<Button x:Name="SelectButton"
        Content="Select"
        Click="SelectButton_Click"
        Tag="{Binding Title}">
</Button>
```

In the `Click` event handler, we can cast the sender to a `Button` and get access to the value of a `Tag`. In the following code, we bound the `Title`:

```
private void SelectButton_Click(object sender, RoutedEventArgs e)
{
    Button templateButton = sender as Button;
    if (templateButton.Tag != null)
    {
        //do something
    }
}
```

Validating the DataGrid

Applies to Silverlight 4 and 5

Validation of your data is a requirement for almost every application in order to make sure that no invalid input is possible. If you're using a `DataGrid`, then you can easily implement validation by using **data annotations** on your classes or properties. This control picks up these validation rules automatically and even provides visual feedback. In this recipe, you'll learn how to get your `DataGrid` to implement this kind of validation.

Getting ready

You can find a starter solution for this recipe located in the `Chapter04\DataGrid_Validation_Starter` folder in the code bundle that is available on the Packt website. The complete solution for this recipe can be found in the `Chapter04\DataGrid_Validation_Completed` folder.

How to do it...

If you're starting from a blank solution, you'll need to create a `Person` class having an `ID`, `FirstName`, `LastName`, and `DateOfBirth` properties. The `MainPage` should contain an `ObservableCollection` of `Person`.

We're going to add a `DataGridView` to this project and we'll make sure that it reacts to the validation attributes that we'll add to the `Person` class. To achieve this, carry out the following steps:

1. Open `MainPage.xaml` and add a `DataGridView` to this control. Your `LayoutRoot` grid looks as shown in the following code:

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="40" /></RowDefinition>
        <RowDefinition /></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Text="Working with the DataGridView"
        Margin="10"
        FontSize="14" />
    <data:DataGridView x:Name="myDataGridView"
        Grid.Row="1"
        Width="400"
        Height="300"
        Margin="10"
        HorizontalAlignment="Left"
        VerticalAlignment="Top">
    </data:DataGridView>
</Grid>
```

Add the following namespace import to `MainPage.xaml` to make sure that the `DataGridView` can be used:

```
xmlns:dataclrnamespace:="System.Windows.Controls,
    assembly=System.Windows.Controls.Data"
```

2. Make sure your project has a reference to the `System.ComponentModel.DataAnnotations` assembly.
3. Open the `Person` class and add the following attributes to the `FirstName` property of this class:

```
[StringLength(30, MinimumLength=3,
    ErrorMessage="First name should have between 3 and 30
    characters")]
[Required(ErrorMessage="First name is required")]
public string FirstName { get; set; }
```

4. We can now build and run our solution. When you enter invalid data into the FirstName field, you notice that you get visual feedback for these validation errors and you aren't able to retain your changes unless they are valid. This can be seen in the following image:

ID	FirstName	LastName
1	Kevin dummy characters to make this field too long	Dockx
2	Gill	Cleeren
3	Judy	Garland

0 1 Error

First name should have between 3 and 30 characters

How it works...

This recipe starts by adding a `DataGrid` and a corresponding namespace import to `MainPage.xaml`. As this is done, we can see how it reacts to data annotations.

In the `Person` class, we've added data annotations to the `FirstName` property—the `RequiredAttribute` and the `StringLengthAttribute`. These data annotations tell that the `FirstName` is required and should have between 3 and 30 characters to any control that can interpret them. We've also added a custom error message by filling out the `ErrorMessage` `NamedParameter`.

As a `DataGrid` is automatically able to look for these validation rules, it will show the validation errors if validation fails. This feature comes out of the box with a `DataGrid`, without us having to do any work. Therefore, you can easily enable validation by just using data annotations.

By using the named parameters in the constructors of your attributes, you can further customize how an attribute should behave. For example, an `ErrorMessage` enables you to customize the message that is shown when validation fails.

There's more...

In this recipe, we've used just a few of the possible data annotations. Others that are available are:

- ▶ `DataTypeAttribute`
- ▶ `RangeAttribute`
- ▶ `RegularExpressionAttribute`
- ▶ `RequiredAttribute`
- ▶ `StringLengthAttribute`
- ▶ `CustomValidationAttribute`

For all of these attributes, named parameters are available to further customize the way validation should occur. `ErrorMessage`, `ErrorMessageResourceName`, and `ErrorMessageResourceType` are available on all the attributes, but many more are available depending on the attribute you use. You can check these parameters by looking at the IntelliSense tool tip that you get on the attribute constructor.

See also...

If you want to learn more about the various uses of data annotations, have a look at *Chapter 10, Talking to REST and WCF Data Services* and *Chapter 11, Using WCF RIA Services*.

5

Working with Local Data

In this chapter, we will cover the following:

- ▶ Reading data from and storing data in the isolated storage
- ▶ Working with `IsolatedStorageSettings`
- ▶ Caching data between different Silverlight applications using Isolated Storage
- ▶ Using the Sterling database

Introduction

By default, Silverlight applications cannot access the local file system. This is because Silverlight applications run under partial-trust: if running inside the browser, they can't escape from the **sandbox** of the browser (things are different with out-of-the-browser applications with elevated permissions: this type of application has access to the local disk). This is quite logical, as it wouldn't be a good thing if just any Silverlight application you're running could access the files on your local drive without any limitation! However, Silverlight has the ability to use **Isolated Storage**. In fact, isolated storage is not specific to Silverlight: it also exists in regular .NET and Windows Phone 7 (which we are covering in *Chapter 13, Windows Phone 7*) also has support for it.

Isolated storage can somewhat be compared to cookies, since it has the same characteristic that it can be used to save data for an application between sessions on the client. However, while a cookie can only store plain text, isolated storage can store just about everything that your application may want to persist, varying from user settings (perhaps serialized into an XML file) to an image that we want to cache instead of downloading it again. Being isolated, isolated storage basically means that storage is specific to an application: a Silverlight application can only access its own storage and not the storage of another application. There is an exception to this rule, at which we'll look in this chapter as well. It also offers a complete file system, which can be described as a virtual file system. From the root of this file system, we can build a directory and file hierarchy, just like you would do with your hard drive. We can create (nested) directories and files in there. While isolated storage is intended to be used solely from the Silverlight application that created it, it should be noted that it is nothing more than a physical directory on the hard disk. Being a normal directory, it can be located by a user. That should ring an alarm bell! It is not safe to put any sensitive data in isolated storage. Encryption of data stored in isolated storage can help make sure that no one can simply read out the information. If the data is vital for the application, a good solution might be to back it up on the server over a service for the case when it gets deleted on the client machine.

In the first recipe of this chapter, we will start by looking at how we can practically use the isolated storage using the `IsolatedStorageFile` class, for example to cache data from a service. Data that doesn't change doesn't need to be downloaded every time. Caching it in isolated storage brings down the load on the server as well as improves the performance of the application since the service doesn't need to be invoked every time. A second way of using isolated storage is through the `IsolatedStorageSettings` class, which is basically a dictionary we can use to persist any kind of object (using serialization) and reload objects from (using deserialization). This class is really helpful if you don't want to be in charge of writing all the persistence logic yourself.

Although isolated storage is limited to the application creating the store, it's possible to have applications running from the same site to access a shared version of the storage. We'll see how we can activate and use this as well.

We'll finish off the chapter by looking at Sterling, a NoSQL object-oriented database that uses isolated storage for persistence of objects and allows LINQ-to-Objects queries to be executed on the data.

Reading data from and storing data in the isolated storage

Applies to Silverlight 3, 4, and 5

As we've seen in the introduction, isolated storage allows our Silverlight applications to store data, files, and so on to a persistent store. Even after the application is shut down, the data remains intact, allowing us to write code that checks for the presence of stored data upon startup of the application.

Isolated storage can be handy to store files such as images, user settings such as selected theme, or even the data the user was entering on a screen when the application was shut down for some reason. In this recipe, we'll look at how we can store and read data.

Getting ready

For this recipe, a starter solution that contains the service we're using has been created. It can be found in the Chapter05/IsoStore_Starter folder. The finished solution for this recipe can be found in the Chapter05/IsoStore_Completed folder.

How to do it...

To demonstrate the use of isolated storage, we'll use it in a scenario where we are accessing a service that returns data. The application mimics the use of a hotel booking application where the user first gets a list of cities. After selecting a city, hotels for the selected city is listed.

Instead of hammering the server every time the user arrives on the application for the same city data, we'll store the data on the local machine after first retrieval using isolated storage. From the second request on, the data will be loaded from the disk rather than from the service. You will need to complete the following steps to learn how to use isolated storage:

1. Open the solution in the starter solution and note that there's a Silverlight-enabled WCF service in the web project (`IsoStore.Web`) called `CityHotelService.cs`. This service uses LINQ-to-Entities code using an Entity Model (`CityHotelModel.edmx`), which can also be found in this project.
2. In the Silverlight project, a service reference has already been created. This reference can be found under the **Service References** node in the **Solution Explorer**.
3. Under the **Views** folder, add a Silverlight page and call it `CityView.xaml`. The XAML code for this page is shown next. It uses a simple `DataGridView` and two `Button` instances that allow the user to load the hotels and refresh the list from the service, respectively.

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="100"></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button Content="Load All Cities"
            VerticalAlignment="Center"
            Name="LoadAllCitiesButton"
            Click="LoadAllCitiesButton_Click">
        </Button>
        <Button Content="Refresh cities"
            VerticalAlignment="Center"
            Name="RefreshCitiesButton"
            Click="RefreshCitiesButton_Click">
        </Button>
    </StackPanel>
</Grid>
```

```
    VerticalAlignment="Center"
    Name="RefreshCitiesButton"
    Click="RefreshCitiesButton_Click">
  </Button>
</StackPanel>
<sdk:DataGrid AutoGenerateColumns="False"
  Grid.Row="1"
  HorizontalAlignment="Stretch"
  Name="CityDataGrid"
  VerticalAlignment="Stretch">
<sdk:DataGrid.Columns>
  <sdk:DataGridTemplateColumn Header="">
    <sdk:DataGridTemplateColumn.CellTemplate>
      <DataTemplate>
        <Button Content="+" 
          Click="NavigateHotelButton">
        </Button>
      </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
  </sdk:DataGridTemplateColumn>
  <sdk:DataGridTemplateColumn
    x:Name="imageUrlColumn"
    Header="Image Url"
    Width="SizeToHeader">
    <sdk:DataGridTemplateColumn.CellTemplate>
      <DataTemplate>
        <Image
          Source="{Binding Path=ImageUrl,
            Converter={StaticResource
              localImageConverter}}" />
      </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
  </sdk:DataGridTemplateColumn>
  <sdk:DataGridTextColumn
    x:Name="cityNameColumn"
    Binding="{Binding Path=CityName}"
    Header="City Name"
    Width="SizeToHeader" />
  <sdk:DataGridTextColumn
    x:Name="areaColumn"
    Binding="{Binding Path=Area}"
    Header="Area"
    Width="SizeToHeader" />
  <sdk:DataGridTextColumn
```

```
        x:Name="cityIdColumn"
        Binding="{Binding Path=CityId}"
        Header="City Id"
        Width="SizeToHeader" />
    <sdk:DataGridTextColumn
        x:Name="countryColumn"
        Binding="{Binding Path=Country}"
        Header="Country"
        Width="SizeToHeader" />
    <sdk:DataGridTextColumn
        x:Name="languageColumn"
        Binding="{Binding Path=Language}"
        Header="Language"
        Width="SizeToHeader" />
    <sdk:DataGridTextColumn
        x:Name="populationColumn"
        Binding="{Binding Path=Population}"
        Header="Population"
        Width="SizeToHeader" />
    <sdk:DataGridTextColumn
        x:Name="websiteColumn"
        Binding="{Binding Path=Website}"
        Header="Website"
        Width="SizeToHeader" />
</sdk:DataGrid.Columns>
</sdk:DataGrid>
</Grid>
```

4. When we click on LoadCitiesButton, we want to show the user a list of all the cities. This list is initially retrieved from the service and the resulting data is stored in isolated storage. To bring down the load on the service, subsequent requests will not load the data again from the service; the data will be read from isolated storage if present. In the following code, in the LoadAllCities() method, we can see this functionality:

```
private void LoadAllCitiesButton_Click
    (object sender, RoutedEventArgs e)
{
    LoadCities();
}

private void LoadCities()
{
    using (var isoStore =
        IsolatedStorageFile.GetUserStoreForApplication())
```

```
{  
    if (isoStore.FileExists("CityInfo.xml"))  
    {  
        CityDataGrid.ItemsSource = LoadCitiesFromXml();  
    }  
    else  
    {  
        LoadCitiesFromService();  
    }  
}
```

5. Using the static `GetUserStoreForApplication` on the `IsolatedStorageFile` class, we can check if a file called `CityInfo.xml` exists. If it does, we'll read it out using the `LoadCitiesFromXml()` method to load in the cities. If it doesn't, we will need to go ahead and load the cities from the service. Both these methods will be implemented next.
6. Let's first look at what should happen with the initial call. The `LoadCitiesFromService()` calls the service and in the callback, the resulting list is used as `ItemsSource` for the `DataGrid`. A call is made to the `SaveXml()` method, which we'll see next.

```
private void LoadCitiesFromService()  
{  
    CityHotelService.CityHotelServiceClient proxy =  
        new CityHotelServiceClient();  
    proxy.GetCitiesCompleted +=  
        new EventHandler<GetCitiesCompletedEventArgs>(  
            proxy_GeCitiesCompleted);  
    proxy.GetCitiesAsync();  
}  
  
void proxy_GeCitiesCompleted  
    (object sender, GetCitiesCompletedEventArgs e)  
{  
    if (e.Error == null)  
    {  
        CityDataGrid.ItemsSource = e.Result;  
        SaveXml(e.Result);  
    }  
}
```

7. In the `SaveToXml()` method, we are using LINQ-to-XML code to create an XML representation of the data that we want to store. More importantly, we need to store this data. We can do so using the `IsolatedStorageFileStream` class, which accepts the filename, `FileMode`, and `IsolatedStorageFile` instance to work with. With this stream, we can save the XML to disk. This is shown in the following code sample:

```
private void SaveToXml(ObservableCollection<City> cities)
{
    var doc = new XDocument(new XElement("Cities"));

    foreach (var item in cities)
    {
        doc.Element("Cities").Add(
            new XElement("CityInfo",
                new XElement("CityId", item.CityId),
                new XElement("ImageUrl", item.ImageUrl),
                new XElement("Area", item.Area),
                new XElement("CityName", item.CityName),
                new XElement("Country", item.Country),
                new XElement("Language", item.Language),
                new XElement("Population", item.Population),
                new XElement("Website", item.Website)
            ) );
    }

    using (var isoStore =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (var isoStream =
            new IsolatedStorageFileStream
                ("CityInfo.xml", FileMode.Create, isoStore))
        {
            doc.Save(isoStream);
        }
    }
}
```

8. When the page is called again, we'll want to load the data from the isolated storage. Again, we can use the `IsolatedStorageFileStream` class to load in a file in this case. The following code shows the `LoadCitiesFromXml()` method where we read in the XML and parse it using LINQ-to-XML methods:

```
private ObservableCollection<City> LoadCitiesFromXml()
{
    ObservableCollection<City> cities =
        new ObservableCollection<City>();

    using (var isoStore =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (var isoStream =
            new IsolatedStorageFileStream
                ("CityInfo.xml", FileMode.Open, isoStore))
        {
            var document = XDocument.Load(isoStream);
            IEnumerable< XElement > elements =
                document.Element("Cities").Elements("CityInfo");

            foreach (var item in elements)
            {
                cities.Add(
                    new City()
                    {
                        CityId =
                            Int32.Parse(item.Element("CityId").Value),
                        Area = Int32.Parse(item.Element("Area").Value),
                        CityName = item.Element("CityName").Value,
                        Country = item.Element("Country").Value,
                        ImageUrl = item.Element("ImageUrl").Value,
                        Language = item.Element("Language").Value,
                        Population =
                            Int32.Parse(item.Element("Population").Value),
                        Website = item.Element("Website").Value
                    });
            }
        }
    }
    return cities;
}
```

9. When the user wants to refresh the data, they can click on RefreshButton. In this case, we want to delete the file. As isolated storage works with simple IO methods, we can do so using the DeleteFile() method of the IsolatedStorageFile class, as shown next:

```
private void RefreshCitiesButton_Click
    (object sender, RoutedEventArgs e)
{
    using (var isoStore =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        isoStore.DeleteFile("CityInfo.xml");
    }
    LoadCities();
}
```

With these steps executed, we have an application which now uses isolated storage as a local caching mechanism that helps bring down the load on the service and speeds up the application as data is read locally - we don't have to wait for a service to respond. Note that this technique is a good fit for data that doesn't change often; caching rapidly-changing data on the client might not be a good solution.

In the completed solution, you can look at the code for the HotelView page as well, which is similar to the page we created here.

How it works...

Isolated storage in Silverlight is the way to store information locally on the user's machine. Each application gets its own location within the isolated storage. For an application, it's as if it has its own file system where it can store files, create directories or perform other IO-related operations. The term isolated actually refers to the way the storage is built: each application has its own storage which is isolated from the rest. Also, from this location, it's impossible to browse to the rest of the file system (there's no way that using ...\\..\\.. could allow access to files and folders such as Program Files or the Windows directory).

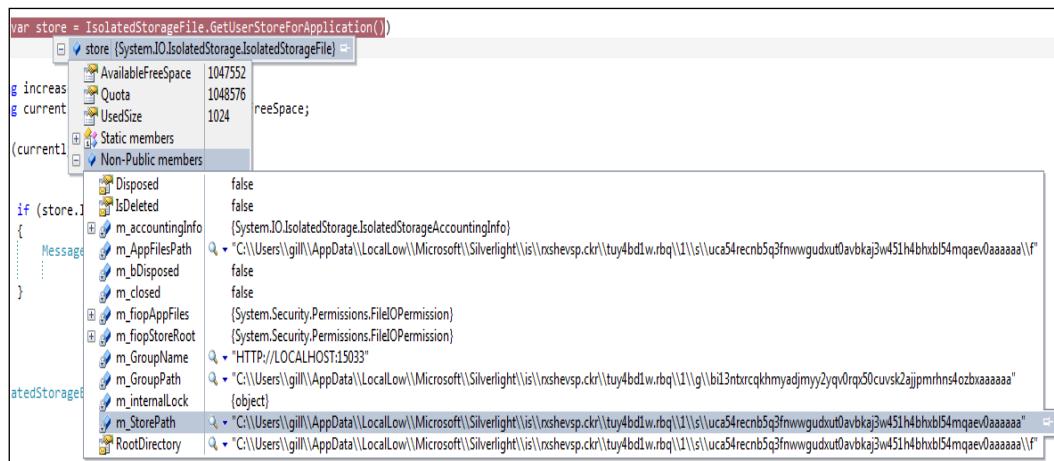
Using the GetUserStoreForApplication method on the IsolatedStorageFile class, we get access to the store (which is nothing more than a directory which is hidden deep in the physical file system), specific for the application and the executing user. In fact, there are two types of stores: the user + application store (which we have been using here) and the user + site store (which we'll look at in the *Caching data between different Silverlight applications using isolated storage* recipe).

Isolated storage can be used to store any type of data in. In this recipe, we used it as a **caching** mechanism: on the local machine, we stored the data coming back from the service so that each page load of the CityView page doesn't make a request to the service. Instead, after initial loading of the data, we load the data from the isolated storage. Here, we used LINQ-to-XML to read and write the data to an XML file. Using XML isn't mandatory; we can use any file type we want, we can write code that saves an image to a jpg file inside the isolated storage. Note that since isolated storage is limited in size, you could fill up the available space quickly when working with images. We'll look at isolated storage size further.

Once we have the data we want to store, we use the `IsolatedStorageFileStream` class to perform the actual reading and writing of the data. This class extends the `FileStream`, which means that familiar IO methods can be used in combination with this class, such as `EndRead`, `Seek`, `WriteByte` (for a complete list, see <http://msdn.microsoft.com/en-us/library/system.io.isolatedstorage.isolatedstoragefilestream.aspx>).

Where's my data

Although obscured deep in the file system, it's possible to locate the files created by isolated storage. To do so, put a breakpoint on a line that uses `IsolatedStorageFile.GetUserStoreForApplication()`. In the properties, we can see the physical path, as shown in the following screenshot:



While it's not really obvious where the data is stored, it should be easy to see that this is *not secure* at all. Therefore, isolated storage is certainly not a good location to store sensitive information such as passwords! As mentioned before, encrypting the data can be useful here as well to increase security.

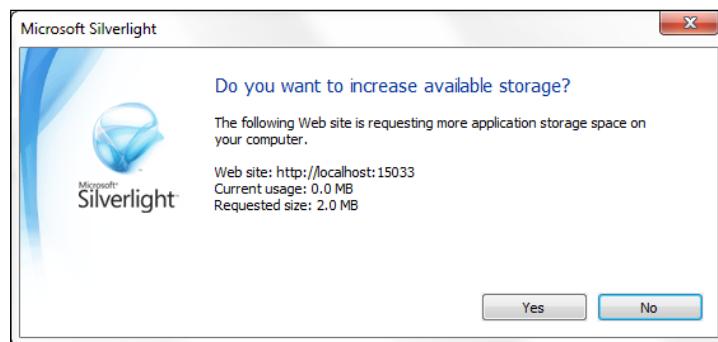
Isolated storage size

Isolated storage is **limited** in size, which is quite logical. If it wasn't limited, an application could start filling the entire disk without the user knowing. By default, an in-browser application is limited to 1 MB (out-of-browser applications get 25 MB). One megabyte is not a lot of space and it certainly won't be enough if you start storing images in isolated storage. Therefore, it's possible to include in our code a check for the currently available size as well as a prompt to the user to request more space. The following code shows how we can do this:

```
public void GetMoreSpace()
{
    try
    {
        using (var store =
IsolatedStorageFile.GetUserStoreForApplication())
        {
            long increaseAmount = 1048576;
            long currentlyAvailable = store.AvailableFreeSpace;

            if (currentlyAvailable < increaseAmount)
            {
                if (store.IncreaseQuotaTo
                    (store.Quota + increaseAmount))
                {
                    MessageBox.Show ("Thanks for increasing the quota!");
                }
            }
        }
    catch (IsolatedStorageException)
    {
    }
}
```

The following screenshot shows the prompt that Silverlight shows the user, asking to increase the quota:



Note that an exception will occur if the requested amount is smaller than what the application already has. Also note that requesting more space can only be triggered after a user-initiated action took place, such as *Click on a button*. It's therefore not possible to prompt the user in the Load event handler.

See also

In the *Working with IsolatedStorageSettings* recipe, we'll see another way of working with isolated storage.

Working with IsolatedStorageSettings

Applies to Silverlight 3, 4, and 5

In the previous recipe, we used the `IsolatedStorageFile` class to manually perform the saving to and reading from isolated storage. While this option is the most powerful one—it allows us to save anything varying from raw data to images—it's quite a cumbersome task to perform all the serializing and deserializing of objects ourselves. Although we used the easy-to-work-with LINQ-to-XML, there is an even easier solution: letting Silverlight take care of this detail. In this recipe, we'll do just that by using the `IsolatedStorageSettings` class.

Getting ready

In this recipe, we are going to refactor the code from the previous recipe. You can therefore continue working on that codebase. Alternatively, you can use the starter solution provided with this book, which is located in the `Chapter05/IsolatedStorageSettings_Starter` folder. The complete solution for this recipe can be found in the `Chapter05/IsolatedStorageSettings_Completed` folder.

How to do it...

Instead of doing all the save and read operations manually through the use of LINQ-to-XML, we are going to make use of the `IsolatedStorageSettings` class that comes with Silverlight. This class offers us a `Dictionary<string, object>` through the `ApplicationSettings` property, which we can use to save and retrieve any type of object. In the following steps, we are going to refactor the previous recipe to make use of this simpler approach:

1. As mentioned in the *Getting ready* section, open the starter solution or use your solution from the previous project. Start by adding a class to the Silverlight project and name it `IsoStoreSettingsHelper`. This class will be our wrapper around `IsolatedStorageSettings.ApplicationSettings`. Make the class static as follows:

- ```
public static class IsoStoreSettingsHelper
{
}
```
2. In the `IsoStoreSettingsHelper` class, add the following code to allow us to save objects by key:
- ```
public static void SaveObject<T>(string key, T value)
{
    IsolatedStorageSettings.ApplicationSettings[key] = value;
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```
3. To this generic method, we are passing a key and a value, which can basically be of any type we want. Inside the method, we use the `ApplicationSettings` property, which implements the `IDictionary<string, object>` to add a value to the dictionary. Finally, we call `Save()` to persist the changes to isolated storage.
4. Again in the `IsoStoreSettingsHelper` class, add the following method to be able to retrieve a value based on a given key:
- ```
public static T GetObject<T>(string key) where T : class
{
 if (IsolatedStorageSettings.ApplicationSettings
 .Contains(key))
 {
 return IsolatedStorageSettings.ApplicationSettings[key]
 as T;
 }
 return default(T);
}
```
5. This method first checks using the `Contains()` method if a value is present for the key. If so, the value is searched and cast to the passed-in type. If not, the default for the given type is returned.
6. Also add the following methods to the `IsoStoreSettingsHelper` class to perform a delete and a check to see if a value for a given key is present:
- ```
public static void DeleteObject(string key)
{
    if (IsolatedStorageSettings.ApplicationSettings
        .Contains(key))
    {
        IsolatedStorageSettings.ApplicationSettings.Remove(key);
    }
}

public static bool ContainsObject(string key)
```

```
{  
    if (IsolatedStorageSettings.ApplicationSettings  
        .Contains(key))  
    {  
        return true;  
    }  
    return false;  
}
```

7. Let's now look at the code behind the `CityView` class, `CityView.xaml.cs`. We'll change this class to use our newly-created wrapper class instead of doing the persisting itself. Up first is the `LoadAllCities()` method, which is shown next. As can be seen, this method now uses `ContainsObject` from `IsoStoreSettingHelper` to check for the presence of a value for the key `Cities`. If this method returns true, the value is retrieved using the `GetObject()` method.

```
private void LoadCities()  
{  
    if (IsoStoreSettingsHelper.ContainsObject("Cities"))  
    {  
        CityDataGrid.ItemsSource =  
            IsoStoreSettingsHelper.  
                GetObject<ObservableCollection<City>>("Cities");  
    }  
    else  
    {  
        LoadCitiesFromService();  
    }  
}
```

8. The callback from the `LoadCitiesFromService`, `proxy_GeGetCitiesCompleted()` can also be changed. Instead of saving to XML, it can now use the `IsoStoreSettingsHelper` to save the cities. This is shown in the following code:

```
void proxy_GeGetCitiesCompleted  
    (object sender, GetCitiesCompletedEventArgs e)  
{  
    if (e.Error == null)  
    {  
        CityDataGrid.ItemsSource = e.Result;  
  
        IsoStoreSettingsHelper.SaveObject  
            <ObservableCollection<City>>("Cities", e.Result);  
    }  
}
```

9. Finally, we can change the refresh functionality so that it uses the `DeleteObject()` method as follows:

```
private void RefreshCitiesButton_Click
    (object sender, RoutedEventArgs e)
{
    using (var isoStore =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        IsoStoreSettingsHelper.DeleteObject("Cities");
    }
    LoadCities();
}
```

With this code in place, we are now letting Silverlight take care of saving and retrieving the data from isolated storage using `IsolatedStorageSettings.ApplicationSettings`. In the completed solution, you can also see the use of this class in the code-behind for the `HotelView` page.

How it works...

While `IsolatedStorageFile` allows us to write and read any file from and to isolated storage, it can be a lot of work to do this task ourselves. As shown in the previous recipe, we as developers needed to take care of serialization of the data.

For this purpose, the `IsolatedStorageSettings.ApplicationSettings` exists. With it, we get access to an `IDictionary<string, object>` that's being persisted automatically for us to isolated storage. Silverlight itself takes care of saving the object that we pass it and it associates the object with a key. This key can then later be used to retrieve the value again.

Behind the scenes, Silverlight is creating a text file named `__LocalSettings` where the data can be read. This file is in the same location as where the data in isolated storage is being stored. As this is again a plain text/XML file, it's not a best practice to store sensitive data such as passwords in the `IsolatedStorageSettings`.

See also

In the previous recipe, we looked at working with the `IsolatedStorageFile` which shows another way of working with isolated storage.

Caching data between different Silverlight applications using isolated storage

Applies to Silverlight 3, 4 and 5

The isolated part of isolated storage means that the data is isolated. Up until now we said that data is stored per-user and only the application that stored the data has access to it. This is the default when we use `IsolatedStorageFile.GetUserStoreForApplication` or `IsolatedStorageSettings.ApplicationSettings`. However, there's another option that allows sharing of data between applications if the applications are originating from the same website. In this recipe, we'll change the code created in the first recipe of this chapter so that the data in isolated storage is also accessible from other Silverlight applications that run from the same hosting website.

Getting ready

In this recipe, we are refactoring the code from the first recipe of this chapter. You can use your own codebase or use the sample code from the code of the book. The starter solution for this recipe is located in the `Chapter05/IsoSiteStore_Starter` folder; the complete solution can be found in the `Chapter05/IsoSiteStore_Completed` folder.

How to do it...

In this recipe, we'll take a look at the refactoring we need to apply to the code of the first recipe of this chapter so that the data stored in isolated storage is available to other applications running from the same site. Follow along with the following steps to learn more:

1. As outlined in the *Getting ready* section, open either your solution from the first recipe of this chapter or use the starter solution provided for this recipe. This Silverlight application uses isolated storage through the `IsolatedStorageFile.GetUserStoreForApplication()` method.
2. In the `SaveToXml()` method, refactor your code so that we are using the static `GetUserStoreForSite()` method of the `IsolatedStorageFile` class, as shown next:

```
...
using (var isoStore =
    IsolatedStorageFile.GetUserStoreForSite())
{
    using (var isoStream =
        new IsolatedStorageFileStream
            ("CityInfo.xml", FileMode.Create, isoStore))
    {
        doc.Save(isoStream);
```

```
    }  
}  
...
```

3. Throughout the entire `CityView.xaml.cs` file, change every occurrence of `GetUserStoreForApplication()` to `GetUserStoreForSite()`.
4. In the `HotelView.xaml.cs` file, also change every occurrence of `GetUserStoreForApplication()` to `GetUserStoreForSite()`.

If we run the application now, we'll see that the application runs the exact same way. However, the isolated storage is now shared between applications that originate from the same site. In the completed solution of this recipe, a test project named `IsoStore_Reader` is added to the solution. It allows you to read out the information from isolated storage shared with the other Silverlight applications running from the same site.

How it works...

By default, isolated storage uses a per-user, per-application location where an application can store and access data. This data is then only accessible for the application which created the data; no other application can access that data. It's easy to understand why this limitation is in place; if any application could access any other's information, a lot of security issues would be raised.

However, there are situations where you may want this approach a bit less strict. Assume you have more than one Silverlight application running from your site. Each of these applications needs a list of countries on some particular form. It would be a good step for each application to cache that data in isolated storage, helping to bring the load on the server down as well as to speed up the application as it doesn't have to access the service for this data. However, we can go one step further and let these applications share this cached country list. It's possible that a Silverlight application opens up (part of) its data in isolated storage for other applications running from the same site.

To have an application share the data, it should use the static `GetUserStoreForSite()` method of the `IsolatedStorageFile` class instead of `GetUserStoreForApplication()`. The application reading the shared data should also use `GetUserStoreForSite()`. This effectively means that we have an opt-in model: the developer can choose to allow other applications from the same site to use the data. Because this is limited to the bounds of the site, no real security issues arise from this: it's assumed that no one that has access to the site will mess around with your data.

What about `IsolatedStorageSettings`?

Just like `IsolatedStorageFile` has a site option, `IsolatedStorageSettings` has one as well. Through `IsolatedStorageSettings.SiteSettings`, which is again an `IDictionary<string, object>`, we can store information that's accessible for other Silverlight applications running from the same site.

Using the Sterling database

Applies to Silverlight 4 and 5

Up until now, we've seen several ways of storing data locally on the system using isolated storage. With `IsolatedStorageSettings`, we are handing our object over to Silverlight which does the serialization to XML itself. There's little we have to say in this process. When using `IsolatedStorageFile`, we are in charge of serialization of the data. We have the option to save as XML, JSON or even as binary. The latter isn't supported out-of-the-box since Silverlight doesn't have a `BinaryFormatter` on board, so we are on our own to write the code for this ourselves.

Instead of doing so, we can use a library called **Sterling**. Sterling provides us with an object-oriented, NoSQL database, based on isolated storage that writes data in a *binary format to isolated storage*. On top of that, we get the ability to write LINQ-to-Objects queries to query our data and it handles things such as foreign keys and indexes as well. In this recipe, we'll look at how we can work with this component.

Getting ready

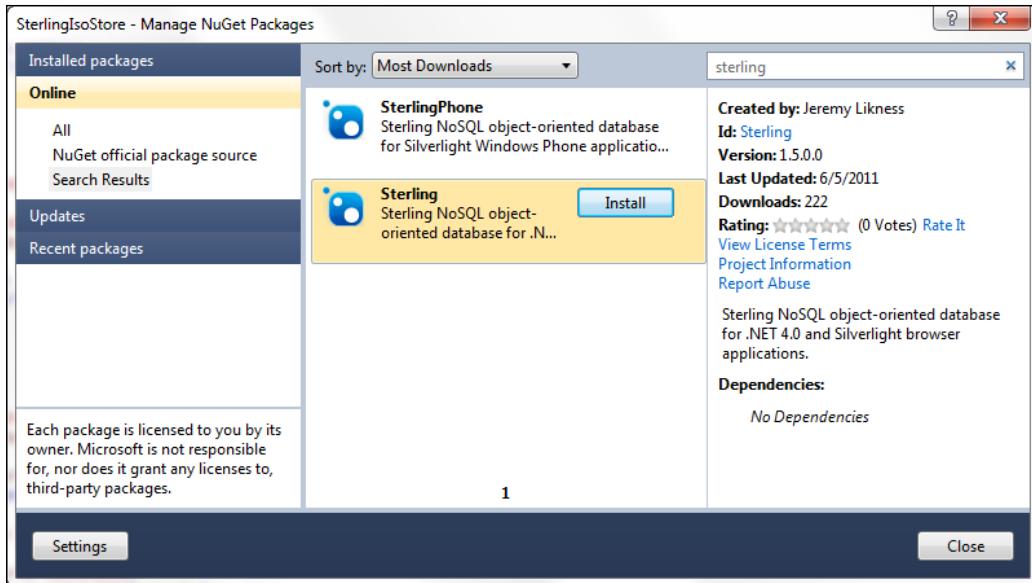
This recipe creates an application from scratch, so there's no starter solution provided. The finished solution for this recipe can be found in the `Chapter05/SterlingIsoStore_Completed` folder.

How to do it...

In this recipe, we'll build a small note application, allowing us to save notes and view a list of notes we've already created. Instead of using isolated storage directly, we'll use the Sterling database. Follow along with the steps given to learn how to use the component:

1. Create a new Silverlight Navigation Application and name it `SterlingIsoStore`. Make sure you select the *navigation application template*.
2. As Sterling is an external component, we need to add it to the Silverlight project. You can do so in two ways. Either you can go to <http://sterling.codeplex.com>, download the binaries from there and add a reference to the downloaded assemblies from your Silverlight project. Alternatively, you can use **NuGet** (<http://www.nuget.org>), a package manager for Visual Studio from Microsoft. In the *Appendix*, you can find information on installing and using NuGet.

With NuGet installed, right-click on your Silverlight project and select **Manage NuGet Packages**. In the dialog shown, search for Sterling and select the non-Windows Phone 7 version to add to your project:



3. Let's first create the XAML for the two pages our application needs. In the Home.xaml file, which is under the Views folder, add the following code:

```
<navigation:Page.Resources>
    <DataTemplate x:Key="NoteTemplate">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
            </Grid.RowDefinitions>
            <TextBlock
                Text="{Binding NoteName}"
                FontWeight="Bold">
            </TextBlock>
            <TextBlock
                Text="{Binding NoteContents}"
                Grid.Row="1">
            </TextBlock>
        </Grid>
    </DataTemplate>
</navigation:Page.Resources>
<Grid x:Name="LayoutRoot">
    <StackPanel
        x:Name="ContentStackPanel"
        Margin="20"
```

```
    HorizontalAlignment="Left">
<TextBlock
    Text="My notes"
    FontSize="14"
    FontWeight="Bold">
</TextBlock>
<ListBox
    Width="200"
    Height="200"
    Name="NotesListBox"
    ItemTemplate="{StaticResource NoteTemplate}"
    HorizontalAlignment="Left">
</ListBox>
<HyperlinkButton
    NavigateUri="/CreateNote"
    Content="Create new note"
    Margin="10">
</HyperlinkButton>
</StackPanel>
</Grid>
```

This code should look familiar, there's nothing new being introduced here.

4. Add another page to the views folder and name it `CreateNote.xaml`. Add the following code to create a form to add a new note:

```
<Grid x:Name="LayoutRoot">
    <Grid
        Width="400"
        HorizontalAlignment="Left"
        Margin="20">
        <Grid.RowDefinitions>
            <RowDefinition Height="30"/></RowDefinition>
            <RowDefinition Height="30"/></RowDefinition>
            <RowDefinition Height="30"/></RowDefinition>
            <RowDefinition Height="30"/></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/></ColumnDefinition>
            <ColumnDefinition Width="*"/></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock
            Text="Add new note"
            FontWeight="Bold"
            FontSize="14"
            Grid.ColumnSpan="2">
```

```
</TextBlock>
<TextBlock
    Text="Name"
    Grid.Row="1"
    Grid.Column="0">
</TextBlock>
<TextBlock
    Text="Contents"
    Grid.Row="2"
    Grid.Column="0">
</TextBlock>
<TextBox
    Name="NoteNameTextBox"
    Grid.Column="1"
    Grid.Row="1"
    Width="200"
    Height="25">
</TextBox>
<TextBox
    Name="NoteContentsTextBox"
    Grid.Column="1"
    Grid.Row="2"
    Width="200"
    Height="25">
</TextBox>
<HyperlinkButton
    Content="Save note"
    Grid.Column="1"
    Grid.Row="3"
    Name="SaveNoteButton"
    Click="SaveNoteButton_Click">
</HyperlinkButton>
</Grid>
</Grid>
```

5. Create a new folder and name it `Model`. Inside the folder, create a new interface called `IBaseModel`. Add the following code to this interface:

```
public interface IBaseModel
{
    int Id { get; set; }
}
```

6. We can now create our Note class and let it implement the IBaseModel interface. The Note class only contains a few properties, as outlined next:

```
public class Note : IBaseModel
{
    public int Id { get; set; }
    public string NoteName { get; set; }
    public string NoteContents { get; set; }
    public DateTime CreationDate { get; set; }
}
```

7. We have now arrived at the point at which we can really start using Sterling. Create a folder called Database and inside of it, create a class and call it NoteDatabase. The code for this class is shown next:

```
public class NoteDatabase : BaseDatabaseInstance
{
    public const string NOTE_NOTEID = "Note_NoteId";

    public override string Name
    {
        get { return "NoteDatabase"; }
    }

    protected override List<ITableDefinition> RegisterTables()
    {
        return new List<ITableDefinition>()
        {
            CreateTableDefinition<Note, int>(n => n.Id)
                .WithIndex<Note, int, int>
                    (NOTE_NOTEID, fd => fd.Id)
            };
    }
}
```

8. The class that represents the database needs to inherit from BaseDatabaseInstance. An application can contain more than one database; each one needs to be represented with a single class. In this class, we override two members: Name and RegisterTables. Inside the RegisterTables() method, we can define how our data needs to be saved. Using the CreateTableDefinition() method, we are indicating that we want to save the Note class. Also, we are defining the key for this table, which is the Id property. Finally, we are creating an index on the Id property as well.

9. Now that we have the database, we need a way to work with it. We are going to create a class called `DatabaseService` inside the `Database` folder which interacts with our created database. In this `DatabaseService` class, add the following code:

```
public static class DatabaseService
{
    private static SterlingEngine _sterlingEngine;
    public static ISterlingDatabaseInstance CurrentDatabase
        { get; private set; }

}
```

10. We are creating a `SterlingEngine` here, as our application needs one of these. The `CurrentDatabase` member will be used to access the database from outside this class.
11. We need to write code to start the database engine. Add a method named `ActivateEngine()` to the `DatabaseService` class, as shown next:

```
public static void ActivateEngine()
{
    _sterlingEngine = new SterlingEngine();

    _sterlingEngine.Activate();

    CurrentDatabase =
        _sterlingEngine.SterlingDatabase
            .RegisterDatabase<NoteDatabase>
                (new IsolatedStorageDriver());
}

}
```

12. To start the engine, we call the `Activate()` method on it. After activation, we can register our database with the engine. We are passing in an `IsolatedStorageDriver` instance, which will cause Sterling to use isolated storage to save its data. By default, data is stored in-memory and is thus not persisted between application runs.
13. With databases, we mostly don't want to worry about indexing the key ourselves. With Sterling, we need to create what is called a **Trigger** to generate the identity key. Add a class called `IdentityTrigger` to the `Database` folder in your project and add the following code:

```
public class IdentityTrigger<T> : BaseSterlingTrigger<T, int>
    where T : class, IBaseModel, new()
{
    private static int _idx = 1;
```

```
public IdentityTrigger(ISTerlingDatabaseInstance database)
{
    if (database.Query<T, int>().Any())
    {
        _idx =
            database.Query<T, int>().Max(key => key.Key) + 1;
    }
}

public override bool BeforeSave(T instance)
{
    if (instance.Id < 1)
    {
        instance.Id = _idx++;
    }

    return true;
}

public override void AfterSave(T instance)
{
    return;
}

public override bool BeforeDelete(int key)
{
    return true;
}
```

14. The trigger needs to be registered with our database so it will fire when we add a new instance. Add the following line as the last line in the `ActivateEngine()` method in the `DatabaseService` class:

```
CurrentDatabase.RegisterTrigger
    (new IdentityTrigger<Note>(CurrentDatabase));
```

15. We now have everything in place to work with the database. Of course, we need to start it as well. It's a good practice to have the `App` class start the service. In the `App.xaml.cs` in the constructor, add the following line:

```
DatabaseService.ActivateEngine();
```

16. Also in the constructor, register a handler for the Application's `Exit` event as follows:

```
this.Exit += new EventHandler(Application_Exit);
```

17. In the handler, we dispose of the engine by calling the `DeactivateEngine()` method as follows:

```
void Application_Exit(object sender, EventArgs e)
{
    DatabaseService.DeactivateEngine();
}
```

18. With the database engine running, we can now in a very easy way use it from our application. In the code-behind of the `Home.xaml`, in the `OnNavigatedTo()` method, add the following code:

```
private ObservableCollection<Note> _notes =
    new ObservableCollection<Note>();

foreach (var item in DatabaseService.CurrentDatabase
    .Query<Note, int>())
{
    _notes.Add(item.LazyValue.Value);
}

NotesListBox.ItemsSource = _notes;
```

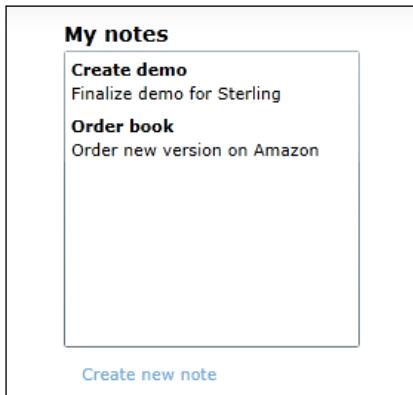
19. We are querying for all notes that can be found in the database and add them to an `ObservableCollection`. This collection is then set as the `ItemsSource` for the `ListBox`.

20. In the `CreateNode.xaml.cs` file, in the event handler of the `Save` button, we create a new instance of the `Note` class and save this to the database.

```
private void SaveNoteButton_Click
    (object sender, RoutedEventArgs e)
{
    Note note = new Note()
    {
        NoteName=NoteNameTextBox.Text,
        NoteContents = NoteContentsTextBox.Text,
        CreationDate=DateTime.Now
    };
    DatabaseService.CurrentDatabase.Save(note);
    DatabaseService.CurrentDatabase.Flush();

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

With this code in place, we can save notes and persist them between application reboots. The following screenshot shows some notes already added to the database.



How it works...

Instead of performing manual serialization like we did when we used the `IsolatedStorageFile` or have Silverlight serialize our data to XML like we did with the `IsolatedStorageSettings`, we can use Sterling. Sterling is a NoSQL object-oriented database for use with Silverlight (an implementation also exists for Windows Phone 7, which works in a similar way).

Sterling saves data in a binary format. While we can do so manually as well using a **BinaryFormatter**, Sterling does all of that for us. On top of that, it fully supports LINQ-to-Objects, giving us a familiar way of working with our data. Although the data isn't stored in the same way as a database, we are getting a comparable behavior for our Silverlight applications.

In the following paragraphs, we'll dig a bit deeper into the several aspects that make up how we work with Sterling.

Getting Sterling

Sterling is free and open-source. That means that we get full access to the source code and can make changes if we want. The project is hosted on CodePlex (<http://sterling.codeplex.com>) where the source code, binaries, and documentation can be found.

Using Sterling in our projects requires that we reference the Sterling assemblies. We can do this manually from the Silverlight project by using **Add Reference**. Alternatively, we can use NuGet (see the appendix for guidance on working with NuGet) to have the assemblies downloaded and referenced automatically for us.

Creating a database

When we say we are creating a database with Sterling, we are not going to start by creating a schema in SQL Server Management Studio. Creating a database is done using code. To do so, every "table" we want needs to be represented by a class. We are in fact creating the database model via code.

The database is a class which inherits from `BaseDatabaseInstance`. In this class, we can override the `Name` property as well as the `RegisterTables()` method. The latter is very important as it allows us to build up the database schema. In our sample application, we had just one table. Most applications will have more than one, of course. In that case, we can use the `RegisterTable()` several times to create more than one table, as shown next:

```
protected override List<ITableDefinition> RegisterTables()
{
    return new List<ITableDefinition>()
    {
        CreateTableDefinition<Note, int>(n => n.Id)
            .WithIndex<Note, int, int>(NOTE_NOTEID, fd => fd.Id),
        CreateTableDefinition<NoteTag, int>(n => n.Id)
            .WithIndex<NoteTag, int, int>
                (NOTETAG_NOTETAGID, fd => fd.Id)
    };
}
```

Note that in one application, we aren't limited to just one database. Each database is simply another class.

The Sterling engine

Once the database is created, an engine needs to be instantiated. In the sample code, we did this in the `DatabaseService.ActivateEngine()` method. The engine is the heart of Sterling, so activating the engine should be the first step. Once up and running, we need to register the database(s) with the engine.

The engine uses what is known as a **database driver**. The default one is an in-memory driver, which never saves to isolated storage but instead keeps all data in-memory. This isn't all that useful; we'll mostly want to store and persist data, just like with regular isolated storage. We can achieve this by passing an `IsolatedStorageDriver` instance to the registration of the database.

When we are finished with the application, we need to dispose of the database object. To do so properly, we wrote a `DeactivateEngine()` method that is called from the `ApplicationExit` event, which is called always when the Silverlight application shuts down. Of course, all the data that is inside our database remains intact in the isolated storage.

Saving and loading data

A database isn't very useful without its ability to save and load data. Saving data with Sterling is very easy. The following line of code allows us to save a note:

```
DatabaseService.CurrentDatabase.Save(note);
```

Sterling itself will decide if we are updating or inserting data, based on the lambda expression we defined for the key (`n => n.Id`). If the key already exists, it will perform an update to the already existing `Note` instance. Otherwise, a new instance will be created.

After the call to the `Save()` method, we call the `Flush()` method on the database. This causes the data to be written. When doing a single save, it's recommended to call the `Flush()` directly after. If we are doing more than one save operation, we can call the `Flush()` at the end.

Loading data is simple as well. We can use the following line to retrieve a single `Note` instance:

```
var myNote = DatabaseService.CurrentDatabase.Load<Note>(1);
```

The only things we need to specify are the type (here, the `Note` type) and the key. Sterling will retrieve the selected `Note`, if present.

Next to loading by ID, we have access to LINQ-to-Objects, allowing us to perform queries on the data. The following sample query retrieves all IDs greater than two:

```
var range =
    from n in
        DatabaseService.CurrentDatabase.Query<Note, int>()
    where n.Key > 2
    orderby n.Key
    select n.LazyValue.Value;
```

The reason this works is that keys and indexes, like we created with the `RegisterTable()` method, are in-memory and are exposed as a list. On this list, we can then execute LINQ-to-Objects code. An important aspect to this is that the actual objects aren't loaded (in our case, the `Notes` instances) until we really need them. If we would add a search option in the application that allows searching on `NoteTitle`, a simple index would be enough since it would cause this list of `NoteTitles` to be in-memory, thus allowing us to perform a query onto the data, without loading all the `Note` data.

Triggers

In some cases, we may want some more control over how the saving of an instance is executed. In our case, we don't want to search manually for the next available ID each time we save an instance. For this very purpose, the notion of a trigger exists in Sterling. A trigger is a class that inherits from `BaseSterlingTrigger<TInstanceType, TKeyType>`. In our trigger, we override three methods: `BeforeSave`, `AfterSave`, and `BeforeDelete`.

In our implementation, we check what's the highest available ID in the constructor and in the `BeforeSave`, we add one to that value and set this value as the ID for the instance we are saving.

See also

Take a look at the Sterling site on CodePlex (<http://sterling.codeplex.com>) and the Sterling documentation site (<http://www.sterlingdatabase.com>) for more information.

6

MVVM

In this chapter, we will cover the following topics:

- ▶ Creating a basic MVVM application
- ▶ Using MVVM Light to enable MVVM applications
- ▶ Connecting a View to a ViewModel using a ViewModelLocator
- ▶ Connecting a View to a ViewModel using MEF
- ▶ Using commands to pass your events to the ViewModel
- ▶ Communicating between different ViewModels
- ▶ Leveraging a messenger to wrap application-wide messages

Introduction

MVVM or **Model-View-ViewModel** is a design pattern that has rapidly gained popularity in the XAML community. Based on Martin Fowler's MVP (Model-View-Presenter) pattern, and related to the MVC (Model-View-Controller) pattern, this pattern is targeted at UI development platforms (WPF, Silverlight, but more recently it's also emerging in JavaScript-based solutions) where the developer roles can have quite different requirements: the UX developer (or **devigner**) has another set of responsibilities than the more traditional developer, but they both work on the same code base, at the same time.

In this chapter, you'll learn about what is needed to enable MVVM in your application, how you can use it, and what parts it typically consists of. After this chapter, you should have a thorough understanding of MVVM and how to use it.

Creating a basic MVVM application

Applies to Silverlight 3, 4, 5 and WP7

Before we can start with MVVM, the concepts have to be explained. So we're going to start from scratch, assuming you have no knowledge of MVVM, and you're not using any MVVM-enabling framework or library.

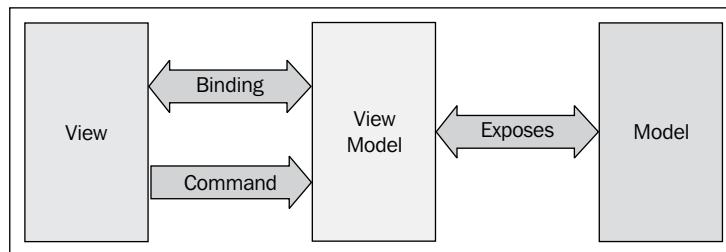
In this recipe, you'll learn about the basic concepts of the MVVM pattern, and you'll see how easy it is to get a bare-bones MVVM application up and running.

Getting ready

The first question that pops up when talking about Model-View-ViewModel is typically: *Why would you want to use it?* The logical answer to that question would be: **separation of concerns**, which in turn leads to **better testability** and **code maintainability**.

But besides that, another important advantage is that it forces you, as a developer, to leverage the power of Silverlight: you have to make good use of Data Binding and the Data Context to get your application to be operational. This, in turn, allows the designer and the developer to work on their own parts of the project, respectively the View and the ViewModel/Model, without too much interference.

To better understand this pattern, let's have a look at the following figure:



In this image, the different parts of the pattern are immediately visible:

- ▶ **The View:** This can be seen as the UI of your application: your Silverlight User Controls or Pages, containing all the UI elements. This is typically designed by the designer.
- ▶ **The ViewModel:** This serves as the **bridge** between the **View** and the **Model**. It's the responsibility of the ViewModel to transform the Model into data that is useable by the View, typically through public properties and commands that can be utilized through the binding system.

- ▶ **The Model:** this should be seen as an **object-based representation** of your data. In some projects, your model comes from generated code: (through proxy generation) or via the client-side version of your entities (when working with RIA Services). However, in other implementations a business layer is injected, for example to parse your DTOs to usable objects. Whichever approach you choose, the main point is: The model is an object-based representation of your data.

To start with this recipe, you can use the starter solution, found in Chapter 6\Basic_MVVM_Starter.

A completed solution can be found in Chapter 6\Basic_MVVM_Completed.

How to do it...

We're starting from the starter solution, and we're going to add the necessary classes and code to create a basic MVVM application. To achieve this, we'll complete the following steps:

1. In the Models folder, add a new class, Person, by right-clicking the folder and selecting **Add | New class**.
2. Implement the class with properties Name, FirstName, and Age as follows:

```
public class Person : NotifyPropertyChangedBase
{
    private string _name;
    /// <summary>
    /// The Name property
    /// </summary>
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
            RaisePropertyChanged("Name");
        }
    }

    private string _firstName;
    /// <summary>
    /// The FirstName property
    /// </summary>
```

```
/// </summary>
public string FirstName
{
    get
    {
        return _firstName;
    }
    set
    {
        _firstName = value;
        RaisePropertyChanged("FirstName");
    }
}

private int _age;
/// <summary>
/// The Age property
/// </summary>
public int Age
{
    get
    {
        return _age;
    }
    set
    {
        _age = value;
        RaisePropertyChanged("Age");
    }
}
```

}

3. In the **ViewModels** folder, add a new class, **viewModel**, by right-clicking the folder and selecting **Add | New class**.

4. Implement the class as follows:

```
using System.ComponentModel;

public class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged(string propertyName)
{
```

```
        if (PropertyChanged != null)
        {
            PropertyChanged.Invoke(this, new PropertyChangedEventArgsA
rgs(propertyName));
        }
    }
}
```

5. In the same folder, create a new class, PersonViewModel, by right-clicking the folder and selecting **Add | New class**.

6. Add the following using statement to this class:

```
using MVVM.Client.Models;
using System;
using System.Collections.ObjectModel;
```

7. In this class, we'll add two properties: People and LastAddedDate. Besides that, we'll also add the code that will be executed to generate the data, and to add a new Person to the People collection:

```
public class PersonViewModel : ViewModel
{
    private ObservableCollection<Person> _people;
    /// <summary>
    /// The People property
    /// </summary>
    public ObservableCollection<Person> People
    {
        get
        {
            return _people;
        }
        set
        {
            _people = value;
            RaisePropertyChanged("People");
        }
    }

    private DateTime? _lastAddedDate;
    /// <summary>
    /// The LastAddedDate property
    /// </summary>
    public DateTime? LastAddedDate
```

```

    {
        get
        {
            return _lastAddedDate;
        }
        set
        {
            _lastAddedDate = value;
            RaisePropertyChanged("LastAddedDate");
        }
    }
}

public PersonViewModel()
{
    LoadData();
}

private void LoadData()
{
    People = DataLoader.GeneratePeople();
}

public void AddPerson()
{
    People.Add(new Person() { FirstName = "Sven", Name =
"Vercauteren", Age = 25 });
    LastAddedDate = DateTime.Now;
}
}

```

8. Open `PeopleView.xaml`, and add a `Button`, `TextBlock`, and `ListBox` to it, bound to the properties we've just created on our `PersonViewModel`:

```

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

    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnAddPerson"
            Content="Add person"

```

```
        Width="200"
        Height="30"
        Click="btnAddPerson_Click"
        Margin="0,0,5,5">></Button>
    <TextBlock Text="{Binding LastAddedDate,
StringFormat='Last person was added on {0}', FallbackValue=' '}"
        Margin="10,0,0,0">

    </TextBlock>
</StackPanel>

<ListBox ItemsSource="{Binding People}"
        Grid.Row="1">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Vertical">
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding Name}" />
                    <TextBlock Text="{Binding FirstName}"
                        Margin="5,0,0,0" />
                </StackPanel>
                <TextBlock Text="{Binding Age}"
                    FontSize="12" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

</Grid>
```

9. Implement it by adding the instantiation of PersonViewModel in Peopleview.xaml.cs and a button-click handler, as such:

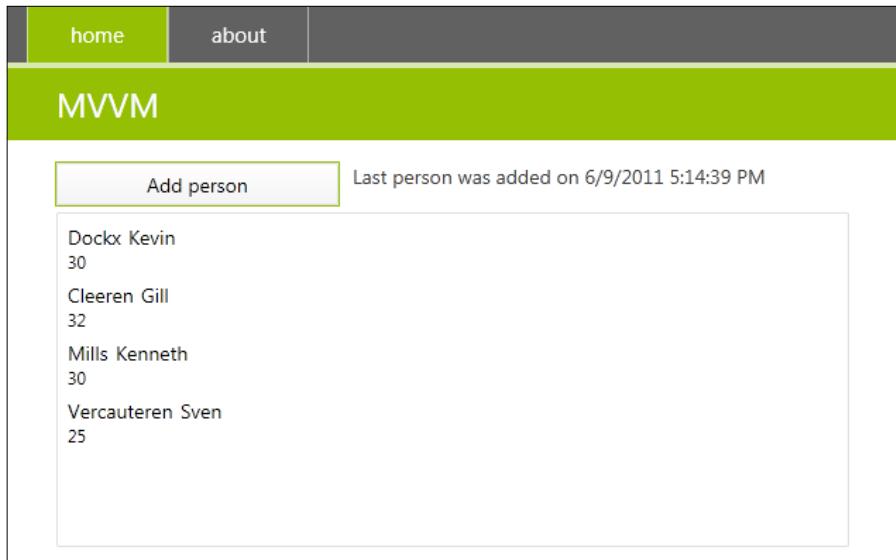
```
public PeopleView()
{
    InitializeComponent();
    this.DataContext = new PersonViewModel();
}

private void btnAddPerson_Click(object sender, RoutedEventArgs e)
{
    ((PersonViewModel)DataContext).AddPerson();
}
```

10. Add a using statement to PeopleView.xaml.cs:

```
using MVVM.Client.ViewModels;
```

11. You can now build and run the application.



How it works...

In step 3, we created a `ViewModel` base class. This class implements the `INotifyPropertyChanged` interface. Through this interface, notifications can be sent when a property is changed, telling Silverlight that the UI should be updated (this of course depends on how you've created your bindings).

After this, we've created a `PersonViewModel` class, inheriting the `ViewModel` base class we've just created, and we've added two properties to it: `People` (an `ObservableCollection<Person>`) and `LastAddedDate`. Both properties raise the `NotifyChanged` event in their property setters, which makes sure the UI is updated when these properties are set or changed.

In this example, our **Model** is the `Person` class, and we fill it with dummy data in the `PersonViewModel` constructor. The `Person` class implements `NotifyPropertyChangedBase` so the UI can be updated when the property is changed.

On to the **View**: our View contains a `ListBox`, bound to the `People` collection, and a `TextBlock`, bound to the `LastAddedDate`. In the View's constructor, the `DataContext` of the View is set to a new instance of our `PersonViewModel` class. This is crucial; as the `DataContext` of the View is now a `PersonViewModel` instance, our binding syntax to properties on the `PersonViewModel` works. In an MVVM pattern, the `ViewModel` is always the `DataContext` of the View.

There's also a button, `btnAddPerson` defined on the View. When this button is clicked, the method `AddNewPerson()` is executed on the `ViewModel`. This adds a new `Person` instance to the `People` list, and sets the `LastAddedDate` to the current date.

Bringing it all together, it works as follows:

- ▶ The View is instantiated, and its `DataContext` is set to a new `PersonViewModel` instance.
- ▶ In the constructor, the `People` collection of `PersonViewModel` is initialized. The `ViewModel` is thus responsible for converting the Model (the `Person` class) to a representation that is useable by the View. Sometimes, the `ViewModel` is called a *converter on steroids*—the reason is obvious.
- ▶ As the properties on the `ViewModel` fire a `NotifyPropertyChanged` event in their setters, the View (which is bound to these properties) is correctly updated.
- ▶ When the **Add Person** button is pressed, a `Person` instance is added to `People`, and the `LastAddedDate` is changed. The View is notified of this change (thanks to the `INotifyCollectionChanged` and `INotifyPropertyChanged` interfaces) and is, again, updated accordingly.

And there we go: our first, very simple MVVM application is up and running. However, this is just the beginning. We've used this application as an example, and a few crucial features are still missing. There are other ways to connect your View to your `ViewModel`, and for the sake of simplicity, **Commanding** (to bind your events, such as the click event, to the `ViewModel`) was omitted. Make sure you read the next recipes for a better understanding of these concepts.

See also

For more on data binding, see *Chapter 2, Introduction to Data Binding* and *Chapter 3, Advanced Data Binding*. For more information on MVVM, have a look at the other recipes in this chapter.

Using MVVM Light to enable MVVM applications

Applies to Silverlight 3, 4, 5 and WP7

Not all necessary building blocks are available out of the box in Silverlight when you want to start using the MVVM pattern. Certain base classes and/or components have to be written—your basic `ViewModel` classes, `ICommand` implementations, a way to connect your View to your `ViewModel`, a way to communicate between `ViewModels`, and so on. Doing all of this by yourself would quickly result in quite a workload.

Due to the popularity of the pattern, quite a few MVVM-enabling frameworks have popped up, of which the best-known are probably Caliburn, Prism (more than just MVVM), and the **MVVM Light Toolkit** by Laurent Bugnion.

In this recipe (and the following ones), we will use the MVVM Light Toolkit to enable MVVM in our projects: it's open, it's lightweight, it's extensible, and it's very easy to use with little to no learning curve. Instead of using project templates (delivered with the MVVM Light Toolkit), we will only use the base classes in these recipes, so as offer a better understanding of how things work and can be set up.

Getting ready

We're going to recreate the application from the previous recipe, *Creating a basic MVVM application*, but this time by using the MVVM Light Toolkit base classes. This means you might have to download the necessary files and install them. They can be found at <http://mvvmlight.codeplex.com/>.

You can start with the provided starter solution, which already includes these necessary dependencies: `Chapter 6\Using_MVVMLight_Starter`.

The completed solution can be found at `Chapter 6\Using_MVVMLight_Completed`.

How to do it...

We're going to create an MVVM application that will display a list of persons and allows us to add a Person to that `ListBox`. To achieve this, we'll complete the following steps:

1. Add a new class to the `ViewModels` folder in `MVVM.Client`. Right-click the folder, select **Add | New class**, and create a new class named `PersonViewModel`, inheriting `ViewModelBase`.

2. Add the following `using` statements to this class:

```
using GalaSoft.MvvmLight;
using MVVM.Client.Models;
```

3. Add two properties to `PersonViewModel`:

```
private ObservableCollection<Person> _people;
/// <summary>
/// The People property
/// </summary>
public ObservableCollection<Person> People
{
    get
    {
        return _people;
    }
    set
    {
        _people = value;
        RaisePropertyChanged("People");
    }
}

private DateTime? _lastAddedDate;
/// <summary>
/// The LastAddedDate property
/// </summary>
public DateTime? LastAddedDate
{
    get
    {
        return _lastAddedDate;
    }
    set
    {
        _lastAddedDate = value;
        RaisePropertyChanged("LastAddedDate");
    }
}
```

-
4. Change the constructor of PersonViewModel and add the LoadData() method:

```
public PersonViewModel()
{
    LoadData();
}
private void LoadData()
{
    People = DataLoader.GeneratePeople();
}
```

5. Add the AddPerson method to PersonViewModel:

```
public void AddPerson()
{
    People.Add(new Person() { FirstName = "Sven"
        , Name = "Vercauteren"
        , Age = 25 });
    LastAddedDate = DateTime.Now;
}
```

6. Open PeopleView.xaml, and implement the bindings to LastAddedDate and People such as:

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnAddPerson"
            Content="Add person"
            Width="200"
            Height="30"
            Click="btnAddPerson_Click"></Button>
        <TextBlock Text="{Binding LastAddedDate,
StringFormat='Last person was added on {0}', FallbackValue=''}"
            Margin="10,0,0,0">
        </TextBlock>
    </StackPanel>
    <ListBox ItemsSource="{Binding People}"
        Grid.Row="1">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel Orientation="Vertical">
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Name}" />
```

```

        <TextBlock Text="{Binding FirstName}"
                   Margin="5,0,0,0" />
    </StackPanel>
    <TextBlock Text="{Binding Age}"
                   FontSize="12" />
    </StackPanel>
</DataTemplate>
<ListBox.ItemTemplate>
</ListBox>
</Grid>

```

7. In the constructor of `PeopleView.xaml.cs`, set its `DataContext` to an instance of `PersonViewModel`:

```

public PeopleView()
{
    InitializeComponent();
    this.DataContext = new PersonViewModel();
}

```

8. Add the following `using` statement to `PeopleView.xaml.cs`:

```
using MVVM.Client.ViewModels;
```

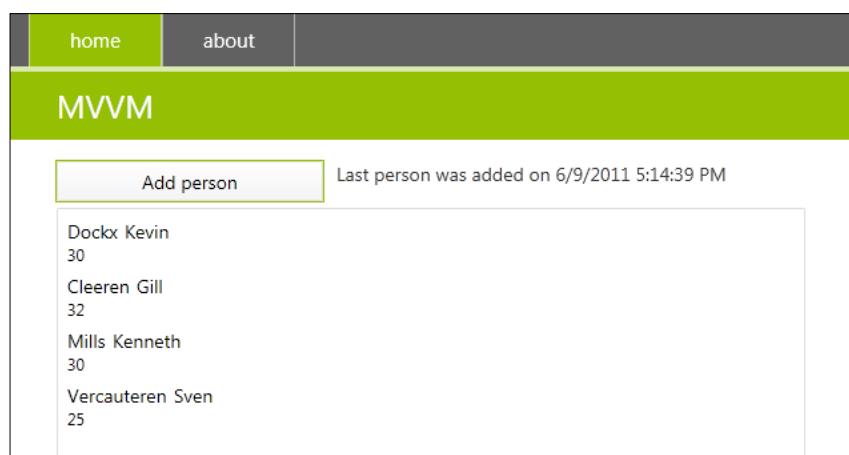
9. Implement the click handler of `btnAddPerson` in `PeopleView.xaml.cs`:

```

private void btnAddPerson_Click(object sender, RoutedEventArgs e)
{
    ((PersonViewModel)DataContext).AddPerson();
}

```

10. You can now build and run the application, and add a `Person` by clicking the **Add person** button.



How it works...

We've started out by creating the `PersonViewModel` class, which inherits MVVM Light's `ViewModelBase` class. Among other things (which will be talked about in later recipes), this `ViewModelBase` class implements `INotifyPropertyChanged`. Through this interface, notifications can be sent when a property is changed, telling Silverlight that the UI should be updated (this of course depends on how you've created your bindings).

In the `PersonViewModel` class, we've added two properties: `People` (an `ObservableCollection<Person>`) and `LastAddedDate`. Both properties raise the `NotifyChanged` event in their property setters, which makes sure the UI is updated when these properties are set or changed.

The starter solution already includes a `Person` class (inheriting `NotifyPropertyChangedBase`, which implements `INotifyPropertyChanged`). This is the business object that is our model. In the `PersonViewModel`, we fill this with dummy data by calling `DataLoader.GeneratePeople()` via the constructor.

On to the View—`PeopleView`. `PeopleView` contains a `ListBox`, bound to the `People` collection, and a `TextBlock`, bound to the `LastAddedDate`. In the View's constructor, the `DataContext` of the View is set to a new instance of our `PersonViewModel` class. This is crucial; as the `DataContext` of `PeopleView` is now a `PersonViewModel` instance, our binding syntax to properties on the `PersonViewModel` works. In an MVVM pattern, the `ViewModel` is always the `DataContext` of the View. There's also a button, `btnAddPerson`, defined on the View. When this button is clicked, the method `AddNewPerson()` is executed on the `ViewModel`. This adds a new `Person` to the `People` list, and sets the `LastAddedDate` to the current date.

Bringing it all together, it works as follows:

- ▶ The View is instantiated, and its `DataContext` is set to a new `PersonViewModel` instance.
- ▶ In the constructor, the `People` collection of `PersonViewModel` is initialized. The `ViewModel` is thus responsible for converting the Model (the `Person` class) to a representation that is useable by the View. Sometimes, the `ViewModel` is called a *converter on steroids*—the reason is obvious.
- ▶ As the properties on the `ViewModel` fire a `NotifyChanged` event in their setters, the View (which is bound to these properties) is correctly updated.
- ▶ When the **Add Person** button is pressed, a `Person` instance is added to `People`, and the `LastAddedDate` is changed. The View is notified of this change (thanks to the `INotifyCollectionChanged` and `INotifyPropertyChanged` interfaces) and is, again, updated accordingly.

There's more...

This is just the beginning, we've now got a very basic MVVM application up and running, but a few crucial features are still missing—connecting the View to the ViewModel can be done differently (built-in in MVVM Light), Commanding can be used to bind Events to Commands in the ViewModel, and there's a way to send messages between ViewModels. Make sure you read the next recipes for a better understanding of these concepts.

See also

For more information on MVVM, have a look at the other recipes in this chapter.

Connecting a View to a ViewModel using a ViewModelLocator

Applies to Silverlight 3, 4, 5 and WP7

One of the first things you'll have to decide on when working with the MVVM design pattern is how you're going to connect your Views to your ViewModels. The simplest method by far is by just instantiating the ViewModel in the View's constructor, but this of course breaks the idea that we're trying to work as loosely coupled as possible (even though a View and ViewModel are somewhat coupled by design, as a ViewModel needs to have the properties used in the View's binding syntax).

The approach used in MVVM Light is that of the `ViewModelLocator`. In this recipe, we'll learn how to use this `ViewModelLocator` to tie your View and ViewModel together.

Getting ready

We're starting with the solution we completed in the previous recipe, *Using MVVM Light to enable MVVM applications*. Or alternatively, you can start with the provided starter solution, which can be found at `Chapter 6\Connecting_View_ViewModel.ViewModelLocator_Starter`.

The completed solution can be found in `Chapter 6\Connecting_View_ViewModel.ViewModelLocator_Completed`.

How to do it...

We're going to adjust the solution from the previous recipe, *Using MVVM Light to enable MVVM applications*, or the starter solution, so it uses the `ViewModelLocator` to tie the View and ViewModel together. To achieve this, we should complete the following steps:

1. Add a new class to the `ViewModel` namespace by right-clicking the `ViewModel` directory and selecting **Add new class**. Name this class `ViewModelLocator`.

2. Implement the class as such:

```
public class ViewModelLocator
{
    private static PersonViewModel _personVM;

    /// <summary>
    /// Initializes a new instance of the ViewModelLocator class.
    /// </summary>
    public ViewModelLocator()
    {
        CreatePersonVM();
    }

    /// <summary>
    /// Gets the PersonVM property.
    /// </summary>
    public static PersonViewModel PersonVMStatic
    {
        get
        {
            if (_personVM == null)
            {
                CreatePersonVM();
            }
            return _personVM;
        }
    }

    /// <summary>
    /// Gets the PersonVM property.
    /// </summary>
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.
Performance",
    "CA1822:MarkMembersAsStatic",
    Justification = "This non-static member is needed for data
binding purposes.")]
```

```
public PersonViewModel PersonVM
{
    get
    {
        return PersonVMStatic;
    }
}

/// <summary>
/// Provides a deterministic way to delete the PersonVM
property.
/// </summary>
public static void ClearPersonVM()
{
    _personVM.Cleanup();
    _personVM = null;
}

/// <summary>
/// Provides a deterministic way to create the PersonVM
property.
/// </summary>
public static void CreatePersonVM()
{
    if (_personVM == null)
    {
        _personVM = new PersonViewModel();
    }
}

/// <summary>
/// Cleans up all the resources.
/// </summary>
public static void Cleanup()
{
    ClearPersonVM();
}
```

3. Open App.xaml, and add the following xmlns import:

```
xmlns:vm="clr-namespace:MVVM.Client.ViewModels"
```

4. Add the following code to `App.xaml`, right below the `<ResourceDictionary>` statement:

```
<vm:ViewModelLocator x:Key="Locator" />
```

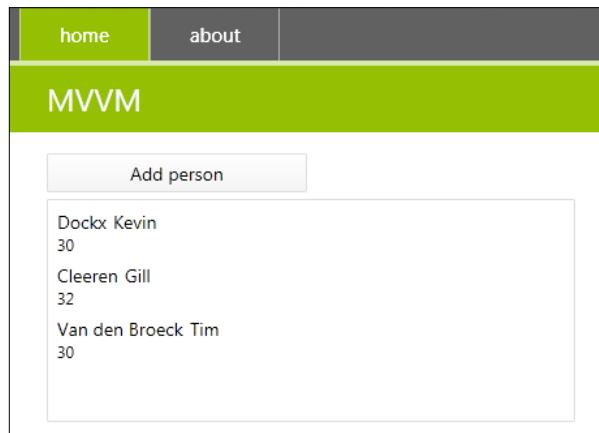
5. Open `PeopleView.xaml.cs`, and change the constructor to this:

```
public PeopleView()
{
    InitializeComponent();
}
```

6. Open `PeopleView.xaml`, and add the following code right below the `xmlns` import statements:

```
DataContext="{Binding Source={StaticResource Locator},
Path=PersonVM}"
```

7. You can now build and run your application.



How it works...

The `ViewModelLocator` allows us to use binding syntax in the View to tie it to the `ViewModel`. The `ViewModelLocator` itself has a property `PersonVM`. When this property's getter is called, it ties back to a static property, `PersonVMStatic`, in which the `PersonViewModel` is instantiated.

With that in mind, we now make the `ViewModelLocator` available for binding by adding it as a `StaticResource` to the application, in the `App.xaml` `ResourceDictionary`. If we now use the binding syntax we've added to `PeopleView.xaml`, the following will happen:

- ▶ The View is instantiated once we navigate to it.

- ▶ The View's DataContext is bound to a property of our ViewModelLocator, PersonVM. This binding works because the ViewModelLocator is made available through the App.xaml ResourceDictionary.
- ▶ PersonVM ties back to PersonVMStatic. This in turn calls the CreatePersonVM() method, which returns a new instance of PersonViewModel. This instance is now the DataContext of our PeopleView, so the bindings all work.

Important to know is that, every time we re-navigate to PeopleView, the same PersonViewModel instance is reused. This is different from how it worked in the previous recipe, *Using MVVM Light to enable MVVM applications*. There, the PersonViewModel was re-instantiated every time.

Typically, in a Silverlight application you want to save the state of a View (kept through the ViewModel), so reusing this PersonViewModel instance is expected, and mostly required, behavior.

Finally, if you need to clean up the PersonViewModel, you can use the respective Cleanup() method.

There's more...

In MVVM, there are two ways of binding your View to your ViewModel. What we've done here is called the View First-approach—when the View gets instantiated, the ViewModel gets instantiated (or reused). In this case, a View is quite strictly tied to a specific ViewModel—we're essentially saying 'View A will always have ViewModel A as its DataContext' (have a look at the next recipe, *Connecting a View to a ViewModel using MEF*, for a more flexible approach).

The ViewModel First-approach works just the other way around—the ViewModel is instantiated, and this takes care of instantiating the View for which it is the DataContext.

There is no real right or wrong way to do this—both approaches are valid, and the one you'll choose typically depends on the needs of your application or architecture. The View First-approach is often done with Data Binding and a ViewModelLocator (as seen in this recipe). ViewModel First is typically achieved through the use of an Inversion of Control container—the ViewModel is instantiated by passing in the View (or rather, a contract the View must obey) in its constructor. This easily enables you to, for example, reuse the same ViewModel for different Views.

Have a look at the following code:

```
public PersonViewModel(IView view)
{
    // execution
}
```

This is a ViewModel first-approach in which any View (abiding by the `IView` contract) can be injected and used.

Other approaches and mixes of both exist—for example, to reach the ability to use a View First-approach in which one View can be tied to a variety of ViewModels, depending on certain rules, you can use the approach detailed in the next recipe, *Connecting a View to a ViewModel using MEF*.

See also

For more information on MVVM, have a look at the other recipes in this chapter.

Connecting a View to a ViewModel using MEF

Applies to Silverlight 4 and 5

In the previous recipe, we used the `ViewModelLocator` to connect our Views to our ViewModels. Another approach you can take is to use the **Managed Extensibility Framework (MEF)** to do this for you.

One of the advantages of this approach is that besides relaying the responsibility of providing the ViewModels to MEF, you can now use multiple ViewModel instances of the same type without having to create a `ViewModelLocator` property for each, for example: you can use this approach if you don't know in advance how many instances you'll need (amongst some other advantages, which will be described in this recipe).

In this recipe, you'll learn how to use MEF for tying your Views to your ViewModels.

Getting ready

We're starting with the solution we completed in the previous recipe, *Connecting a View to a ViewModel using a ViewModelLocator*. Or alternatively, you can start with the provided starter solution, which can be found in `Chapter 6\Connecting_View.ViewModel.MEF_Starter`.

The completed solution can be found in `Chapter 6\Connecting_View.ViewModel.MEF_Completed`.

How to do it...

We're going to adjust the solution from the previous recipe, *Connecting a View to a ViewModel using a ViewModelLocator*, or the starter solution, so it uses MEF to tie the View and ViewModel together. To achieve this, we should complete the following steps:

1. Right-click on **MVVM.Client**, select **Add Reference**, and add references to `System.ComponentModel.Composition` and `System.ComponentModel.Composition.Initialization`.
2. Remove `ViewModelLocator.cs` from `MVVM.Client`.
3. Open `App.xaml`, and remove the `ViewModelLocator` resource:

```
<vm:ViewModelLocator x:Key="Locator" />
```
4. Add a new folder to `MVVM.Client`, **Contracts**, by right-clicking `MVVM.Client` and selecting **Add | New Folder**. In this folder, add a new interface, `IPersonViewModel`, as such:

```
public interface IPersonViewModel
{
}
```
5. Open `PersonViewModel.cs`, and add the following `using` statements:

```
using System.ComponentModel.Composition;
using MVVM.Client.Contracts;
```
6. Add the following attributes to the class signature:

```
[Export(typeof(IPersonViewModel))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class PersonViewModel : ViewModelBase
```
7. Open `PeopleView.xaml` and remove the `DataContext` binding statement:

```
DataContext="{Binding Source={StaticResource Locator},
Path=PersonVM}"
```
8. Open `PeopleView.xaml.cs`, and add the following `using` statements:

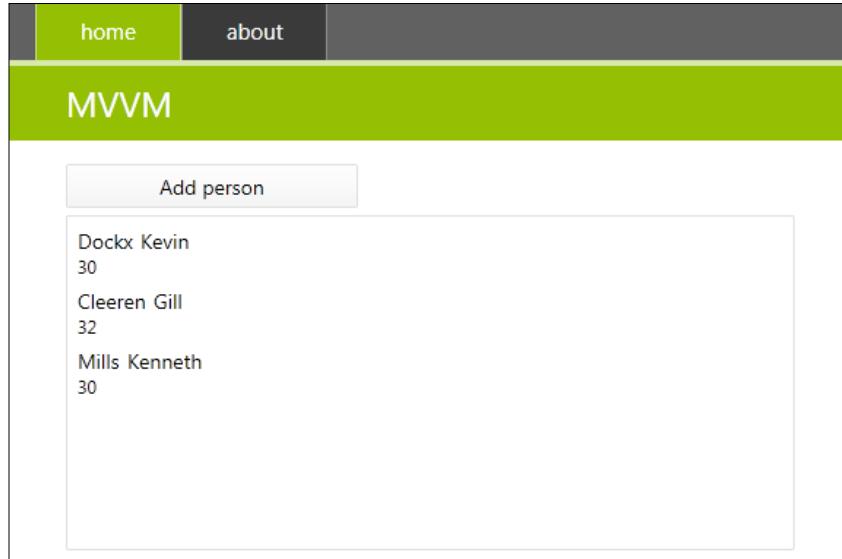
```
using GalaSoft.MvvmLight;
using System.ComponentModel.Composition;
using MVVM.Client.Contracts;
```
9. Change the constructor and add a `ViewModel` property as such (`IsInDesignModeStatic` isn't necessary, but it's added to ensure the design view still works):

```
public PeopleView()
{
    InitializeComponent();
    ViewModel = new PersonViewModel();
}
```

```
if (!ViewModelBase.IsInDesignModeStatic)
{
    // load VM through MEF
    CompositionInitializer.SatisfyImports(this);
}

[Import(typeof(IPersonViewModel))]
public ViewModelBase ViewModel
{
    set
    {
        this.DataContext = value;
    }
    get
    {
        return this.DataContext as ViewModelBase;
    }
}
```

10. You can now build and run your application.



How it works...

In our `PeopleView.xaml.cs`, we've added a property `ViewModel`, which is decorated with an `import` statement as such:

```
[Import(typeof(IPersonViewModel))]  
public ViewModelBase ViewModel
```

With this `import` statement, we're stating that the `ViewModel` property should be of type `IPersonViewModel`. In the constructor of the `PeopleView`, we've added a call to `CompositionInitializer.SatisfyImports(this)`. This means that whenever the View is instantiated, all the imports in this View should be satisfied. As we've added an `import` statement on the `ViewModel` property, that import will be satisfied.

To satisfy this, MEF will go and look for anything that has been exported as an `IPersonViewModel` (which is used as a marker interface—regular strings are also accepted, but are prone to type errors). If it finds one, it will instantiate (the composition), and set it as the value of the `ViewModel` property in `PeopleView.xaml.cs`. The setter of that `ViewModel` property sets the View's `DataContext` to that imported value, for example: to an instance of anything that is exported as `IPersonViewModel`.

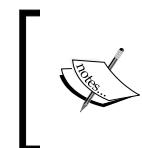
In this case, we've exported the `PersonViewModel` as something of type `IPersonViewModel`. This means that MEF will set the `ViewModel` property on our View to an instance of `PersonViewModel`. That is thus the `DataContext` of our View, so the View and `ViewModel` are connected now, and everything works as it should.

There's more...

There's a lot more to MEF than this simple example, but let's start with an explanation of what MEF does. MEF, or the Managed Extensibility Framework, was designed to simplify the design of extensible applications and components. A lot of applications have requirements that change frequently, and it might become difficult to extend the application with new functionality after some amount of time; MEF tries to solve this problem by making it easy to plug new functionality into your application.

- ▶ The first keyword is **Composition**: you start by telling MEF where it can find the parts of the application it needs to provide by creating a catalog (this might be, for example, a catalog of all types in the currently executing assembly). From this catalog, you then create a `CompositionContainer`. This container will do the work for you from now on.
- ▶ The second keyword is **Export**: by exporting a component with the `Export` attribute, you tell MEF you want to export this component (type, property), so it can be composed via the container.

-
- ▶ The third keyword is **Import** (or **ImportMany**), which tells MEF what type of components should be imported via the container in a specific part of your application.



These three simple concepts are the heart and brains of MEF. While this isn't a book on MEF, this should give you a basic understanding of what MEF is and what it does. A lot more information can be found at <http://mef.codeplex.com/>.



With that in mind, we can look at some of the specifics of this recipe: what about the `PartCreationPolicy` attribute we've used on `PersonViewModel`? We've exported our `ViewModel` with a `CreationPolicy` of `Shared`—this means the same `ViewModel` instance will be reused whenever it is asked for by an `Import` statement. However, if you want to ensure new instances are used every time (typically the case when you want to reuse the same View/ViewModel combination for an unknown number of times), you can make it `NonShared`. Have a look at the following code example:

```
[Export(typeof(IPersonViewModel))]  
[PartCreationPolicy(CreationPolicy.NonShared)]  
public class PersonViewModel : ViewModelBase
```

The preceding code will ensure a new instance is created each time an `Import` on this type is called.

But what about tying different Views to the same `ViewModel` type? Well, the same approach can be used: identify your `ViewModel` as `Exported` in a `Shared` way, and the same `ViewModel` instance (holding the state) will be reused as `DataContext` for whichever View imports it. This makes it very easy to reuse the same underlying data, but presenting it in a different way. And if you need different instances of the same `ViewModel` type for different Views, simply export the `ViewModel` as `NonShared`.

Note that, in essence, we're using MEF for **dependency injection** in this approach. Not everyone (as with almost all patterns, frameworks, and building blocks) agrees that this is a correct approach, or something that should be done. If you don't want to use MEF, you can easily use an IoC container, such as Castle (<http://www.castleproject.org/>) or Ninject (<http://ninject.org/>), to get this result; that will, however, require external references to those containers, while MEF is part of the core .NET Silverlight CLR.

MEF is not an Inversion of Control container, but it of course uses IoC to do its business. And it even allows you to do more than your typical IoC container: an IoC container typically works on known dependencies. MEF was designed to manage your unknown dependencies, hence the plugability promise. However, that doesn't mean you can't manage your known dependencies with it as well—at least up to a certain point. Some applications use MEF as a replacement for their standard IoC containers like Castle or Ninject, other applications use both: an IoC container for known dependencies, MEF for unknown ones. The choice is yours, but for more demanding, enterprise-grade applications, a mix of an IoC container and MEF is often the way to go.

See also

For more information on MVVM, have a look at the other recipes in this chapter.

Using commands to pass your events to the ViewModel

Applies to Silverlight 3, 4, 5 and WP7

Up until now, we've passed through click events (or other events) by setting up a handler in the code behind of our View. In this view, we then cast the DataContext to a specific `ViewModel` type, which contains the implementation of the handler.

This is easy, but it's not the way to go in MVVM. By doing this, you're adding in some extra tight coupling between your View and ViewModel, which is exactly what MVVM tries to avoid. A better way to tackle this problem is by using commands. This will allow you to bind an event on your View to a command on your ViewModel, thus eliminating the need to pass through events via the code behind your View.

In this recipe, we'll see how we can use commands to pass events to the ViewModel.

Getting ready

We're starting with the solution we completed in the recipe *Connecting a View to a ViewModel using a ViewModelLocator*. Or alternatively, you can start with the provided starter solution, which can be found in `Chapter 6\Using_Commands_Starter`.

The completed solution can be found in at `Chapter 6\Using_Commands_Completed`.

How to do it...

We're going to adjust the solution from the recipe *Connecting a View to a ViewModel using a ViewModelLocator*, or the starter solution, to use commanding. To achieve this, we should complete the following steps:

1. Open `PeopleView.xaml.cs`, and remove the button-click handler.
2. Open `PeopleView.xaml`, locate the **Add Person** button, and change the tag as follows:

```
<Button x:Name="btnAddPerson"
        Content="Add person"
        Width="200"
        Height="30"
        Command="{Binding AddPersonCommand}"></Button>
```

-
3. Open PersonViewModel.cs, and remove the AddPerson() method.

4. Add the following using statement to the top of PersonViewModel.cs:

```
using GalaSoft.MvvmLight.Command;
```

5. Add the following command definition:

```
public RelayCommand AddPersonCommand
{
    get;
    private set;
}
```

6. Add an InstantiateCommand() method with the following implementation, and change the constructor so it calls this method:

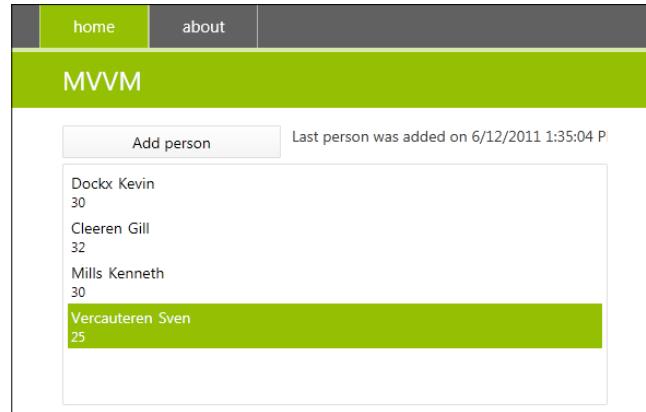
```
public PersonViewModel()
{
    InstantiateCommands();
    LoadData();
}

private void InstantiateCommands()
{
    AddPersonCommand = new RelayCommand(() =>
AddPersonExecution(), () => true);
}
```

7. Add the implementation of AddPersonExecution as such:

```
private void AddPersonExecution()
{
    People.Add(new Person() { FirstName = "Sven", Name =
"Vercauteren", Age = 25 });
    LastAddedDate = DateTime.Now;
}
```

8. You can now build and run your application.



How it works...

The Command property of a Button can be bound to any class that implements `ICommand`, so that's the first thing we need—an implementation of `ICommand`. `ICommand` looks as follows:

```
public interface ICommand
{
    // Summary:
    //      Occurs when changes occur that affect whether the command
    //      should execute.
    event EventHandler CanExecuteChanged;

    // Summary:
    //      Defines the method that determines whether the command can
    //      execute in its
    //      current state.
    //
    // Parameters:
    //      parameter:
    //          Data used by the command. If the command does not require
    //          data to be passed,
    //          this object can be set to null.
    //
    // Returns:
    //      true if this command can be executed; otherwise, false.
    bool CanExecute(object parameter);
    //
    // Summary:
}
```

```
//      Defines the method to be called when the command is
invoked.
//
// Parameters:
//   parameter:
//      Data used by the command. If the command does not require
data to be passed,
//      this object can be set to null.
void Execute(object parameter);
}
```

To have a valid implementation of `ICommand`, we'd need to implement an `Execute` method, a `CanExecute` method, and an `EventHandler`. If needed, you can do this by yourself, but MVVM Light already includes an implementation of this interface, named `RelayCommand`, so we can use that one.

When you instantiate a `RelayCommand`, two parameters can be passed into the constructor: the first one is of type `Action`—typically, you will pass in (a call to) the code that is to be executed when the command is activated. The second parameter, a `Func<bool>`, allows you to add a condition, which will state whether or not the command can be executed. If this `Func` returns `true`, the command will be executed, if it returns `false`, it will not. This last one is an optional parameter. If you do not pass it in, it will return `true` by default, so the command will always be executed.



There's also a `RelayCommand` implementation that accepts a typed parameter—have a look at the *There's More...* section for more information on this.

Back to our solution: we've created a command on our `PersonViewModel`, `AddPersonCommand`, and provided it with code that must be executed (`AddPersonExecution`) and a `Func<bool>` that always returns `true` (meaning the command will always be executed).

In `PersonView`, we've added a `Command={Binding AddPersonCommand}` property to the **Add Person** button tag.

By doing these two things, we've successfully implemented commanding in our project: when the user clicks the button, the `AddPersonExecution` method will be executed.

There's more...

The approach we've used here is easy and fast. Use the `Command` property on the `Button` and bind a command to it, which will get executed when you click the button. However, not all controls have a `Command` property (the controls that inherit from `ButtonBase` do). And in some cases, you might want to bind to another event than the `Click` event. Examples are binding to a `SelectionChanged` event on a `ListBox`, or to a `Loaded` event on a `Button`.

In these cases, another approach can be used: you can use the `triggers` collection from the `System.Windows.Interactivity` assembly.

First, import the `System.Windows.Interactivity` assembly (add a reference to it if there isn't one already—you will have it if you've installed Microsoft Expression Blend, or alternatively, it's provided in the `Dependencies` folder of the solution provided with this recipe) and import the `Command` namespace from MVVM Light:

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
xmlns:cmd="clr-namespace:GalaSoft.MvvmLight.Command;assembly=GalaSoft.MvvmLight.Extras.SL4"
```

Then, change the `Button` tag as such:

```
<Button x:Name="btnAddPerson"
        Content="Add person"
        Width="200"
        Height="30">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Click">
            <cmd:EventToCommand Command="{Binding AddPersonCommand}"></cmd:EventToCommand>
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

With this syntax, you can bind any event to a command. The `Interaction.Triggers` collection can contain multiple event triggers: you can bind different commands to different events and handle the same event through different commands.

Besides a regular command, you also have commands that allow you to pass in a parameter. For example, you might want to pass through the selected item on the command used to handle the `SelectionChanged` event of a `ListBox`. This is possible as well. Let's take our completed solution, and change the following:

1. In `PeopleView`, change the `ListBox` opening tag to this:

```
<ListBox ItemsSource="{Binding People}"
        Margin="0,5,0,0"
```

```

        x:Name="lstPeople"
        Grid.Row="1">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="SelectionChanged">
            <cmd:EventToCommand Command="{Binding
SelectedPersonChangedCommand}"
                CommandParameter="{Binding
SelectedItem, ElementName=lstPeople}"></cmd:EventToCommand>
        </i:EventTrigger>
    </i:Interaction.Triggers>

```

2. In PersonViewModel, define a new command, SelectedPersonChangedCommand:

```

public RelayCommand<Person> SelectedPersonChangedCommand
{
    get;
    private set;
}

```

3. Instantiate and implement it as such:

```

private void InstantiateCommands()
{
    ...
    SelectedPersonChangedCommand = new RelayCommand<Person>((p) =>
SelectedPersonChangedExecution(p), (p) => true);
}

private void SelectedPersonChangedExecution(Person selectedPerson)
{
    // implementation
}

```

Now, when the selection in the ListBox is changed, the SelectedPersonChangedExecution method will be executed, passing in the newly selected person.

4. Lastly, MVVM Light also allows you to pass through the event parameters as command parameters. To achieve this, you can use the following code:

```

<Button x:Name="btnAddPerson"
        Content="Add person"
        Width="200"
        Height="30">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Click">

```

```
<cmd:EventToCommand Command="{Binding  
AddPersonCommand}"  
PassEventArgsToCommand="True" ></  
cmd:EventToCommand>  
</i:EventTrigger>  
</i:Interaction.Triggers>  
</Button>
```

See also

For more information on MVVM, have a look at the other recipes in this chapter.

Communicating between different ViewModels

Applies to Silverlight 3, 4, 5 and WP7

Often, you'll find yourself in need of sending messages between ViewModels. A very typical example would be: you've got a View with a `ListBox`, and once an item is selected, you want that item to be displayed in detail in another View. In other words, the standard master-detail requirement.

How do you achieve this with MVVM? If you didn't follow the pattern, you could easily say: "well, I need to execute a method on ViewModel B when something happens in ViewModel A, so I'll keep a reference to ViewModel B in ViewModel A, so I can call the method when appropriate". This, of course, is **tight coupling**—your ViewModels aren't independent of each other anymore, and ViewModel A now knows that ViewModel B exists. What's more: if ViewModel B is removed, or heavily changed somehow, you'll need to update ViewModel A as well.

This is, of course, not a good approach. In this recipe, we'll see how to tackle this problem. We will use MVVM Light's Messenger to communicate between ViewModels.

Getting ready

We're starting with the provided starter solution, which can be found in `Chapter 6\Communicating_Between_ViewModels_Starter`.

The completed solution can be found in `Chapter 6\Communicating_Between_ViewModels_Completed`.

How to do it...

We're going to start from the provided starter solution, which includes two Views (`PeopleView` and `PersonEditView`) with their respective ViewModels, and we're going to ensure that once a person is selected in `PeopleView`, it is sent to `PersonEditView` and displayed on that View. We will use MVVM Light's Messenger for this. To achieve this, we'll complete the following steps:

1. Open `LocalStateContainer.cs` and add the following:

```
public static Messenger PersonMessenger = new Messenger();
```

2. Right-click on the Messages folder, select **Add New class**, and add a class named `PersonSelectedMessage`. Implement it as such:

```
public class PersonSelectedMessage
{
    public Person SelectedPerson { get; set; }
    public PersonSelectedMessage(Person person)
    {
        SelectedPerson = person;
    }
}
```

3. Open `PersonViewModel.cs`, and add the following code to the `InstantiateCommands()` method:

```
SelectedPersonChangedCommand = new RelayCommand<Person>((p) => SelectedPersonChangedExecution(p), (p) => true);
```

4. Add the following method:

```
private void SelectedPersonChangedExecution(Person selectedPerson)
{
    // send a message to notify anyone who's registered that the
    selection
    // has changed
    LocalStateContainer.PersonMessenger.
    Send<PersonSelectedMessage>(new PersonSelectedMessage(selectedPerson));
}
```

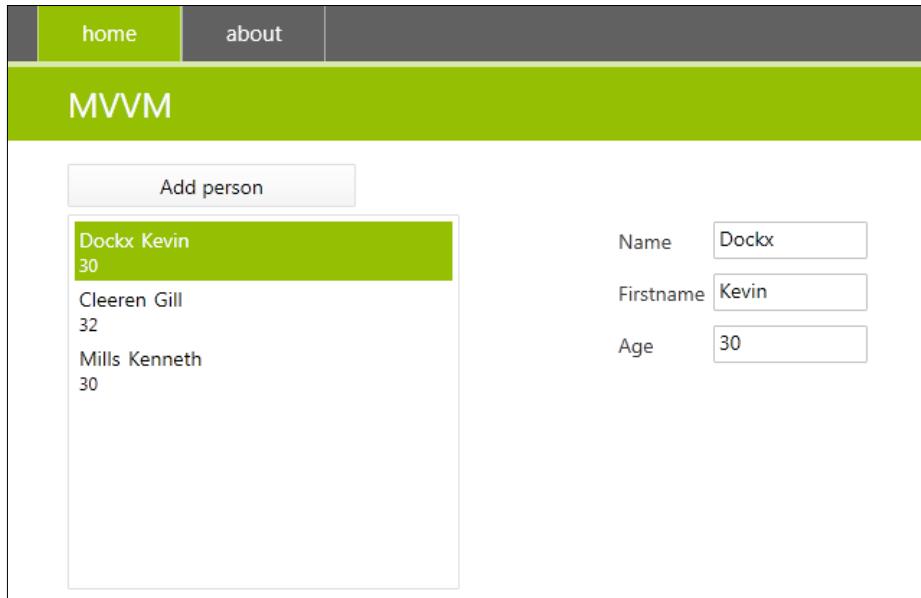
5. Open `PersonEditViewModel.cs`, and add the following method:

```
private void InstantiateMessenger()
{
    LocalStateContainer.PersonMessenger.Register<PersonSelectedMessage>(this, (msg) => PersonSelectedMessageReceived(msg));
}
```

6. Call this method from the PersonEditViewModel constructor.
7. Add the following method:

```
private void PersonSelectedMessageReceived
(PersonSelectedMessage msg)
{
    CurrentPerson = msg.SelectedPerson;
}
```

8. You can now build and run your application.



How it works...

The Messenger is a classic messaging service provided by MVVM Light. It works via **subscription/broadcasting** principles. If you want to receive a message of a certain type, you have to subscribe yourself to receive that type of message through the messenger. Once a message of the type you've subscribed to is broadcasted through that same messenger, you will receive it and can execute the necessary code.

In our example, the Messenger mediates between PersonViewModel and PersonEditViewModel. They can communicate with each other, without knowing each other's existence. Like this, our loose coupling is kept in place.

In the first steps of our example, we're instantiating a Messenger: PersonMessenger. This instance will be used to send messages between PersonViewModel and PersonEditViewModel.

We want to pass a `Person` instance from `PersonViewModel` to `PersonEditViewModel`, so we start out by creating a `PersonSelectedMessage`, which can be instantiated by passing in a `Person` instance. The `PersonEditViewModel` must be able to receive a message of this type, and execute the necessary code to set its `CurrentPerson` property to the `Person` instance it received. That's what the code in steps 5 and 6 is for: we're using our `PersonMessenger` instance, and telling it that it should register this `ViewModel` to receive messages of type `PersonSelectedMessage`. When such a message is received, the `PersonSelectedMessageReceived` method is executed.

Only one thing is left now—sending the actual message. We're handling the `SelectionChanged` event of the `ListBox` on `PeopleView` through commanding and, in the handler, we're creating a new message of type `PersonSelectedMessage`, instantiating it by passing in the currently selected `Person`. This is then broadcasted via `PersonMessenger`, and will be received by our `PersonEditViewModel`, as this `ViewModel` is registered to receive messages of that type from our `PersonMessenger`.

There's more...

The Messenger we've used in this case was an instance we created to communicate between two specific `ViewModels`: `PersonViewModel` and `PersonEditViewModel`. This is perfect if you're sure certain messages should only be sent between a specific set of `ViewModel` instances (for example, if your application consists of a Reporting module and a User Management module, messages concerning the one or the other will typically only be sent between `ViewModels` belonging to that specific module), you would use a self-created Messenger instance.

You've also got the possibility to send application-wide messages, as every `ViewModel` in MVVM Light has access to the default Messenger. Have a look at the recipe *Leveraging a messenger to wrap application-wide messages*, for an implementation of this.

See also

For more information on MVVM, have a look at the other recipes in this chapter.

Leveraging a messenger to wrap application-wide messages

Applies to Silverlight 3, 4, 5 and WP7

Sending messages between `ViewModel` instances is something that will have to be done in almost every application, as the previous recipe, *Communicating between different ViewModels*, explained in detail. However, most applications also need a way to send general messages, which should be able to be sent from everywhere in the application to a specific place. One fine example would be an error-handling message, and how to show this via a `MessageBox` to the user.

In MVVM Light, every `ViewModel` has access to the default `Messenger` instance and a few built-in message types, which can be used for this purpose. In this recipe, we'll learn how to do that.

Getting ready

We're starting from the starter solution provided in Chapter 6\Leveraging_Messenger_Starter.

The completed solution can be found in Chapter 6\Leveraging_Messenger_Completed.

How to do it...

We're going to adjust the starter solution so it will accept `DialogMessages` in a central location in our application. To achieve this, we'll complete the following steps:

1. Open `MainPage.xaml.cs` and add the following code to the constructor:

```
Messenger.Default.Register<DialogMessage>(this, (msg) =>
HandleDialogMessage(msg));
```

2. Add the following code:

```
private void HandleDialogMessage(DialogMessage msg)
{
    MessageBoxResult msgBox = MessageBox.Show(msg.Content,
"Error", msg.Button);
    msg.ProcessCallback(msgBox);
}
```

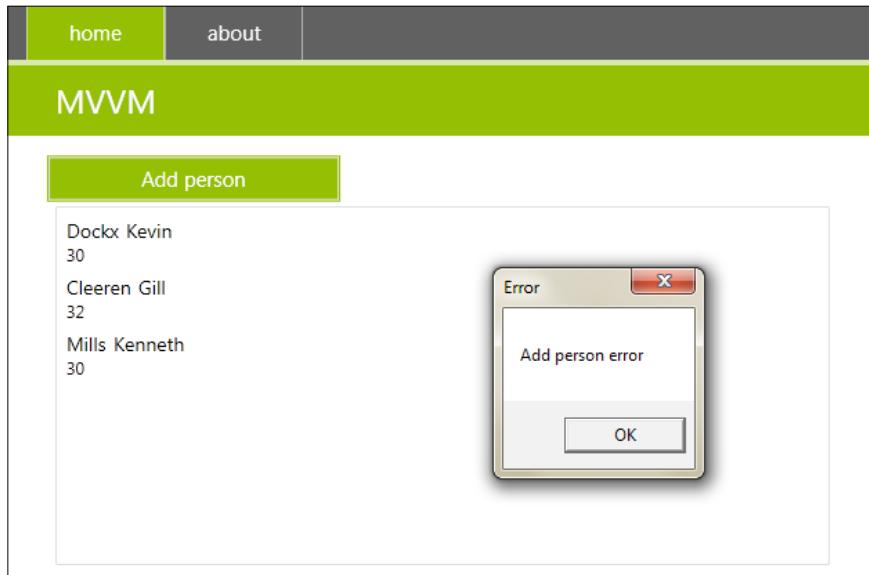
3. Open PersonViewModel.cs and change the AddPersonExecution() method:

```
private void AddPersonExecution()
{
    // simulate an error
    Messenger.Default.Send<DialogMessage>(new DialogMessage("Add
    person error"
        , (result) => ExecuteLoadErrorCallback(result)));
}
```

4. Add the following code:

```
private void ExecuteLoadErrorCallback(System.Windows.
    MessageBoxResult result)
{
    // action to execute on this VM, depending on the messagebox
    result
}
```

5. You can now build and run your application.



How it works...

We're trying to end up with a central place in the application to handle DialogMessages (used to show a message to the user), without having to create a specific Messenger instance nor a message type.

We're starting out by creating a central place to handle messages of this type—the `MainPage`. `MainPage` is registered in steps 1 and 2 to receive messages of type `DialogMessage` through the default Messenger.

Next, we have to create a message to send; this is done in `PersonViewModel`. We also pass in a method that will be executed on callback. This is code that's executed once a user has closed the `MessageBox`.

This message is sent through the default Messenger, available throughout the application (as you notice, we do not have to instantiate this ourselves). Once the message is received in `MainPage`, the `MessageBox` is shown. After this `MessageBox` is closed by the user, the callback is processed. This takes us back to our `ViewModel`, where we can now execute code depending on what the user clicked in the `MessageBox`.

What we've reached now is a central place to handle messages of this type, and we didn't have to create a new `Messenger` instance as we're using the default built-in static `Messenger` instance, available throughout all `ViewModels`.

There's more...

This example was written to show you how to use the default `Messenger` instance, accessible throughout the application, using a built-in `DialogMessage` type. Other types are built-in as well:

- ▶ `NotificationMessage`: built-in type to send a notification to any recipient
- ▶ `NotificationMessage<T>`: built-in type to send a notification (string) and generic content to any recipient
- ▶ `NotificationMessageAction`: built-in type to send a message that provides a callback
- ▶ `NotificationMessageAction<T>`: generically typed version of `NotificationMessageAction`
- ▶ `PropertyChangedMessage<T>`: used to propagate a property changed notification to any recipient; `OldValue (T)` and `NewValue (T)` can be passed via this message

Besides that, the `Messenger` can actually be used to communicate between any two loosely coupled types—not only `ViewModels`, as you have seen in this recipe.

And finally, the default `Messenger` can, just like any `Messenger` instance, receive and send messages of any type, not just built-in types.

See also

For more information on MVVM, have a look at the other recipes in this chapter.

7

Working with Services

In this chapter, we will cover:

- ▶ Connecting and reading from a standardized service
- ▶ Persisting data using a standardized service
- ▶ Configuring cross-domain calls
- ▶ Working cross-domain from a trusted application
- ▶ Reading XML using `HttpWebRequest`
- ▶ Reading out an RSS feed
- ▶ Accessing a database in SQL Azure
- ▶ Accessing a service in Windows Azure
- ▶ Running a Silverlight application hosted in Windows Azure
- ▶ Using socket communication

Introduction

Looking at the namespaces and classes in the Silverlight assemblies, it's easy to see that there are no ADO.NET-related classes available in Silverlight. Silverlight does not contain a `DataReader`, a `DataSet`, or any option to connect to a database directly. Thus, it's not possible to simply define a connection string for a database and let Silverlight applications connect with that database directly.

The solution adds a layer on top of the database in the form of services. The services that talk directly to a database (or, more preferably, to a business and data access layer) can expose the data so that Silverlight can work with it. However, the data that is exposed in this way does not always have to come from a database. It can come from a third-party service, by reading a file, or be the result of an intensive calculation executed on the server.

Silverlight has a wide range of options to connect with services. This is important as it's the main way of getting data into our applications. In this chapter, we'll look at the concepts of connecting with several types of services and external data. In *Chapter 8* through *Chapter 10*, we'll take a detailed look at communicating with WCF and REST services respectively, which are the most widely-used types of services when working with Silverlight.

We'll start our journey by looking at how Silverlight connects and works with a regular service. We'll see the concepts that we use here recur for other types of service communications as well. One of these concepts is cross-domain service access. In other words, this means accessing a service on a domain that is different from the one where the Silverlight application is hosted. We'll see why Microsoft has implemented cross-domain restrictions in Silverlight and what we need to do to access externally hosted services.

Next, we'll talk about working with the **Windows Azure Platform**. More specifically, we'll talk about how we can get our Silverlight application to get data from a SQL Azure database, how to communicate with a service in the cloud, and even how to host the Silverlight application in the cloud, using a hosted service or serving it from Azure Storage.

Finally, we'll finish this chapter by looking at **socket communication**. This type of communication is rare and chances are that you'll never have to use it. However, if your application needs the fastest possible access to data, sockets may provide the answer.

Connecting and reading from a standardized service

Applies to Silverlight 3, 4 and 5

If we need data inside a Silverlight application, chances are that this data resides in a database or another data store on the server. Silverlight is a client-side technology, so when we need to connect to data sources, we need to rely on services. Silverlight has a broad spectrum of services to which it can connect.

In this recipe, we'll look at the concepts of connecting with services, which are usually very similar for all types of services Silverlight can connect with. We'll start by creating an **ASMX webservice**—in other words, a regular web service. We'll then connect to this service from the Silverlight application and invoke and read its response after connecting to it.

Getting ready

In this recipe, we'll build the application from scratch. However, the completed code for this recipe can be found in the `Chapter07/SilverlightJackpot_Read_Completed` folder in the code bundle that is available on the Packt website.

How to do it...

We'll start to explore the usage of services with Silverlight using the following scenario. Imagine we are building a small game application in which a unique code belonging to a user needs to be checked to find out whether or not it is a winning code for some online lottery. The collection of winning codes is present on the server, perhaps in a database or an XML file. We'll create and invoke a service that will allow us to validate the user's code with the collection on the server. The following are the steps we need to follow:

1. We'll build this application from scratch. Our first step is creating a new Silverlight application called **SilverlightJackpot**. As always, let Visual Studio create a hosting website for the Silverlight client by selecting the **Host the Silverlight application in a new Web site** checkbox in the **New Silverlight Application** dialog box. This will ensure that we have a website created for us, in which we can create the service as well.
2. We need to start by creating a service. For the sake of simplicity, we'll create a basic ASMX web service. To do so, right-click on the project node in the **SilverlightJackpot** **Web** project and select **Add | New Item...** in the menu. In the **Add New Item** dialog, select the **Web Service** item. We'll call the new service as **JackpotService**. Visual Studio creates an ASMX file (**JackpotService.asmx**) and a code-behind file (**JackpotService.asmx.cs**).
3. To keep things simple, we'll mock the data retrieval by hardcoding the winning numbers. We'll do so by creating a new class called **CodesRepository.cs** in the web project. This class returns a list of winning codes. In real-world scenarios, this code would go out to a database and get the list of winning codes from there. The code in this class is very easy. The following is the code for this class:

```
public class CodesRepository
{
    private List<string> winningCodes;
    public CodesRepository()
    {
        FillWinningCodes();
    }
    private void FillWinningCodes()
    {
        if (winningCodes == null)
        {
            winningCodes = new List<string>();
            winningCodes.Add("12345abc");
            winningCodes.Add("azertyse");
            winningCodes.Add("abcdefgh");
            winningCodes.Add("helloall");
            winningCodes.Add("ohnice11");
        }
    }
}
```

```
        winningCodes.Add("yesigot1");
        winningCodes.Add("superwin");
    }
}

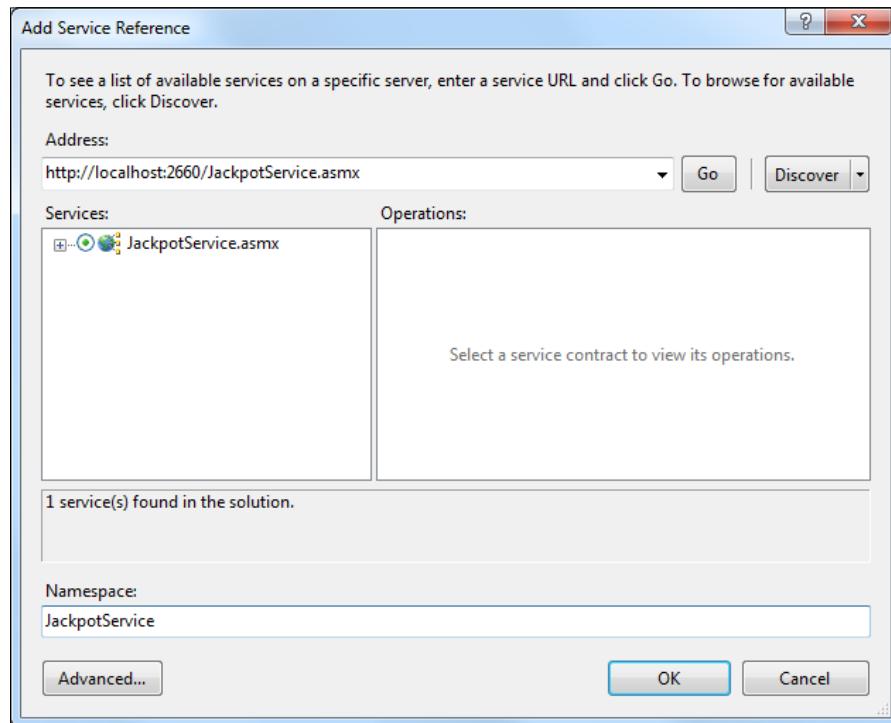
public List<string> WinningCodes
{
    get
    {
        return winningCodes;
    }
}
```

4. At this point, we need only one method in our **JackpotService**. This method should accept the code sent from the Silverlight application, check it with the list of winning codes, and return whether or not the user is lucky to have a winning code. Only the methods that are marked with the `WebMethod` attribute are made available over the service. The following is the code for our service:

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class JackpotService : System.Web.Services.WebService
{
    List<string> winningCodes;
    public JackpotService()
    {
        winningCodes = new CodesRepository().WinningCodes;
    }
    [WebMethod]
    public bool IsWinningCode(string code)
    {
        if(winningCodes.Contains(code))
            return true;
        return false;
    }
}
```

5. Build the solution at this point to ensure that our service will compile and can be connected from the client side.
6. Now that the service is ready and waiting to be invoked, let's focus on the Silverlight application. To make the service known to our application, we need to add a reference to it. This is done by right-clicking on the **SilverlightJackpot** project node, and selecting the **Add Service Reference...** item.

7. In the dialog that appears, we have the option to enter the address of the service ourselves. However, we can click on the **Discover** button as the service lives in the same solution as the Silverlight application. Visual Studio will search the solution for the available services. If there are no errors, our freshly created service should show up in the list. Select it and rename the **Namespace:** as **JackpotService**, as shown in the following screenshot. Visual Studio will now create a proxy class:



8. The UI for the application is kept quite simple. An image of the UI can be seen a little further ahead. It contains a **TextBox**, where the user can enter a code, a **Button** that will invoke a check, and a **TextBlock** that will display the result. This can be seen in the following code:

```
<StackPanel>
    <TextBox x:Name="CodeTextBox"
        Width="100"
        Height="20">
    </TextBox>
    <Button x:Name="CheckForWinButton"
        Content="Check if I'm a winner!"
        Click="CheckForWinButton_Click">
    </Button>
```

```
<TextBlock x:Name="ResultTextBlock">
</TextBlock>
</StackPanel>
```

9. In the Click event handler, we'll create an instance of the proxy class that was created by Visual Studio as shown in the following code:

```
private void CheckForWinButton_Click(object sender,
RoutedEventArgs e)
{
    JackpotService.JackpotServiceSoapClient client = new
        SilverlightJackpot.JackpotService.JackpotServiceSoapClient();
}
```

10. All service communications in Silverlight happen asynchronously. Therefore, we need to provide a callback method that will be invoked when the service returns:

```
client.IsWinningCodeCompleted += new EventHandler
<SilverlightJackpot.JackpotService.
    IsWinningCodeCompletedEventArgs>
(client_IsWinningCodeCompleted);
```

11. To actually invoke the service, we need to call the IsWinningCodeAsync method as shown in the following line of code. This method will make the actual call to the service. We pass in the value that the user entered:

```
client.IsWinningCodeAsync(TextBox.Text);
```

12. Finally, in the callback method, we can work with the result of the service via the Result property of the IsWinningCodeCompletedEventArgs instance. Based on the value, we display another message as shown in the following code:

```
void client_IsWinningCodeCompleted(object sender,
SilverlightJackpot.JackpotService.
    IsWinningCodeCompletedEventArgs e)
{
    bool result = e.Result;
    if (result)
        ResultTextBlock.Text = "You are a winner! Enter your data
            below and we will contact you!";
    else
        ResultTextBlock.Text = "You lose... Better luck next time!";
}
```

13. We now have a fully working Silverlight application that uses a service for its data needs. The following screenshot shows the result from entering a valid code:



How it works...

As it stands, the current version of Silverlight does not have support for using a local database. Silverlight thus needs to rely on external services for getting external data. Even if we had local database support, we would still need to use services in many scenarios. The sample used in this recipe is a good example of data that would need to reside in a secure location (meaning on the server). In any case, we should never store the winning codes in a local database that would be downloaded to the client side.

Silverlight has the necessary plumbing on board to connect with the most common types of services. Services such as **ASMX**, **WCF**, **REST**, **RSS**, and so on, don't pose a problem for Silverlight. While the implementation of connecting with different types of services differs, the concepts are similar.

In this recipe, we used a plain old web service. Only the methods that are attributed with the `WebMethodAttribute` are made available over the service. This means that even if we create a public method on the service, it won't be available to clients if it's not marked as a `WebMethod`. In this case, we only create a single method called `IsWinningCode`, which retrieves a list of winning codes from a class called `CodesRepository`. In real-world applications, this data could be read from a database or an XML file. Thus, this service is the entry point to the data.

For Silverlight to work with the service, we need to add a reference to it. When doing so, Visual Studio will create a proxy class. Visual Studio can do this for us because the service exposes a **Web Service Description Language (WSDL)** file. This file contains an overview of the methods supported by the service. A proxy can be considered a copy of the server-side service class, but without the implementations. Instead, each copied method contains a call to the actual service method. The proxy creation process carried out by Visual Studio is the same as adding a service reference in a regular .NET application.

However, invoking the service is somewhat different. All communication with services in Silverlight is carried out asynchronously. If this wasn't the case, Silverlight would have had to wait for the service to return its result. In the meantime, the UI thread would be blocked and no interaction with the rest of the application would be possible.

To support the asynchronous service call inside the proxy, the `IIsWinningCodeAsync` method as well as the `IIsWinningCodeCompleted` event is generated. The `IIsWinningCodeAsync` method is used to make the actual call to the service. To get access to the results of a service call, we need to define a callback method. This is where the `IIsWinningCodeCompleted` event comes in. Using this event, we define which method should be called when the service returns (in our case, the `client_IIsWinningCodeCompleted` method). Inside this method, we have access to the results through the `Result` parameter, which is always of the same type as the return type of the service method.

See also

Apart from reading data, we also have to persist data. In the next recipe, *Persisting data using a standardized service*, we'll do exactly that.

Persisting data using a standardized service

Applies to Silverlight 3, 4 and 5

In the previous recipe, we connected with the service and read the result of the service call. However, we can also send data back to the service as a type known by the service. If a type is used as the type for a parameter or as the return type for a service method, that type will be exposed by the service as well. Through the proxy generation, we have access to this type inside the Silverlight application as well.

In this recipe, we'll add another method to the web service that uses a custom type as the type for its parameter.

Getting ready

This recipe builds on the code from the previous recipe. If you want to follow along with this recipe, you can either continue using your code or use the provided starter solution located in the `Chapter07/SilverlightJackpot_Persist_Starter` folder in the code bundle that is available on the Packt site. The finished solution for this recipe can be found in the `Chapter07/SilverlightJackpot_Persist_Completed` folder.

How to do it...

We'll extend the **SilverlightJackpot** solution that we built in the previous recipe. When the user enters a valid code, a registration form appears that allows the user to enter his or her credentials. We'll extend the service as well, so that it accepts this data and stores it accordingly. Let's look at the way this is done in the following steps:

1. We will start by adding a new class called `Winner` in the web project. This class will be used to store the information about a winner:

```
public class Winner
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```

2. We'll also create another class called `WinnerRepository`. This class would allow us to store all the winners in a static `List<Winner>`. Just as the previous recipe, this data would be stored in a database in real-world scenarios:

```
public static class WinnerRepository
{
    private static List<Winner> winners = new List<Winner>();
    public static void AddWinner(Winner newWinner)
    {
        winners.Add(newWinner);
    }
}
```

3. We can now extend the service (`JackpotService.asmx.cs`), so that it includes a method (again marked as `WebMethod`) that accepts a `Winner`, and stores it in the `List<Winner>` of the `WinnerRepository` class. This is shown in the following code:

```
[WebMethod]
public void SaveWinner(Winner winner)
{
    WinnerRepository.AddWinner(winner);
}
```

4. Build the solution to make sure that the service works correctly.
5. To make the new method available in the Silverlight application, we need to right-click on the **JackpotService** node within the Silverlight project in the **Solution Explorer**, and select the **Update Service Reference** item. Visual Studio will reconnect to the service and rebuild the proxy. It will also include our extra method.

-
6. We need to add some XAML code to add the fields to collect the required information. For the complete listing, refer to the `MainPage.xaml` file in the sample code. The following is the most relevant part of the XAML code, as it contains the fields in which the user can enter his/her details:

```
<TextBlock x:Name="NameTextBlock"
           Text="Name: ">
</TextBlock>
<TextBox x:Name="NameTextBox">
</TextBox>
<TextBlock x:Name="FirstNameTextBlock">
</TextBlock>
<TextBox x:Name="FirstNameTextBox">
</TextBox>
<TextBlock x:Name="EmailTextBlock">
</TextBlock>
<TextBox x:Name="EmailTextBox">
</TextBox>
<Button x:Name="SubmitButton"
        Click="SubmitButton_Click">
</Button>
<TextBlock x:Name="SubmissionResultTextBlock">
</TextBlock>
```

7. In the `Click` event handler, we start off by creating an instance of the proxy similar to the previous recipe. We'll define the callback method when the service call completes, as well as make the asynchronous call. However, the latter requires an instance of the `Winner` class, which was originally created on the server. This instance is now sent to the service. In the callback method, we'll update the user interface, so that the result of the service call is shown:

```
private void SubmitButton_Click(object sender, RoutedEventArgs e)
{
    JackpotService.JackpotServiceSoapClient client = new
        SilverlightJackpot.JackpotService.JackpotServiceSoapClient();
    client.SaveWinnerCompleted += new
        EventHandler<System.ComponentModel.AsyncCompletedEventArgs>
        (client_SaveWinnerCompleted);
    client.SaveWinnerAsync(
        new JackpotService.Winner()
    {
        FirstName = FirstNameTextBox.Text,
        LastName = NameTextBox.Text,
        Email = EmailTextBox.Text
    }
);
```

```

        }
        void client_SaveWinnerCompleted(object sender,
            System.ComponentModel.AsyncCompletedEventArgs e)
        {
            if (e.Error == null)
            {
                ResultTextBlock.Text = string.Empty;
                SubmissionResultTextBlock.Text = "Submission successful!";
                UserDetailGrid.Visibility = Visibility.Collapsed;
            }
        }
    }
}

```

- After completing these steps, we have successfully allowed the user to send data from the Silverlight application to the service. In turn, the service can store this data in any data store. The following screenshot shows the expanded fields. The application will bundle the entered information into a `Winner` object and send it to the service:



How it works...

Apart from reading data, we also need to send data back to a service. Behind this service façade, code can be used to store data into a database or any other data store. An important thing to understand here is that persisting data can be carried out by passing in values for the parameters of a service method.

When the service uses a particular class (such as the `Winner` class in this recipe) as a parameter type, this class gets sent down the wire to the Silverlight application as well. On generating the proxy based on the WSDL file, a local copy of this class will be generated in the Silverlight application as well. This class can be used in the same way as a normal class. In this sample, we have created an instance of this class using the following code:

```
client.SaveWinnerAsync(new JackpotService.Winner()
{ FirstName = FirstNameTextBox.Text, LastName = NameTextBox.Text,
Email = EmailTextBox.Text } );
```

This instance is then serialized in a SOAP message (because of the ASMX service), and sent to the service. On the service side, the instance is rebuilt and used as a parameter for the service method through deserialization.

See also

We have looked at connecting with a service and reading data from it in the first recipe of this chapter, *Connecting and reading from a standardized service*.

Configuring cross-domain calls

Applies to Silverlight 3, 4 and 5

In the previous recipes, we have used services that were hosted within the same website (and consequently the same domain) as the Silverlight application itself. However, more often than not, this will not be the case in real-world scenarios. We may need to connect to a third-party service (such as Flickr's API services) from Silverlight. Even if we wrote the service layer ourselves, it may be hosted on a different domain.

Silverlight imposes restrictions on communication with external services. However, Silverlight allows services to opt-in to allow being called from a Silverlight application. In this recipe, we'll look at the actions we need to take to make it possible to connect to a self-written service that is hosted in a different domain.

Getting ready

To follow along with this recipe, a starter solution is provided in the `Chapter07/SilverlightCrossDomain_Starter` folder in the code bundle that is available on the Packt site. The finished solution for this recipe can be found in the `Chapter07/SilverlightCrossDomain_Completed` folder.

How to do it...

In this recipe, we'll build a Silverlight application that shows flight information based on the values entered by the user. However, this information is located in a service hosted by the airline companies, which means it runs externally and on a different domain than our Silverlight application. In the following steps, we'll see what we need to change so that Silverlight can access externally hosted services:

1. Start by opening the solution as outlined in the *Getting Ready* section. The solution contains two projects: **SilverlightCrossDomain** (the Silverlight project) and **SilverlightCrossDomain.Web** (the hosting web project).
2. In this solution, add a new **ASP.NET Web Application** and name it as **FlightInformation**. You can do this by right-clicking on the solution in the **Solution Explorer** and selecting **Add | New Project....** In the dialog that appears, select **ASP.NET Web Application** under the **Visual C# | Web node**.
3. We'll create a WCF service in this web application. Add a new WCF service called `FlightInformationService.svc`. Visual Studio will create the `FlightInformationService.svc` file, the `IFlightInformationService.cs` interface, and the concrete implementation—`FlightInformationService.svc.cs` (Note that we'll be using more WCF services in *Chapter 8, Talking to WCF and ASMX Services* and *Chapter 9, Talking to WCF and ASMX Services - One Step Beyond*).
4. To make the service accessible from Silverlight, we need to change the binding type from `wsHttpBinding` to `basicHttpBinding` in the `web.config` of the project. The change we need to make is shown in the following code:

```
<endpoint address=""  
          binding="basicHttpBinding"  
          contract="FlightInformation.IFlightInformationService">  
    <identity>  
      <dns value="localhost" />  
    </identity>  
</endpoint>
```

5. Let's now define the contract of the service in the `IFlightInformationService` interface. We'll keep it simple by making it possible for the service to return only a list of `Flight` instances.

```
[ServiceContract]  
public interface IFlightInformationService  
{  
    [OperationContract]  
    List<Flight> GetFlights(string fromAirport, string toAirport,  
                           DateTime date);  
}
```

6. The Flight class is not yet defined, so we should go ahead and create this class as shown in the following code:

```
[DataContract]
public class Flight
{
    [DataMember]
    public string Airway { get; set; }
    [DataMember]
    public DateTime DepartureTime { get; set; }
    [DataMember]
    public DateTime ArrivalTime { get; set; }
    [DataMember]
    public double Price { get; set; }
}
```

7. We can now add the implementation method for the service in the FlightInformationService class. For now, we'll create a hard-coded list of Flight instances. In real-world scenarios, this data would come from a real data store, such as a database. The following is the code for the service:

```
public class FlightInformationService : IFlightInformationService
{
    public List<Flight> GetFlights(string fromAirport, string
        toAirport, DateTime date)
    {
        //some data access code here...
        return new List<Flight>()
        {
            new Flight()
            {
                Airway = "Silverlight Airways",
                DepartureTime = DateTime.Now,
                ArrivalTime = DateTime.Now.AddHours(3),
                Price=300
            },
            new Flight()
            {
                Airway = "Packt Airways",
                DepartureTime = DateTime.Now.AddHours(5),
                ArrivalTime = DateTime.Now.AddHours(10),
                Price=1000
            },
            new Flight()
            {
```

```
Airway = "New Airways",
DepartureTime = DateTime.Now,
ArrivalTime = DateTime.Now.AddHours(9),
Price=1200
},
new Flight()
{
    Airway = "Silverlight Airways",
    DepartureTime = DateTime.Now.AddHours(3),
    ArrivalTime = DateTime.Now.AddHours(5),
    Price=200
}
};
```

```
}
```

8. Build the solution at this point.
9. Now that the external service is ready, we can use it from our Silverlight application. Or can we? Let's try and see how it goes! Add a service reference in the Silverlight application by right-clicking on the Silverlight project in the **Solution Explorer** and selecting **Add Service Reference....** In the dialog, search for the `FlightInformationService` using the **Discover** button. Once it's located, set `FlightInformationService` as the namespace.
10. In the Silverlight application, we create a basic UI (shown in the image a bit later in this recipe). The most important element of the UI is a `ListBox` control, which we'll use to bind the flight information. The UI also contains a `Button` that will trigger a request to the service. Refer to the sample code for the complete listing.

11. In the `Click` event of the `Button`, we perform an asynchronous call to the service. This is shown in the following code:

```
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    FlightInformationService.FlightInformationServiceClient proxy =
        new SilverlightCrossDomain.FlightInformationService.
        FlightInformationServiceClient();
    proxy.GetFlightsCompleted += new
        EventHandler<SilverlightCrossDomain.
        FlightInformationService.GetFlightsCompletedEventArgs>
        (proxy_GetFlightsCompleted);
    proxy.GetFlightsAsync("New York", "London",
        DateTime.Now.Date);
}
```

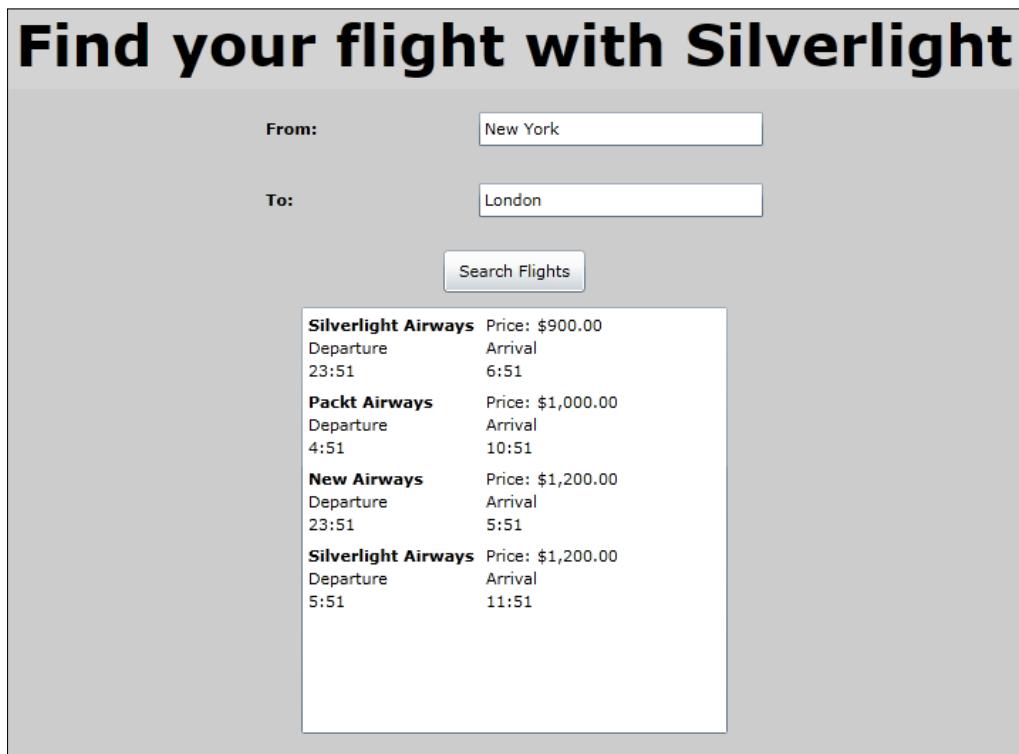
12. In the `proxy_GetFlightsCompleted` event handler, we can read the results from the `e.Result` property and bind them to the `ListBox`, as shown in the following code:

```
void proxy_GetFlightsCompleted(object sender,
    SilverlightCrossDomain.FlightInformationService.
    GetFlightsCompletedEventArgs e)
{
    FlightListBox.ItemsSource = e.Result;
}
```

13. On running this code now, we'll get an exception telling us that we're most likely trying to do a cross-domain service call. The exception is of the `CommunicationException` type.
14. To solve this, we need to add a file called `clientaccesspolicy.xml` to our web project (the project where the service resides). Silverlight will check for the existence of this file in the root of the service domain when performing a cross-domain call. The content of this file is shown in the following code:

```
<?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>
```

15. When trying to run the code again, Silverlight will make the call to the service and the results are displayed as shown in the following screenshot:



How it works...

When talking to services that are not hosted in the same domain as the Silverlight application, Silverlight is quite restrictive. By default, it does not allow so-called cross-domain calls. This is purely for security reasons. Let's look at what would happen if Silverlight would allow making cross-domain calls.

Let's assume that a website hosted on `SomeFriendlySite.com` requires the user to log in. The credentials are stored on the user's PC, so that on the next visit they wouldn't need to log in again. This site also exposes a service that contains secret information about the user, which is only accessible when logged in to the site.

An attacker could create a Silverlight application that would try to retrieve this secret information and host this application on `SomeFakeSite.com`. Unaware of any danger, the user accesses the Silverlight application on `SomeFakeSite.com`, and thereby installs the hacker's Silverlight application on his machine. This application can now try to make a request to the `SomeFriendlySite.com` service using the stored credentials and can send the secret information to `BadAttackerSite.com`. Worst of all, the user would not even know of all this happening as it would take place behind the scenes. This type of attack from Silverlight is comparable to a cross-site scripting attack, in which the same technique is used. To make it impossible for these kinds of attacks to occur, Silverlight does not make a request to an external service; that is, unless this service explicitly allows us to do so (usually because it does not expose any sensitive data).

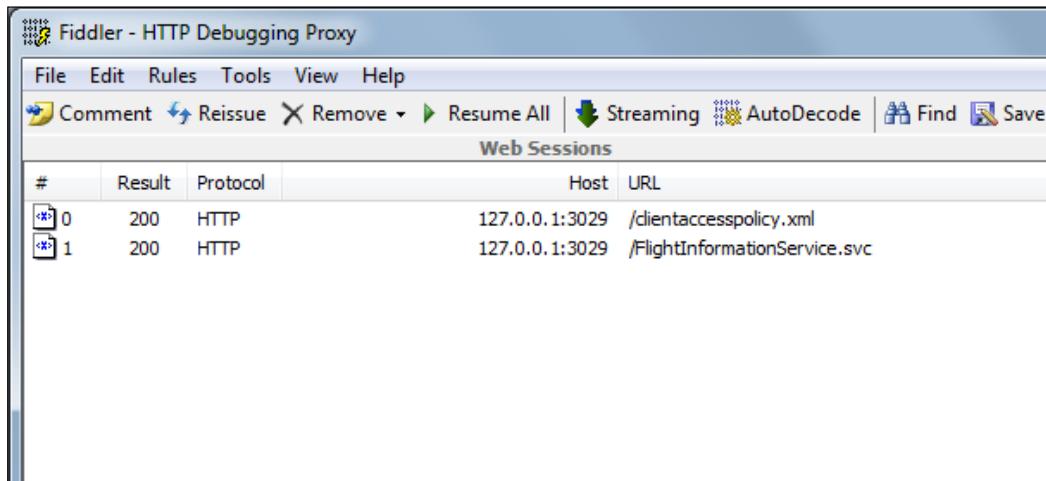
When making a call to such an external service, Silverlight will first check the existence of a file called `clientaccesspolicy.xml`. If this file exists, it will be analyzed by Silverlight. If the file allows all domains to call the service (`domain uri="*"`), or if the domain in which the Silverlight application is running is in the specified list, Silverlight will make the service call. The following listing shows a `clientaccesspolicy.xml` file, which allows only some domains to call it:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
    <cross-domain-access>
        <policy>
            <allow-from http-request-headers="*">
                <domain uri="http://www.snowball.be"/>
                <domain uri="http://api.snowball.be"/>
                <domain uri="http://www.codeflakes.com"/>
            </allow-from>
            <grant-to>
                <resource path="/" include-subpaths="true"/>
            </grant-to>
        </policy>
    </cross-domain-access>
</access-policy>
```

The approach that Silverlight uses here is similar to what **Flash** does. Flash also does not allow cross-domain calls. When making a request to an external service, it will check for the existence of a file called `crossdomain.xml`. The following is a sample `crossdomain.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<cross-domain-policy>
    <allow-access-from domain="*"/>
</cross-domain-policy>
```

Silverlight can work with both `clientaccesspolicy.xml` as well as `crossdomain.xml`. When calling a service, Silverlight will first check if `clientaccesspolicy.xml` exists at the root of the domain. If it does not, it will check if `crossdomain.xml` is located at the root. If neither of the files is present, the request is blocked. The following screenshot shows that Silverlight searched for `clientaccesspolicy.xml` before allowing the call to the service:



Of course, the need for one of these two files is bad news when we are creating mashups.

Mashups are applications that are composed of data coming from different services.

However, the good news is that several services (such as, **Flickr**) expose a file that allows cross-domain calls. That said, some services do not, while others are restrictive (such as **Twitter**). Nothing is lost in this case, but we are required to do some additional work. We need to create an extra service layer in the same domain as the one in which the Silverlight application is hosted. These services can connect with every service (as this is not a cross-domain call from a client). Thus, when calling the external service, we would actually call the local service from Silverlight, which would in turn call the external service. The local service acts as a pass-through for data.

[ Please note that while this is a good solution, it will put extra load on the server as every service call will pass by that server. In *Chapter 10, Talking to REST and WCF Data Services*, the *Talking to Twitter* recipe, looks at how to do this using the Twitter API.]

Working cross-domain from a trusted Silverlight application

Applies to Silverlight 4 and 5

In the previous recipe, we looked at the restrictions enforced by the Silverlight runtime when accessing services that do not live in the same domain as the Silverlight application. These are called **cross-domain restrictions**. To access a service that lives in another domain, a cross-domain policy file should be in place. If not, Silverlight won't communicate with the service.

Starting with Silverlight 4, a new application model was added, namely **trusted applications**. A **trusted application** or an application with elevated permissions is similar to an out-of-browser application. However, it gets more permissions on the system. One of these permissions is accessing services in a cross-domain manner without requiring a cross-domain file to be in place. In Silverlight 5, in-browser applications can also run with elevated permissions. The concepts explained in this recipe therefore apply for both in-browser and out-of-browser trusted applications.

In this recipe, we'll make cross-domain calls from a trusted Silverlight application.

Getting ready

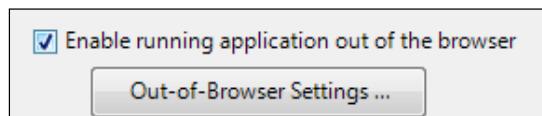
For this recipe, you can use any of the other Silverlight applications. In the following steps, we use the starter solution for this recipe that can be found in the `Chapter07/Silverlight_TrustedCrossDomain_Starter` folder in the code bundle that is available on the Packt site. The completed solution can be found in the `Chapter07/Silverlight_TrustedCrossDomain_Completed` folder.

How to do it...

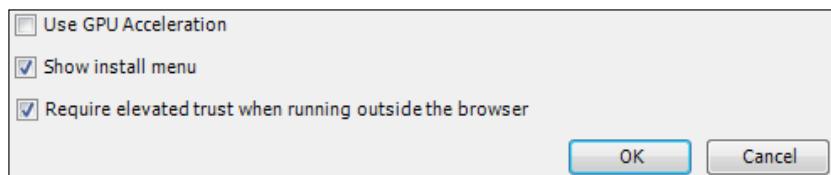
To allow a Silverlight application to make cross-domain calls without having a cross-domain file in place, the application should run as an out-of-browser application having elevated permissions. The following are the steps we need to perform to get this working:

1. Open a Silverlight project (starter solution or your own project) as outlined in the *Getting ready* section of this recipe.
2. Right-click on the **SilverlightCrossDomain** project and select the **Properties** item to open the **Project Properties** window.

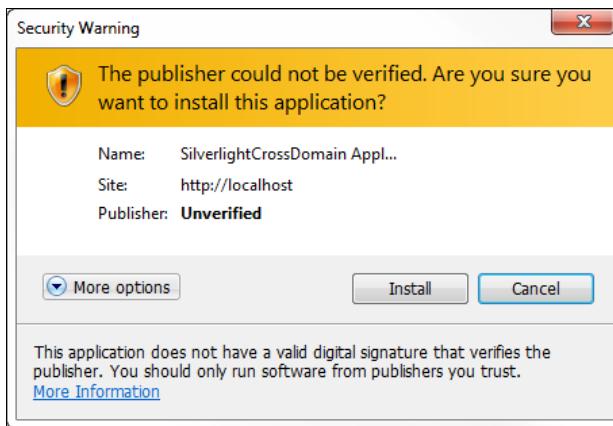
3. In this window, select the **Enable running application out of the browser** option. This will allow us to install the application locally, so that it can run as a standalone application and will not require a browser to be opened:



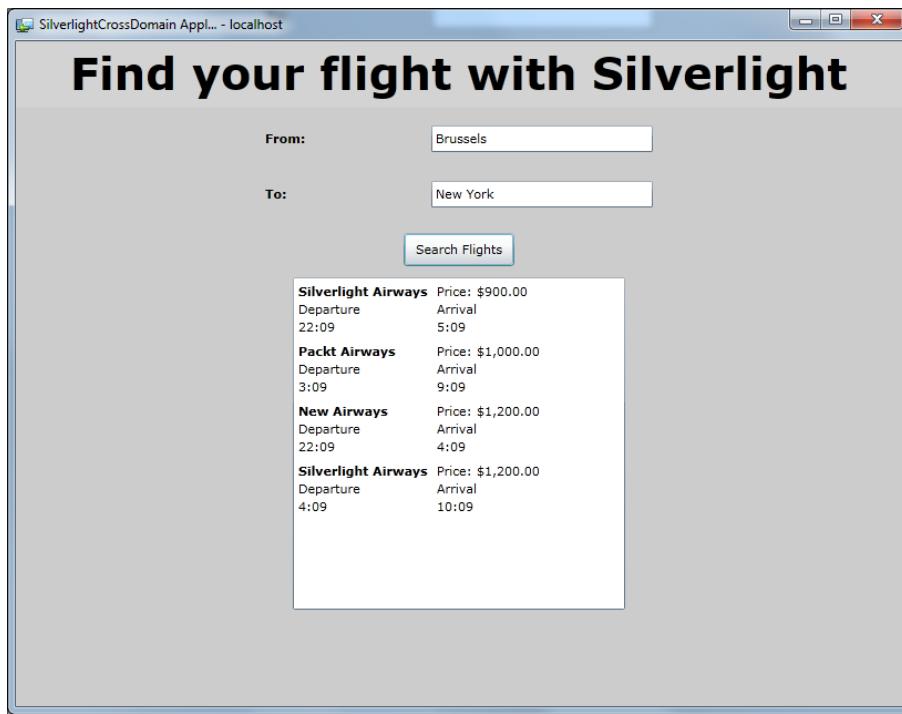
4. Running without a browser is not enough; the application should run with elevated permissions. To enable this, click on the **Out-of-Browser Settings...** button as shown in the previous screenshot, and mark the **Require elevated trust when running outside the browser** option in the dialog box that appears. This is shown in the following screenshot:



5. In the **FlightInformation** project, we don't need the `clientaccesspolicy.xml` anymore. It can be deleted from the project.
6. You can now run the application. Make sure that the hosting web application, **SilverlightCrossDomain.Web**, is set as the startup project for the solution. Before interacting with it, right-click on the interface and choose the second option—**Install SilverlightCrossDomain Application** onto this computer. As shown in the following screenshot, Silverlight will display a prompt warning that the application requires elevated permissions and thus has access to system resources. Click on the **Install** button to finish installing the application on the local machine:



7. Our application now runs as a trusted application and can access the service located on another domain, without cross-domain restrictions preventing us from accessing this service. The running application is shown in the following screenshot:



How it works...

The default behavior of Silverlight is that it will check for the existence of a cross-domain policy file, when accessing services that are hosted on a domain different from the one the Silverlight application itself is hosted on.

Starting with Silverlight 4, a new option was added: an application that runs with elevated permissions. This so-called trusted application has permissions that regular in-browser or regular out-of-browser Silverlight applications don't have. For example, it can access the local filesystem, perform COM interop or P/Invoke calls, and also perform cross-domain service access without checking for the policy file.

 **COM interop** is a technology, part of .NET, that makes it possible to interact with COM objects, directly from .NET code. This way, .NET code can call into existing COM components without these components needing to be changed. Working with Microsoft Office components requires that COM interop code is written.



P/Invoke (short for **Platform Invocation Services**) is another **CLR** feature that allows .NET code to invoke native (for example, C++) code.



With Silverlight 5, the concept of trusted applications is extended into the browser: in-browser applications can run with elevated permissions as well. In case you are building this type of application, it will also be able to access a service in a cross-domain manner without a cross-domain file being in place.

See also

In the previous recipe, we looked at the default cross-domain behavior, where Silverlight restricts the access to services that are not hosted in the same domain. In *Chapter 10, Talking to REST and WCF Data Services*, the recipe *Talking to Twitter from a non-trusted application* applies the concepts learned in this recipe by communicating with Twitter.

Reading XML using `HttpWebRequest`

Applies to Silverlight 3, 4 and 5

The services we used in the first two recipes of this chapter were self-describing services. By this we mean that the service itself exposes information about its methods and data types by means of a WSDL file.

However, often we might need to access data that is not exposed by such a service. For example, we may need to read out XML data. This data could be available as a physical file on the server. It could perhaps be dumped by a process in a specific location. Alternatively, while sending a request to a specific URL, some services, such as REST services, return XML data. Communicating with REST services is the topic of *Chapter 10, Talking to REST and WCF Data Services*.

Whether the XML comes from a REST service or lives in a file on the server, reading out XML is done using the `WebClient` class most of the time. We'll use this class extensively in *Chapter 8, Talking to WCF and ASMX Services*. However, if we need more control over the call, we should use the `HttpWebRequest` class. Everything we can do with the `WebClient` class can also be done using the `HttpWebRequest` class, but not vice-versa.

In this recipe, we'll look at how we can use the `HttpWebRequest` class to read out the XML data returned by a handler file.

Getting ready

If you want to follow along with this recipe, you can use the starter solution located in the Chapter07/SilverlightHttpRequest_Starter folder in the code bundle that is available on the Packt site. The completed solution for this recipe can be found in the Chapter07/SilverlightHttpRequest_Completed folder.

How to do it...

To find out how to work using the `HttpWebRequest` class, we'll develop a handler that generates an XML string containing news items. Our Silverlight client will create an `HttpWebRequest` instance to request news items newer than a specified date from the handler and receive a response, the XML literal. We can use such a literal in a data binding scenario by parsing it using LINQ to XML. Following are the steps we need to execute to get the `HttpWebRequest` class working:

1. Open the starter solution as outlined in the *Getting Ready* section of this recipe. In the web project, add a new **Generic Handler** in the **Add new item** dialog named **NewsHandler.ashx**. A handler can be recognized by its `.ASHX` file extension and is comparable to an ASP.NET webform as both implement the `IHttpHandler` interface and act like an endpoint. The `IHttpHandler` interface defines one method (`ProcessRequest`) and one boolean property (`IsReusable`). The `ProcessRequest` method will be executed when sending a request to the handler.
2. Before we add the code to the handler, let's create a class named `NewsRepository` that represents a data access layer containing some hardcoded data (which is news items here). This class is located in the `NewsRepository.cs` file in the web project, and can be found in the samples. The most relevant part is shown in the following code. It contains the `GetNewsAsXml` method, which accepts the start date parameter. The method will carry out a LINQ query to retrieve the news items and return them in an XML format using the `XElement` class (part of LINQ to XML). The rest of the code for this class can be found in the completed solution of the code bundle:

```
public string GetNewsAsXml(DateTime startDate)
{
    XElement element = new XElement("news");
    var result = from n in CreateNews()
                 where n.DatePosted > startDate
                 select n;
    foreach (var newsItem in result.ToList())
    {
        element.Add(new XElement("newsitem",
            new XElement("newsitemid", newsItem.NewsItemId),
            new XElement("title", newsItem.Title),
            new XElement("content", newsItem.Content),
            new XElement("dateposted", newsItem.DatePosted)));
    }
}
```

```

        new XElement("dateposted", newsItem.DatePosted),
        new XElement("image", newsItem.Image)
    ));
}
return element.ToString();
}

```

For compiling this code, a reference has to be made to the `System.Xml.Linq` assembly.

- Let's now look back at the handler. In the `ProcessRequest` method, we'll call the previous method, which required a parameter of the `DateTime` type. We can get access to objects such as `Server` and `Request` using the `context` parameter of the `ProcessRequest` method. In this case, we'll use it to get the value sent from the client. In the following steps, we'll write the code that passes a `DateTime` object. Finally, we'll use the same `Context` instance to send the result back to the caller in an XML format. All this can be achieved using the following code:

```

public void ProcessRequest(HttpContext context)
{
    string news;
    using (System.IO.StreamReader reader = new
        System.IO.StreamReader(context.Request.InputStream))
    {
        string input = reader.ReadToEnd();
        DateTime startDate = DateTime.Parse(input);
        news = new NewsRepository().GetNewsAsXml(startDate);
    }
    context.Response.ContentType = "text/xml";
    context.Response.Write(news);
}

```

- That's all for the server side! The next stop is the client side. The UI for the application is quite simple. The code can be found in the `MainPage.xaml` file of the code bundle. It mainly contains a `Button` that will load the news items into a `ListBox` when clicked. The `ListBox` has a `DataTemplate` applied to it, which uses data binding features to tie the items to the template.
- When clicking on the button, we need to send a request to the URI of the handler and later work with the response. We can use the `Create` method of the `HttpWebRequest` class to create the request. This method accepts a URI as a parameter and returns a `WebRequest` instance. We can use this instance to call the service. The following code creates a `WebRequest` using the `Create` method of the `HttpWebRequest` class:

```

private void LoadButton_Click(object sender, RoutedEventArgs e)
{
    WebRequest request = HttpWebRequest.Create(new

```

```
        Uri("http://localhost:61639/NewsHandler.ashx",
        UriKind.Absolute));
    }
```

6. We need a stream to send data to a service (the start date). Getting this stream is carried out asynchronously using the `BeginGetRequestStream` method of the request. We also need to specify that we'll be sending data, so we need to set the `Method` to `POST`. This is shown in the following code:

```
request.ContentType = "text/xml";
request.Method = "POST";
request.BeginGetRequestStream(RequestCallback, request);
```

7. The callback method implements the `AsyncCallback` delegate. It has an `IAsyncResult` parameter. The `AsyncResult` property of the `IAsyncResult` instance is cast to `HttpWebRequest`. Using this instance, we retrieve the `EndGetRequestStream` method to get the request stream. We use a `StreamWriter` instance to write the current date as shown in the following code:

```
void RequestCallback(IAsyncResult result)
{
    HttpWebRequest request = result.AsyncState as HttpWebRequest;
    Stream stream = request.EndGetRequestStream(result);
    using (StreamWriter writer = new StreamWriter(stream))
    {
        writer.WriteLine(new DateTime(2009, 12, 1));
    }
}
```

8. We can now use the request to invoke the service and get its result asynchronously. This is done using the `BeginGetResponse` method. The invocation is carried out using the following line of code, which again specifies a callback method—`ResponseCallback`.

```
request.BeginGetResponse(ResponseCallback, request);
```

9. The `ResponseCallback` method is similar to the `RequestCallback`. It has the same `IAsyncResult` parameter as the `RequestCallback` had. We get access to the `HttpWebRequest` object again using the `AsyncResult` property of the `IAsyncResult`. On this instance, we call the `EndGetResponse` to get an `HttpWebResponse` instance. This is shown in the following code:

```
void ResponseCallback(IAsyncResult result)
{
    HttpWebRequest request = result.AsyncState as HttpWebRequest;
    HttpWebResponse response = request.EndGetResponse(result) as
        HttpWebResponse;
}
```

10. We invoke the `GetResponseStream` method on the `HttpWebResponse` instance. This call returns a `Stream` that can be used in combination with a `StreamReader` to read out the response from the service. This is shown in the following lines of code:

```
using(StreamReader reader = new  
    StreamReader(response.GetResponseStream()))  
{  
    string responseMessage = reader.ReadToEnd();  
}
```

11. This string now contains the XML data as returned from the service. We can parse it and generate instances of a custom class called `NewsItem` using LINQ to XML. This is shown in the following code:

```
public class NewsItem  
{  
    public int NewsItemId { get; set; }  
    public string Title { get; set; }  
    public string Content { get; set; }  
    public DateTime DatePosted { get; set; }  
    public string NewsImage { get; set; }  
}
```

12. The parsing of the XML code generates a `List<NewsItem>`. The code uses the `XDocument` and the `XElement` classes that are located in the `System.Xml.Linq` namespace. This namespace lives in the corresponding `System.Xml.Linq` assembly to which we need to create a reference. The parsing process is shown in the following lines of code:

```
XDocument document = XDocument.Parse(responseMessage);  
var query = from n in document.Descendants("newsitem")  
            select n;  
List<NewsItem> newsItems = new List<NewsItem>();  
foreach ( XElement element in query)  
{  
    NewsItem newsItem = new NewsItem();  
    newsItem.NewsItemId = int.Parse  
        (element.Descendants("newsitemid").First().Value);  
    newsItem.Title = element.Descendants("title").First().Value;  
    newsItem.Content = element.Descendants("content").First().Value;  
    newsItem.DatePosted = DateTime.Parse  
        (element.Descendants("dateposted").First().Value);  
    newsItem.NewsImage= element.Descendants("image").First().Value;  
    newsItems.Add(newsItem);  
}
```

13. Now that we have a collection of items, we want to display them in a `ListBox`. However, the asynchronous service call operates in the background thread and thus cannot directly access the controls in the UI that live in the UI thread. However, we can use the `Dispatcher` class to invoke the code in the UI thread from the background thread, as shown in the following line of code:

```
this.Dispatcher.BeginInvoke(() => NewsListBox.ItemsSource = newsItems);
```

14. At this point, we have successfully communicated with a service using the `HttpWebRequest` class.

How it works...

Silverlight can connect with several types of services. They can be categorized into two types—services that describe themselves and services that do not describe themselves. The latter can be used from Silverlight by using either the `WebClient` class or the `HttpWebRequest` class. Most of the time, the more easy-to-use `WebClient` class will suffice. It contains all that is needed to carry out communication with most services.

However, if more control is needed over the call to the service, the `HttpWebRequest` class can be used. It has more options than the `WebClient` class. In spite of that, the `WebClient` class does use the `HttpWebRequest` class under the hood. In *Chapter 10, Talking to REST and WCF Data Services*, we will be using the `WebClient` class where we will discuss communication with REST services. However, all those samples can be performed using the `HttpWebRequest` class as well.

All calls carried out using the `HttpWebRequest` class take place asynchronously. Using the `HttpWebRequest` class, we can send data to the service using the request as well as read out data using the response.

The request

A `WebRequest` instance is created using the `Create` method of the `HttpWebRequest` class. This method accepts one parameter, which is the URI of the service. On the created `WebRequest` instance, we can specify the HTTP verb. Silverlight's `HttpWebRequest` class supports the `GET` and the `POST` methods. If we want to send data to the service (for example a search term), we need to get the request stream. This can be done using the `BeginGetRequestStream` method of the `HttpWebRequest` class, which starts an asynchronous call to get the request. This method requires two parameters—a callback method and a state. The callback method is executed when the request stream is ready. For the state, we pass in the request itself, so that we have access to it in the callback.

In the callback method, we can get the request by casting the `AsyncResult` property of the `IAsyncResult` interface back into the `HttpWebRequest` class. On this instance, we can call the `EndGetRequestStream` method that gives us access to the request stream. We can write to this stream using a `StreamWriter`.

The response

The request instance can be used to invoke a service and get back its result. To carry out this invocation, we use the `BeginGetResponse` method on the request, which is, quite logically, also asynchronous. In the specified callback, we can use the `EndGetResponse` method, which will give us an `HttpWebResponse` instance. On this instance, we can use the `GetResponseStream` method, which returns a stream. This stream can then be used to read the textual response from the server.

Threading headaches

Calls using the `HttpWebRequest` class take place on the background thread. The code that runs in this thread has no access to the UI and its controls. This means that we can't update the UI from the callback method. Luckily, we can use the `Dispatcher` class. Using the `Dispatcher`, we can cross threads quite easily by means of the `BeginInvoke` method. The `Dispatcher.BeginInvoke` method allows us to call from the background thread code that will be executed on the UI thread.

See also

In *Chapter 10, Talking to REST and WCF Data Services*, we'll use the `WebClient` class intensively while working with REST services.

Reading out an RSS feed

Applies to Silverlight 3, 4 and 5

Most modern, regularly updated websites (such as, news sites, blogs, and so on) allow the users of these sites to subscribe to a **Really Simple Syndication (RSS)** feed (which is in fact an XML file that follows a certain schema), which will typically contain excerpts of site updates. A lot of people use specialized RSS readers for this (such as Google Reader) or integrate the RSS feeds into applications that they use on a day-to-day basis (such as Outlook).

Silverlight includes a few classes to make it very easy to write your own RSS reader. In this recipe, we'll learn how to achieve this.

Getting ready

We'll need to start from a Silverlight solution that contains a few sample feeds and some code to show the feeds you're going to read on screen. You can find a starter solution in the `Chapter07\Reading_RSS_Starter` folder in the code bundle that is available on the Packt site. The completed solution can be found in the `Chapter07\Reading_RSS_Completed` folder.

How to do it...

We're going to add code to our Silverlight project to read out and show an RSS feed. To achieve this, carry out the following steps:

1. Open `MainPage.xaml` and have a look at the code. This file contains the code for a `ListBox` with data binding that will be used to visualize the RSS feed. Also, familiarize yourself with the project layout. It contains the RSS XML files that will be parsed, which contain the RSS updates.

2. Add a new class to the Silverlight project and name it `SimpleSyndicationItem.cs`.
3. Add the following code to this new class:

```
public class SimpleSyndicationItem
{
    public DateTimeOffset Date { get; set; }
    public string Title { get; set; }
    public Uri Link { get; set; }
    public SimpleSyndicationItem(DateTimeOffset Date, string Title,
        Uri Link)
    {
        this.Date = Date;
        this.Title = Title;
        this.Link = Link;
    }
}
```

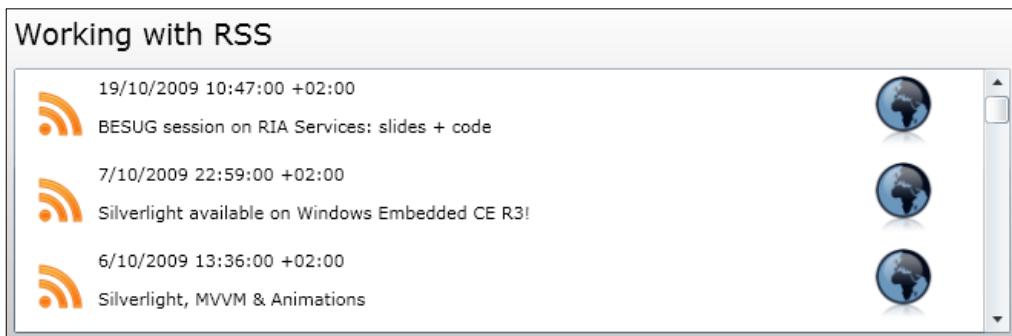
4. Open `MainPage.xaml.cs` and add a new property to it, which will be used to store a collection of `SimpleSyndicationItem`. This is shown in the following line of code:

```
public ObservableCollection<SimpleSyndicationItem>
    lstSimpleItems { get; set; }
```

5. Locate the `FetchItems()` method and add the following code to the method body:

```
XmlReader reader = XmlReader.Create("rssicecream.xml");
SyndicationFeed feed = new SyndicationFeed();
feed = SyndicationFeed.Load(reader);
lstSimpleItems = new
    ObservableCollection<SimpleSyndicationItem>();
foreach (var item in feed.Items)
{
    lstSimpleItems.Add (new SimpleSyndicationItem(item.PublishDate,
        item.Title.Text, item.Links[0].Uri));
}
lstbxRSS.ItemsSource = lstSimpleItems;
```

6. You can now build and run the application. You'll notice a `ListBox` with all the updates from our RSS file. This can be seen in the following screenshot:



How it works...

Silverlight contains classes that make it very easy to work with RSS feeds. These classes are contained in the `System.ServiceModel.Syndication` assembly. The Silverlight project requires a reference to this assembly.

One of these useful classes is the `SyndicationFeed` class. With this class, you can easily load RSS feeds and work with them. The statement used to do this is `SyndicationFeed.Load`. This statement accepts an `XmlReader`. We created an `XmlReader` that points to the RSS feed XML we want to parse, and passed this reader to the `SyndicationFeed.Load` method.

Now, our `SyndicationFeed` instance contains all the items from the RSS feed, which are nicely organized. We run through this collection to create `SimpleSyndicationItem` objects (which are simplified objects that contain only the data we want to visualize), and add these items to an `ObservableCollection` of `SimpleSyndicationItem`.

Finally, we set this collection as an `ItemsSource` of our `ListBox`. Our collection is now nicely visualized because of Silverlight's data binding abilities.

There's more...

You can't just read out any RSS feed (XML file). Silverlight doesn't allow you to access any domain you want. The owner of the domain has to allow you to access files/services on it by publishing a `clientaccesspolicy.xml` or a `crossdomain.xml` file in which the owner can describe who has access to what. If this file is missing, Silverlight will not be able to communicate with the requested service.

A way to work around this would be to design an RSS reader that doesn't communicate with RSS feeds directly, but communicates with a service that in turn talks to the RSS feeds. The service, which you'd write yourself, can then be published on your own web server along with a `clientaccesspolicy.xml` or a `crossdomain.xml` file to allow your Silverlight application to access your service, if this service lives in a different domain than the Silverlight application.

RSS versions

There are different versions of RSS available. But luckily, most of these versions are supported by most RSS readers. You can find out more about these different versions at <http://en.wikipedia.org/wiki/RSS#Variants>.

Accessing a database in the cloud

Applies to Silverlight 3, 4 and 5

Cloud computing is a hot topic nowadays. Microsoft introduced Windows Azure as its cloud platform a couple of years ago, and we are starting to see a lot of businesses moving their applications and data to the cloud.

Windows Azure itself is a combination of several components. We can host services in the cloud as a **Hosted Service** (the component is often referred to as Windows Azure Compute). An implementation of SQL server in the cloud, namely **SQL Azure** is a second component part of the Azure platform. **Azure Storage** is another component, which offers us (nearly) unlimited storage of files and non-relational data. Other components, such as the **Service Bus** and **Access Control** exist as well, but they are out of the scope of this book.

Silverlight applications can benefit from Windows Azure as well. In this and the two following recipes, we'll take a look at some practical uses. We'll start this recipe by looking at how we can set up and use a database hosted in SQL Azure.

Getting ready

To work with Windows Azure, you'll need an account. You can register for an account at <http://windows.azure.com>. This account will then give you access to the previously mentioned components, such as SQL Azure. At the moment of writing, you can create a test account; however, you do need a credit card to sign up.

During development, you can use the **local development cloud**, known as **Windows Azure Emulator**. This tool allows you to do most of your testing on your local machine during development, before moving your application to the real cloud.

A sample database, `SilverlightAirways.mdf`, is included with the source code for this recipe. This database needs to be attached to your local SQL server instance.

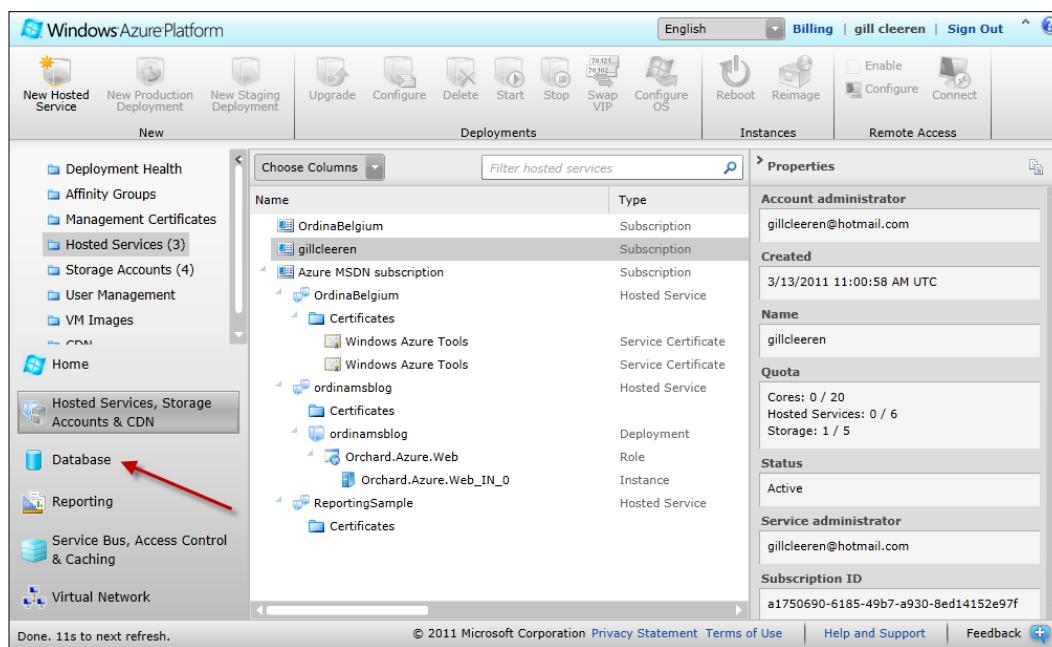
To follow along with this recipe, a starter solution is provided in Chapter07/Database_Cloud_Starter folder. The finished solution can be found in the Chapter07/Database_Cloud_Completed folder.

How to do it...

In this recipe, we are starting from a rather basic solution. We have a Silverlight application that reads out flight information from a service. In the default setup, this service reads data from a local SQL server database (this database can be on a physically different server than where the service is hosted).

Since we expect a lot of load on our database and we don't want to be bothered by managing our database servers ourselves, we decide to move the database to SQL Azure instead. Let's look at the steps we need to do on the database and see what changes, if any, are required for the Silverlight application using the service.

1. We first want to move the database (which is initially installed locally or on a database server that is not in the cloud) to SQL Azure. The first step is creating a database in the cloud from within the Windows Azure portal. To do so, go to <http://windows.azure.com>, log on with your account, and click on **Database**, as shown in the following screenshot:

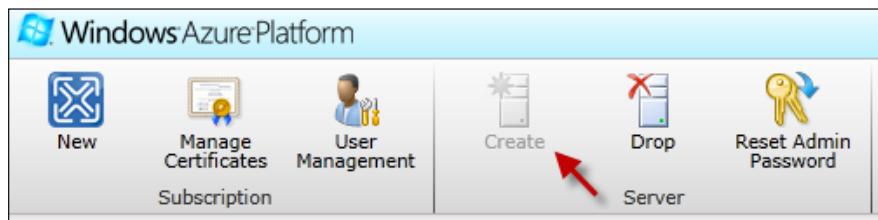


The screenshot shows the Windows Azure Platform portal. On the left, there is a navigation menu with options like New Hosted Service, New Production Deployment, New Staging Deployment, Home, Hosted Services, Storage Accounts & CDN, Database (which has a red arrow pointing to it), Reporting, Service Bus, Access Control & Caching, and Virtual Network. The main content area shows a list of databases under the 'Choose Columns' dropdown. The list includes:

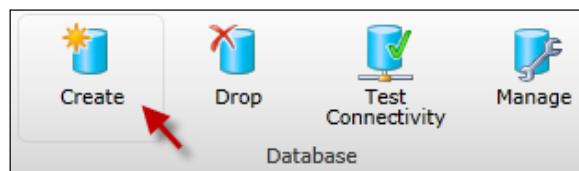
Name	Type
OrdinaBelgium	Subscription
gillcleeren	Subscription
Azure MSDN subscription	Subscription
OrdinaBelgium	Hosted Service
Certificates	Service Certificate
Windows Azure Tools	Service Certificate
Windows Azure Tools	Service Certificate
ordinamsblog	Hosted Service
Certificates	
ordinamsblog	Deployment
Orchard.Azure.Web	Role
Orchard.Azure.Web_IN_0	Instance
ReportingSample	Hosted Service
Certificates	

On the right, there is a properties panel for the 'gillcleeren' database, showing details such as Account administrator (gillcleeren@hotmail.com), Created (3/13/2011 11:00:58 AM UTC), Name (gillcleeren), Quota (Cores: 0 / 20, Hosted Services: 0 / 6, Storage: 1 / 5), Status (Active), Service administrator (gillcleeren@hotmail.com), and Subscription ID (a1750690-6185-49b7-a930-8ed14152e97f).

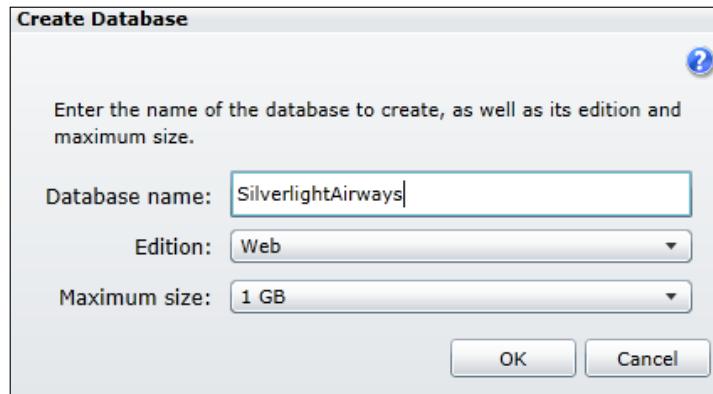
2. In the **Database** page, in the menu, click on the **Create** button in the **Server** tab to create a database server, as indicated in the following screenshot:



3. With the server created, click on the **Create** button in the **Database** tab, shown as follows:



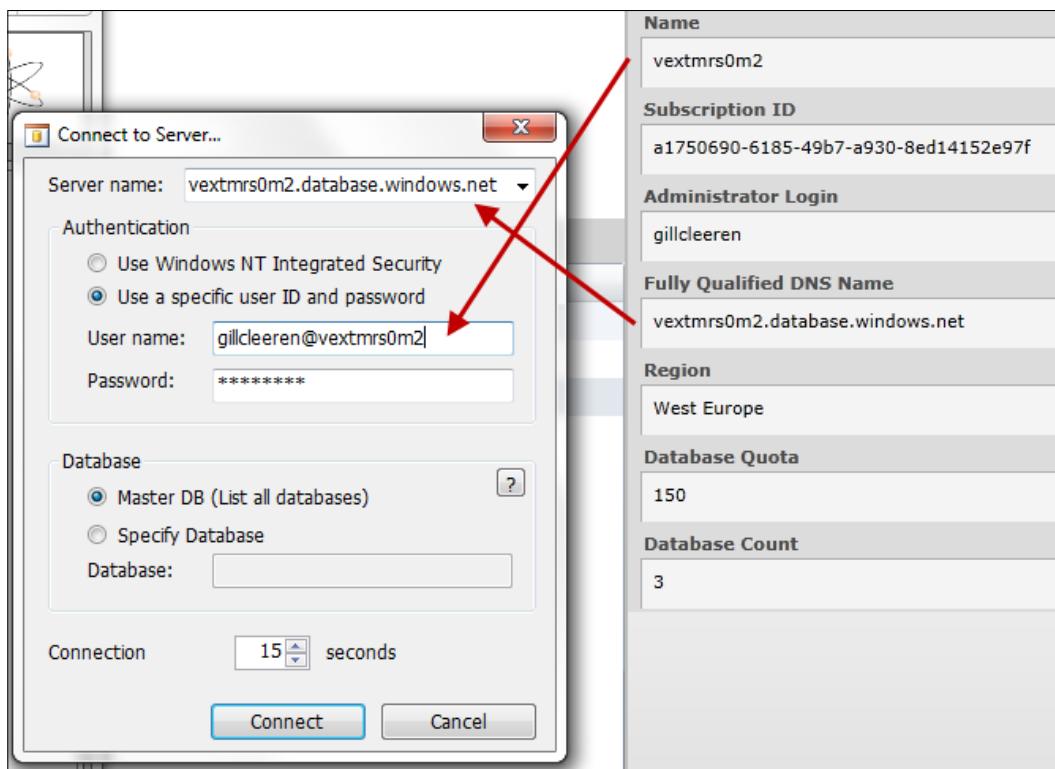
4. Give the database a name (here, **SilverlightAirways**). The other fields can keep their default value:



The database will now appear in the list of databases on your SQL Azure database server.

5. We are now ready to execute the migration of the local database to SQL Azure. To perform this migration, we'll use an open-source tool called **SQL Azure Migration Wizard**. This tool can be downloaded from <http://sqlazurermw.codeplex.com/>. When downloaded, run the tool.

6. In the first window, indicate you want to perform **Analyze/Migrate** of a SQL database.
7. In the next window, enter your credentials for your local database. Select the local **SilverlightAirways** database from the list.
8. Next, indicate you want to script all database objects. The tool will prompt you, asking to generate SQL script. Note that the tool also will migrate data for you. After this step, you'll get an overview, indicating whether or not issues were found.
9. With the local database scripted and ready to be migrated, we can move on to the cloud part. Enter the data for your SQL Azure database, as shown in the following screenshot. The data you need is shown in the **Database** page in the Azure portal. Note that your **User name** should be followed by @<YOUR_SERVER_NAME>. Click on **Connect** to connect with the database in the cloud:

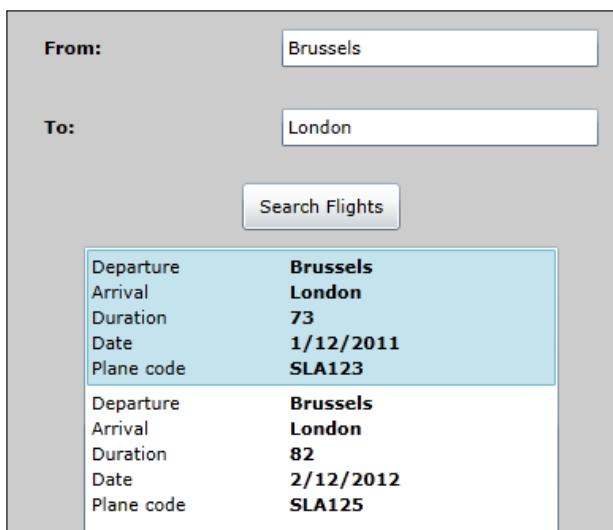


10. In the next screen, select your database that you created earlier in SQL Azure. The tool will now prompt you to execute the script. After clicking **OK**, you'll see the progress. Any errors that may occur are shown here as well. We're now ready with the migration of the database.

11. With the database moved to the cloud, we can now turn our attention to the application. In the web.config of the SilverlightAirwaysInTheCloud.Services project, we can change the connection string (the service is using EntityFramework here). Currently, this string is pointing to the **localhost**. Change it, so it points to your SQL Azure database as follows:

```
data source=YOUR_DATABASE_NAME.database.windows.net;initial  
catalog=SilverlightAirways;user id=YOUR_USER_NAME;password=  
YOUR_PASSWORD;multipleactiveresultsets=True;App=EntityFramework
```

12. Run the application now. You'll get a list of data, served directly from the cloud, as shown in the following screenshot:



How it works

SQL Azure is one of the components of Windows Azure. It's SQL server, but then in the cloud. Since it's hosted by Microsoft as a service, we get all the benefits such as scalability.

If we have a scenario where Silverlight, the service layer, and the database are hosted on our own managed servers, we can perform a migration of just the database to the cloud. The database is available using a connection string. By adding this new connection string to the configuration of the service, the service can communicate with the cloud-hosted database in the exact same way as it did with the **local** database.

For the Silverlight application, this process is entirely transparent. It continues to communicate with the service layer, which in turn communicates with the database in the cloud.

It should be clear that a migration to the cloud does not require an all-in approach: if we want to move just one tier to the cloud, we can do so.

Accessing a service in the cloud

Applies to Silverlight 3, 4 and 5

In the previous recipe, we moved the database, exposed over a service layer and used by a Silverlight application, to the cloud. This way, we leverage the power of the cloud on the database tier.

We can, however, go one step further. Instead of hosting the service layer on our own servers, we can also move the service layer into the cloud. In this case, the Silverlight application will communicate with the services, hosted in the cloud. These services in turn connect with the database, hosted in SQL Azure (as we looked at in the previous recipe).

In this recipe, we'll look at how we can move the services to become a Hosted Service in Windows Azure.

Getting ready

Just like the previous recipe, you'll need a Windows Azure account. You also need the **Windows Azure SDK** installed. You can download these from <http://www.microsoft.com/windowsazure/sdk>

This recipe builds on the previous recipe, where we moved the SQL server database into SQL Azure. It's required that you follow that recipe first.

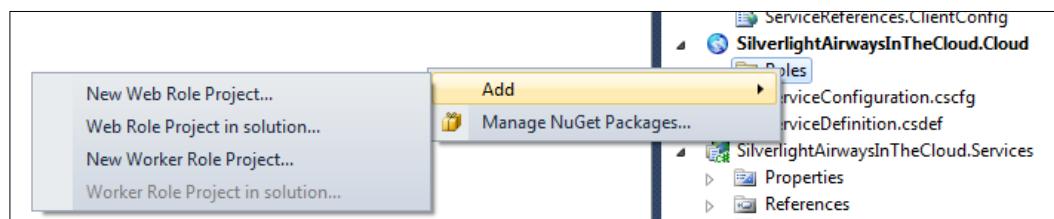
To follow along with this recipe, you can continue with the solution from the previous recipe. The finished solution for this recipe can be found in the `Chapter07/Service_Cloud_Completed`.

How to do it...

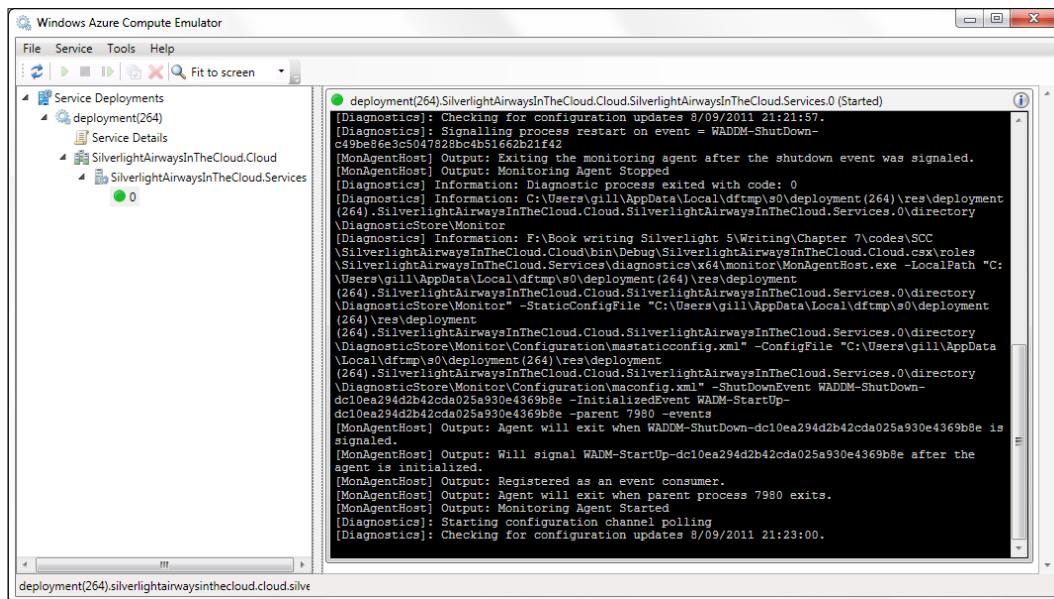
To benefit more from the advantages, such as the reliability offered by cloud computing, we can decide to move the service layer to Windows Azure as well. In this case, we can create a hosted service in Windows Azure, host the services there, and have a Silverlight application still hosted on our own servers, then connect to the cloud-hosted services. During development, we can use the local Windows Azure Emulator to test our development. Let's look at how we can move the service layer to Azure:

1. Make sure you have the solution open as outlined in the *Getting ready* section from this recipe. We'll first make changes to this solution to make the service ready to be hosted in Azure.

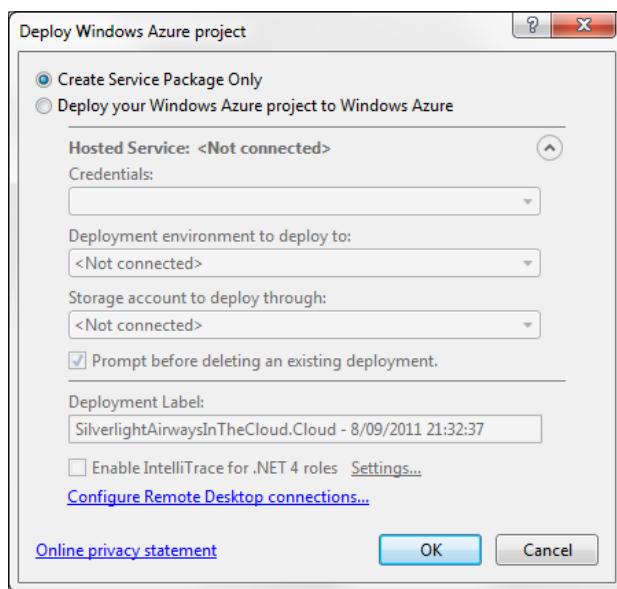
2. For each service project we want to host, we must add a Windows Azure project. With the tools installed, right-click on the solution and select **Add | New Project**. In the **Add New Project** window, select **Visual C# | Cloud** to add a new cloud project to your solution. Name it **SilverlightAirwaysInTheCloud.Cloud**. In the dialog that appears titled **New Windows Azure Project**, don't select anything and just click **OK**.
3. In the **Solution Explorer**, right-click the **Roles** folder in the newly added project, and select **Add | Web Role Project in Solution**. Select the **SilverlightAirwaysInTheCloud.Services** project. A role is added for the service project. You can consider a role as an instance that will execute code for you:



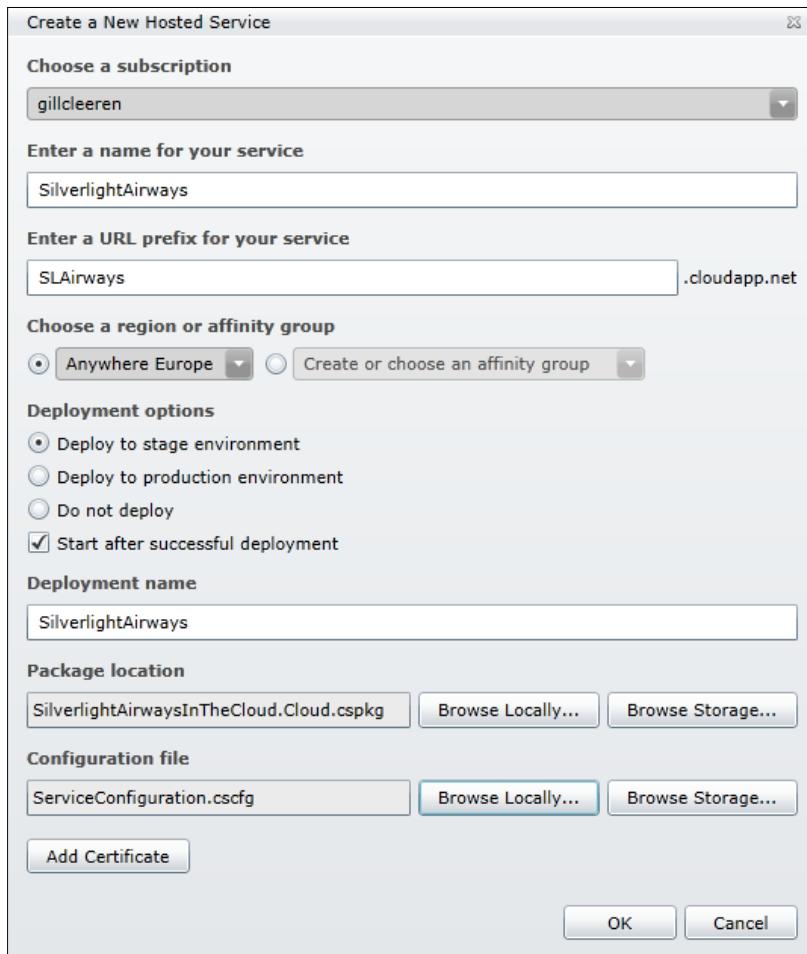
4. Test that the service still works as previously. Right-click on the project and select **Debug | Start New Instance**. The **Windows Azure Compute Emulator** now starts (you may only see an icon in your task bar, which you can click to show the emulator UI). This emulator can be seen as a local cloud, allowing you to test your applications. The emulator is shown as follows:



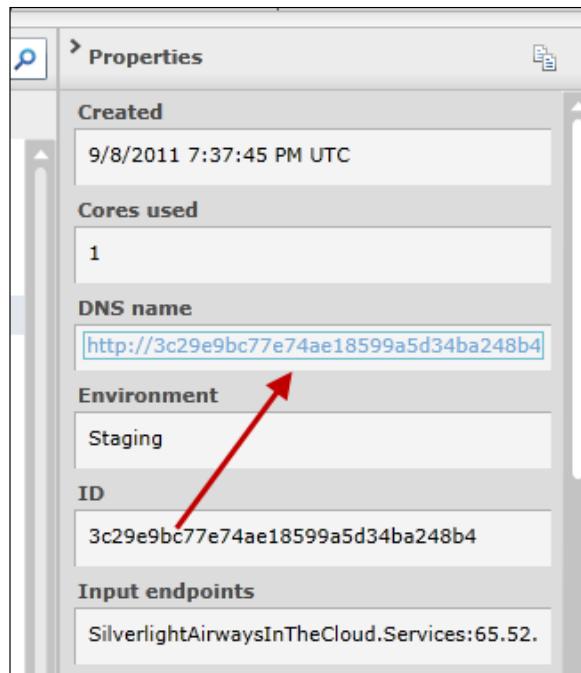
5. Browse to `http://127.0.0.1:81/FlightService.svc` (the port number may vary on your system, normally a browser window will open for you). You are now accessing the service hosted locally.
6. We're ready to create the package that we'll upload to the cloud. Right-click on the project and click on **Publish**. Select **Create Service Package Only**, as indicated in the following screenshot, so Visual Studio will create a Service Package file and a configuration file:



7. We are now going to move this service to the cloud. First, we need to create a hosted service in the Azure portal. Go to `http://windows.azure.com` and click on **Hosted Services, Storage accounts and CDN**. Select your subscription and click on **New Hosted Service**. Enter values in the dialog as shown in the following screenshot. Finish by selecting both the package and the configuration file created in the previous step:



8. After clicking **OK**, the service will be deployed. Note that this can take up to 20 minutes. If you get a dialog warning your service will only be deployed on one server, just ignore it for now (a Service Level Agreement only applies if the service is at least deployed on two machines). When ready, you can access your service in the cloud. The root address of the service can be found in the DNS name field. The service itself can be accessed by appending /FlightService.svc:



- Now that the service is hosted in the cloud, we can use it from our Silverlight application. Remove the existing service reference and add a new one, pointing to the address of the service in Azure.

The Silverlight application is now communicating with a service hosted in Windows Azure. The database is still hosted in SQL Azure.

How it works

One of, if not the most important aspect of Windows Azure is the ability to run code in the cloud. This is possible using a Hosted Service. A Hosted Service can basically host an application for you. Since it's in the cloud, you benefit again from things such as reliability and reduced **Total Cost of Ownership (TCO)**.

In a Hosted Service, we can run WCF services and ASP.NET applications. However, we can do more than that. WCF and ASP.NET are executed in what is known as a web role. A **web role** is basically an instance of **IIS** running in the cloud, executing your code. On the other hand, worker roles exist. These allow you to run any executable applications or code, even non-.NET code. This way, you can set up a custom database engine, such as MySQL or a web server, such as Apache. These in turn will then let you run non-managed code such as PHP.

Developing .NET applications or services to run in the cloud isn't difficult. In many cases, we just add a web role to the project, point to the service/web project. After publication, a configuration file and a package are created that you can upload to the Windows Azure portal. Azure will then create a virtual machine, running your code.

During the development, we have the ability to test our cloud projects locally in the **Windows Azure Compute Emulator**. This **local cloud** allows testing Hosted Services " and Azure Storage solutions.

Once deployed, the Silverlight application needs to update its service reference to now point to the service in the cloud. The application itself doesn't need any changes; it will just work in the same way with the service in Azure as it did with the service on our own servers.

See also

In the previous recipe, where we moved the database to SQL Azure, you can learn more about how to configure the data tier in Azure so that a Silverlight application can access data in the cloud.

Running a Silverlight application from the cloud

Applies to Silverlight 3, 4 and 5

In the two previous recipes, we moved both the database and the service layer into the cloud. So far, the Silverlight application still runs from a locally-hosted page. This means that both the XAP as well as the page (which can be ASPX, HTML, or even PHP) are still running from our own servers, while the service and the database being accessed are in the cloud already.

In this recipe, we'll take things one last step further—we will host the website hosting the Silverlight application in the cloud as well.

Getting ready

For this recipe, of course you need an account on Windows Azure as well. To follow along with this recipe, you need to complete the two previous recipes as well in order to have the database and the service already hosted in the cloud. The finished solution for this recipe can be found in the Chapter07/Silverlight_Cloud_Completed folder.

How to do it...

With the service hosted as a Hosted Service in Windows Azure and the database already running in SQL Azure, we can create a second Hosted Service, running the application that hosts Silverlight. The Silverlight XAP file will be packaged as well and run from the cloud. Let's take a look at the steps we need to take to set this up:

1. To accomplish running the site from the cloud, we have two options. Either we create another cloud project or we add a second web role to the already-existing project. For simplicity, let's take the latter here. In the **Solution Explorer**, right-click on the **Roles** folder in the **SilverlightAirwaysInTheCloud.Cloud** project, and select **Add | Web Role Project**. Select the **SilverlightAirwaysInTheCloud.Web** project. The result is a new role being added.
2. Run the **Publish** wizard by right-clicking on the cloud project and selecting **Publish**. In the dialog that appears, select **Create Service Package Only**. Windows Explorer will open and show you the location of the files.
3. Go to the Azure Management portal (windows.azure.com) and remove the current deployment. Create a new one (as outlined in the previous recipe).
4. Upload (again as outlined in the previous recipe) the Service Package and the configuration file.

After your deployment is ready, you can browse to the test page hosting your Silverlight application (you can still navigate to the service as well). Note that it is accessible only when adding a port to the URL. In the *How it works* section, we will look at why this is.

How it works

Running a Silverlight application hosted in the cloud comes down to publishing the site/page that does the hosting to the cloud as well. The XAP file is packaged as well and gets unpacked on the cloud server. From then on, the page requests the XAP from the `ClientBin` directory, which is now on the cloud server as well.

In the previous steps, we created a second cloud project and pointed this to the Silverlight-hosting site. Because of the fact that we are adding more than one role to the same cloud project, they are accessible individually by adding a port number. Why this happens is quite logical: we have one unique URI pointing to the single server now hosting two sites. This isn't an ideal situation in real life—we now have two sites (one for the services and one for the website hosting Silverlight) running on one server. This is the cheapest solution, but also the one that's the least able to scale!

This may not be what you want. You may want two separate machines, hosting one site each and also both accessible via a URI not appended with a port number. This is perfectly possible. Simply create a second cloud project and point it to the Silverlight-hosting site. You now need to create two separate Hosted Services in the Azure portal, one for each site. Note that this solution requires two machines, therefore doubling the cost.

There's more

In the way of working we choose, the XAP file was packaged in the Service Package file created by the **Publish** option. Now if we want to upgrade only the XAP file, we need to re-package and re-publish again.

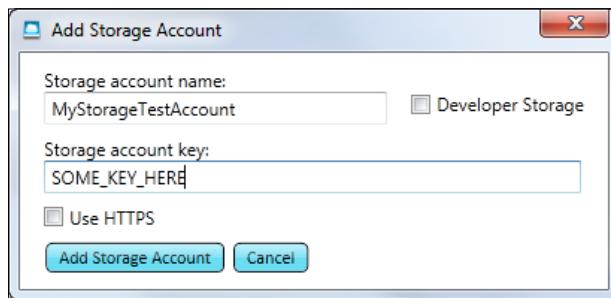
An easier approach might be placing the XAP file in Azure Storage. Azure Storage is an almost infinite file storage where we can place images, videos, documents, and why not, the XAP file. Files are stored in what is known as **blob storage**.

In the Azure Management Portal, go to **Hosted Services | Storage accounts**. If none have been created, create a new Storage account by clicking on the **New Storage Account** button at the top. Note that on the right, keys are generated, which you'll need next:

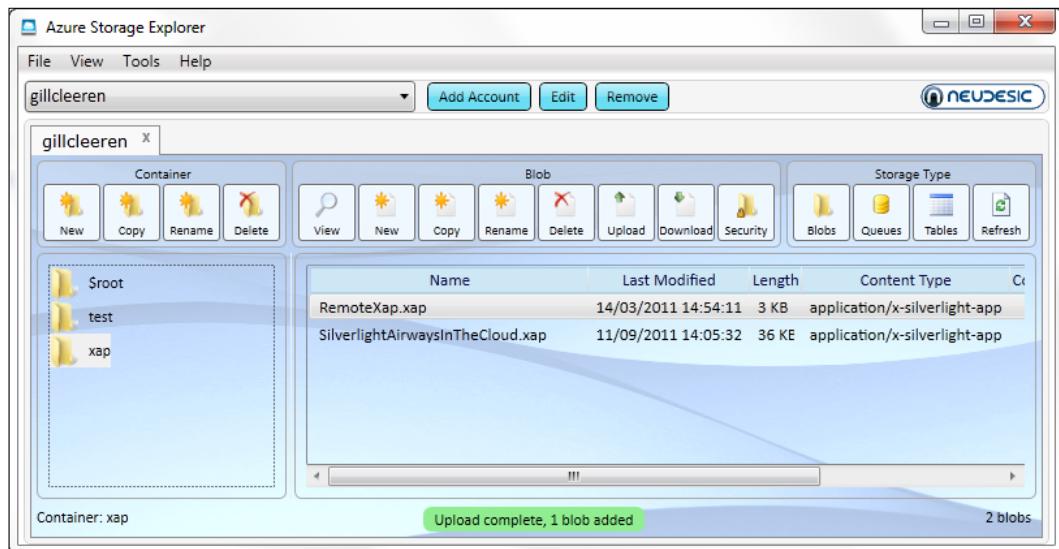
The screenshot shows the Windows Azure Platform Management Portal. The top navigation bar includes links for English, Billing, gill cleeren, and Sign Out. Below the navigation bar, there are several buttons: New Storage Account (highlighted with a red arrow), Delete Storage, View Access Keys, Regenerate Access Keys, Add Domain, Validate Domain, Delete Domain, and Custom Domain. The left sidebar contains links for Deployment Health, Affinity Groups, Management Certificates, Hosted Services (4), Storage Accounts (4) (highlighted with a red arrow), User Management, VM Images, Home, Hosted Services, Storage Accounts & CDN, Database, Reporting, Service Bus, Access Control & Caching, and Virtual Network. The main content area displays a table titled 'Choose Columns' with columns for Name and Type. It lists four storage accounts: OrdinaBelgium (Subscription), ordinasamples (Storage account), gillcleeren (Subscription), and gillcleeren (Storage account). A red arrow points to the 'gillcleeren' entry. To the right of the table is a 'Properties' pane for the selected 'gillcleeren' account. The properties include Primary access key (<Hidden> View), Secondary access key (<Hidden> View), Blob URL (gillcleeren.blob.core.windows.net), Table URL (gillcleeren.table.core.windows.net), Queue URL (gillcleeren.queue.core.windows.net), Name (gillcleeren), Last updated (9/11/2011 1:57:08 PM UTC), and Country/Region. At the bottom of the page, there is a note: 'Done. 24s to next refresh.' and links for © 2011 Microsoft Corporation, Privacy Statement, Terms of Use, Help and Support, and Feedback.

Managing files and folders (known as Containers in Azure Storage) can't be done with the portal. You can do so from code (programmatic access is possible), although for our case, we'll use an external tool called **CloudBerry Explorer** (available for free via <http://www.cloudberrylab.com/free-microsoft-azure-explorer.aspx>).

In this tool, add a new account (the name of the storage account you just created) and copy the Primary Keys from the Azure Portal.



CloudBerry will now give you an Explorer-like interface of your storage account. Create a new container and name it XAP. In the container, upload your XAP file of your application (don't upload the Azure package file!). In the case of our sample application, the result is as shown in the following screenshot:



Now we need to change the HTML/ASPX file so that it points to the XAP file hosted in Azure storage. From the Azure portal, copy the URL for your blob storage (in the form of NAME_OF_STORAGE_ACCOUNT.blob.core.windows.net).

In the HTML code, change the value of the source parameter as follows:

```
<div id="silverlightControlHost">
<object
    data="data:application/x-silverlight-2,"
```

```
type="application/x-silverlight-2"
width="100%"
height="100%">
<param name="source"
      value="http://NAME_OF_STORAGE_ACCOUNT.blob.core.windows.net/xap/SilverlightAirwaysInTheCloud.xap" />
```

When you publish and run the application again, the XAP file will be downloaded from blob storage, instead of being serviced from the Hosted Service itself.

See also

In the two previous recipes, we looked at moving the database to SQL Azure and moving the external service to a Hosted Service in Windows Azure.

Using socket communication in Silverlight

Applies to Silverlight 3, 4 and 5

Until now, all the communication with services has been initialized by the client. The Silverlight application makes a call to a service, which can be a WCF service, an ASMX service, or an RSS feed. After receiving the call, the service will start working by sending the response back to the Silverlight application. This kind of communication is typically carried out by HTTP, which is based on a request model. A response will be sent based on a request.

Sometimes, we need the opposite. The initiative needs to be taken by the server by pushing data to its clients. Initially, this client should register with the server, but from then on, the server can decide when it needs to send data to the connected applications.

This type of communication can be achieved using sockets in Silverlight. Sockets are the endpoints on both sides. The server and the client can both send data to an endpoint, making it possible for the server to push data back into the socket on the client side.

Sockets aren't used often (we'll take a look at the reason for this in the *How it works...* section), but they have their advantages, such as speed, less overhead, bi-directional communication, and so on. In this recipe, we'll take a look at what we need to do to get sockets working in a Silverlight application.

Getting ready

We're building the entire application in this recipe from scratch. The finished code for this recipe can be found in the `Chapter07/SilverlightStockSocket_Completed` folder in the code bundle that is available on the Packt site.

How to do it...

In this recipe, we'll build a service that sends up-to-date stock information and is similar to a stock ticker. It's possible that more than one client is connected simultaneously. In this case, all connected clients will receive updates from the server. We'll see that some of the techniques we used to communicate with a socket service are similar to connecting with a regular service, such as WCF. However, some are entirely different. For example, socket services also require a `clientaccesspolicy.xml` to be present. On the other hand, socket services can push data to the clients, which is typical for this type of communication.

We need to carry out the following steps to get a working socket service and client:

1. We'll start from scratch and build the application. Create a new Silverlight solution in Visual Studio called **SilverlightStockSocket**.
2. Every Silverlight application (unless it runs as a trusted application having elevated permissions) that calls a service in a domain, other than the one in which it is hosted requires a `clientaccesspolicy.xml` or a `crossdomain.xml` file (as we saw in the *Configuring cross-domain calls* recipe). The same holds true for a socket-based application. When dealing with services hosted by a web server, serving the `clientaccesspolicy.xml` is handled by the server software itself. This isn't the case with a socket server. With that being said, our first task will be to build a policy server, which is a piece of software that will open a socket and serve the `clientaccesspolicy.xml` when requested. We'll build the policy server as a Console application. To do this, add a new Console application to the solution and name it **StockSocketPolicyServer**.
3. In this new project, we'll first include the policy file itself. Add a new XML file to the project and name it as `clientaccesspolicy.xml`. In the **Properties** window, set its **Build Action** to **None** and **Copy to Output Directory** to **Copy Always**. Due to this, while building, the XML file is copied to the same directory as the executable of the application. The following code represents the content of the file:

```
<?xml version="1.0" encoding = "utf-8"?>
<access-policy>
    <cross-domain-access>
        <policy>
            <allow-from>
                <domain uri="*" />
            </allow-from>
            <grant-to>
                <socket-resource port="4530" protocol="tcp" />
            </grant-to>
        </policy>
    </cross-domain-access>
</access-policy>
```

4. To serve this file, we need a policy server containing the code that reads out the file and sends its contents to the connecting Silverlight application. The policy server's main task is to listen for incoming connections and respond with the contents of the policy file. Add a new class called `PolicyServer.cs` to the project.
5. The policy server reads the content of the XML file when it starts up. The content is loaded into a byte array as shown in the following code:

```
public class PolicyServer
{
    private byte[] clientaccesspolicy;
    public void StartPolicyServer()
    {
        using (FileStream stream = new
            FileStream("clientaccesspolicy.xml", FileMode.Open))
        {
            clientaccesspolicy = new byte[stream.Length];
            stream.Read(clientaccesspolicy, 0,
                clientaccesspolicy.Length);
        }
    }
}
```

6. After reading out the content, the policy server has to start listening for incoming requests for the policy file. A `TcpListener` object is required to do this. These requests will always be sent to port 943. So, we configure the `TcpListener` instance to listen to that port. As shown in the following code, the `BeginAcceptTcpClient` method starts a new thread on which the actual listening is done. Because of this, it returns immediately:

```
private TcpListener tcpListener;
public void StartPolicyServer()
{
    ...
    tcpListener = new TcpListener(IPAddress.Any, 943);
    tcpListener.Start();
    tcpListener.BeginAcceptTcpClient(new AsyncCallback
        (OnBeginAcceptTcpClient), null);
}
```

7. When a client eventually connects, the `OnBeginAcceptTcpClient` callback will be invoked. In this method, we begin the process of receiving data asynchronously using the resulting `TcpClient` instance. The data that will be sent by the client is the request for the policy file:

```
private TcpClient tcpClient;
private byte[] receivedBytes;
private static string policyRequestString =
```

```

    "<policy-file-request/>";
private void OnBeginAcceptTcpClient(IAsyncResult ar)
{
    receivedBytes = new byte[policyRequestString.Length];
    tcpClient = tcpListener.EndAcceptTcpClient(ar);
    tcpClient.Client.BeginReceive(receivedBytes, 0,
        policyRequestString.Length, SocketFlags.None,
        new AsyncCallback(OnReceive), null);
}

```

- The OnReceive callback is invoked when the operation is complete. In this method, we check if the received string data is equal to <policy-file-request/>. This string data is a request for the policy file. Now, we find out that the request was a policy request, and we start sending the content of the clientaccesspolicy.xml file. Finally, we need to start listening again for new incoming requests. This is shown in the following code:

```

private void OnReceive(IAsyncResult ar)
{
    int receivedLength = tcpClient.Client.EndReceive(ar);
    string policyRequest = Encoding.UTF8.GetString
        (receivedBytes, 0, receivedLength);
    if (policyRequest.Equals(policyRequestString))
    {
        tcpClient.Client.BeginSend(clientaccesspolicy, 0,
            clientaccesspolicy.Length, SocketFlags.None,
            new AsyncCallback(OnSend), null);
    }
    tcpListener.BeginAcceptTcpClient(new
        AsyncCallback(OnBeginAcceptTcpClient), null);
}
private void OnSend(IAsyncResult ar)
{
    tcpClient.Client.Close();
}

```

- The policy server also needs to close at some point. The Stop method is used for this as shown in the following code:

```

public void StopPolicyServer()
{
    tcpListener.Stop();
}

```

10. Managing the policy server (starting and stopping) can be carried out from the `Main` method in the `Program.cs` file that was added automatically when we created the project. This is shown in the following code:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Starting policy server");
        PolicyServer policyServer = new PolicyServer();
        policyServer.StartPolicyServer();
        Console.WriteLine("Policy server successfully started");
        Console.WriteLine("Press any key to exit the server");
        Console.ReadLine();
        policyServer.StopPolicyServer();
    }
}
```

11. Now that the policy server is in place, it's time to build the real socket server. The code for this is quite similar to the code of the policy server. Add another Console application to the solution and name it as **StockSocketServer**. Add a new class to this project called `SocketServer`.
12. A client connects to the socket server to receive updates on stock quotes. We'll simulate this using a `Timer` that sends new, random stock data to all connected clients every two seconds. Also, all the clients that are connected to the socket server at any given point of time need to be stored by the server. As shown in the following code, we create a `List<StreamWriter>` to achieve this:

```
Timer stockTimer;
List<StreamWriter> clientConnections;
public SocketServer()
{
    stockTimer = new Timer();
    clientConnections = new List<StreamWriter>();
}
public string GetStockInfo()
{
    double randomStockValue =
        30 + Math.Round(10 * new Random().NextDouble(), 2);
    return "MSFT: " + randomStockValue.ToString();
}
```

The `Timer` instance in this code, requires adding a using statement to the `System.Timers` namespace.

13. Just like the policy server, the socket server will listen for incoming requests using the `TcpListener`. However, here the listener needs to work on a port between 4502 and 4532. Instantiating and starting the listener is carried out in the `StartSocketServer` method. We also configure the `Timer` instance in this method, as shown in the following code:

```
private TcpListener tcpListener;
public void StartSocketServer()
{
    stockTimer.Interval = 2000;
    stockTimer.Enabled = true;
    stockTimer.Elapsed += new
        ElapsedEventHandler(stockTimer_Elapsed);
    stockTimer.Start();
    tcpListener = new TcpListener(IPAddress.Any, 4530);
    tcpListener.Start();
    tcpListener.BeginAcceptTcpClient
        (OnBeginAcceptTcpClient, null);
}
```

14. In the `BeginAcceptTcpClient` callback, we access the stream associated with the client connection. This stream is first stored in the previously mentioned `List<StreamWriter>`. We push data from the server to the client using the `Write` method. Finally, we start listening for new incoming requests. All this is shown in the following code:

```
private TcpClient tcpClient;
public void OnBeginAcceptTcpClient(IAsyncResult ar)
{
    Console.WriteLine("Client connected successfully");
    tcpClient = tcpListener.EndAcceptTcpClient(ar);
    StreamWriter streamWriter = new
        StreamWriter(tcpClient.GetStream());
    clientConnections.Add(streamWriter);
    streamWriter.AutoFlush = true;
    streamWriter.Write(GetStockInfo());
    //wait again for new connection
    tcpListener.BeginAcceptTcpClient(OnBeginAcceptTcpClient,
        null);
}
```

15. When the `Elapsed` event of the `Timer` triggers, we want to update all clients by sending new data. To do this, we loop over the `List<StreamWriter>` and call the `Write` method again, thereby passing new data to the clients. This is shown in the following code:

```
void stockTimer_Elapsed(object sender, ElapsedEventArgs e)
{
    if (clientConnections != null)
    {
        foreach (var clientConnection in clientConnections)
        {
            if (clientConnection != null)
                clientConnection.Write(GetStockInfo());
        }
    }
}
```

16. Closing the socket server consists of two things—closing each `StreamWriter` instance followed by stopping the listener as shown in the following code:

```
public void StopSocketServer()
{
    foreach (var streamWriter in clientConnections)
    {
        streamWriter.Close();
    }
    tcpListener.Stop();
}
```

17. Similar to the policy server, the socket server is managed from the `Main` method in the `Program.cs` file, as shown in the following code:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Starting socket server");
        SocketServer socketServer = new SocketServer();
        socketServer.StartSocketServer();
        Console.WriteLine("Socket server successfully started");
        Console.WriteLine("Press any key to exit the server");
        Console.ReadLine();
        socketServer.StopSocketServer();
    }
}
```

18. Finally, we have created both the policy server and the socket server. Let's now turn our attention to the Silverlight application that needs to connect with this socket server. The UI layout is pretty straightforward and can be found in the code bundle. Its primary components are a Button and a TextBlock. We want to connect with the socket server, once the Button has been clicked. The results sent back from the server should be shown in the TextBlock.
19. In the Click event of the Button, we start by defining the socket endpoint using the DnsEndPoint class. Due to this, the Silverlight application knows where the host is located. It then creates the Socket object as well as the SocketAsyncEventArgs instance as shown in the following code. The latter represents the asynchronous operation with the socket:

```
private void StartButton_Click(object sender, RoutedEventArgs e)
{
    DnsEndPoint endPoint = new DnsEndPoint
        (Application.Current.Host.Source.DnsSafeHost, 4530);
    Socket socket = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
    SocketAsyncEventArgs args = new SocketAsyncEventArgs();
    args.UserToken = socket;
    args.RemoteEndPoint = endPoint;
    args.Completed += new
        EventHandler<SocketAsyncEventArgs>(OnSocketConnected);
    socket.ConnectAsync(args);
}
```

- 20 The OnSocketConnected callback is fired when the client has finished attempting to connect with the server socket. In this method, we check if the connection was a success. If it was, we create a buffer that will contain the received data, and start the receiving process using the ReceiveAsync method. This is shown in the following code:

```
private void OnSocketConnected(object sender,
    SocketAsyncEventArgs e)
{
    if (e.SocketError == SocketError.Success)
    {
        byte[] response = new byte[1024];
        e.SetBuffer(response, 0, response.Length);
        e.Completed -= new
            EventHandler<SocketAsyncEventArgs>(OnSocketConnected);
        e.Completed += new
            EventHandler<SocketAsyncEventArgs>(OnSocketStartReceive);
        Socket socket = (Socket)e.UserToken;
        socket.ReceiveAsync(e);
    }
}
```

21. We can access the received data in the `OnSocketStartReceive` callback. We can't update the UI directly as the code is now executing on a background thread. Therefore, we need to use `Dispatcher.BeginInvoke` to get access to the UI thread. To register to receive new updates coming from the server, we call the `ReceiveAsync` again as shown in the following code:

```
private void OnSocketStartReceive(object sender,
    SocketEventArgs e)
{
    string data = Encoding.UTF8.GetString(e.Buffer, e.Offset,
        e.BytesTransferred);
    Dispatcher.BeginInvoke(() => StockTickingTextBlock.Text = data);
    Socket socket = (Socket)e.UserToken;
    socket.ReceiveAsync(e);
}
```

22. We have now arrived at a point where we can test the application. To test the application, we need to start the Silverlight application, the policy server, and the socket server. Starting these two servers is necessary, because they run as separate projects and need to be available, so that the Silverlight application can connect with them. To achieve this, first start the application by setting the web project as a Startup project and running it by pressing the *F5* key. Secondly, right-click on both the policy server and the socket server projects in the **Solution Explorer** and select **Debug | Start new instance**.

The following screenshot shows the Silverlight application as well as both the server applications running:



How it works...

When we need a type of service that is able to push data from the server to the client, a socket-based application might be a good solution. Socket-based applications can also be considered as an option when pure speed for communicating with a service is our top priority. This can be explained by the TCP protocol they use. All other services use HTTP, which is a higher layer on top of TCP. HTTP is slower than pure TCP, and it can't push data back to the client as it requires a request for every response that goes out.

When would we actually need this increased performance or this bi-directional, push-enabled communication? If raw data is all that passes over a wire, and it needs to move fast, a socket can be used. A stock ticker is a great example of this. The client (some application) registers with the socket server and starts receiving updates from then on without asking!

The policy server

Just like other service communications, Silverlight needs to know that the server allows connecting from another domain. In other words, it will check for cross-domain restrictions. For example, when connecting to a WCF service, Silverlight will send a request for the `clientaccesspolicy.xml` file to the web server. If not found, it will request for the `crossdomain.xml` file. If neither of these files is found, or the file enforces restrictions on who can access the service (only requests from specific domains are allowed), Silverlight will not allow the call to the service. Otherwise, it will go ahead and let us communicate with the service.

In the case of a socket-based application, the server can be any type of application, ranging from a console application to a Windows service. These types of applications aren't web servers that will automatically serve the policy file when requested. Therefore, we need to write a policy server.

This policy server uses a `TcpListener` to listen for incoming requests. Policy requests always arrive at port 943. This port needs to be specified when creating the `TcpListener` instance.

The `BeginAcceptTcpClient` method of the `TcpListener` class actually starts the listening process on another thread. That is why this method returns immediately and then waits for a client to connect. Once data is received, it is checked to see if the request is really a policy request. If it is so, the contents of the `clientaccesspolicy.xml` file is sent back to the requesting application (in our case, a Silverlight application). This way, Silverlight can determine if the application can access the service.

However, since Silverlight 4, it is possible to have a Silverlight application connect over HTTP port 80 instead of TCP port 443. In this case, the `clientaccesspolicy.xml` file should also be at the root of the domain, just like with other cross-domain communication. This way, we can have an HTTP server (such as IIS) authorize a socket connection from a Silverlight application, without the need to deploy the policy server and without opening communication on port 943.

To enable this, we should set the `SocketAsyncEventArgs`.

`SocketClientAccessPolicyProtocol` property to `SocketClientAccessPolicyProtocol.Http` of the `SocketAsyncEventArgs` instance, passed to the `Socket.ConnectAsync`. By default, the `SocketAsyncEventArgs.SocketClientAccessPolicyProtocol` is set to `SocketClientAccessPolicyProtocol.Tcp`.

The socket server

The actual socket-based communication will take place between the Silverlight application and the socket server application. Just like the policy server, this server will also listen for incoming requests using the `TcpListener` class. Multiple client applications can connect to the server. To manage this, the server uses a `List<StreamWriter>`. In this list, we store a `StreamWriter` that references the network stream to the connected client application. Whenever new data needs to be sent to the clients, we move through the list and data is sent over each encountered writer.

The socket server can listen for incoming requests on any port varying between 4502 and 4532. It's important that whenever a new client has connected, we start listening for new incoming requests using the following line of code:

```
tcpListener.BeginAcceptTcpClient(OnBeginAcceptTcpClient, null);
```

Connecting the Silverlight application

When connecting to the socket server from the client side, we can perform three tasks: connect to the server, send data, and receive data. We can do this using the `ConnectAsync`, `SendAsync`, and `ReceiveAsync` methods respectively. In this sample, we haven't sent data to the server. However, it's similar to receiving data. We can't work with the `TcpClient` as it's not available in Silverlight. We need to use the `Socket` class directly.

A `SocketAsyncEventArgs` instance is required to communicate with the socket server. With this instance, we can store a reference to the socket and the endpoint for the connection, stored as a `DnsEndPoint`.

The `ConnectAsync` method on the `Socket` instance will make the actual call connecting with the server, using the `SocketAsyncEventArgs` instance. The connection call is carried out asynchronously and when it returns, we can check if the attempt to connect with the server has succeeded. If it does succeed, we can create a buffer, so that some received data can be stored inside it.

Using the following code:

```
Socket socket = (Socket)e.UserToken;
socket.ReceiveAsync(e);
```

We can tell the client to start listening for any incoming messages from the server. The callback method is invoked whenever a message is received. As we're now in a different thread than the UI thread, we can no longer access the UI elements, such as the `TextBlock` control. Thus, we need to use `Dispatcher.BeginInvoke`. This allows us to execute the specified delegate on the thread that is associated with the `Dispatcher` (in this case, the UI thread). The delegate is simply a call for updating the value of the `TextBlock` control.

8

Talking to WCF and ASMX Services

In this chapter, we will cover:

- ▶ Invoking a service that exposes data
- ▶ Invoking a service such as Bing.com
- ▶ Optimizing performance using binary XML
- ▶ Debugging WCF service errors with Silverlight
- ▶ Using ASP.NET Membership authentication in Silverlight
- ▶ Uploading files to a WCF service
- ▶ Displaying images as a stream from a WCF service

Introduction

When building services in .NET, most developers will choose WCF (Windows Communication Foundation). WCF was introduced with .NET 3.0 as an API that eased building service-oriented applications. It aims at unifying several communication APIs, such as remoting, classic web services, and so on.

Due to its wide adoption, it's easy to understand that Microsoft included support for communication with WCF services in Silverlight. In version 5, the combination of Silverlight and WCF has the most options when it comes to communicating with services from Silverlight. When starting the development of an enterprise, WCF should default your choice (that is, if you have the ability to influence the technology choice for the service layer).

ASMX or classic web services are still widely used as well. We can also connect with classic services from Silverlight, although not all options available with WCF are possible with these types of services.

In this chapter, we'll look at communicating with services we create ourselves, as well as third-party services. Sites such as Bing offer an API that includes service endpoints.

With Silverlight 3, support was added for binary encoding of the exchanged information, which results in faster data transfer. Also, Silverlight supports fault handling. Until the introduction of Silverlight 3, faults thrown by the service were not visible in Silverlight. This resulted in a bad debugging experience. In this chapter, we'll see how we can leverage these features.

If you already have an existing ASP.NET site using ASP.NET Membership, and you now want to build (part of) that site with Silverlight, you can allow Silverlight to connect to a service implementation of the Membership API. We'll see how we can use these so-called application services.

Services aren't limited to sending and receiving textual data. Silverlight can send or receive an image from a service. In this case, the image is sent as binary data. We'll look at how we can create a service that is capable of this type of communication and how Silverlight can work with it.

Invoking a service that exposes data

Applies to Silverlight 3, 4 and 5

When working on a Silverlight project that involves services, WCF is the preferred choice for building a service if we have to create both the service and the Silverlight client application that will use it. Using WCF gives us complete control over what types will be sent to the client and for each type. We can also specify whether or not a field should be included in the client-side copy of the type.

When we want to build a Silverlight application that works with the data available on the service (perhaps coming from a database or another external service), we need to ask ourselves two questions. How should we design the service that exposes the data so that it can be accessible from Silverlight? Secondly, how should we go about designing the Silverlight application so that it communicates with the service?

In this recipe, we'll take a look at finding answers to both these questions.

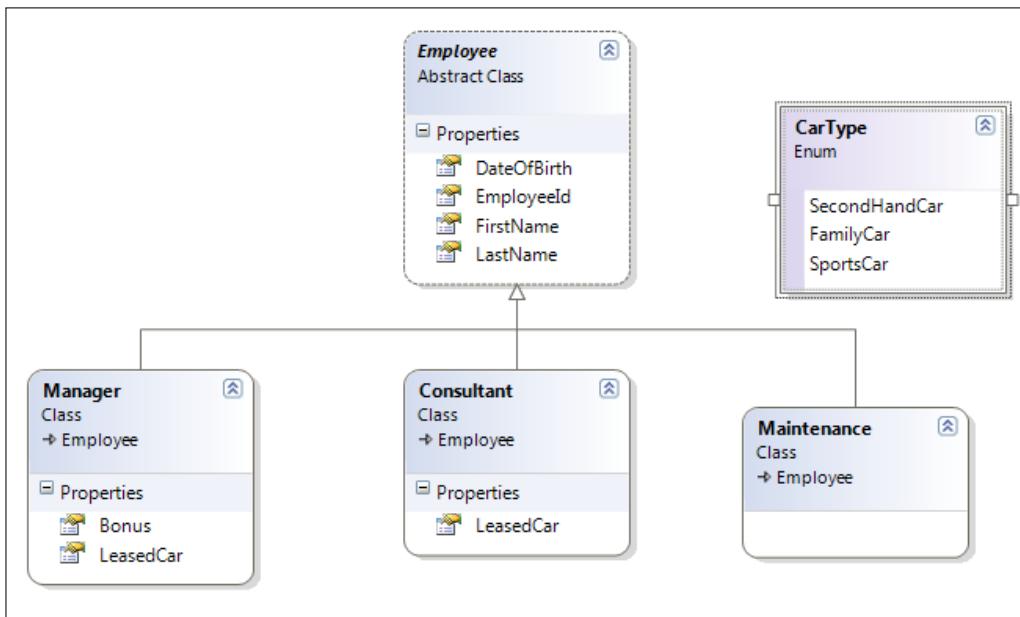
Getting ready

We'll build the application in this recipe from scratch. However, the complete code for this recipe can be found in the `Chapter08/SilverlightEmployeeBrowser` folder in the downloads for this book.

How to do it...

To show how we can connect from Silverlight to a WCF service that exposes data, we'll start quite logically with designing the service itself. For this recipe, we'll assume that we are building an easy employee overview screen, where the user can see all the employee information. The employee data is exposed by the service. The Silverlight application will connect to the service and work with the data on the client side. The following are the steps we need to execute to get this working:

1. For this recipe, we'll start from an empty Silverlight application. Create a new Silverlight solution in Visual Studio, and name it **SilverlightEmployeeBrowser**. As usual, Visual Studio will create a solution containing both a Silverlight application called **SilverlightEmployeeBrowser** and a hosting website called **SilverlightEmployeeBrowser.Web**.
2. Let's first focus on the web application. The data we want to expose consists of employee data. The following class diagram shows the relation. Add all the classes and the CarType enumeration in a folder called Model within the project. Note that Employee is an abstract base class:



3. The following is the code for the `Employee` class. Note the attributes added to the class. By specifying the `DataContractAttribute`, we state that this type can be sent over the wire, therefore becoming available on the client-side. Also, we have marked all fields with the `DataMemberAttribute`. Every field that is attributed with the `DataMemberAttribute` will be included in the type being sent over the wire to the client side. Finally, we also add the `KnownTypeAttribute` in the base class. This attribute marks the types that should be included in the serialization process as follows:

```
[DataContract]
[KnownType(typeof(Manager))]
[KnownType(typeof(Consultant))]
[KnownType(typeof(Maintenance))]
public abstract class Employee
{
    [DataMember]
    public int EmployeeId { get; set; }
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    [DataMember]
    public DateTime DateOfBirth { get; set; }
}
```

4. The new attributes require a new using statement for the `System.Runtime.Serialization` added at the top of this class.

The following is the code for the `Manager` class that inherits from `Employee`:

```
public class Manager : Employee
{
    [DataMember]
    public CarType LeasedCar { get; set; }
    [DataMember]
    public double Bonus { get; set; }
}
```

The other classes in the hierarchy are similar and can be found in the completed sample.

5. Next, we will create a class called `EmployeeRepository` that will load in the sample data. In a real-world scenario, we would load data from a database or an external service. The following is the code for this class. It uses a static `List<Employee>` that is filled upon first creation of the `EmployeeRepository` class:

```
public class EmployeeRepository
```

```
{  
    private static List<Employee> allEmployees;  
    public EmployeeRepository()  
    {  
        FillEmployees();  
    }  
    private void FillEmployees()  
    {  
        if(allEmployees == null)  
        {  
            allEmployees = new List<Employee>()  
            {  
                new Manager()  
                {  
                    EmployeeId=1,  
                    FirstName="Gill",  
                    LastName="Cleeren",  
                    LeasedCar=CarType.SportsCar,  
                    Bonus=10000.00,  
                    DateOfBirth=new DateTime(1980, 1, 1)  
                },  
                new Consultant()  
                {  
                    EmployeeId=2,  
                    FirstName="John",  
                    LastName="Smith",  
                    LeasedCar=CarType.FamilyCar,  
                    DateOfBirth=new DateTime(1976, 11, 9)  
                },  
                new Consultant()  
                {  
                    EmployeeId=3,  
                    FirstName="Jeff",  
                    LastName="Jones",  
                    LeasedCar=CarType.SecondHandCar,  
                    DateOfBirth=new DateTime(1983, 3, 12)  
                },  
                new Consultant()  
                {  
                    EmployeeId=4,  
                    FirstName="Lindsey",  
                    LastName="Clarks",  
                    LeasedCar=CarType.FamilyCar,  
                    DateOfBirth=new DateTime(1984, 6, 7)  
                }  
            };  
        }  
    }  
}
```

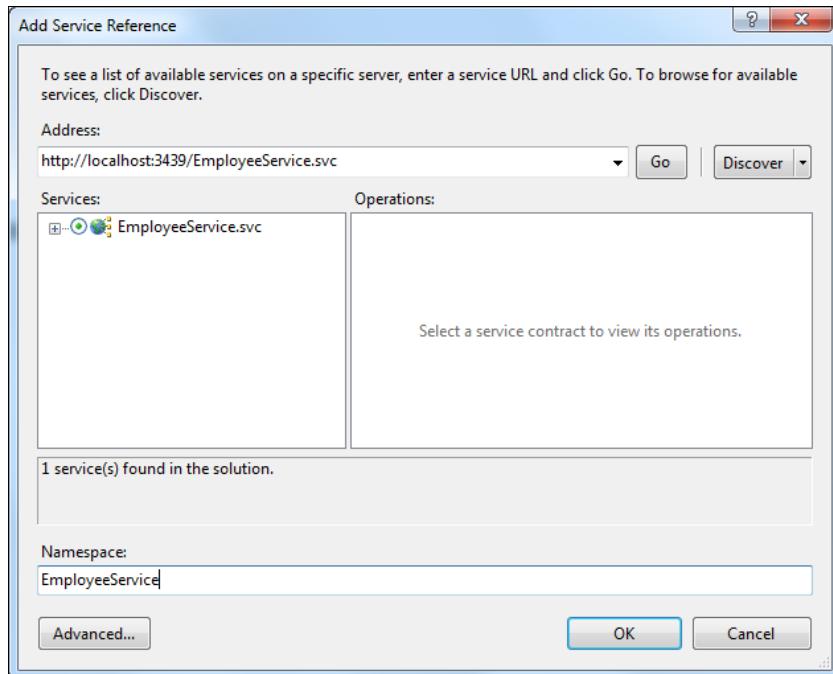
```
        },
        new Maintenance()
    {
        EmployeeId=5,
        FirstName="Clay",
        LastName="Richards",
        DateOfBirth=new DateTime(1960, 9, 22)
    },
    new Maintenance()
    {
        EmployeeId=6,
        FirstName="Marie",
        LastName="Smith",
        DateOfBirth=new DateTime(1963, 2, 19)
    },
},
}
public List<Employee> AllEmployees
{
    get
    {
        return allEmployees;
    }
}
```

6. Now that we have the model and the repository, we're ready to add a WCF service to this project. Right-click on the **SilverlightEmployeeBrowser.Web** project node in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** dialog, we have two options for adding a WCF service—a regular WCF Service and a Silverlight-enabled WCF Service. Select the latter and name the service **EmployeeService**. Visual Studio will add a *.svc file, a *.svc.cs file, and make some changes in the web.config file to make the service accessible.
7. The service exposes two methods. The first one retrieves all employees, while the second one retrieves an employee based on the employee's ID. Each method or operation that should be available on the client side needs to be attributed with the **OperationContract** attribute. In a WCF scenario, the service itself should have the **ServiceContract** attribute. The service with its two methods is shown in the following code:

```
[ServiceContract(Namespace = "")]
[AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Allowed)]
public class EmployeeService
```

```
{
    [OperationContract]
    public List<Employee> GetAllEmployees()
    {
        return new Model.EmployeeRepository().AllEmployees;
    }
    [OperationContract]
    public Employee GetEmployeeById(int employeeId)
    {
        return new Model.EmployeeRepository().AllEmployees.Where
            (e => e.EmployeeId == employeeId).FirstOrDefault();
    }
}
```

8. We're now ready on the server side. Build the project before continuing.
9. In the Silverlight application, right-click on the project node in the **Solution Explorer** and select **Add Service Reference....** In the dialog box that appears, click on the **Discover** button, and if the service can be connected to without errors, it will appear in the list as shown in the following screenshot. Select the service and enter **EmployeeService** in the **Namespace:** field. An error will occur if the service wasn't built. After clicking on the **OK** button, Visual Studio will attempt to build a proxy that is roughly a client-side copy of the service class:



-
10. The UI of the application is very simple. The complete code list can be found in the code downloads. As shown a bit later in this recipe, it contains a TextBox and a Button, which should trigger a call to the GetEmployeeId service operation. The retrieved details are to be shown in a detailed grid. Also, while loading the application for the first time, a call to the GetAllEmployees method should be triggered and the result of this call should be shown in the DataGrid. Here we'll focus on the latter.
 11. When the control has loaded, a call to the LoadEmployees method is invoked. In this method, we create an instance of the proxy class, which was generated by Visual Studio when we connected with the service. Next, we specify the callback method. Communication with a WCF service, similar to all other types of service connections, will take place asynchronously. Finally, we perform the actual invocation of the service method. All this is shown in the following code:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    LoadAllEmployees();
}

private void LoadAllEmployees()
{
    EmployeeService.EmployeeServiceClient proxy = new
        SilverlightEmployeeBrowser.EmployeeService
            .EmployeeServiceClient();
    proxy.GetAllEmployeesCompleted +=
        proxy_GetAllEmployeesCompleted;
    proxy.GetAllEmployeesAsync();
}
```

12. If no errors occurred in the callback method, we get access to the result of the service call via the Result property of the SilverlightEmployeeBrowser.EmployeeService.GetAllEmployeesCompletedEventArgs parameter. This property is of the same type as returned by the service method (in this case a List<SilverlightEmployeeBrowser.Employee>). For the DataGrid, we set the list as ItemsSource as shown in the following code:

```
void proxy_GetAllEmployeesCompleted(object sender,
    SilverlightEmployeeBrowser.EmployeeService
        .GetAllEmployeesCompletedEventArgs e)
{
    if (e.Error == null)
        EmployeeDataGrid.ItemsSource = e.Result;
    else
        ErrorTextBlock.Text = "An error occurred while communicating
        to the service";
}
```

13. All employees should now be shown in the `DataGridView`, as shown in the following screenshot. Calling the `GetEmployeeById` operation is similar. The code for this can be found in the code downloads:



How it works...

WCF services are a preferred way of authoring services in the .NET framework. From the Silverlight point of view, the WCF services are also the best choice. Let's take a look at the specifics for both the service and the Silverlight application connecting to the service.

The WCF service

Adding a WCF service is possible using one of the two item templates available in Visual Studio—the WCF Service template or the Silverlight-enabled WCF Service template. The first one configures the service to use `wsHttpBinding`, which is the default for WCF. However, Silverlight can't work with this type of binding. So, if we choose to use this template, we need to configure the `web.config` manually, so that the service uses `basicHttpBinding` as shown in the following code:

```
<system.serviceModel>
  <services>
    <service behaviorConfiguration="SilverlightEmployeeBrowser
      .Web.EmployeeServiceBehavior"
      name="SilverlightEmployeeBrowser.Web.EmployeeService">
      <endpoint address=""
        binding="basicHttpBinding"
```

```
contract="SilverlightEmployeeBrowser
    .Web.IEmployeeService">
<identity>
    <dns value="localhost" />
</identity>
</endpoint>
</service>
</services>
</system.serviceModel>
```

However, if we use the Silverlight-enabled WCF Service template, the service will get configured automatically, so that it'll work when accessed from a Silverlight application. This is shown in the following code snippet:

```
<system.serviceModel>
    <behaviors>
        ...
    </behaviors>
    <bindings>
        <customBinding>
            <binding name="customBinding0">
                <binaryMessageEncoding/>
                <httpTransport>
                    <extendedProtectionPolicy policyEnforcement="Never"/>
                </httpTransport>
            </binding>
        </customBinding>
    </bindings>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
    <services>
        <service behaviorConfiguration="SilverlightEmployeeBrowser
            .Web.EmployeeServiceBehavior"
            name="SilverlightEmployeeBrowser.Web.EmployeeService">
            <endpoint address=""
                binding="customBinding"
                bindingConfiguration="customBinding0"
                contract="SilverlightEmployeeBrowser.Web
                    .EmployeeService"/>
            <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange"/>
        </service>
    </services>
</system.serviceModel>
```

Note that the service is using binary message encoding by default, so that the data will get compressed when sent over the wire, resulting in quicker transfers.

When creating a WCF service, we need to specify what will be exposed over the service. In more detail, we need to specify the operations that can be invoked on the service through the use of a service contract and the types that will be sent over the wire for the client application to use. Each class that should be available on the client side needs to be marked with the `DataContract` attribute. In these classes, we can specify which fields should go over the wire. Only those fields marked with the `DataMemberAttribute` will be sent to the client. Thus, we have granular control over what will and what will not be sent to the client-side application.

Connecting to the service

Once the service is in place, we can connect to it from the Silverlight application. This can be done using the **Add Service Reference** dialog box. Visual Studio will create a proxy class when the connection is made. This class contains a client-side copy of the types exposed by the service (for example, the `Employee` and the `Manager` class) and a copy of the service methods among others. However, code is generated for each service method that makes it possible to call the service asynchronously. The actual service code is not copied to the proxy. To view this code, click on the **Show All Files** button located at the top of the **Solution Explorer**. Then, select the **+** sign in front of the service reference and expand it, so that the `Reference.cs` file is shown, which contains the code for the proxy class.

The requirement of calling the service asynchronously can also be seen in the code that's written in the Silverlight application itself. We start by instantiating the proxy class. On this instance, we define the callback method for the `xxx_Completed` event. Finally, we perform the service invocation by calling the `xxx_Async` method, which launches the request to the service. If the call is not carried out asynchronously, the application will cause the UI thread to hang until the service returns.

The callback method is invoked once the service returns. In this method, we have access to the result via the `e.Result` property. This property will be of the same type as returned by the service method (using the client-side copy of the data contracts).

See also

The communication with a WCF service is not very different from the general pattern used for communicating with a service in Silverlight. Take a look at the *Connecting and reading from a standardized service* recipe in the previous chapter, where we discussed all general aspects of this topic.

Invoking a service such as Bing.com

Applies to Silverlight 3, 4 and 5

Many (large) websites expose services that we can use in our applications. Some of these expose a WSDL file, and thus expose metadata that we can use to develop an application around them. One of these sites is **Bing** (www.bing.com), the search engine from Microsoft. The **Bing API** allows us to connect in several ways and interact with its services using many protocols. It also exposes a WSDL file to which we can connect from Visual Studio.

In this recipe, we'll build a Silverlight application that incorporates the services in Bing.

Getting ready

In this recipe, we're building the sample application from scratch. The finished solution for this recipe can be found in the Chapter08/SilverBing folder.

How to do it...

We can integrate search functionalities into our Silverlight applications using the Bing API. Bing exposes a WSDL file to which we can connect from Visual Studio. Visual Studio can create a proxy based on the metadata in this file. Thus, we get IntelliSense on the types and methods exposed by the service. The following are the steps we need to perform to get this working:

1. We need an API key to interact with the Bing API. This key is required, so that Bing can check that the request has arrived from a registered developer. A key can be obtained for free at <http://www.bing.com/developers/createapp.aspx>.
2. To build the actual application, we'll start from a clean slate. Create a new Silverlight application in Visual Studio and name it **SilverBing**.
3. We don't need to create a service ourselves; we just have to connect with the service from Bing. To do this, right-click on the Silverlight project node in the **Solution Explorer** and select **Add Service Reference...**. In the dialog box that appears, enter the URL `http://api.search.live.net/search.wsdl?AppID=XXX` for the WSDL file, where XXX needs to be replaced with the obtained API key. Enter **BingService** in the **Namespace:** field. Visual Studio will create the proxy after you click on the **OK** button.
4. The UI of the application consists of a search `TextBox`, a `Button`, and a templated `ListBox`. The complete XAML listing can be found in the code downloads.

5. When clicking on the previously mentioned button, in the event handler, we need to create a request to send to Bing. The request is encapsulated in a SearchRequest object, which is a type exposed by the service for us to use in our application. It contains the parameters we need to pass to the service (such as the API key) and the type of search we want to conduct. After constructing the SearchRequest instance, we define the callback method that should be executed when the service call returns. Finally, we perform the search asynchronously. The following code is used to illustrate this:

```
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    BingService.BingPortTypeClient soapClient =
        new SilverBing.BingService.BingPortTypeClient();
    SearchRequest request = new SearchRequest();
    request.AppId = "PERSONAL API KEY GOES HERE";
    request.Sources = new SourceType[] { SourceType.Web };
    if (SearchTextBox.Text != string.Empty)
    {
        request.Query = SearchTextBox.Text;
        soapClient.SearchCompleted +=
            new EventHandler<SearchCompletedEventArgs>
            (soapClient_SearchCompleted);
        soapClient.SearchAsync(request);
    }
}
```

6. We define a data-only class called BingSearchResult in the Silverlight project, in order to work with the results. This class is shown in the following code:

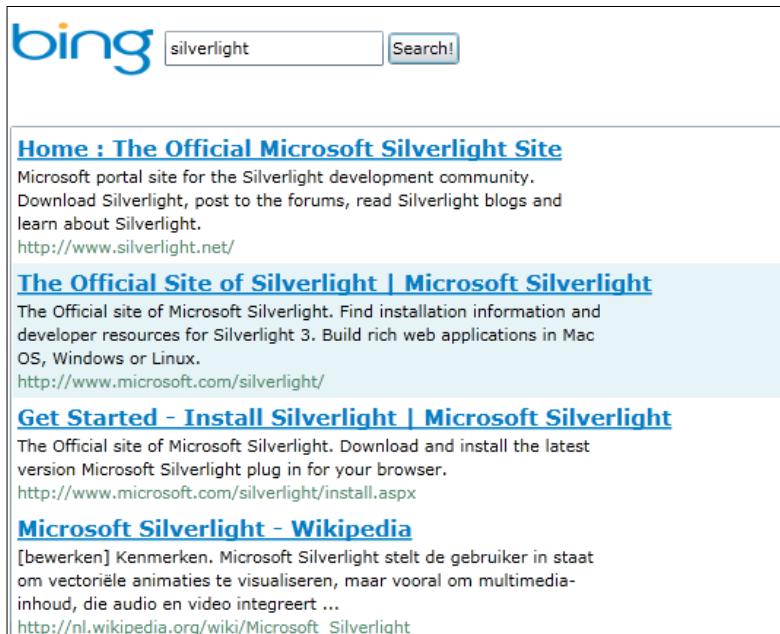
```
public class BingSearchResult
{
    public string Title { get; set; }
    public string Uri { get; set; }
    public string Description { get; set; }
}
```

7. When the service returns, the results are packaged in a SearchResponse, which is a type defined by Bing's service. We can loop through the results using a LINQ query and build BingSearchResult instances as shown in the following code:

```
void soapClient_SearchCompleted(object sender,
    SearchCompletedEventArgs e)
{
    SearchResponse response = e.Result;
    if (response.Web.Results.Count() > 0)
    {
        var results = from result in response.Web.Results
```

```
        select new BingSearchResult
    {
        Title = result.Title,
        Uri = result.Url,
        Description = result.Description
    };
    ResultListBox.ItemsSource = results.ToList();
}
}
```

The results are now shown in a `ListBox` with a `DataTemplate` applied to it. The following screenshot shows the results obtained on searching for Silverlight:



How it works...

Publicly accessible services, such as the services from Bing, allow us to integrate functionality into our Silverlight applications. Some services expose a REST API (refer to *Chapter 10, Talking to REST and WCF Data Services*, for more information about REST access), while some expose SOAP services. Bing exposes both, and we used the latter option in this recipe. If a service exposes both, using SOAP is recommended. One advantage is full IntelliSense, because Visual Studio can generate a proxy in the project based on the WSDL file. This proxy also contains all types and enumerations exposed by the service. Another advantage is that we get deserializing of the SOAP response done automatically, and it does not require us to do manual processing of XML.

The entire API for Bing can be found at <http://www.bing.com/developers/s/APIBasics.html>. Working with the API requires an API key, which Bing can use to tell if the request is authorized. We can find the address of the WSDL file in the documentation as well.

To perform a search, we need to instantiate an object of the `SearchRequest` class. This class is exposed by the service and is available on the client side because of the proxy generation. When Visual Studio connects with the service, it reads the exposed metadata (in the form of a WSDL file) and generates the client-side proxy. The `SearchRequest` class is used to pass parameters to the service, such as the API key and the search term. Invoking the service is done asynchronously, as with all other services.

The callback method is invoked once the service request is complete and the result is available. The result is of the `SearchResponse` type. Looping through the `Web.Results` property of the `SearchResponse` allows us to retrieve the results as returned by Bing.

There's more...

Bing exposes a `SourceType` enumeration that can be used to specify if we want to perform a regular web search, an image search, a news search, and so on. Bing exposes a specific type for some of the options of the enumerations. For example, it exposes the `ImageRequest` class for `SourceType.Image`. This class exposes more specific properties related to image searching. This `ImageRequest` can then be linked with the original request using its `Image` property.

See also

Communicating with Bing's service is similar to what we did in the previous recipe of this chapter.

Optimizing performance using binary XML

Applies to Silverlight 3, 4 and 5

When working with data in Silverlight, you'll typically get your data through some kind of service call. This means that this data has to be sent over the wire. In some cases, the data that is to be sent back and forth is relatively small, but in other cases, the amount of transported data can become quite large, which means more bandwidth consumption and a longer wait for the user.

When using WCF services with Silverlight, it's possible to compress this data, thus ensuring lower bandwidth consumption. In this recipe, you'll learn how to achieve this.

Getting ready

If you want to follow along with this recipe, you can use the provided starter solution located in the `Chapter08/UsingBinaryXML_Starter` folder, in the downloads for the book. The completed solution can be found in the `Chapter08/UsingBinaryXML_Completed` folder.

How to do it...

To look at how binary XML works, we're starting from a simple Silverlight solution that includes a Silverlight project that calls a WCF service method. This method returns a `Person` object and shows it on the screen. This starter solution uses `basicHttpBinding`. We're going to encode our data to binary, so the bandwidth consumption is smaller. To achieve this, complete the following steps:

1. Start by opening the solution as outlined in the *Getting ready* section of this recipe.
2. Open the `web.config` file and locate the endpoint exposed by `MyWCFService`. At the moment, this endpoint uses `basicHttpBinding`. Change this endpoint to use a custom binding to encode the data to binary as shown in the following code:

```
<bindings>
  <customBinding>
    <binding name="UsingBinaryXML.Web.MyWCFService
      .customBinding0">
      <binaryMessageEncoding />
      <httpTransport />
    </binding>
  </customBinding>
</bindings>
<serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
<services>
  <service name="UsingBinaryXML.Web.MyWCFService">
    <endpoint address=""
      binding="customBinding"
      bindingConfiguration="UsingBinaryXML.Web
        .MyWCFService.customBinding0"
      contract="UsingBinaryXML.Web.MyWCFService" />
    <endpoint address="mex"
      binding="mexHttpBinding"
      contract="IMetadataExchange" />
  </service>
</services>
```

3. Open the Silverlight application and update the Service Reference by right-clicking on **MyWCFServiceReference** and selecting **Update Service Reference**.
4. You can now build and run the solution. The data will be sent over the wire, which will be binary-encoded.

How it works...

When using a WCF service, changing the way data is encoded is a matter of defining the correct binding in the corresponding config files. WCF takes care of the rest.

In the second step of this recipe, we've added a `customBinding` that defines the `binaryMessageEncoding` element. This binding is then used by our endpoint (instead of `basicHttpBinding`) to make sure that binary encoding will be used. In the third step, we make sure that the correct corresponding client-side endpoint configuration is generated in `ServiceReferences.ClientConfig` by updating the Service Reference.

The code doesn't have to be changed to achieve binary encoding.

The default configuration is different depending on the version of Silverlight you're using

In Silverlight 3 and higher, a custom binding that provides binary encoding is automatically generated. So, you don't need to execute the previous steps to get the advantages of binary encoding.

Using binary encoding is not a security measure

Although the data sent over the wire isn't as easily readable as plain text, it's not that hard to write a decoder. Thus, binary encoding should be used as a means of lowering your bandwidth consumption, but never as a security measure!

See also

As mentioned before, using binary encoding is not a security measure. To learn how to secure your communication, have a look at the *Ensuring data is encrypted* and *Securing service communication using message-based security* recipes.

Debugging a service in Silverlight

Applies to Silverlight 3, 4 and 5

One of the things that were missing in Silverlight 2 when it came to working with services was the ability to debug them. In the event of an error occurring in the service code, it was not possible to get information about this error in the Silverlight client application. The only information we got when something went wrong was **The remote server returned an error: NotFound**, and that was it. The user wasn't provided with any inner exception with a clue about the real error on the service, any messages, and so on. While this was a problem during development, it was a worse a problem that we could not give any indication to the end user on what he/she should do to solve the issue. This was because we could not distinguish let's say a divide-by-zero error from an error caused by the database being down inside the Silverlight application.

If we are developing both the service and the Silverlight application, we can still debug the service code and find out the eventual errors. However, it's a real show-stopper when working with external, perhaps third-party services to which we don't have access as we can't see what's happening on the service side.

Luckily, Silverlight has evolved quite a lot since version 2, as it now includes the option to debug the service by displaying the returned error from the service side. In this recipe, we'll look at how to set this up.

Getting ready

To try out this sample, a starter solution is provided in the `Chapter08/SilverlightServiceDebugging_Starter` folder in the downloads available on the Packt website. The finished solution can be found in the `Chapter08/SilverlightServiceDebugging_Completed` folder.

How to do it...

To explain the way Silverlight makes it possible to debug a service, we'll start from an application that could have been easily created in Silverlight 2, and we'll follow the changes we need to make to both the service and the Silverlight application. We do this so that we can debug the service and get error information in the Silverlight client.

The scenario is very simple. The application allows us to search for an employee or a list of employees in a directory based on username. If a result is found, a `DataGridView` is shown containing the results. Let's take a look at the steps we need to perform to make it possible to debug this application:

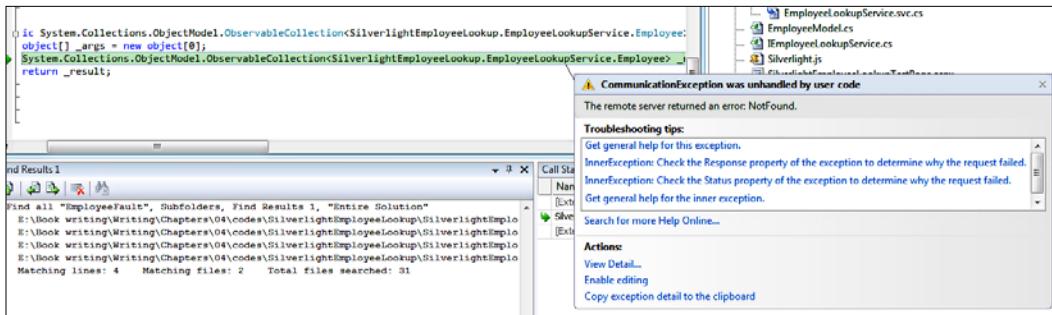


Note that this recipe requires either **Fiddler** or **Web Development Helper** to view the traffic from and to the service. See the Appendix for more info on these. We'll use Fiddler in this recipe.

1. Open the solution located in the Chapter08/SilverlightServiceDebugging_Starter folder in the code downloads and take a look at the SilverlightEmployeeLookup.Web/EmployeeLookupService.svc.cs file. The RetrieveEmployeesByUserName(string userName) method has a bug residing in this class. If no Employee items are found, we still try to retrieve the first item. Of course, this will generate an error. The following code snippet shows the faulty code:

```
if (results.Count<Employee>() == 0)
{
    Employee employee = results.First();
}
```

2. Let's try if we can see in the Silverlight application what went wrong. Run the application and search for George. Visual Studio will show the sequence contains no element's error inside the service code. However, on the client side, the error information is not available anymore, and the dreaded `NotFound` exception is thrown, as shown in the following screenshot:



3. At this point, even Fiddler is of no help. Let's run the application with Fiddler open again. When the error occurs, a message with status 500 is shown, as well as the following result is shown, containing no information about the error:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <s:Fault>
      <faultcode xmlns:a="http://schemas.microsoft.com/
        net/2005/12/windowscommunicationfoundation/dispatcher">
        a:InternalServiceFault
      </faultcode>
```

```
<faultstring xml:lang="nl-BE">
The server was unable to process the request due to an
internal error. For more information about the error,
either turn on IncludeExceptionDetailInFaults (either from
ServiceBehaviorAttribute or from the <serviceDebug>;
configuration behavior) on the server in order to send the
exception information back to the client, or turn on tracing as
per the Microsoft .NET Framework 3.0 SDK documentation and inspect
the server trace logs.
</faultstring>
</s:Fault>
</s:Body>
</s:Envelope>
```

The previous message says that we need to turn on `IncludeExceptionDetailInFaults` to receive information about the error. We'll come to this later.

4. The reason that Silverlight can't read the error is that the message has a status 500. Due to the browser networking stack that Silverlight uses under the covers, plugins such as Silverlight don't get access to this message. Hence, we should try to change the status to 200, so that Silverlight can access the error. This change can be brought about by creating a WCF endpoint behavior for Silverlight faults. This will do the conversion from HTTP status 500 to HTTP status 200.
5. We'll create this behavior in a class in a separate project. To do this, create a new class library (not a Silverlight class library, as this library will be used by the web project, not the Silverlight project). Name the new project **FaultBehavior**.
6. In this new project, we create a class containing the new behavior. The complete code for this behavior can be found at [http://msdn.microsoft.com/en-us/library/dd470096\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/dd470096(VS.95).aspx) as well as in downloads of the book. The most relevant part, which does the status conversion, is shown in the following code:

```
public void BeforeSendReply(ref Message reply,
    object correlationState)
{
    if (reply.IsFault)
    {
        HttpResponseMessageProperty property =
            new HttpResponseMessageProperty();
        // Here the response code is changed to 200.
        property.StatusCode = System.Net.HttpStatusCode.OK;
        reply.Properties[HttpResponseMessageProperty.Name] =
            property;
    }
}
```

7. Build the class library and reference the project from the **SilverlightEmployeeLookup.Web** web project.
8. We need to make some configuration changes in the `web.config` file, so that our service can use the newly created behavior. Change the code in the `web.config` file as shown in the following code (note that `includeExceptionDetailInFaults` is set to `true` in this code):

```
<system.serviceModel>
  <extensions>
    <behaviorExtensions>
      <add name="silverlightFaults"
           type="FaultBehavior.SilverlightFaultBehavior,
                  FaultBehavior, Version=1.0.0.0, Culture=neutral,
                  PublicKeyToken=null" />
    </behaviorExtensions>
  </extensions>
  <behaviors>
    <endpointBehaviors>
      <behavior name="SilverlightFaultBehavior">
        <silverlightFaults />
      </behavior>
    </endpointBehaviors>
    <serviceBehaviors>
      <behavior name="SilverlightEmployeeLookup.Web
                  .EmployeeLookupServiceBehavior">
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service behaviorConfiguration="SilverlightEmployeeLookup.Web
                  .EmployeeLookupServiceBehavior"
           name="SilverlightEmployeeLookup.Web
                  .EmployeeLookupService">
      <endpoint address=""
                binding="basicHttpBinding"
                behaviorConfiguration="SilverlightFaultBehavior"
                contract="SilverlightEmployeeLookup.Web
                          .IEmployeeLookupService">
        <identity>
          <dns value="localhost"/>
        </identity>
      </endpoint>
```

```
<endpoint address="mex"
           binding="mexHttpBinding"
           contract="IMetadataExchange" />
</service>
</services>
</system.serviceModel>
```

9. Now that all the plumbing is done, we can focus on the service again. Let's start by adding a new class called `EmployeeFault`. This class will contain information on the error that occurred. The following code specifies our own fault type called `EmployeeFault`. It contains two properties that will be filled with data when an error occurs:

```
public class EmployeeFault
{
    public string Message { get; set; }
    public string AdditionalInfo { get; set; }
}
```

10. In the contract of the `IEmployeeLookupService`, we need to add the `FaultContract` attribute. This is shown in the following code:

```
[ServiceContract]
public interface IEmployeeLookupService
{
    [OperationContract]
    [FaultContract(typeof(EmployeeFault))]
    List<Employee> RetrieveEmployeesByUserName(string userName);
}
```

11. In the service implementation, we can now create an instance of the `EmployeeFault` when an exception occurs. In the following code, we check if there are no results and if so, we return an instance of the `EmployeeFault` that contains information regarding the results:

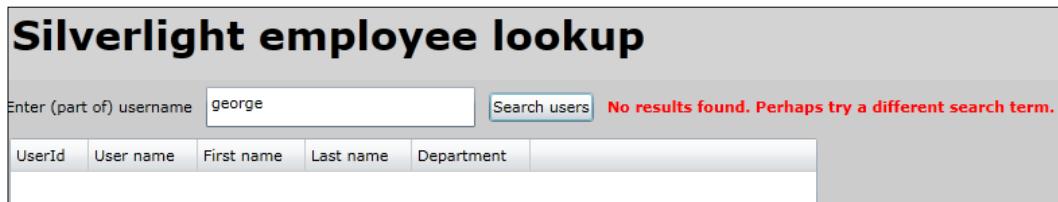
```
if (results.Count<Employee>() == 0)
{
    EmployeeFault fault = new EmployeeFault();
    fault.Message = "No results found.";
    fault.AdditionalInfo = "Perhaps try a different search term.";
    throw new FaultException<EmployeeFault>(fault,
        "Bad search term");
}
```

12. Build the solution again. We also need to update the service reference, so that the client-side proxy code is recreated. To do this, right-click on the service reference and select **Update Service Reference**. The generated code in the proxy class will now also include the `EmployeeFault`.

13. In the callback method of the Silverlight application, we can now check if the result contains an error, and if it does, we can check the type of the returned fault. This fault also includes relevant information that we can use to display to the user:

```
if (e.Error == null)
{
    EmployeesDataGrid.ItemsSource = e.Result;
}
else if (e.Error is FaultException<EmployeeFault>)
{
    FaultException<EmployeeFault> serviceFault =
        e.Error as FaultException<EmployeeFault>;
    ErrorTextBlock.Text = serviceFault.Detail.Message + " " +
        serviceFault.Detail.AdditionalInfo;
}
```

If we run the application now and search for a non-existing employee, we see the information contained in the fault returned by the service, as shown in the following screenshot:



How it works...

When a service encounters an error, it returns an HTTP status code 500. Silverlight can't receive these responses, because of the browser networking stack. Due to this, when working with services, we often encounter the **The remote server returned an error: NotFound error** error message. This isn't very helpful, because no information on what really went wrong is included.

To make it possible for Silverlight to read service faults, we need to change the status code from 500 to 200. This can be achieved by adding a behavior that will do this for our WCF service. When using this behavior on a service, we also need to make configuration changes in the `web.config` file of the service project.

After this change, service faults are accessible in Silverlight. The fault—a SOAP fault—needs to be translated into managed code. Silverlight 2 could not do this. However, this changed with the introduction of Silverlight 3, and remained the same for Silverlight 4 and 5 as well. This way, error information sent by the service can be translated into a managed exception that can be inspected and used for error-handling purposes.

Types of faults

There are two types of faults that can be sent by a service—a **declared fault** and an **undeclared fault**.

Declared faults

Declared faults are the type of faults we should use in a production environment. These define a custom type that will be sent back to the client-side application, containing information on the error that occurred. Declared faults occur in operations annotated with the `FaultContractAttribute`. By annotating an operation with this attribute, we specify that this operation may throw this type of exception, and if it does, the resulting fault instance should be sent to the client.

When building the proxy, the custom fault type will be generated on the client side as well. This way, we can build code that will check if the returned error is an instance of the custom fault type. If it is, we can get the error information from the instance.

Undeclared faults

During development, we often want all the information about an error that occurs. Remember the error message we got earlier in this recipe using Fiddler (in step 3)? Hidden in the response of the service, it was mentioned that we could get more information on the error that occurred by setting `IncludeExceptionDetailInFaults` to `true` in the service configuration. This is shown in the following line of code:

```
<serviceDebug includeExceptionDetailInFaults="true"/>
```

Doing this (and thus not using the declared faults), results in the original error being sent over the wire. Silverlight will also be able to pick up this type of fault (assuming we still have the behavior in place that converts status 500 to status 200). The following code can be used to capture undeclared faults:

```
void proxy_RetrieveEmployeesByUserNameCompleted(object sender,
    SilverlightEmployeeLookup.EmployeeLookupService
    .RetrieveEmployeesByUserNameCompletedEventArgs e)
{
    if (e.Error == null)
    {
        EmployeesDataGrid.ItemsSource = e.Result;
    }
    else if (e.Error is FaultException<ExceptionDetail>)
    {
        FaultException<ExceptionDetail> serviceFault =
            e.Error as FaultException<ExceptionDetail>;
        ErrorTextBlock.Text = serviceFault.Detail.Message;
    }
}
```

If we run this code, the Silverlight application will show the error message, as it was sent when the error occurred on the service.

The main reason to favor declared faults for production code is that we don't want to show real service errors to the end users. These might even include sensitive information, such as database passwords.

Using ASP.NET Authentication in Silverlight

Applies to Silverlight 3, 4 and 5

When creating ASP.NET web applications, we often have to authenticate visitors. Not every user has the permission to see just any information contained in the website. For example, only paying subscribers have access to read all articles or perform searches on the site. Also, an administrator should be able to log in to the application and make changes, such as changing the content or adding an article.

This typical behavior is supported by some authentication mechanism. ASP.NET 2.0 introduced the **ASP.NET Membership API**, a built-in authentication system in ASP.NET that supports user authentication, role management, profile management, and so on. Through **ASP.NET Application Services**, added with ASP.NET 3.5, this system is exposed for client-side use. Technologies, such as ASP.NET, AJAX, and Silverlight can use these services to authenticate a user from a client-side application on the server side.

In this sample, we'll look at how Silverlight applications can integrate with an (existing) authentication system built with ASP.NET Membership.

Getting ready

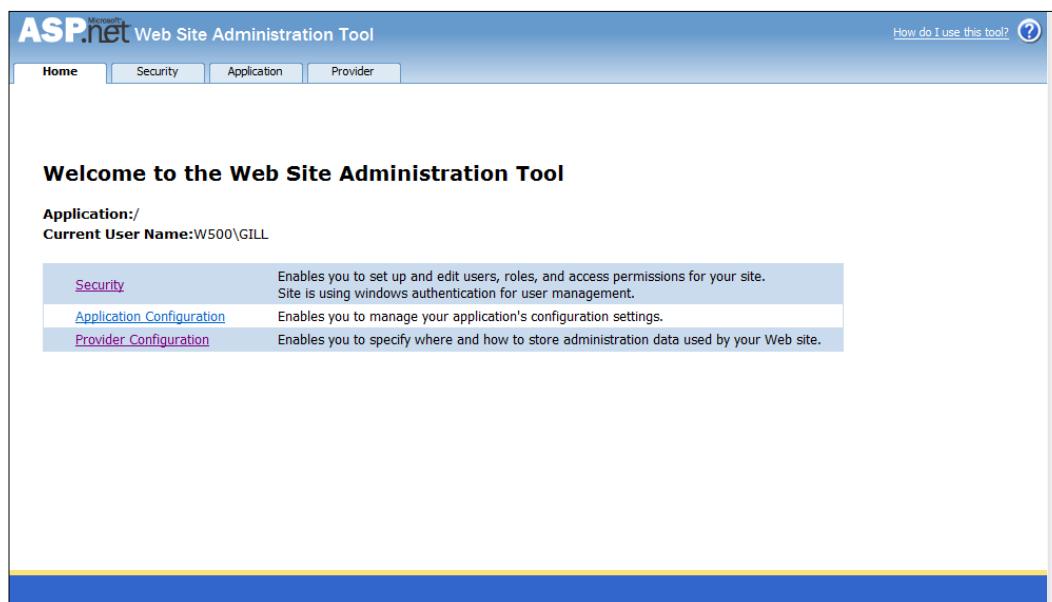
This recipe requires that SQL Server or SQL Server Express is installed on your machine, since the code we are creating is talking to a database. The SQL Server service should be started as well. You can check if the service is active via **Control Panel | Administrative Tools | View Local Services**.

To follow along with this recipe, you can use a starter solution located in the `Chapter08/UserAuthentication_Starter` folder in the code bundle available on the Packt website. The completed code for this recipe can be found in the `Chapter08/UserAuthentication_Completed` folder.

How to do it...

For this recipe, we'll build an application where a user can log in. If the authentication succeeds, he or she can view the employee information. Should he or she fail to provide correct authentication information, the employee data will not be accessible for that user. All the authentication data (usernames and passwords) is contained in a database and is managed by the ASP.NET Membership API, and thus lives at the server side. However, we will allow the user to log in from the client-side Silverlight application. Let's take a look at the steps we need to follow to support ASP.NET Authentication from Silverlight:

1. Open the starter solution as outlined in the *Getting Ready* section.
2. To kick things off, we should first do some work on that website. We'll start by adding a database that contains the user information. For this sample, we'll use the default ASPNETDB .MDF database, but all that is explained here can be added to any existing database. To have Visual Studio create this default database for us, select any file in the website in the **Solution Explorer** and then go to **Project | ASP.NET Configuration**. An administration website called **ASP.NET Web Site Administration Tool** will appear, as shown in the following screenshot. Make sure that the SQL server service is running for the next steps to succeed:



3. In this tool, head over to the **Security** tab. On this page, we must start by changing the authentication type, which is by default set to **Windows Authentication**. Click on the **Select Authentication Type** link, and in the page that appears next, select **From the internet**. Confirm by clicking on **Done**.

4. We are now sent back to the **Security** page. We can now add some users using this tool. Click on the **Create user** link, and add the details for a user. For the sample, we use the username **normaljoe** and the password **normal?**.
5. At this point, we are ready to start exposing the ASP.NET Membership information. We can do so by creating a service. We'll want this service to expose the possibility to authenticate a user from the client side by verifying his or her username and password combination. However, this functionality is already available in ASP.NET in the form of the ASP.NET Application Services. Thus, we only need to create a service endpoint and enable the service (it is disabled by default). Let's start with the service endpoint. Since we don't need to write any functionality for this service, we can add an empty text file and rename it to have the .svc extension. Name the newly added file as AuthService.svc.

6. In this file, we should now only add a line of code that will link the code already available in ASP.NET with our service endpoint. Add the following line of code:

```
<%@ ServiceHost Language="C#"
    Service="System.Web.ApplicationServices.AuthenticationService" %>
```

7. Next, we need to configure this service as if it were a regular WCF service. As we did not add the service as a service, but as a text file, Visual Studio did not add any configuration code to the web.config file as it usually does. The following code needs to be added between the <configuration></configuration> tags in the web.config file.

Note that the basicHttpBinding is used as the binding type for the communication between Silverlight and the ASP.NET Application service.

```
<system.serviceModel>
    <services>
        <service
            name="System.Web.ApplicationServices.AuthenticationService"
            behaviorConfiguration="AuthenticationServiceBehaviors">
            <endpoint
                contract="System.Web.ApplicationServices.AuthenticationService"
                binding="basicHttpBinding" />
        </service>
    </services>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
    <behaviors>
        <serviceBehaviors>
            <behavior name="AuthenticationServiceBehaviors">
                <serviceMetadata httpGetEnabled="true"/>
            </behavior>
        </serviceBehaviors>
    </behaviors>
</system.serviceModel>
```

```
</serviceBehaviors>
</behaviors>
</system.serviceModel>
```

8. As already mentioned, the application services' functionality are not enabled by default; we need to enable it using configuration code. Again in the web.config file, add the following code between the <configuration></configuration> tags:

```
<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true"
        requireSSL="false"/>
    </webServices>
  </scripting>
</system.web.extensions>
```

9. Finally, let's check that the authentication mode is set to **Forms** (it should be fine already, since we set this using the **ASP.NET Web Site Administration Tool**).

```
<authentication mode="Forms" />
```

10. We're done with what concerns the server side. Let's turn our attention to the client side. We'll start by adding a service reference. Adding this reference is nothing different from any other service reference we have already added. Right-click on the Silverlight project and select the **Add service reference** node. In the dialog box, click on the **Discover** button, and the service should be located. Set the service namespace to AuthenticationService.

11. The Silverlight application contains two screens. The first screen, the main window, contains some Button instances and a DataGrid, which shows employee data. No data is shown by default; the user has to click on the **View Users** button. However, this button is only enabled after the user has authenticated. Authentication can be done using the second screen, a ChildWindow called LoginWindow that contains fields to enter username and password. Both screens can be seen at the end of this recipe, and the XAML code can be found in the downloads of the book.

12. The application checks at startup if the user is already authenticated. It will do so using the operations exposed by the authentication service, namely the IsLoggedIn method. We create an instance of the AuthenticationServiceClient (the proxy class), and call the IsLoggedInAsync method on it. An event handler is attached on the IsLoggedInCompleted event as follows:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    AuthenticationService.AuthenticationServiceClient proxy = new
        UserAuthentication.AuthenticationService.
    AuthenticationServiceClient();
```

```
proxy.IsLoggedInCompleted += new EventHandler<UserAuthentication
    .AuthenticationService.IsLoggedInCompletedEventArgs>
    (proxy_IsLoggedInCompleted);
proxy.IsLoggedInAsync();
}
```

13. In the callback method, we can check the result of the service call. If the result is `false`, the user is asked to enter his or her credentials in the `LoginWindow`, an instance of a Silverlight `ChildWindow`. If it is `true`, we enable the `ViewUsersButton` as well as the `LogoutButton`. The following code performs the check of the result:

```
void proxy_IsLoggedInCompleted(object sender, UserAuthentication
    .AuthenticationService.IsLoggedInCompletedEventArgs e)
{
    if (e.Result)
    {
        ViewUsersButton.IsEnabled = true;
        LogoutButton.IsEnabled = true;
    }
    else
    {
        var loginWindow = new LoginWindow();
        loginWindow.Closed += new EventHandler(loginWindow_Closed);
        loginWindow.Show();
    }
}
```

14. To actually log in the user in the `LoginWindow`, we can use the `Login` method. This method is called when the user clicks on the `OKButton`. When the call is completed, in the callback method, the `DialogResult` property of the `ChildWindow` class is evaluated and set to `true` if the login attempt was successful. This can be seen in the following lines of code:

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    AuthenticationService.AuthenticationServiceClient proxy = new
        UserAuthentication.AuthenticationService.
    AuthenticationServiceClient();
    proxy.LoginCompleted += new EventHandler<UserAuthentication
        .AuthenticationService.LoginCompletedEventArgs>
        (proxy_LoginCompleted);
    proxy.LoginAsync(UserNameTextBox.Text, UserPasswordBox.Password,
        "", true);
}
void proxy_LoginCompleted(object sender,
```

```
UserAuthentication.AuthenticationService.LoginCompletedEventArgs  
e)  
{  
    if (e.Error != null)  
        StatusTextBlock.Text = e.Error.ToString();  
    else  
    {  
        this.DialogResult = true;  
    }  
}
```

15. The MainPage can now react according to the result of the login attempt in the child window. In step 13, we added an event handler to the Closed event of the LoginWindow instance. In this method, we can check what the DialogResult property contains. If it is true, we enable the ViewUsersButton. This is shown in the following code snippet:

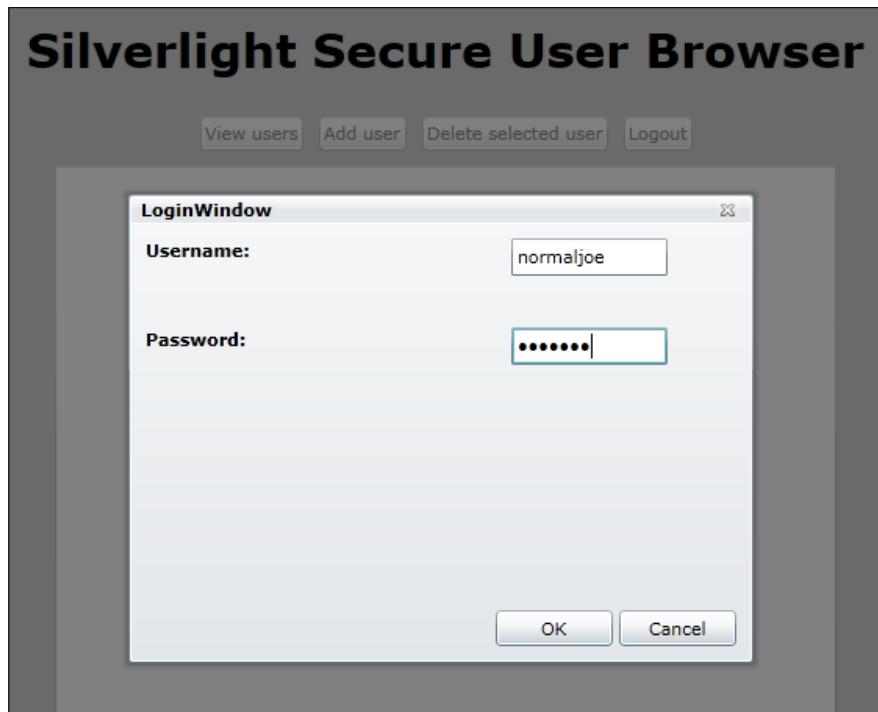
```
void loginWindow_Closed(object sender, EventArgs e)  
{  
    LoginWindow loginWindow = (LoginWindow)sender;  
    bool? result = loginWindow.DialogResult;  
    if (result.HasValue && result.Value)  
    {  
        ViewUsersButton.IsEnabled = true;  
    }  
}
```

16. When the user wants to leave the application, he or she can use the LogoutButton. The following code logs the user out of the application:

```
private void LogoutButton_Click(object sender, RoutedEventArgs e)  
{  
    AuthenticationService.AuthenticationServiceClient proxy = new  
        UserAuthentication.AuthenticationService.  
    AuthenticationServiceClient();  
    proxy.LogoutCompleted += new EventHandler  
        <System.ComponentModel.AsyncCompletedEventArgs>  
        (proxy_LogoutCompleted);  
    proxy.LogoutAsync();  
}  
void proxy_LogoutCompleted(object sender,  
    System.ComponentModel.AsyncCompletedEventArgs e)  
{  
    if (e.Error == null)  
    {  
        LogoutButton.IsEnabled = false;  
    }  
}
```

```
        ViewUsersButton.IsEnabled = false;
    }
}
```

We have now created a Silverlight application that uses the ASP.NET Membership API, more specifically ASP.NET authentication, to authenticate users. The following screenshot shows the `LoginWindow` being displayed, because the user is not logged in:



How it works...

When we want to add a Silverlight application to an already existing ASP.NET application that uses ASP.NET Membership, we can easily reuse the authentication system. Since version 3.5, ASP.NET has contained the so-called Application Services, which expose some functionality of the server-side API to use by client-side platforms, such as Silverlight.

ASP.NET Membership uses cookies to validate a user request. When we use the API from Silverlight, it creates the same cookies on the client side. When we are authenticated in the Silverlight application, we are also authenticated for the ASP.NET application; the cookies are identical.

The functionality for the Application Services is already contained in the ASP.NET platform. Therefore, we don't have to write any code: we only need to create an endpoint (*.svc file), which links the endpoint with the code of ASP.NET. We do need to configure the service; we need to add configuration code similar to a normal WCF service. Also, we need to enable the Application Services using configuration code.

A Silverlight application works with this service just in the same way as it would with any other service: Visual Studio creates a proxy when we add a service reference. This proxy class can be instantiated to use the exposed methods. The service contains methods to check if the user is already logged in and log the user in and out.

There's more

In this recipe, we've connected with the AuthenticationService. The ASP.NET Membership API also has functionality to work with roles and profiles. Both are also exposed through an application service, the RoleService and the ProfileService respectively. Silverlight can connect to both these services in the same way as we did with the AuthenticationService.

The RoleService gives us the ability in our client-side Silverlight application to verify if a user is part of a certain role. Based on this information, we can enable or disable a particular functionality in the application, or show or hide a screen from the user.

The ProfileService allows us to store typical user setting information. For example, we can expose the functionality to let the user change the background color of the application. If we want to persist this information between sessions, we can store it in the profile of the user.

Uploading files to a WCF service

Applies to Silverlight 3, 4 and 5

While most of the time we'll want to download files, such as images from a server, sometimes a scenario may arise where we might need to send files to a server. For example, imagine we are building a Silverlight image editor, where the user is editing an image locally. After completion, the user may want to save this image on the server, perhaps in a folder or even in a SharePoint library. In this case, the locally created file (in this case an image) somehow needs to be transmitted to the server.

WCF services can help in implementing this scenario. At its heart, a file is nothing more than binary data that can be sent to a service endpoint. In this recipe, we'll build such a service and a client application that uses this service.

Getting ready

We'll build the application in this recipe from scratch. However, the finished solution can be found in the Chapter08/SilverlightImageUpload folder in the downloads available on the Packt website.

How to do it...

To start sending files to the server, we need to create a service that is capable of receiving binary data. The service can take care of converting the binary data into a physical file that it can store on the local file system. The client-side Silverlight application can then send data to the service. The following are the steps we need to perform to complete this task:

1. We'll start from an empty Silverlight application in this recipe. Create a new Silverlight solution in Visual Studio and name it as `SilverlightImageUpload`.
2. Let's first take a look at how to create a service. In the **SilverlightImageUpload.Web** web project, add a new WCF service by right-clicking on the project and selecting the **Add | New Item...** option. In the dialog box that appears, select the regular WCF service template and name the service `ImageUploadService.svc`. Visual Studio creates the `ImageUploadService.svc`, the `ImageUploadService.svc.cs`, and the `IImageUploadService.cs` files. The latter contains the service contract.
3. The service contract in the `IImageUploadService.cs` file is quite simple, containing only one operation. This is shown in the following code. The operation accepts one parameter, namely an instance of the `ImageUpload` class that we'll create in the next step:

```
[ServiceContract]
public interface IImageUploadService
{
    [OperationContract]
    bool Upload(ImageUpload imageUpload);
}
```

4. The `ImageUpload` class that is attributed with the `DataContract` attribute contains a byte array as one of its members. This is shown in the following code:

```
[DataContract]
public class ImageUpload
{
    [DataMember]
    public string ImageName { get; set; }
    [DataMember]
    public byte[] Image { get; set; }
}
```

5. Of course, we need to implement the service as well. The following code will convert the byte array sent by the client application into a file. The code relies on classes in the `System.IO` namespace, such as `FileStream` and `BinaryWriter`. Note that we also add the `AspNetCompatibilityRequirementsAttribute` to this service. Setting this attribute results in the service running in an ASP.NET compatibility mode. This in turn results in the WCF service behaving like an ASMX service for some ASP.NET features:

```
[AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
public class ImageUploadService : IImageUploadService
{
    public bool Upload(ImageUpload imageUpload)
    {
        FileStream fileStream = null;
        BinaryWriter writer = null;
        string imagePath;
        try
        {
            imagePath = HttpContext.Current.Server.MapPath(".") +
                ConfigurationManager.AppSettings["ImageUploadDirectory"] +
                imageUpload.ImageName;
            if (imageUpload.ImageName != string.Empty)
            {
                fileStream = File.Open(imagePath, FileMode.Create);
                writer = new BinaryWriter(fileStream);
                writer.Write(imageUpload.Image);
            }
            return true;
        }
        catch (Exception)
        {
            return false;
        }
        finally
        {
            if (fileStream != null)
                fileStream.Close();
            if (writer != null)
                writer.Close();
        }
    }
}
```

6. In the previous step, we used `ConfigurationManager.AppSettings["ImageUploadDirectory"]`. This line of code refers to an entry in the `appSettings` collection in the `web.config` file, where we store settings that are specific to the application. Add the following code in the `web.config` file (note that it is to be inserted between the opening and closing `appSettings` tags):

```
<appSettings>
  <add key="ImageUploadDirectory"
       value="/Images/" />
</appSettings>
```

7. Make sure the folder you want to store the images in exists before using it!
8. Finally, we need to tweak the configuration of the service to make it possible to receive larger than normal messages. In the `web.config` file, change the `system.ServiceModel` node so that it reflects the following code:

```
<system.serviceModel>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
  <bindings>
    <basicHttpBinding>
      <binding name="ImageUploadBinding"
               maxReceivedMessageSize="2000000"
               maxBufferSize="2000000">
        <readerQuotas maxArrayLength="2000000"
                      maxStringContentLength="2000000"/>
      </binding>
    </basicHttpBinding>
  </bindings>
  <behaviors>
    ...
  </behaviors>
  <services>
    <service behaviorConfiguration="SilverlightImageUpload.Web
              .ImageUploadServiceBehavior"
              name="SilverlightImageUpload.Web.ImageUploadService">
      <endpoint address=""
                binding="basicHttpBinding"
                bindingConfiguration="ImageUploadBinding"
                contract="SilverlightImageUpload.Web
                          .IImageUploadService">
        ...
      </endpoint>
    </service>
  </services>
</system.serviceModel>
```



Note that the binding of the service is also set to basicHttpBinding.

9. Now that the service is ready, build it.
10. Add a service reference to the Silverlight application by right-clicking on the project node and selecting the **Add Service Reference...** option. In the dialog box that appears, click on the **Discover** button. Visual Studio should find the service. Set the namespace to `IImageUploadService`.
11. The UI of the application is very simple. The most important element is a Button. The complete code can be found in the code bundle.
12. In the event handler of the `Click` event of the Button, an `OpenFileDialog` is presented to the user, in which he or she can select an image file (only *.jpg files are allowed in our scenario). The image is then converted into a byte array. This array along with a filename is used for the creation of an `IImageUpload` instance. Finally, an instance of the proxy class is created and a service call is made asynchronously, passing in the `IImageUpload` instance. This is shown in the following code:

```
private void UploadButton_Click(object sender, RoutedEventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "JPEG files|*.jpg";
    if (openFileDialog.ShowDialog() == true)
    {
        Stream stream = (Stream)openFileDialog.File.OpenRead();
        byte[] bytes = new byte[stream.Length];
        stream.Read(bytes, 0, (int)stream.Length);
        string fileName = openFileDialog.File.Name;
        ImageUploadService.ImageUpload imageUpload = new
            SilverlightImageUpload.ImageUploadService.ImageUpload();
        imageUpload.ImageName = fileName;
        imageUpload.Image = bytes;
        ImageUploadService.ImageUploadServiceClient proxy =
            new SilverlightImageUpload.ImageUploadService
                .ImageUploadServiceClient();
        proxy.UploadCompleted += new EventHandler
            <SilverlightImageUpload.ImageUploadService
                .UploadCompletedEventArgs>(proxy_UncleUploadCompleted);
        proxy.UploadAsync(imageUpload);
    }
}
```

13. In the callback method, we have access to the result. We use this result to give feedback to the user as shown in the following code:

```
void proxy_UploadCompleted(object sender, SilverlightImageUpload
    .ImageUploadService.UploadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        if (e.Result)
        {
            ResultTextBlock.Text = "Image uploaded successfully!";
        }
        else
        {
            ResultTextBlock.Text = "An error occurred while uploading
                the image";
        }
    }
}
```

14. We have now successfully uploaded the image to the server using the WCF service. The following screenshot shows the interface with the `OpenFileDialog` open, thus allowing the user to select an image. The selected image will then be uploaded to the service:



How it works...

To upload files to the server, WCF services can be the solution. The process is actually quite simple. We need to create a service that accepts a byte array. In the service code, this byte array can then be converted into a file. In this recipe, we have used images, but the process explained works for all types of files (a Word document, an Excel file, and so on).

By default, a WCF service does not accept incoming messages larger than 65536 bytes. This is an issue when sending files, because most files will be larger than this limit. Due to this, we need to configure the binding in the `web.config` file of the service, so that larger messages aren't blocked. In our recipe, we specified the maximum file size to be 2 megabytes.

The client application can open a file on the user's hard drive by using `OpenFileDialog`. We get a read-only stream to the file. This resulting stream can then be used to create a byte array that is then passed to the service.

See also

In the next recipe, we carry out a similar operation, namely, displaying images sent to the Silverlight application as binary data.

Displaying images as a stream from a WCF service

Applies to Silverlight 3, 4 and 5

In the previous recipe, we looked at how we could send files to a server using a WCF service that lies in between. The data is sent in that scenario as a stream of bytes.

Instead of uploading files, we may also come across a situation where a service sends data, such as images in the form of binary data (instead of just sending a link to the file). This can be the case if the files are stored in a database in binary format. In this recipe, we'll look at exactly this.

Getting ready

A starter solution containing some sample images is provided in the code bundle available on the Packt website. This solution can be found in the `Chapter08/SilverlightImageDownload_Starter` directory. The complete code for this recipe can be found in the `Chapter08/SilverlightImageDownload_Completed` directory.

How to do it...

In this recipe, we'll build a simple image search application in which the user can enter a value to find an image. This service will search for an image in a specific directory and if found, it will return this image as binary data. This way, we can send images to the Silverlight application that live on the server and not in a web folder. The following are the steps we need to perform to get this working:

1. Open the starter solution as outlined in the *Getting ready* section of this recipe.
2. Let's first focus on the server side. Add a new regular WCF service and name it `ImageDownloadService.svc`. Visual Studio will add three files: the service endpoint (`ImageDownloadService.svc`), the service contract (`IImageDownloadService.cs`), and the service implementation (`ImageDownloadService.cs`).

The service contract defines the `Download` method that returns an `ImageDownload` instance and accepts one parameter of the `string` type. This is the name of the image that the user is searching for. This method is also attributed with the `OperationContract` attribute as shown in the following code:

```
[ServiceContract]
public interface IImageDownloadService
{
    [OperationContract]
    ImageDownload Download(string imageName);
}
```

3. The `ImageDownload` class contains a string and a byte array. This is the data contract for the service, and is attributed as shown in the following code:

```
[DataContract]
public class ImageDownload
{
    [DataMember]
    public string ImageName { get; set; }
    [DataMember]
    public byte[] Image { get; set; }
}
```

4. The service implementation searches for an image with a specific name in a specified directory. If found, the image file is read and the bytes are placed in a byte array. This byte array is then wrapped into an `ImageDownload` instance that can be returned to the client side. The following code allows us to do this:

```
[AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
public class ImageDownloadService : IImageDownloadService
```

```

{
    public ImageDownload Download(string imageName)
    {
        FileStream fileStream = null;
        BinaryReader reader = null;
        string imagePath;
        byte[] imageBytes;
        try
        {
            imagePath = HttpContext.Current.Server.MapPath(".") +
                ConfigurationManager.AppSettings["ImageDirectory"] +
                imageName + ".jpg";
            if (File.Exists(imagePath))
            {
                fileStream = new FileStream(imagePath, FileMode.Open,
                    FileAccess.Read);
                reader = new BinaryReader(fileStream);
                imageBytes = reader.ReadBytes((int)fileStream.Length);
                return new ImageDownload() { ImageName = imageName,
                    Image = imageBytes };
            }
            return null;
        }
        catch (Exception)
        {
            return null;
        }
    }
}

```



Note that the service is attributed with `AspNetCompatibilityRequirementsAttribute`. This is done as we're using ASP.NET-specific features, such as `HttpContext`.

5. We have used `ConfigurationManager.AppSettings["ImageDirectory"]` in step 5. This line of code refers to an entry in the `appSettings` collection in the `web.config` file, where we store settings that are specific to the application. Add the following code in the `web.config` file (note that it is to be inserted in between the opening and closing `appSettings` tags):

```

<appSettings>
    <add key="ImageDirectory"
        value="/Images/" />
</appSettings>

```

Create the `Image` directory as a subdirectory within the web application.

- As the service runs in ASP.NET Compatibility mode, we need to allow this from the configuration code. This is done by adding the following code in the web.config file:

```
<system.serviceModel>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
</system.serviceModel>
```

- Finally, we need to change the type of binding for the server side. As we used a regular WCF service as opposed to a Silverlight-enabled WCF service, the binding added in the configuration was wsHttpBinding. Silverlight can't work with this binding, so we need to change it to a basicHttpBinding. This is shown in the following code:

```
<endpoint address=""
           binding="basicHttpBinding"
           contract="SilverlightImageDownload.Web
           .IImageDownloadService">
```

- Compile the project, so that the service can be connected to it.
- In the Silverlight project, add a service reference to the newly created service. If the service was created correctly, Visual Studio will go ahead and add a proxy to the Silverlight project. Set the namespace as ImageDownloadService.
- The UI of the application is intentionally kept simple. It can be seen a bit further in this recipe. It allows the user to enter a search term. In the Click event of the Button, the service will be called and the image will be shown, if found. The code for this UI can be found in the code bundle.
- We instantiate the proxy in the Click event. In the asynchronous call, we pass in the search term entered by the user. This is shown in the following code:

```
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    ImageDownloadService.ImageDownloadServiceClient proxy =
        new SilverlightImageDownload.ImageDownloadService
        .ImageDownloadServiceClient();
    proxy.DownloadCompleted += new EventHandler
        <SilverlightImageDownload.ImageDownloadService
        .DownloadCompletedEventArgs>(proxy_DownloadCompleted);
    proxy.DownloadAsync(SearchImageTextBox.Text);
}
```

- In the code of the callback method, we have access to the image in binary format (if one was returned by the service). The byte array in which the image data resides is loaded into the memory using a MemoryStream instance. We then create a BitmapImage and set its source to a loaded stream using the SetSource method. This BitmapImage is finally set as the Source for the image. The code for this is as follows:

```
void proxy_DownloadCompleted(object sender,
    SilverlightImageDownload.ImageDownloadService
    .DownloadCompletedEventArgs e)
{
    NoImageTextBlock.Visibility = Visibility.Collapsed;
    BitmapImage image = new BitmapImage();
    if (e.Error == null)
    {
        if (e.Result != null)
        {
            ImageDownloadService.ImageDownload imageDownload = e.Result;
            MemoryStream stream = new MemoryStream(imageDownload.Image);
            image.SetSource(stream);
            ResultImage.Source = image;
        }
        else
        {
            NoImageTextBlock.Visibility = Visibility.Visible;
        }
    }
}
```

13. As shown in the following screenshot, when searching for a specific image, it is loaded from the service and displayed in an `Image` control:



How it works...

When images are stored in a non-accessible location on the server (meaning there's no URL through which we can access them), or when an image is stored in a database, a service that returns an image to the client as an array of bytes might be available. WCF services can handle this type of traffic without problems.

Silverlight applications can capture this incoming binary data and recreate the original image from that byte array. For this, it can create a stream from the bytes. This stream can be used in combination with the `BitmapImage` class. The latter can then be used to set the `Source` for an `Image` control from the code-behind.

See also

In the previous recipe, we also worked with images: we uploaded images as binary data to a service.

9

Talking to WCF and ASMX Services—One Step Beyond

In this chapter, we will cover:

- ▶ Using duplex communication over HTTP
- ▶ Using duplex communication with the net.tcp protocol
- ▶ Ensuring that data is encrypted
- ▶ Securing service communication using message-based security
- ▶ Integrating Windows Identity Foundation in Silverlight
- ▶ Calling a WCF service from Silverlight using ChannelFactory

Introduction

In the previous chapter, we looked at the core concepts around communicating with WCF services from a Silverlight standpoint. We've seen how proxies are generated in the Silverlight application, so that we get typed access to the service types from within the Silverlight code itself. This makes working with WCF and ASMX very enjoyable, since we are getting access to a full IntelliSense experience while coding.

In .NET, WCF offers many options to the developer. From a Silverlight perspective, WCF offers a lot of possibilities as well. We've seen some of the simpler concepts in the previous chapter. The focus of this chapter is taking things further by looking at more complex ways of interacting with WCF from Silverlight code.

In some scenarios, services may want to initiate the communication with the client (instead of traditional client-server communication). We've seen earlier how sockets are a way of allowing this. However, WCF also has two ways of allowing duplex (also referred to as bidirectional) communication, namely using `HttpPollingDuplex` and `net.tcp` bindings. In this chapter, we'll take a deep look at how we can work with these, and when to choose which.

Another important but often complex aspect is security. WCF as a technology offers a wide range of options in the security area. However, a lot of these aren't working in combination with Silverlight. HTTPS, combined with message-based security, can be a good solution to implement security in your projects. Next to this, in this chapter, we also take a look at working with the **Windows Identity Foundation (WIF)**.

We'll round up this chapter by looking at using the `ChannelFactory` itself to call a service.

Using duplex communication over HTTP

Applies to Silverlight 3, 4 and 5

Most types of communication between a client and a server are initiated by the client. The client sends a request to the server and the server sends its response back to the client. However, we may sometimes need to inform the client about the changes that take place in the service. This will then be a server-initiated request to the client, as the server needs to push information to the client. This leads us to the fact that sometimes we may need so-called duplex communication, in which both the server and the client side can initiate the communication. This isn't possible out of the box because of the HTTP stack. HTTP is based on a client-initiated request, to which the server can respond.

`PollingDuplexBinding` was introduced to mimic this duplex communication. In this type of binding, the client will almost continuously poll for new messages on the server, creating the illusion of a duplex communication channel.

Silverlight 2 already had basic support for duplex communication. However, Silverlight 3 extended this model and dramatically reduced its complexity. It's now very easy to create duplex communication using WCF. In Silverlight 4 and 5, the model remained largely the same.

Getting ready

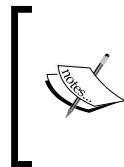
We'll build the application in this recipe from scratch. To follow along with this recipe, a starter solution is provided in the `Chapter09/DuplexStockService_Starter` folder in the code bundle that is available on the Packt site. This solution contains some images that are used in the application code. The finished solution can be found in the `Chapter09/DuplexStockService_Completed` folder.

How to do it...

We'll build a stock ticker in this recipe. A **stock ticker** is a typical scenario in which the server should be able to send new information to the client, when the value of a stock has changed. We'll start by building the service and then we'll connect the Silverlight application to the newly created service. Using `PollingDuplexBinding`, the client will receive updates as they happen on the server. The following are the steps we need to complete to make duplex communication work:

1. Open the starter solution as outlined in the *Getting ready* section. It's basically a blank solution, but it already contains the correct images used further in the XAML code.
2. Let's first focus on the service. The support for duplex communication, both on the server and the client side, is provided by two assemblies that are automatically installed with the Silverlight SDK. For the server side, there's the `System.ServiceModel.PollingDuplex.dll` assembly that is located in the `%ProgramFiles%\Microsoft SDKs\Silverlight\v5.0\Libraries\Server` directory. Note that when using a 64-bit OS, such as Windows 7 64 bit, this file will be located in the `%ProgramFiles (x86)%\Microsoft SDKs\Silverlight\v5.0\Libraries\Server` directory. Add a reference to this assembly in the web project of the solution.
3. To add the service, right-click on the web project and add a Silverlight-enabled WCF Service. Name this service `StockService`.
4. We need to make some modifications to the service so that it can work as a duplex service. First, add a new interface to the project by right-clicking on the web project, and selecting the **Interface** item in the dialog. Name this interface `IStockService`.
5. In this interface, we will define the contract the service exposes. In this case, the service exposes one operation called `Connect`, which will be invoked from the client application, passing in the stock symbol that we want to receive updates from. However, we also have to specify a client callback interface called `IStockServiceClient`. This is shown in the following code:

```
[ServiceContract(Namespace = "Silverlight", CallbackContract =
    typeof(IStockServiceClient))]
public interface IStockService
{
    [OperationContract(IsOneWay = true)]
    void Connect(string stockSymbol);
}
```



Note that the `OperationContract` was defined with the `IsOneWay` parameter set to `true`. This indicates that the operation will not return a reply message and the updates will be pushed to the client. Also, note that you may need to add a using statement for the `System.ServiceModel` namespace.

6. In the `IStockService.cs` file, we still need to define the client interface called `IStockServiceClient`. This interface defines that we can send a message of the `Update` type to the client (we'll add the `Update` class in the next step). This is shown in the following code:

```
[ServiceContract]
public interface IStockServiceClient
{
    [OperationContract(IsOneWay = true)]
    void SendUpdate(Update update);
}
```

7. The `Update` class contains information about the stock, such as the new value, the percentage change, and an indication whether it increased or decreased. The status of the stock is wrapped as an enumeration type called `UpdateType` as shown in the following code:

```
public class Update
{
    public double Amount { get; set; }
    public UpdateType UpdateType { get; set; }
    public double Percentage { get; set; }
}
public enum UpdateType
{
    Increase,
    Decrease,
    NoChange
}
```

8. Let's now implement the `IStockService` interface. The implementation should be added to the `StockService.svc.cs` file. We'll introduce a `Timer` (of the `System.Threading` namespace) that will simulate new stock values coming in. Using the `OperationContext.Current.GetCallbackChannel`, we get access to the client callback interface. This instance is then used to send updates using the `SendUpdate` method in the `Tick` event of the `Timer` as shown in the following code:

```
public class StockService : IStockService
{
```

```
private IStockServiceClient client;
private Timer updateTimer;
private Update update;
private Update previousUpdate;
public void Connect(string stockSymbol)
{
    client = OperationContext.Current.GetCallbackChannel
        <IStockServiceClient>();
    updateTimer = new Timer(new TimerCallback(TimerTick),
        null, 500, 5000);
}
void TimerTick(object state)
{
    if (client != null)
    {
        try
        {
            RefreshUpdate();
            client.SendUpdate(update);
        }
        catch (Exception)
        {
            client = null;
            updateTimer.Dispose();
        }
    }
}
private void RefreshUpdate()
{
    Random r = new Random();
    double value = Math.Round(30 + 10 * r.NextDouble(), 2);
    if (update == null)
    {
        update = new Update()
        {
            Amount = value,
            UpdateType = UpdateType.NoChange,
            Percentage = 0.0
        };
        previousUpdate = update;
    }
    else
    {
        //calculate difference
    }
}
```

```
        double diff = value > previousUpdate.Amount ? value -
            previousUpdate.Amount : previousUpdate.Amount - value;
        double percentage = diff / previousUpdate.Amount*100;
        UpdateType updateType = value > previousUpdate.Amount ?
            UpdateType.Increase : UpdateType.Decrease;
        update = new Update()
        {
            Amount = value,
            UpdateType = updateType,
            Percentage = Math.Round(percentage, 2)
        };
        previousUpdate = update;
    }
}
```

9. The last step for the service is changing its configuration. In `web.config`, in the `system.serviceModel` node, we need to specify that the service will be using polling duplex communication. The code for the configuration is as follows:

```
<system.serviceModel>
    <extensions>
        <bindingExtensions>
            <add name="pollingDuplexHttpBinding"
                type="System.ServiceModel.Configuration
                    .PollingDuplexHttpBindingCollectionElement,
                    System.ServiceModel.PollingDuplex, Version=5.0.0.0,
                    Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
        </bindingExtensions>
    </extensions>
    <behaviors>
        <serviceBehaviors>
            <behavior name="DuplexStockService.Web
                .StockServiceBehavior">
                <serviceMetadata httpGetEnabled="true"/>
                <serviceDebug includeExceptionDetailInFaults="false"/>
            </behavior>
        </serviceBehaviors>
    </behaviors>
    <bindings>
        <pollingDuplexHttpBinding>
        </pollingDuplexHttpBinding>
    </bindings>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="false"/>
    <services>
```

```

<service behaviorConfiguration="DuplexStockService.Web
    .StockServiceBehavior"
    name="DuplexStockService.Web.StockService">
    <endpoint address=""
        binding="pollingDuplexHttpBinding"
        contract="DuplexStockService.Web.IStockService"/>
    <endpoint address="mex"
        binding="mexHttpBinding"
        contract="IMetadataExchange"/>
</service>
</services>
</system.serviceModel>

```

10. Working with a duplex service from a client-side perspective is not very different from working with a regular WCF service. The biggest difference is that there's no support for working with *.config files when working with duplex services. This results in more manual work that needs to be done. Start by adding a service reference to the service. To do this, right-click on the **Service References** node in the Silverlight project and select **Add Service Reference....** In the dialog box that appears, we can find the service using the **Discover** button, as it's located in the same solution as the Silverlight project itself. Enter StockService into the **Namespace:** field. This process is entirely similar to a non-duplex service.
11. Similar to the service, we also need to add a reference to the System.ServiceModel.PollingDuplex.dll file in the Silverlight application. This file can be found in %ProgramFiles%\Microsoft SDKs\Silverlight\v5.0\ Libraries\Client directory (or %ProgramFiles(x86)%\Microsoft SDKs\Silverlight\v5.0\ Libraries\Client in the case of a 64-bit OS).
12. We want to receive updates when the user enters a stock symbol and clicks on the button. We start by creating an `EndPointAddress` instance that points to the URI of the service. After that, we create an instance of the `CustomBinding` class to make communication with the service possible. The rest of the code consists of instantiating the proxy, attaching the callback method, and invoking the `Connect` operation using the `ConnectAsync` method. Note that the `Connect` will not return a message and the callback is registered for the `SendUpdateReceived` event. This is shown in the following code:

```

private StockService.StockServiceClient proxy;
private void StartButton_Click(object sender, RoutedEventArgs e)
{
    StockSymbolTextBlock.Text = StockSymbolTextBox.Text + ":";

    EndpointAddress address = new EndpointAddress
        ("http://localhost:6138/StockService.svc");

```

```
CustomBinding binding = new CustomBinding()
    (new PollingDuplexBindingElement(),
     new BinaryMessageEncodingBindingElement(),
     new HttpTransportBindingElement());  
  
proxy = new DuplexStockService.StockService.  
StockServiceClient(binding, address);  
  
proxy.SendUpdateReceived +=  
    new EventHandler<DuplexStockService.StockService.  
SendUpdateReceivedEventArgs>  
    (proxy_SendUpdateReceived);  
  
proxy.ConnectAsync(StockSymbolTextBox.Text);  
}
```

[ Note that the port number (localhost:6138) may be different in your own project. You'll need to add a using statement for the System.ServiceModel and the System.ServiceModel.Channels namespaces as well.]

13. We have access to the UI from the callback (so no crossing threads are needed here). We can update the UI with a new value for every message that comes from the server. This can be achieved using the following code:

```
void proxy_SendUpdateReceived(object sender, DuplexStockService
    .StockService.SendUpdateReceivedEventArgs e)
{
    this.DataContext = e.update;
}
```

We have now created a service that allows duplex communication with a Silverlight application. The result is shown in the following image:



How it works...

To explain how duplex communication works, we need to cover several items. Let's start with what really happens behind the scenes.

PollingDuplexBinding: polling, binding, and assemblies

There's a real need of supporting two-way communication in real-world applications. In some scenarios, in addition to requesting data from the client, the server must be capable of initiating the action itself and sending data to the client side. In Silverlight 3, two options were available: using sockets (which is explained in the *Using socket communication in Silverlight* recipe in Chapter 7, *Working with Services*) and using duplex bindings. (In fact, a third option was added with version 4, namely the `net.tcp` binding, which is the subject of the following recipe.) While sockets can be a great solution, they have their disadvantages as well. They communicate over an exotic range of ports, which may not always be accessible through firewalls, making it impossible to use them.

However, duplex communication is implemented through `PollingDuplexBinding` in WCF. By looking at the name of this binding, it's easy to see that it's implemented using some sort of polling, sometimes referred as smart polling. The reason for this name is writing a polling client is something we could do ourselves as well. In that case, the Silverlight client application would contact the server quite often for new information. However, this would have its complications. Finding a perfect poll interval would be rather difficult. If we set the interval too small, we'll bring down the server. If we set it too large, we won't be notified about updates in time. This solution would also be difficult to scale out, resulting in server overload, when too many clients start polling.

Due to this, Microsoft introduced the smart polling technique. The client has to contact the server to initiate the communication. The reason for this is that we're working over HTTP, and HTTP first requires a request to start the communication. If the communication had taken place over TCP, this wouldn't have been the case as TCP is a duplex protocol. Once a connection is made, the `PollingDuplexBinding` allows a channel to remain connected until a timeout occurs, disconnecting only to refresh the channel or to send received messages to the client. In between that time, the client polls the service almost continuously. The timeout is defined by the `PollTimeout` property on `PollingDuplexBindingElement` and defaults to five minutes. When the timeout occurs, no exceptions are thrown if the client and the server haven't aborted the session. If they have, a `TimeoutException` is raised. When the channel needs to refresh, no exception is thrown; an empty response is returned, and the channel is opened up again. As a result, the polling starts again. If messages are created within this period, they are stored on the server and are sent to the client on the first poll.

A second type of timeout called `InactivityTimeout` is also available. It defines the interval of time that can pass without activity from either the client or the server. It has a default value of ten minutes, after which the channel is set to a faulted state.

To support duplex binding both on the server and the client side, we need to add a reference to an assembly that provides support for duplex binding. This assembly is named `System.ServiceModel.PollingDuplex.dll` on both sides. On the server side, this assembly is located by default at `%ProgramFiles%\Microsoft SDKs\Silverlight\v5.0\Libraries\Server`. On the client side, it is located at `%ProgramFiles%\Microsoft SDKs\Silverlight\v5.0\Libraries\Client`. If you are using an older version of Silverlight (3 or 4), simply replace the 5.0 directory with the correct version name to locate these binaries.

The service

The service does not differ that much from a WCF service written for one-way communication. However, some specifics need to be taken into account to make duplex bindings work. Let's take a look at them. In the service contract (both `IStockService` and `IStockServiceClient`), we specified the `IsOneWay` property as `true` in the `OperationContract` attribute. The result of setting this value is that the client will not wait for the service to return, which it would normally do in a one-way scenario. In this case, the client calls the method, but returns immediately and does not wait for the service code to execute. Due to this, the operations need to have `void` as the return type as no value can be returned.

The service contract is attributed with the `CallbackContract` attribute. The specified contract—`IStockServiceClient`—is used as a callback contract. This allows client applications to listen to the incoming operation calls sent independently by the service.

Using the `GetCallbackChannel` method in the service, we get access to the client instance that made the call to the service. This instance is of the `IstockServiceClient` type, and the interface is defined as a callback contract. Because of this, we can call the `SendUpdate` method on it, thus triggering the operation on the client side.

Finally, the service requires configuration. In this configuration, we specify that the service will be communicating using duplex communication. The most important part in this is the addition of `pollingDuplexHttpBinding` in the bindings collection:

```
<bindings>
  <pollingDuplexHttpBinding>
    </pollingDuplexHttpBinding>
</bindings>
```

This binding is then used as the binding for the endpoint of the service using the following code:

```
<endpoint address=""
           binding="pollingDuplexHttpBinding"
           contract="DuplexStockService.Web.IStockService"/>
```

The client

With the service in place, one or more clients can connect to the service. We need to define an `EndPointAddress` instance in the client code. The `EndPointAddress` class contains the address of the service endpoint that will be used for duplex communication. We need to create the binding instance manually. To do this, we use `CustomBinding`. This instance is initialized with a list of binding elements that define the binding when taken together.

The actual communication is carried out using a proxy similar to a normal service. However, there are a few differences. The first one is the instantiation of the proxy. It requires both the `EndPointAddress` and the `CustomBinding` instance to be passed in. Secondly, the method we call asynchronously is not the same as the one we register to receive callbacks from. This is quite logical. The asynchronously called method is used in this example to connect to the service. The callbacks are received when the service invokes the client operations.

See also

Duplex services are one way to perform duplex communication. Another way is to use socket services, which are discussed in the *Using socket communication in Silverlight* recipe in Chapter 7, *Working with Services*.

Using duplex communication with the WCF net.tcp binding

Applies to Silverlight 4 and 5

So far, we have looked at two technologies that allow us to perform duplex communications. In Chapter 7, *Working with Services*, we looked at how we could set this up using TCP sockets communication. The biggest advantage of using sockets is their speed. The second option is duplex WCF services, which is the focus of the previous recipe. Being WCF services, the programming model is much simpler. Based on contracts, we get typed access to the objects with which we're working. However, they rely on HTTP for their communication, resulting in slower speeds.

With Silverlight 4, a new option was added which brings the best of both worlds—WCF `net.tcp` binding. We can still use the same simple programming model from WCF, and benefit from better transport speed. However, note that `net.tcp` isn't targeted only at duplex communication. We can also use this binding for regular service communications.

In this recipe, we'll see how we can change the WCF duplex communication sample created in the previous recipe to use the `net.tcp` binding.

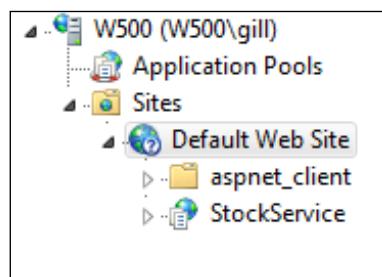
Getting ready

This recipe will change the code created in the previous recipe to use the WCF net.tcp binding. However, it's advisable to use the starter solution located in the Chapter09/DuplexCommunication_NetTcp_Starter folder in the code bundle that is available on the Packt website. The finished solution for this recipe can be found in the Chapter09/DuplexCommunication_NetTcp_Completed folder.

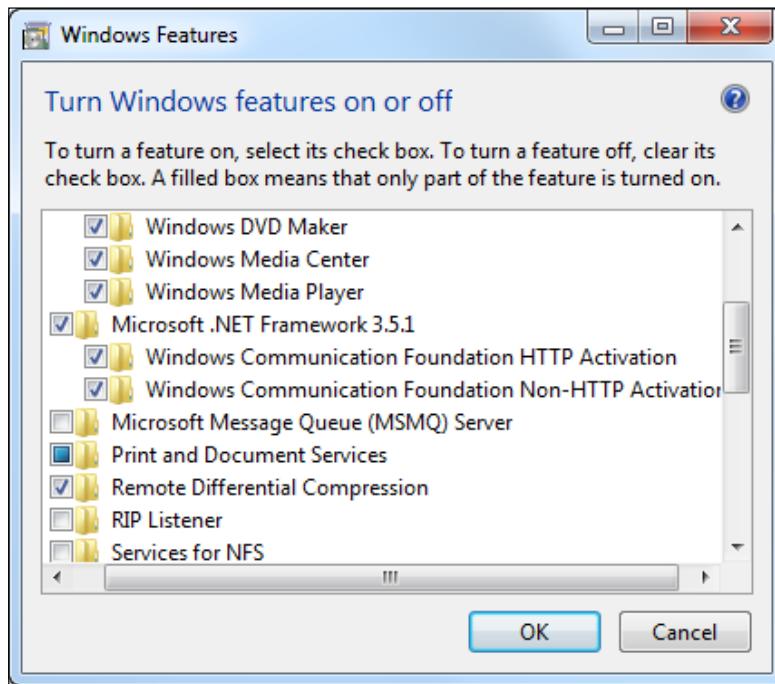
How to do it...

In this recipe, we'll start from the sample we created in the previous recipe that uses WCF duplex communication over HTTP. Changing the application to use net.tcp, instead of HTTP for its communication in itself, boils down to changing the configuration code within web.config. However, we also need to make some changes in the way the services are hosted. The following are the steps we need to perform:

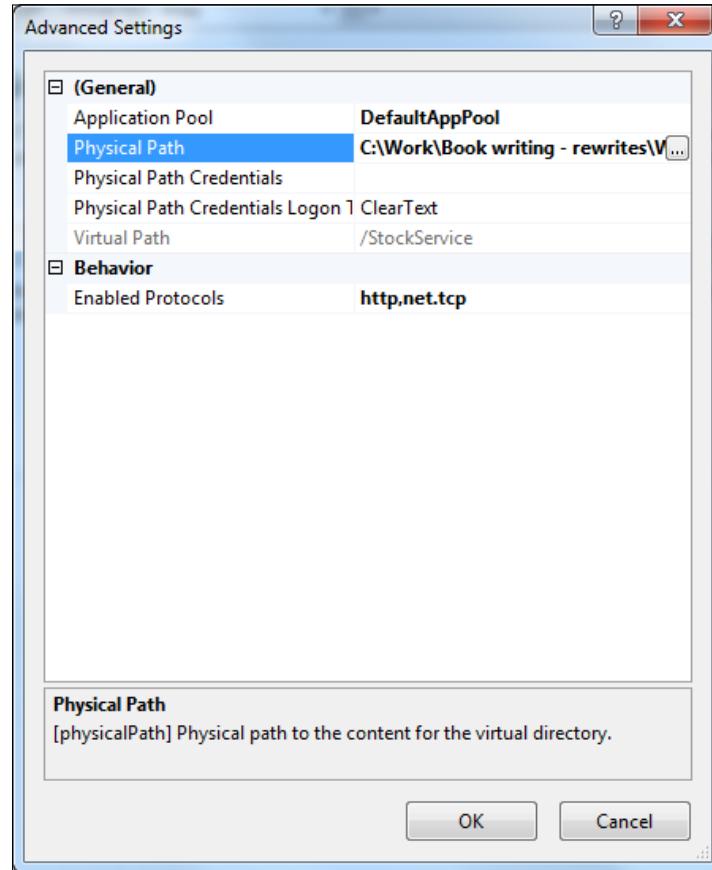
1. Open the solution as outlined in the *Getting ready* section of this recipe.
2. To set up a net.tcp communication, we need to make some changes on the web server software. The built-in ASP.NET Web Server of Visual Studio does not support net.tcp communication. So, we'll need to use **Internet Information Services (IIS)** for this.
3. Deploying the hosting site to IIS can be carried out in several ways. In this recipe, we'll take the manual route. Open IIS and create a new application under the **Default Web Site** node. Name this application StockService. The physical path of the site should point to the folder of the web project of the solution. The result is shown in the following image:



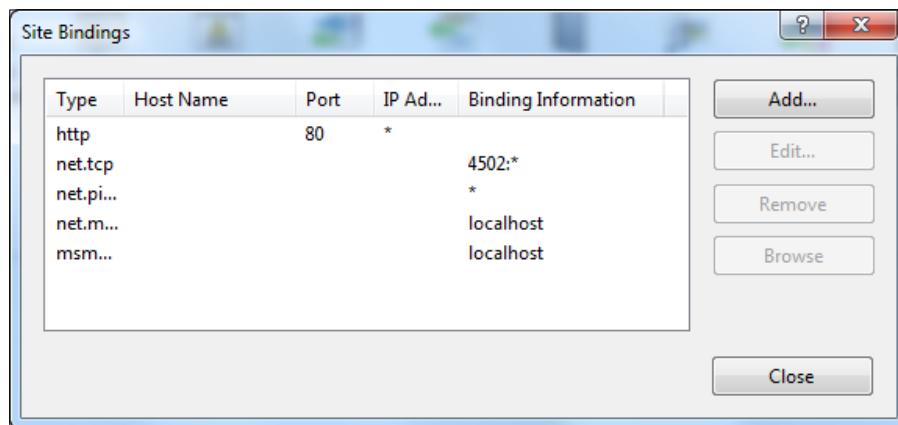
4. When hosting WCF services with `net.tcp` in IIS 7, we need to enable **Windows Process Activation Service (WAS)**. Web applications are normally activated based on incoming HTTP requests. With `net.tcp` communication, a web application will be activated by a `net.tcp` request. This feature is not enabled on IIS by default. To enable it, go to **Windows Features** and under **Microsoft .NET Framework 3.5.1**, check **Windows Communication Foundation Non-HTTP Activation**. This is shown in the following screenshot:



5. In the IIS, we have to enable the `net.tcp` protocol. To do this, right-click on the **StockService** application in the **IIS Manager**, and select **Manage Application | Advanced settings**. In the **Enabled Protocols** field, add `http,net.tcp` as shown in the following screenshot:



6. We should also open the port for communication over `net.tcp`. The available ports are the same as with TCP communication. To do this, right-click on the website hosting the application (in our case, **Default Web Site**) and select **Edit bindings**. In the dialog box that appears, add the port **4502** (lowest available port number) for the `net.tcp` binding type, as shown in the following screenshot:



7. Now that we have IIS configured, we need to specify in Visual Studio that the `DuplexCommunication`.`Web` website should start from this IIS site, instead of the built-in ASP.NET server. To do this, right-click on the hosting website in the **Solution Explorer** and select **Properties**. In the properties window, under the **Web** tab, select the **Use Local IIS Web server** option and point to the web application in IIS (for example, `http://localhost/StockService`).
8. We have to make changes to `web.config`, so that the service will support the `net.tcp` binding. To do this, change the code in `web.config` as follows:

```

<system.serviceModel>
  <extensions>
    <bindingExtensions>
      <add name="pollingDuplexBinding"
           type="System.ServiceModel.Configuration
                  .PollingDuplexHttpBindingCollectionElement,
                  System.ServiceModel.PollingDuplex"/>
    </bindingExtensions>
  </extensions>
  <behaviors>
    <serviceBehaviors>
      <behavior name="DuplexCommunicationNetTcpBehavior">
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>

```

```
</behaviors>
<bindings>
    <pollingDuplexBinding>
        <binding name="DuplexNetTcpBinding"
            useTextEncoding="true"/>
    </pollingDuplexBinding>
    <netTcpBinding>
        <binding name="DuplexNetTcpBinding">
            <security mode="None" />
        </binding>
    </netTcpBinding>
</bindings>
<services>
    <service behaviorConfiguration="DuplexCommunication
        NetTcpBehavior"
        name="DuplexCommunication.Web.StockService">
        <endpoint address=""
            binding="pollingDuplexBinding"
            bindingConfiguration="DuplexNetTcpBinding"
            contract="DuplexCommunication.Web.IStockService"/>
        <endpoint address=""
            binding="netTcpBinding"
            bindingConfiguration="DuplexNetTcpBinding"
            contract="DuplexCommunication.Web.IStockService"/>
        <endpoint address="mex"
            binding="mexHttpBinding"
            contract="IMetadataExchange"/>
    </service>
</services>
</system.serviceModel>
```

9. The service code (the `IStockService` contract and the `StockService.svc.cs` implementation) doesn't need any changes. However, the `Update` class does need a few changes. The new version is shown in the following code:

```
[MessageContract]
public class Update
{
    [MessageBodyMember]
    public double Amount { get; set; }

    [MessageBodyMember]
    public double Percentage { get; set; }
}
```

10. You'll need to add a using statement to the `System.ServiceModel` namespace for the `MessageBodyMember` attribute.
11. Just like TCP socket communication (refer to *Chapter 7, Working with Services*, for more information), `net.tcp` communication needs a policy server. The code for this is not included in this recipe. However, it can be found in the `PolicyServer` project in the code bundle of this recipe.
12. As we're now ready with the service, we can build the solution.
13. Since we're communicating with a WCF service, we can create a proxy class using the **Add Service Reference...** option inside the Silverlight application. Along with the proxy generation, configuration code is also generated in the `ServiceReferences.ClientConfig` code. The generated code refers to the service using `net.tcp`. It is shown in the following code:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="NetTcpBinding_IStockService">
          <binaryMessageEncoding />
          <tcpTransport maxReceivedMessageSize="2147483647"
                         maxBufferSize="2147483647" />
        </binding>
      </customBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://localhost:4502/StockService/
                     StockService.svc"
                 binding="customBinding"
                 bindingConfiguration="NetTcpBinding_IStockService"
                 contract="StockService.IStockService"
                 name="NetTcpBinding_IStockService" />
    </client>
  </system.serviceModel>
</configuration>

```

14. The Silverlight client code can be made easier than the code with regular duplex WCF services. We can create an instance of the proxy by passing in the name of the binding. This can be seen in the following code:

```

private void StartButton_Click(object sender, RoutedEventArgs e)
{
  StockService.StockServiceClient client = new
    StockService.StockServiceClient("NetTcpBinding
      _IStockService");
  client.SendUpdateReceived += new EventHandler
    <StockService.SendUpdateReceivedEventArgs>
    (client_SendUpdateReceived);
}

```

```
        client.ConnectAsync("MSFT");
    }
    void client_SendUpdateReceived(object sender,
        StockService.SendUpdateReceivedEventArgs e)
    {
        AmountTextBlock.Text = (e.request as Update).Amount.ToString();
        PercentageTextBlock.Text = (e.request as
            Update).Percentage.ToString();
    }
}
```

We have now completed the switch of the application to use `net.tcp` binding, instead of duplex communication over HTTP.

How it works...

With Silverlight 2 and Silverlight 3, we were basically limited to using the `BasicHttpBinding`. Silverlight 4 (and of course also Silverlight 5) have support for the `net.tcp` binding. The major benefit of this binding is improved performance, while still using the easy programming model offered by WCF.

Another advantage is the fact that it can also perform duplex communication. This results in Silverlight now having the following three ways to perform two-way communication between a service and a client:

- ▶ **Duplex WCF communication:** This uses HTTP for its communication, but benefits from the easy programming model of WCF.
- ▶ **TCP communication:** This enables fast communication because of TCP. It's more complex to set up.
- ▶ **Net.tcp:** This combines the speed of TCP with the easy programming model of WCF.

`net.tcp` builds `System.Net.Sockets` underneath the covers. Due to this, it has the same restrictions. Silverlight applications can connect to TCP services only on ports between 4502 and 4534. The same holds true for `net.tcp`. Due to this, like TCP, `net.tcp` is a good fit for intranet applications and not for Internet applications. Also, as for TCP, when connecting with `net.tcp` services, Silverlight requires the presence of a policy server.

On the client side, we can use the **Add Service Reference...** dialog box to build a proxy class. The use of the `net.tcp` binding is detected and the configuration code is generated as well. The actual code to perform the call to the service uses the same model, as used for communication with other services. By specifying the name of the binding in the client configuration code, we can instantiate the proxy using the following code:

```
StockService.StockServiceClient client = new
    StockService.StockServiceClient
    ("NetTcpBinding_IStockService");
```

Running with elevated permissions

Since Silverlight 4, applications can run with elevated permissions. In Silverlight 4, this mode was only supported in out-of-browser applications, whereas Silverlight 5 also allows running in-browser applications with elevated permissions.

When you are building an application that runs with these permissions, you aren't limited to connecting only to ports within the previously mentioned range (4502-4534). This makes it possible to create Silverlight applications that use the `net.tcp` binding over the Internet.

Setup requirements

When working with `net.tcp`, we can't use the ASP.NET Development Server (built-in with Visual Studio) or IIS Express. Neither of these supports the `net.tcp` protocol. Due to this, we need to host the website containing the services in IIS. However, by default, IIS isn't configured to support this protocol out-of-the-box.

See also

This recipe used the `net.tcp` binding for its communication and we used it for building a duplex communication mechanism. In the *Using duplex communication* recipe, we built the same application using duplex communication over HTTP. In the *Using socket communication in Silverlight* recipe in Chapter 7, *Working with Services*, we used TCP for this two-way communication.

Ensuring data is encrypted

Applies to Silverlight 4 and 5

In Silverlight, you'll typically work with services oriented through WCF services, REST services, and so on. However, this implies that your data is sent over the wire in an unencrypted format. This is not much of a problem in most controlled environments. But what if the data is sent through an uncontrolled environment (for example the Internet)? It could be easily intercepted by someone with bad intentions. While that's bad when you're sending confidential data, it's even worse when your message header includes a username/password combination.

In this recipe, you'll learn how to make sure that the data sent over the wire is encrypted using **Secure Socket Layer communication (SSL)**.

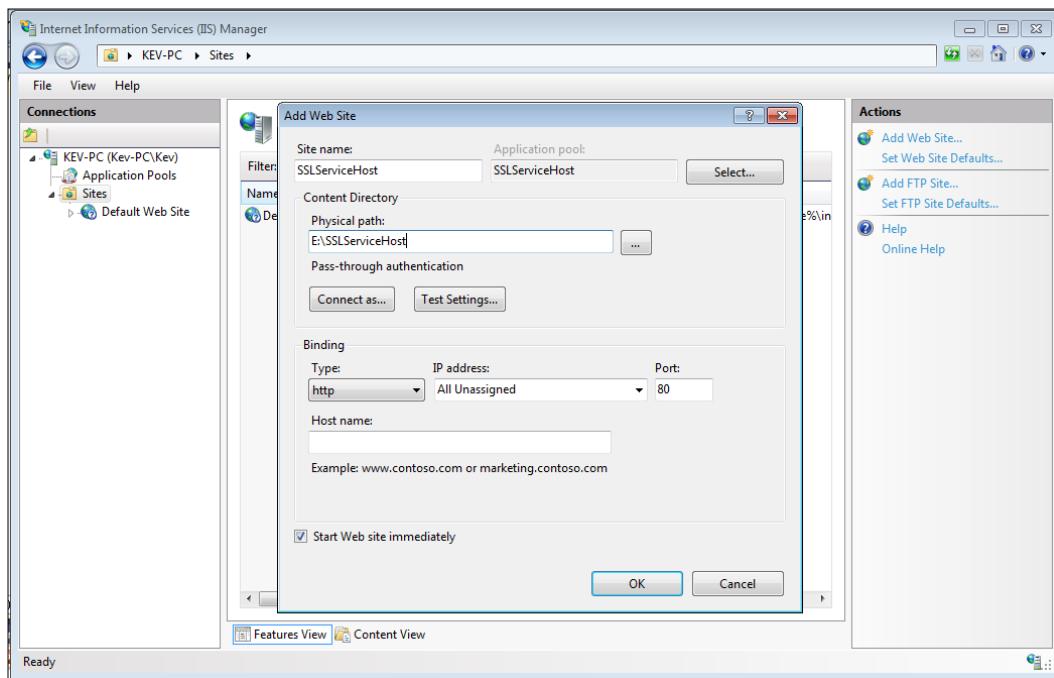
Getting ready

This recipe is mainly about configuring IIS and WCF, and it's much less about writing code. What you need is a simple Silverlight application calling a WCF service and a correctly configured/installed IIS web server, where your Silverlight application and service host are hosted. A starter solution is located in the `Chapter09/EnsuringDataIsEncrypted_Starter` folder in the code bundle available on the Packt website. The completed solution can be found in the `Chapter09/EnsuringDataIsEncrypted_Completed\` folder.

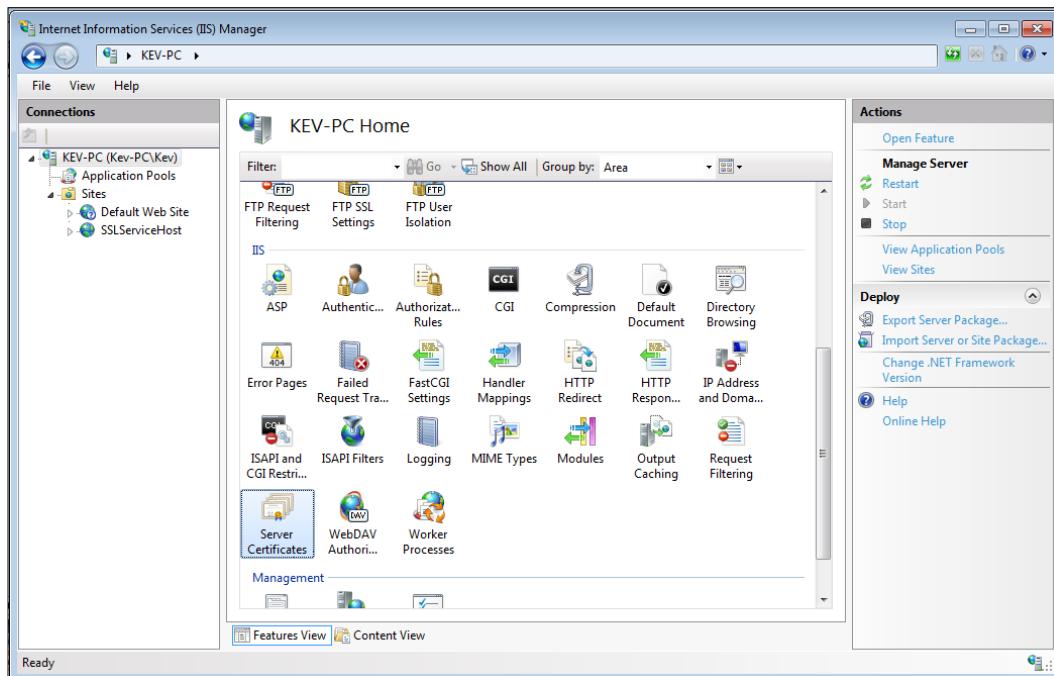
How to do it...

We're going to set up a website in IIS 7.0 to use SSL through a self-signed certificate. After this, we're going to make sure that our WCF service is hosted on this website and securely callable from Silverlight. To achieve this, carry out the following steps:

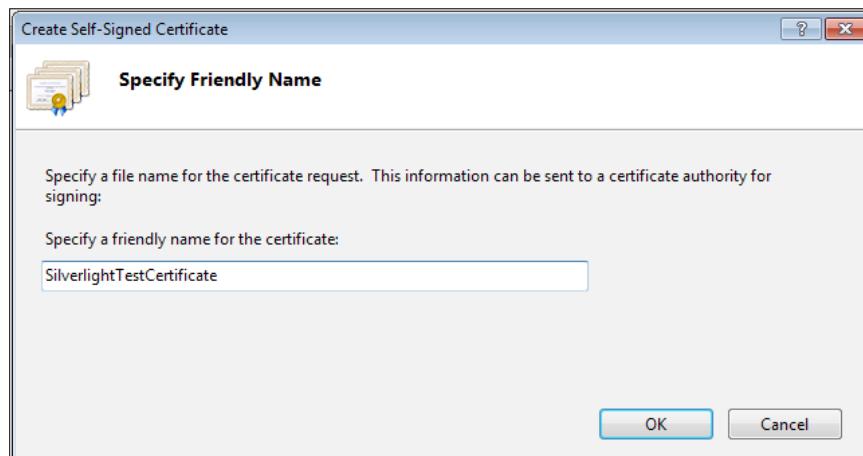
1. Open **Internet Information Services (IIS) Manager** and create a new website by right-clicking on **Sites** on the left side of the IIS Manager and selecting **Add Web Site**. We'll name this website as `SSLServiceHost`. This can be seen in the following screenshot:



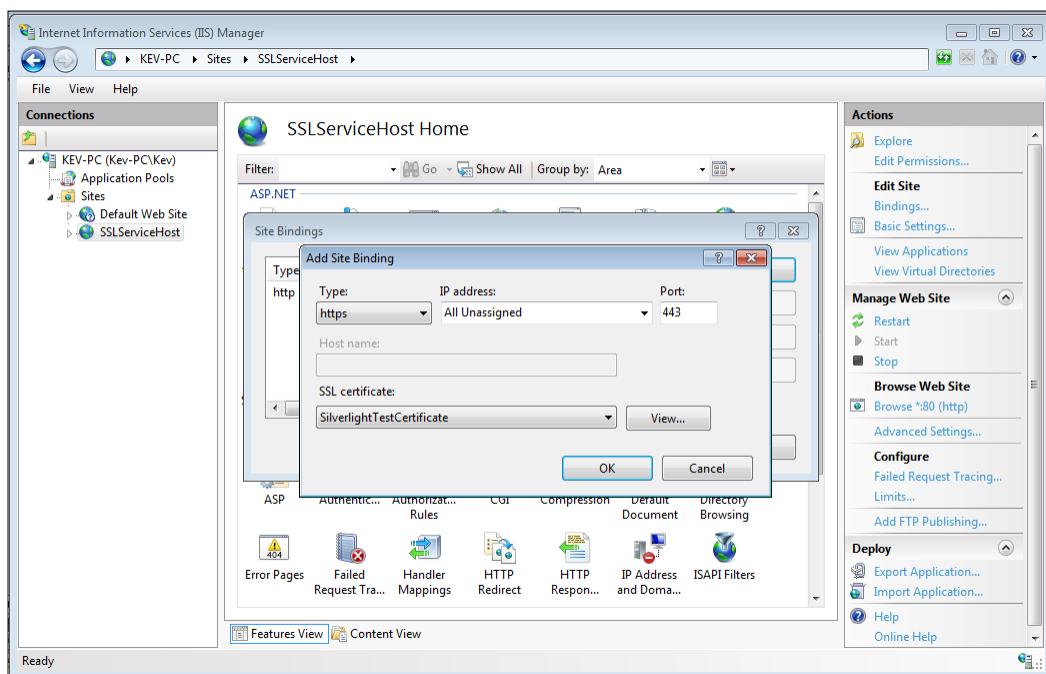
2. We're going to need a security certificate to use with the SSL binding. For real applications, you might want to get one from one of the many vendors on the Internet (for example **Verisign**), but in this example, we're going to create a self-signed one for testing purposes. Self-signed certificates are not meant to be used on live web sites; however, they provide an easy way of working with certificates during development. Click on the root machine node on the left side of your IIS Manager and double-click on **Server Certificates**. This is shown in the following screenshot:



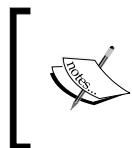
3. In the right-hand pane, click on **Create Self-Signed Certificate** and complete the wizard by providing **SilverlightTestCertificate** as a friendly name for this certificate. This can be seen in the following screenshot:



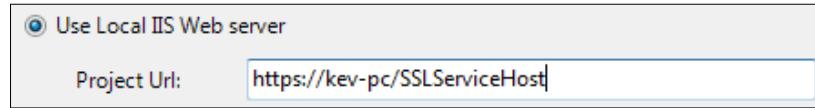
4. Click on **SSLServiceHost** on the left side of your IIS Manager, click on **Bindings...** on the right-hand pane, and add an SSL binding for this website using our freshly-created security certificate:



- Our website is now set up for secure communication over https. We're now going to host our WCF service on this site. Open the provided starter solution and navigate to the **Web** tab in the project properties of the `EnsuringDataIsEncrypted.Web` application. Select **Use Local IIS Web server** and make sure that the **Project Url:** refers to the website (https) we've created in the previous steps:



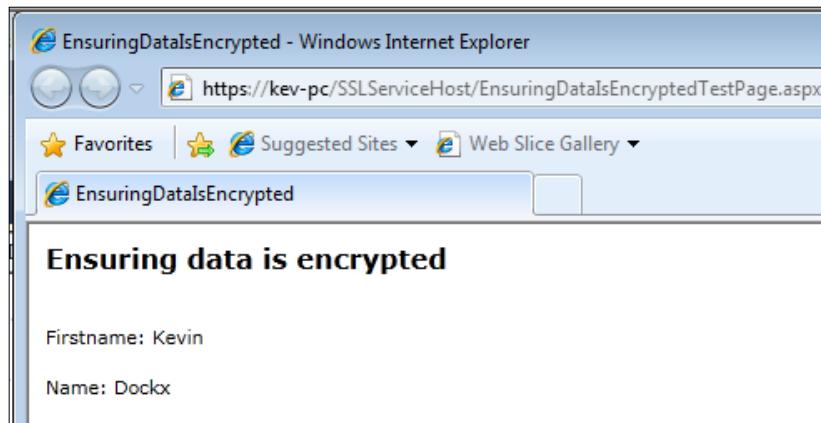
Refer to the website by the server name and not using `localhost`. If you use `localhost`, the certificate will throw a warning (**computer name does not match**, because `localhost` isn't the same as your server name), and you won't be able to access your service.



- Open the `web.config` file, locate the custom binding with the name `EnsuringDataIsEncrypted.Web.MyWCFService.customBinding0`, and change the `HttpTransport` element to an `HttpsTransport` element.
- Open your Silverlight application, add a Service Reference to `MyWCFService`, and name it `MyWCFServiceReference`. If you get a warning concerning the security certificate, ignore it (as this is a self-signed certificate).
- Locate the `btnCall_Click` handler in `MainPage.xaml.cs` and replace it with the following code:

```
private void btnCall_Click(object sender, RoutedEventArgs e)
{
    MyWCFServiceReference.MyWCFServiceClient client = new
        MyWCFServiceReference.MyWCFServiceClient();
    client.GetPersonCompleted += new EventHandler
        <MyWCFServiceReference.GetPersonCompletedEventArgs>
        (client_GetPersonCompleted);
    client.GetPersonAsync();
}
void client_GetPersonCompleted(object sender,
    MyWCFServiceReference.GetPersonCompletedEventArgs e)
{
    if (e.Error == null)
    {
        this.DataContext = e.Result;
    }
}
```

9. You can now build and run your application. Your application will securely call the WCF service over HTTPS as follows:



How it works...

To ensure communication is secure (using SSL), we need to complete two steps. The first one is creating a website in IIS setup to use a certificate, either the one you bought (for production sites) or the one you've created yourself (for testing purposes). This is done in steps 1 to 4.

When that's done, we need to configure our application to use this website and the correct bindings. Configuring the application to use this website is done through the project properties window. By changing the web server to our previously created virtual directory instead of the development server, we ensure SSL communication is possible. It's impossible to use SSL with the built-in Visual Studio development server.

Starting from step 6, we make changes to our application. To be able to use transport security (SSL), we need to tell WCF to use this. This is done in step 6. We make sure that communication with our endpoint is possible only through SSL (https), and not through regular http by changing the transport element in the `web.config` to `HttpsTransport`.

Let's now go back to the Silverlight application. As we've done in previous recipes, we're adding a service reference to the service at the https endpoint. When we do this, Visual Studio generates the accompanying proxy classes, as well as generates the correct bindings in `ServiceReferences.ClientConfig`. If you look at this file, you'll notice that the generated binding is enabled for `HttpsTransport` security.

All that's left now is adding code to call the `GetPerson()` service operation. This is done in step 8. If you build and run your solution now, all communication will be done through SSL and your messages will be encrypted. You could use a tool like Fiddler to look at the network traffic: you'll see the traffic isn't in a readable format anymore, it's encrypted instead. Have a look at the appendix for more information about Fiddler.

See also

If you want to learn how to use message-based security (through SSL), have a look at the next recipe.

Securing service communication using message-based security

Applies to Silverlight 4 and 5

Service-oriented solutions typically require the services to know which user is accessing them, whether this user is authenticated or not, and even whether this user is authorized or not. If you don't implement these checks and if your services are hosted in a publicly available, uncontrolled environment, everyone will be able to access them. This is definitely something you don't want, especially if confidential data is fetched or even edited using service calls.

To resolve this, we need to make sure that certain service methods are accessible only to certain persons. A typical way of doing this is by using message-based security, which in this case means that you'd include the username/password combination in every message that's being sent to the service, where the received combination is then checked. If the combination is valid, the service operation can be executed. If not, typically a fault is returned.



Note that it's not possible to implement full message-based security as defined by the WS Security specification, as Silverlight does not support this. Passing the credentials in the message header is the farthest we can go.

In this recipe, you'll learn how to do this with your Silverlight application using WCF services.

Getting ready

We're starting from a simple Silverlight solution that includes a Silverlight project that calls a WCF service method through HTTPS. Make sure that you've set up your IIS for HTTPS as explained in the previous recipe, because this recipe uses the website you've created in that recipe for hosting your WCF Service. If you want to follow along with this recipe, you can find a starter solution located in the `Chapter09/MessageBasedSecurity_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter09/MessageBasedSecurity_Completed` folder.

How to do it...

We're going to modify the starter solution to ensure that credentials are passed to the server and are checked before allowing further execution of the service method. To achieve this, complete the following steps:

1. Navigate to the **Web** tab in the project properties of the web application and change the virtual directory to a valid, SSL-enabled virtual directory on your computer.



Have a look at the recipe *Ensuring data is encrypted* in this chapter to learn how to enable your applications for SSL.

2. Add a reference to `System.IdentityModel` and `System.IdentityModel.Selectors` to the web application.
3. Open the web application and add a new class. Name this class `UsernamePWValidator`, let it inherit `UsernamePasswordValidator`, and implement it as shown in the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.IdentityModel.Selectors;
namespace MessageBasedSecurity.Web
{
    public class UsernamePWValidator : UserNamePasswordValidator
    {
        public UsernamePWValidator()
        {
        }
        public override void Validate(string userName,
            string password)
        {
            if (userName != "Kevin" || password != "Dockx")
            {
                throw new FaultException("The provided credentials
                    are invalid.");
            }
        }
    }
}
```

4. Open the `web.config` file, locate the included service behavior, and change it to the following code:

```
<behavior name="MyServiceBehaviour">
<serviceMetadata httpGetEnabled="true" />
<serviceDebug includeExceptionDetailInFaults="false" />
<serviceCredentials>
    <userNameAuthentication userNamePasswordValidationMode=
        "Custom"
        customUserNamePasswordValidatorType=
        "MessageBasedSecurity.Web"
```

```

        .UsernamePWValidator,
    MessageBasedSecurity.Web" />
</serviceCredentials>
</behavior>
```

5. In the same file, locate the custom binding called `MessageBasedSecurity.Web.MyWCFService.customBinding0` and change the code as follows:

```

<customBinding>
    <binding name="MessageBasedSecurity.Web.MyWCFService
        .customBinding0">
        <security authenticationMode="UserNameOverTransport" />
        <binaryMessageEncoding />
        <httpsTransport />
    </binding>
</customBinding>
```

6. In the same file, locate the service endpoint and change it as follows (remember to replace the `baseAddress` with your local IIS server name):

```

<service name="MessageBasedSecurity.Web.MyWCFService"
    behaviorConfiguration="MyServiceBehaviour">
    <host>
        <baseAddresses>
            <add baseAddress="https://kev-pc"/>
        </baseAddresses>
    </host>
    <endpoint address=""
        binding="customBinding"
        bindingConfiguration="MessageBasedSecurity.Web
            .MyWCFService.customBinding0"
        contract="MessageBasedSecurity.Web.MyWCFService" />
    <endpoint address="mex"
        binding="mexHttpBinding"
        contract="IMetadataExchange" />
</service>
```

7. In your Silverlight project, add a service reference to `MyWCFService` and name it `MyWCFServiceReference`.
8. Replace the `Click` handler of the `btnWrongCredentials` button with the following code:

```

private void btnWrongCredentials_Click(object sender,
    RoutedEventArgs e)
{
    WebRequest.RegisterPrefix("https://",
        WebRequestCreator.ClientHttp);
    this.DataContext = null;
    MyWCFServiceReference.MyWCFServiceClient client = new
        MyWCFServiceReference.MyWCFServiceClient();
```

```
        client.GetPersonCompleted += new EventHandler
            <MyWCFServiceReference.GetPersonCompletedEventArgs>
            (client_GetPersonCompleted);
        client.ClientCredentials.UserName.UserName = "WrongUsername";
        client.ClientCredentials.UserName.Password = "WrongPassword";
        client.GetPersonAsync();
    }
    void client_GetPersonCompleted(object sender,
        MyWCFServiceReference.GetPersonCompletedEventArgs e)
    {
        if (e.Error == null)
        {
            this.DataContext = e.Result;
        }
        else
        {
            MessageBox.Show(e.Error.Message);
        }
    }
}
```

9. Replace the Click handler of the btnCredentials button with the following code:

```
private void btnCredentials_Click(object sender,
    RoutedEventArgs e)
{
    WebRequest.RegisterPrefix("https://",
        WebRequestCreator.ClientHttp);
    this.DataContext = null;
    MyWCFServiceReference.MyWCFServiceClient client = new
        MyWCFServiceReference.MyWCFServiceClient();
    client.GetPersonCompleted += new EventHandler
        <MyWCFServiceReference.GetPersonCompletedEventArgs>
        (client_GetPersonCompletedWithCredentials);
    client.ClientCredentials.UserName.UserName = "Kevin";
    client.ClientCredentials.UserName.Password = "Dockx";
    client.GetPersonAsync();
}
void client_GetPersonCompletedWithCredentials(object sender,
    MyWCFServiceReference.GetPersonCompletedEventArgs e)
{
    if (e.Error == null)
    {
        this.DataContext = e.Result;
    }
    else
    {
        MessageBox.Show(e.Error.Message);
    }
}
```

10. You can now build and run your solution. Depending on which button you click, the right credentials will be sent and your service call will either succeed or fail.

How it works...

As you've noticed, most of the work is done through WCF configuration and not through code changes. We want to make sure that our WCF service expects credentials and we need to provide a way to validate these credentials.

In step 4, we've changed the behavior our endpoint uses to expect credentials. This is done through the `userNameAuthentication` element. By providing this element, we make sure that any endpoint using this behavior expects client credentials in the form of a username/password combination.

As well as that, we define the way these credentials must be validated by setting the value of the `customUserNamePasswordValidatorType` attribute to a custom validator. This custom validator is the one that we've created in step 3. It must inherit the `UserNamePasswordValidator` class and we override the `Validate` method to define our own validation. In this specific example, validation will succeed when the username/password combination is Kevin/Dockx. All other combinations will fail and will prevent the service method from being executed. Of course, you can write your own way of validating by changing the `Validate` method. For example, you could check your own custom user tables in an underlying database.

We've already defined our custom behavior. What's next is defining the custom binding to be used by our service endpoint. This is done in step 5. The binding is changed to expect the `UserNameOverTransport` authentication and to use `HttpsTransport` (SSL).

We now need to bring this together, which is done in step 6. Our service is configured to use the custom behavior that we created in step 4 by setting the `BehaviorConfiguration` attribute on the service element to our custom behavior. The endpoint is configured to use our custom binding by setting the `bindingConfiguration` attribute on the endpoint element.

After this, everything that is to be done on the service side of our application is done. Next up is the client side. First, a service reference is added that will generate the necessary proxy classes and a client config file. The two different buttons in our application are set to include client credentials on the client proxy instance (this has to be done only once per proxy instance). In the first button handler, the wrong credentials are set. In the second handler, the correct credentials will be passed to the service.

When you call the `GetPerson()` service method, the client credentials are securely sent over the wire in the message header (through SSL) and they're automatically checked by the `Validate` method of our custom validator. Only if they check out will the method be executed.

Transport security is required

Due to security measures, Silverlight doesn't allow a binding as used in this recipe (with credentials sent over the wire) to be used over regular HTTP. Allowing this would mean that the credentials would be sent in clear text. When you're using a binding that expects client credentials, you're obligated to use SSL (as defined by the `HttpsTransport` element in the `web.config` file).

See also

To learn how to secure your communication using SSL, have a look at the previous recipe.

Integrating Windows Identity Foundation in Silverlight

Applies to Silverlight 4 and 5

Authentication and authorization are the requirements for almost every serious Line of Business application. There are quite a few options to get this up and running with Silverlight, and one of them is by using **Windows Identity Foundation (WIF)**. This consists of a set of classes with which you can build identity-aware applications. It allows us to externalize authentication and authorization: the application relies on an identity provider for this.

In this recipe, we'll learn how to use WIF with Silverlight, using an external **Security Token Service** (the identity provider), which will take care of authentication/authorization for us.

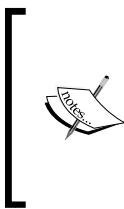
Getting ready

This recipe relies on quite a few prerequisites. You'll have to have at least Windows Vista SP2 (Windows 7 or 2008 server advised), and IIS 7.0 (or better).

First of all, you'll need to install the **Windows Identity Foundation Runtime**, which can be found at <http://www.microsoft.com/download/en/details.aspx?id=17331>. Choose the correct version, depending on your system.

Next, you'll need to install the **Windows Identity Foundation SDK**, which can be found at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=4451>. Choose the SDK version for .NET 4.0.

After this, you'll need to install the **Windows Identity Foundation Developer Training Kit** (April 2011), which can be found at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=14347>. This kit contains sample implementations of, amongst others, a Security Token Service, which is used in this recipe. But more importantly, it also contains the necessary certificates, which you'll have to register to get up and running.



Note: these certificates and scripts are also included in this recipe, in the \Setup folder of the sample solutions. If you do not want to install the **Developer Training Kit**, you can use these to set up your system by running the SetupCertificates.cmd file. However, as the Developer Training Kit includes a lot of extra samples, guidelines, and best practices, it's a good idea to completely install it.

You can check if the certificates are installed correctly in IIS: in the main overview, select **Server Certificates**, and ensure the **IdentityTKStsCert** and **localhost** certificates are available and valid:

Name	Issued To	Issued By	Expiration Date	Certificate Hash
WP0367.ATOMIUM.local	ESPSEC01-CA	9/02/2012 9:06:08	87862BF038EAF2AEAE...	
DefaultApplicationCertificate	DefaultApplicationCertificate	16/11/2011 18:05:43	49372ACA17C9A01A9...	
IdentityTKStsCert	IdentityTKStsCert	1/01/2036 4:00:00	40A1D2622BFB DAC80...	
localhost	localhost	1/01/2036 4:00:00	308EFDEE6453FFF68C...	

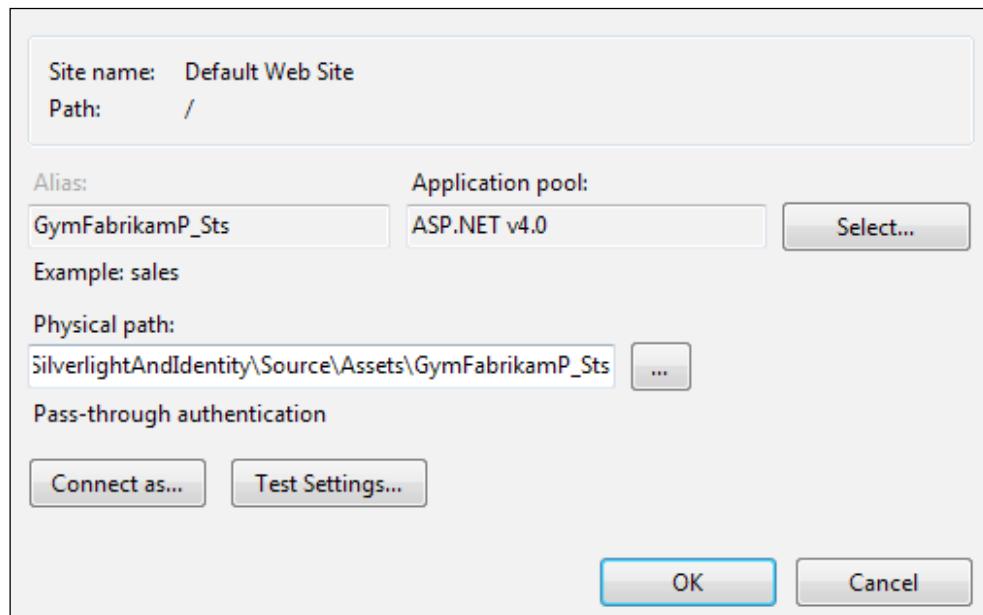
Also, ensure the site binding on your default website is using the **localhost** certificate for https (check this in IIS: select the Default website (or whichever site you're using to host this solution) | **SSL Settings** | **Bindings...** (on the right-hand side)):

A starter solution can be found in Chapter09\Integrating_WIF_Starter. The completed solution can be found in Chapter09\Integrating_WIF_Completed.

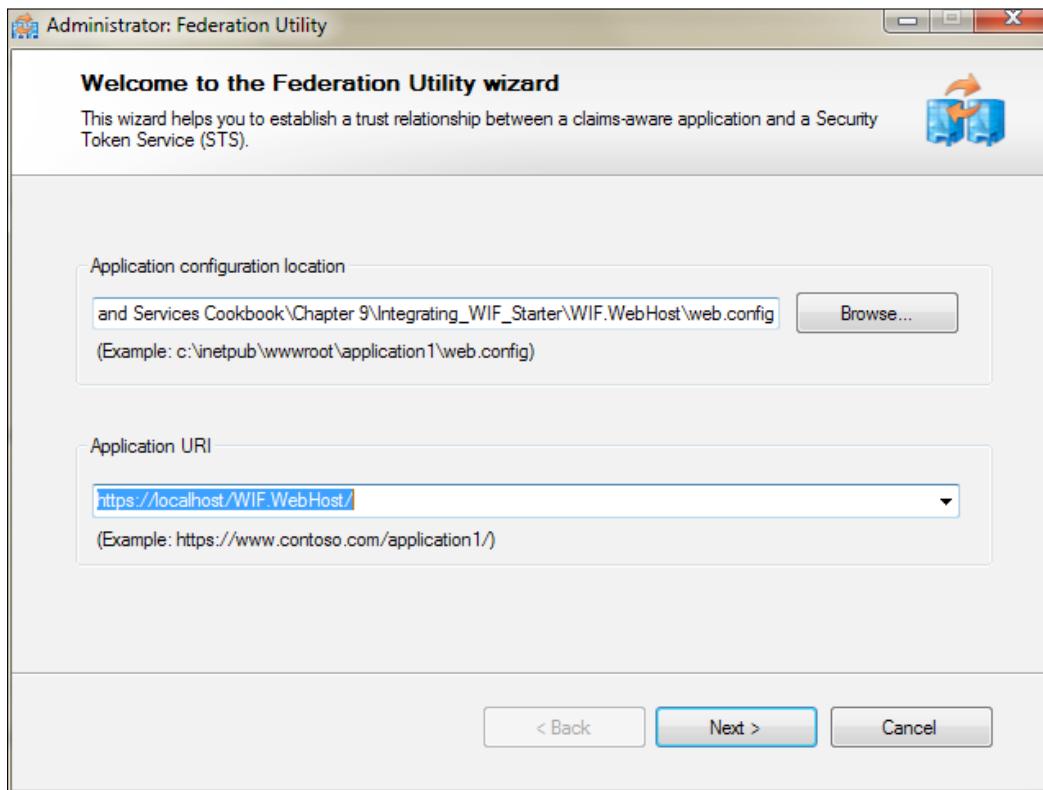
How to do it...

We're going to set up our starter solution to use Windows Identity Foundation for authentication/authorization. In order to do this, we need to complete the following steps:

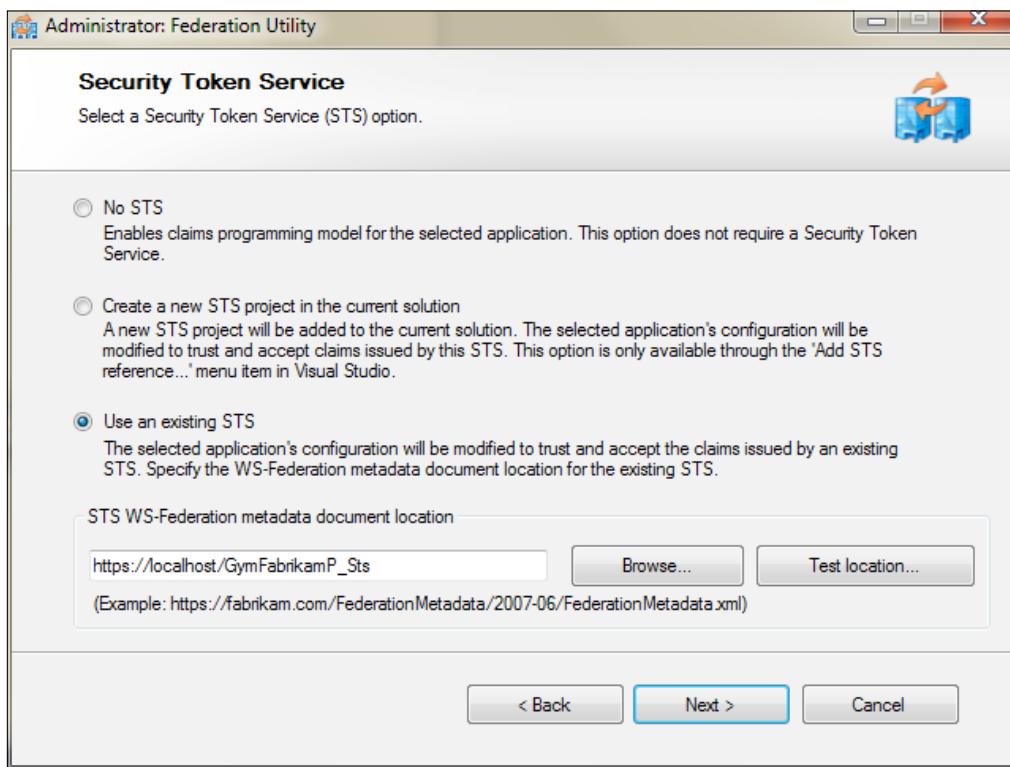
1. First of all, we're using the provided STS (Security Token Service) from the Windows Identity Foundation Training Kit. In case you haven't installed it, the STS has been provided for you with this solution, but you'll have to add it to IIS manually. Add a new application in IIS in the default website, naming it `GymFabrikamP_Sts`, and point it to the location of the provided website. Do this before opening the starter solution:



2. Right-click on **WIF.WebHost**, and select **Add STS Reference....** Set the **Application Uri** to **https://localhost/WIF.WebHost/**:



3. On the next screen, choose to use an existing STS, and locate the Federation metadata. If step one was executed correctly, this should point to **https://localhost/GymFabrikamP_Sts**:



4. Complete the wizard, leaving all the other options on their default.
5. Locate the `system.webServer`-tag in the `web.config` file of the **WIF.WebHost**, and add the following element:

```
<validation validateIntegratedModeConfiguration="false" />
```
6. Include the provided `SampleRequestvalidator.cs` file in **WIF.WebHost**.
7. Locate the `system.web`-tag in the root of the `web.config` file of **WIF.WebHost**, and add the following element:

```
<httpRuntime requestValidationType="SampleRequestValidator" />
```
8. Right-click **WIF.WebHost**, choose **Add... New Item**, and add a new Silverlight-enabled WCF Service, naming it `AuthenticationService.svc`. Delete `AuthenticationService.svc.cs`, open the `svc` file, and change its contents to the following:

```
<%@ ServiceHost Language="C#" Debug="true"
```

```
Factory="SL.IdentityModel.Server.
AuthenticationServiceServiceHostFactory" Service="SL.
IdentityModel.Server.SL.IdentityModel.Server"
%>
```

9. Add a new location-element to the configuration-tag of the web.config file of **WIF.WebHost**:

```
<location path="AuthenticationService.svc">
<system.web>
    <authorization>
        <allow users="*"/>
    </authorization>
</system.web>
</location>
```

10. Open App.xaml in **WIF.Client**, and add the following xmlns directive:

```
xmlns:slid="clr-namespace:SL.IdentityModel.Services;assembly=
SL.IdentityModel"
```

11. Add the following code to App.xaml:

```
<Application.ApplicationLifetimeObjects>
    <slid:ClaimsIdentitySessionManager>
        <slid:ClaimsIdentitySessionManager.IdentityProvider>
            <slid:WSFederationSecurityTokenService />
        </slid:ClaimsIdentitySessionManager.IdentityProvider>
    </slid:ClaimsIdentitySessionManager>
</Application.ApplicationLifetimeObjects>
```

12. Open HomeViewModel.cs, and add the following code:

```
public HomeViewModel()
{
    ClaimsIdentitySessionManager.Current.GetClaimsIdentityComplete
+= new EventHandler<ClaimsIdentityEventArgs>(Current_
GetClaimsIdentityComplete);
}

void Current_GetClaimsIdentityComplete(object sender,
ClaimsIdentityEventArgs e)
{
    if (ClaimsIdentitySessionManager.Current.User.Identity.
IsAuthenticated)
    {
```

```
ClaimsInformation += "Welcome, " +
ClaimsIdentitySessionManager.Current.User.Identity.Name
+ "\n\nThese are the claims provided by the STS:\n\n";

foreach (var item in ClaimsIdentitySessionManager.Current.
User.ClaimsIdentity.Cclaims)
{
    // exclude the picture claim, as this is binary data
    if (item.ClaimType != pictureType)
    {
        ClaimsInformation += item.ClaimType + ":" + item.
Value + "\n";
    }
}
}
```

13. You can now build and run the application. When you log in, you'll see the claims for the logged in user, provided by the STS:

The screenshot shows a web page with a green header bar containing the text "WIF". Below the header, the text "WINDOWS IDENTITY FOUNDATION INTEGRATION" is displayed. Underneath that, it says "Welcome, John". A section titled "These are the claims provided by the STS:" lists several URLs, each representing a claim: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: John", "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/gender: male", "http://gym.fabrikam.com/2010/03/identity/claims/hrest: 62", "http://gym.fabrikam.com/2010/03/identity/claims/maximumlift: 236", and "http://gym.fabrikam.com/2010/03/identity/claims/membershiplevel: gold".

How it works...

The general idea of using a technique like this is that the logic for authentication/authorization isn't the responsibility of the Silverlight developer anymore; the STS (Security Token Service) takes care of this for us, this is the identity provider. The authentication process works **the other way around**, instead of having to write code in your application to call a service for a token, it will receive this from the STS.

In this recipe, we've used the example STS from the Windows Identity Foundation training kit, but any STS can be used, providing it returns a correct token (this token is what is used by the application to decide what to do). While you can write your own STS, it will take some time, and it's generally advised to use an existing, proven, production-ready STS (like **ADFS: Active Directory Federation Services**).

Next to the STS, there are two more projects in this solution that are of importance. By default, WIF classes aren't available in Silverlight, so we need some plumbing to get this up and running. `SL.IdentityModel` contains the claims object model: this is essentially a subset of WIF, built against the Silverlight CLR. The resulting assembly is referenced in the Silverlight client project. `SL.IdentityModel.Server` is referenced in the host project, and it contains logic to trigger authentication when necessary.

After setting up the STS in the first step, the next few steps (2 to 4) are all about adding an STS reference. This is essentially a wizard that helps us generate the necessary settings in the `web.config` file of the webhost: the site hosting our Silverlight application, which we want to secure with WIF. It establishes a trust relation between our webhost and the Security Token Service. It will register the necessary assemblies in the `web.config` file, but the three most important parts are these:

```
<federatedAuthentication>
    <wsFederation passiveRedirectEnabled="true" issuer="https://localhost/GymFabrikamP__STS" realm="https://localhost/WIF.WebHost/"
        requireHttps="true" />
        <cookieHandler requireSsl="true" />
</federatedAuthentication>
```

This tells the application that it's going to use a **Federation Authentication** of type `wsFederation`. It's going to be passive federation, and the issuer of the token is `GymFabrikamP_STS` (the provided Security Token Service). It's going to provide this token to our webhost (**WIFWebHost**).

The next important part is the one in which we define the required claims:

```
<claimTypeRequired>
    <claimType type="http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/name" optional="true" />
        <claimType type="http://schemas.microsoft.com/ws/2008/06/
        identity/claims/role" optional="true" />
</claimTypeRequired>
```

This code is **generated depending on the metadata of our STS**, but it can be changed by editing this, so that we can ensure every user has the claims our application relies on. If, for example, our applications needs the gender-claim of the logged in user, we simply uncomment it (and change the optional attribute), so it becomes one of the required claim types.

And the last important part:

```
<trustedIssuers>
  <add thumbprint="40A1D2622BFBDAC80A38858AD8001E094547369B"
    name="https://localhost/GymFabrikamP_Sts" />
</trustedIssuers>
```

This makes sure our application can only get a token from the GymFabrikamP_Sts—only this STS is trusted (the thumbprint value originates from the certificate we use).

By bringing these three parts together, we've now defined which STS we're going to use for federation, how we're going to do this (passive), and which claims are required by our specific application.

If you would run the application at this point, it would work. What would happen? The site hosting our Silverlight app now uses the STS for authentication. Because our Silverlight app is hosted in this webhost, it is protected as well. However, the claims, sent by the STS in the Security Token (which, if you look back at the wizard used to generate the STS Reference, could be encrypted), are not received by the Silverlight application, but by the hosting website. We need a way to get these claims from the WebHost to the Silverlight application.

That's what happens in the next steps. In step 8, we're adding an AuthenticationService.svc file, changing its contents to:

```
<%@ ServiceHost Language="C#" Debug="true"
  Factory="SL.IdentityModel.Server.
  AuthenticationServiceServiceHostFactory"
  Service="SL.IdentityModel.Server.SL.IdentityModel.Server"
%>
```

This means that the provided SL.IdentityModel.Server contains an implementation of this service, and a Factory to create it. If we have a look at this service implementation (source code is included in SL.IdentityModel.Server), we see the expected SignIn(), SignOut(), and SignInWithIssuedToken(string xmlToken) methods, but also a GetCurrentIdentity() method, which gets the current user from the HttpContext (our IClaimsPrincipal), and returns his list of claims.

In step 9, we ensure this service is accessible for unauthenticated users (as it's the authentication service used for signing in, we need to allow this).

Next, on the client. In step 10 and step 11, we're adding a ClaimsIdentitySessionManager to the Application's ApplicationLifetimeObjects. This class registers an object used for accessing claims from anywhere in the application (much like you would do when you're adding the WebContext when using WCF RIA Services). The IdentityProvider indicates where we expect the claims to originate from—in this case, it's the hosting web application, so we do not need to provide any more information.

The two last steps, 12 and 13, are standard Silverlight code, we're using the current `ClaimsIdentitySessionManager`, adding a callback to be executed when the claims have arrived on the client (remember, everything in Silverlight is `async!`). When we're in the callback, we simply list the received claims, so they're visible on our view.

And like that, we've now got a WIF-enabled Silverlight application.

See also

Have a look at *Chapter 12, Advanced WCF RIA Services*, for an implementation of WIF in combination with WCF RIA Services.

Calling a WCF service from Silverlight using ChannelFactory

Applies to Silverlight 4 and 5

A lot of Silverlight applications using services are typically developed using the **Add Service Reference** dialog box to add service references that generate the client-side proxy classes. However, this isn't sufficient or advisable for some applications or pieces of code. You might want to call variable endpoints, or you might be developing a library or a framework, in which you wouldn't want to have service references for pluggability or for extension purposes (for example, through a provider pattern).

This is where a `ChannelFactory` comes in handy. Through a `ChannelFactory`, you can combine a binding and an address from the code with a contract to open a channel to a service endpoint that implements a matching contract. This can be done without ever using the **Add Service Reference** dialog box.

In this recipe, we'll learn how to achieve this.

Getting ready

We're starting from a simple Silverlight solution that includes a Silverlight project containing our UI and a WCF service in a hosting web application project. If you want to follow along with this recipe, you can use the provided starter solution located in the `Chapter09/UsingChannelFactory_Starter` folder in the code bundle available on the Packt website. The completed solution can be found in the `Chapter09/UsingChannelFactory_Completed` folder.

How to do it...

We're going to modify the starter solution to make sure that we can call WCF service using a ChannelFactory. To achieve this, complete the following steps:

1. We're going to start by adding a new interface called the `IPersonServiceClient` to the Silverlight application. This is an asynchronous representation of the synchronous server-side `IPersonService` contract. Implement it as shown in the following code:

```
[ServiceContract(Name = "IPersonService")]
public interface IPersonServiceClient
{
    [OperationContract(AsyncPattern = true)]
    IAsyncResult BeginGetPerson(int ID, AsyncCallback callback,
        Object state);
    Person EndGetPerson(IAsyncResult result);
}
```

Next, we'll add a `Person` class to the Silverlight application that contains the following code:

```
public class Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string FirstName { get; set; }
}
```

2. Locate the Click handler for `btnFetch` and implement it as shown in the following code:

```
private void btnFetch_Click(object sender, RoutedEventArgs e)
{
    BasicHttpBinding basicHttpBinding = new BasicHttpBinding();
    EndpointAddress endpointAddress = new EndpointAddress
        ("http://localhost:2967/PersonService.svc");
    IPersonServiceClient personService = new ChannelFactory<
        IPersonServiceClient>(basicHttpBinding,
        endpointAddress).CreateChannel();
    var y = personService.BeginGetPerson(Convert.ToInt32(txtID.Text)
        , (asyncResult) =>
    {
        this.Dispatcher.BeginInvoke(delegate
        {
```

```

        var returnedPerson = personService
            .EndGetPerson(asyncResult);
        this.DataContext = returnedPerson;
    });
}
, null);
}

```

3. We can now build and run the application. When a user inputs a value in the textbox and clicks on the **Fetch** button, the GetPerson service method is called using a channel factory. The result can be observed in the following image:



How it works...

We're starting from a solution that includes a WCF Service exposing an endpoint using `basicHttpBinding` and the `IPersonService` service contract. When we want to use a channel factory to access this service, we need to pass in the contract when we create the channel factory. As Silverlight makes its service calls asynchronously, we just can't copy or use the same `IPersonService` service contract, as this doesn't include asynchronous methods. We need to define a new `ServiceContract` that consists of asynchronous method signatures matching the synchronous `GetPerson` method, and that can be matched to the server-side `IPersonService` contract.

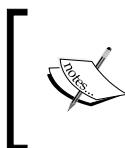
We do this by defining a `BeginGetPerson` method (that accepts an `ID` parameter, callback, and state, returns an `IAsyncResult`, and is decorated with the `[OperationContract(AsyncPattern = true)]` attribute), an `EndGetPerson` method (that accepts an `IAsyncResult` and returns a `Person`), and by setting the contract name as `IPersonService`. This way, we can link our newly created asynchronous contract to its matching synchronous contract, allowing us to create a channel factory using the asynchronous, client-side `IPersonServiceClient` contract to communicate with a service implementing the synchronous `IPersonService` contract.

Next, we've created a `Person` class in the Silverlight application, which is a type returned by the service method. There's one important thing to notice here. The `Person` class in the Silverlight application and the `Person` class in the web application are essentially the same. They exist in the same namespace and have a same name. You can check this by having a look at the `Person` class in the web application. You'll notice that the namespace refers to `UsingChannelFactory` and not to `UsingChannelFactory.Web` as you'd expect. If you don't match these namespaces, the returned object will be null, as .NET does not know how to cast a `UsingChannelFactory.Web.Person` class to a `UsingChannelFactory.Person` class (you could alternatively extend your class to make it implicitly convertible).

The pieces are in place now, so all that's left to do is actually create the channel and calling the service method. This is done in the `Click` handler of the **Fetch** button.

As you can see, the channel is created using the `CreateChannel()` method. Once this is done, we can call the `BeginGetPerson` method and pass in the ID to fetch the correct person. What's important is that we have to make a call to `EndGetPerson` in the callback. The callback will be executed after the service method has run and `EndGetPerson` will return the value the service method returns (in this case a `Person`).

Another important fact to notice is that the callback makes a call to `Dispatcher.BeginInvoke`. We do this because we're accessing the UI thread (setting the `DataContext`) in the callback. If we omit this, we'll get an invalid cross-thread access exception.



A `ChannelFactory` creates and manages the channels used by clients to send messages to service endpoints. It's typically used instead of a proxy when you share a common service contract between client and server.

10

Talking to REST and WCF Data Services

In this chapter, we will cover:

- ▶ Reading data from a REST service
- ▶ Parsing REST results with LINQ To XML
- ▶ Persisting data using a REST service
- ▶ Working with the ClientHttpStack
- ▶ Communicating with a REST service using JSON
- ▶ Using WCF Data Services from Silverlight
- ▶ Reading data from WCF Data Services
- ▶ Persisting data using WCF Data Services
- ▶ Talking to Flickr
- ▶ Talking to Twitter from a non-trusted application
- ▶ Passing credentials and cross-domain access to Twitter from a trusted Silverlight application

Introduction

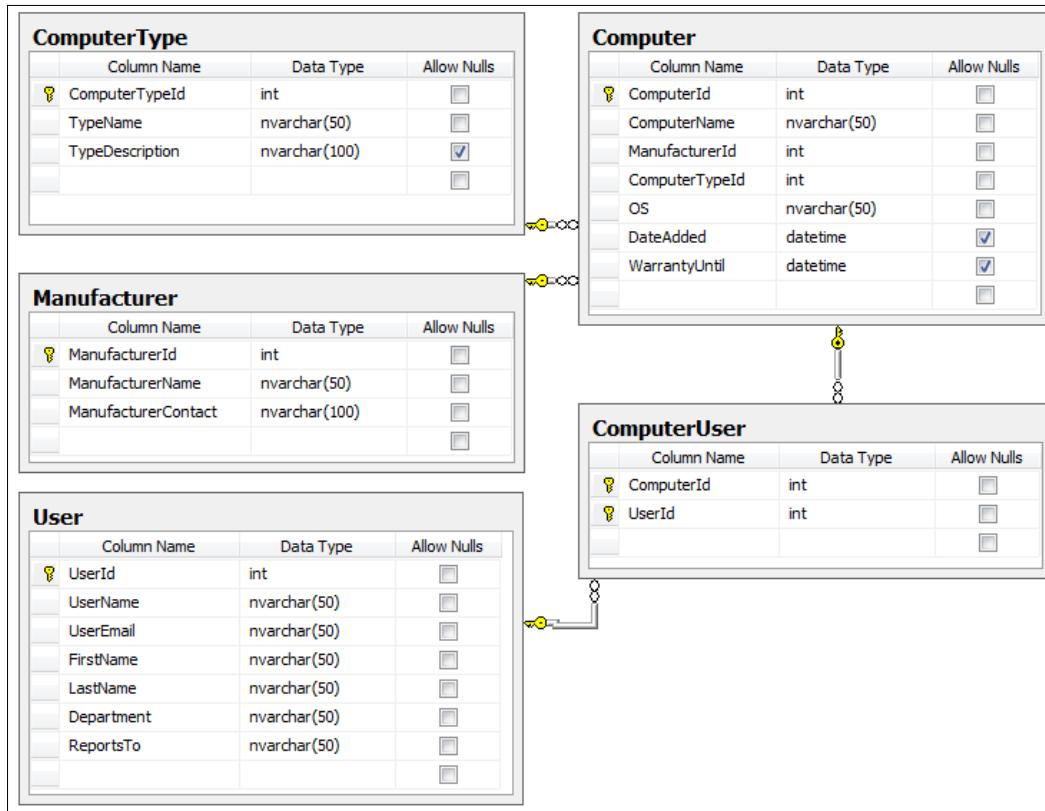
While WCF and ASMX services are very powerful and can address almost every situation, these services might be overkill for some scenarios. Sometimes a simple exchange of textual information, preferably in XML or JSON (JavaScript Object Notation—an easy-to-read data exchange format), might be enough.

The protocol used for this type of communication is REST (Representational State Transfer). Compared to web services (WCF or ASMX), REST has some advantages that can be significant in the case of Silverlight. The exchanged information is human-readable text, mostly in XML format. The XML is clean, meaning there is not a lot of markup being added. SOAP, the format for web services, is also XML, but a lot of extra overhead is added in the so-called SOAP envelope. Using REST will result in less data being sent over the wire, resulting in better performance from a bandwidth perspective. In general, REST is easier to use and entirely platform-independent. It does not require any extra software as it relies on standard HTTP methods.

Are RESTful services (a service that follows REST principles is often referred to as being RESTful) a trend? Definitely! Today, many large web applications, such as Flickr, Facebook, Twitter, YouTube, and so on, offer (part of) their functionality using a RESTful API (a collection of REST services). In .NET, creating RESTful services is fully supported. Moreover, Silverlight can easily connect to REST services.

In this chapter, we'll first look at talking with REST services from Silverlight. Secondly, we'll look at how to work with WCF Data Services (formerly known as ADO.NET Data Services), which are also pure REST services at their base. However, through the use of the client-side library available for use with Silverlight, a lot of plumbing code (necessary to work with RESTful services) is abstracted away, and we get typed access to the entities made available over the service. In other words, it provides a wrapper around REST-based access.

Throughout this chapter, all recipes (except where we use Flickr or Twitter) use the same scenario—the **Computer Inventory** application. This application could be used by an internal IT department of an organization to keep track of PCs, laptops, and so on, as well as by the users registered on a particular system. It consists of two parts—the **User Management**, which we'll build using pure REST services, and the **Computer Management**, which will be built through the use of WCF Data Services. The following image is the schema for the database used. It shows that a Computer is of a certain ComputerType and has a Manufacturer. Each Computer can be registered with one or more User instances:



Reading data from a REST service

Applies to Silverlight 3, 4 and 5

Let's assume that we are writing a Silverlight application that needs to work with data exposed by a RESTful service. The first question that comes to mind is: how can we communicate with such a service and read out the data returned by the service?

This recipe focuses on the communication aspect of REST services, such as how we can connect to a RESTful service from Silverlight and get data into our application.

In this recipe, we'll retrieve a list of all users in the `Users` table using a REST service. For now, we'll show the results in the same format as they are returned, which is plain XML.

Getting ready

The finished solution for this recipe can be found in the Chapter10/TalkingToSimpleRESTServices_ReadingFromRest_Completed folder in the code bundle available on the Packt website. To follow along with this recipe, the starter solution located in the Chapter10/TalkingToSimpleRESTServices_ReadingFromRest_Starter folder can be used.

In this recipe, we're working with a local REST service. The good thing is that building REST services ourselves using WCF is pretty easy. In the sample code, some REST services have already been constructed, such as a service that returns all users (`GetAllUsers`), a service that retrieves a user based on the passed-in user ID (`GetUserById`), and a service that searches for a user based on his/her username (`GetUserByUserName`). These services can be found in the `TalkingToSimpleRESTServices.Services` project in both the starter and the completed solution.

To find out how to create REST services from WCF, please refer to *Appendix A*.

For this sample (as well as the samples of the other recipes in this chapter) to work, we need the `ComputerInventory` database. This database is included both as a **Microsoft SQL Server Database File (MDF)**, file (`CompterInventory.mdf`), and as a `*.sql` script file. Both the files are located in the `Chapter10` folder. For an explanation on how to install them, refer to *Appendix B*.

How to do it...

This recipe will mainly focus on the aspect of communication with REST services. We'll call the RESTful service that returns all users and display the result in its original format—plain XML. The UI of this recipe is kept very basic, containing just enough to trigger a call to the service and show the results, so it won't be in our way while exploring the communication features. The following image shows the application containing a `TextBox` that displays the result of a REST service call, namely an XML string. (Don't worry about any formatting. We'll look at working with XML in the next recipe.):

Computer Inventory - User Management

Controls: [Reload data](#)

```
<ArrayOfUser xmlns:i="http://www.w3.org/2001/XMLSchema-instance"><User><Department>IT</
Department><Email>gill@example.com</Email><FirstName>Gill</
FirstName><LastName>Cleeren</LastName><ReportsTo>Bill Smith</ReportsTo><UserId>1</
UserId><UserName>Gill Cleeren</UserName></User><User><Department>IT</
Department><Email>kevin@example.com</Email><FirstName>Kevin</
FirstName><LastName>Dockx</LastName><ReportsTo>Bill Smith</ReportsTo><UserId>2</
UserId><UserName>Kevin Dockx</UserName></User><User><Department>FINANCE</
Department><Email>john@example.com</Email><FirstName>John</
FirstName><LastName>Smith</LastName><ReportsTo>Lindsey Smith</ReportsTo><UserId>3</
UserId><UserName>John Smith</UserName></User><User><Department>FINANCE</
Department><Email>an@somewhere.com</Email><FirstName>An</
FirstName><LastName>Smith</LastName><ReportsTo>Lindsey Smith</ReportsTo><UserId>4</
UserId><UserName>An Smith</UserName></User><User><Department>IT</
Department><Email>bill@example.com</Email><FirstName>Bill</FirstName><LastName>Smith</
LastName><ReportsTo>CEO</ReportsTo><UserId>6</UserId><UserName>Bill Smith</
UserName></User><User><Department>FINANCE</Department><Email>lindsey@example.com</
Email><FirstName>Lindsey</FirstName><LastName>Smith</LastName><ReportsTo>CEO</
ReportsTo><UserId>7</UserId><UserName>Lindsey Smith</UserName></User></ArrayOfUser>
```

In order to begin reading data from a REST service, we'll need to complete the following steps:

1. Open the solution file in the `Chapter10/TalkingToSimpleRESTServices_ReadingFromRest_Starter` folder. It will open a solution containing a Silverlight application, a hosting website (`TalkingToSimpleRESTServices.Web`), and a website where the REST services are located (`TalkingToSimpleRESTServices.Services`).
2. The easiest way to communicate with a RESTful service is through the use of the `WebClient` class. This class is part of the `System.Net` namespace, which resides in the `System.Net` assembly. If you're still working with Visual Studio 2008 in combination with Silverlight 3, this assembly reference has to be created manually. To do so, right-click on the Silverlight project, select **Add reference**. In the dialog that appears, on the tab titled **.NET**, select **System.Net**. For Silverlight 4 or 5 in Visual Studio 2010, this isn't needed.
3. Let's add some XAML code to `MainPage.xaml` to build the necessary UI for the Silverlight application. We'll add a `Button` that will trigger the call to the service. We'll also add a non-editable `Textbox`, in which the results will be shown as plain text. This can be achieved using the following code:

```
<Grid x:Name="LayoutRoot"
      Background="LightGray">
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="40" />
    <RowDefinition />
  </Grid.RowDefinitions>
```

```
</Grid.RowDefinitions>
<TextBlock x:Name="TitleTextBlock"
           Text="Computer Inventory - User Management"
           FontSize="30"
           FontWeight="Bold"
           HorizontalAlignment="Left"
           Margin="5">
</TextBlock>
<StackPanel Grid.Row="1"
            Orientation="Horizontal" >
    <TextBlock x:Name="ControlsTextBlock"
               Text="Controls: "
               Margin="3"
               VerticalAlignment="Center">
    </TextBlock>
    <Button x:Name="ReLoadButton"
            Content="Reload data"
            Click="ReLoadButton_Click"
            HorizontalAlignment="Left"
            Margin="3"
            VerticalAlignment="Center">
    </Button>
</StackPanel>
<TextBox x:Name="ResultTextBox"
         Grid.Row="2"
         VerticalScrollBarVisibility="Visible"
         TextWrapping="Wrap"
         Width="600"
         IsReadOnly="True">
</TextBox>
</Grid>
```

4. Let's now look at the service that will be called. The contract for this service is located in the TalkingToSimpleRESTServices.Services project in the IUserManagementService.svc.cs file. Calling a RESTful service is nothing more than sending a request to the URI of the service and reading the returned response. In this case, we're sending a request to our own service. In fact, we're sending a request to a method of the service, each method has its own address (a unique URI). The format of this URI is defined by the UriTemplate. For the GetAllUsers method, the value of the UriTemplate is set to userlist as shown in the following code:

```
[OperationContract]
[WebGet(UriTemplate = "userlist",
        BodyStyle = WebMessageBodyStyle.Bare,
```

```
RequestFormat = WebMessageFormat.Xml)]  
List<DTO.User> GetAllUsers();
```

5. In our Silverlight code, we need to match this format. The URI is composed of the base URI (the address of the service itself, assigned to the `serviceBaseUrl` variable in the following code), appended with the `userlist` suffix (defined in the previous code as the value for the `UriTemplate` and assigned to the `getAllUser` variable in the following code). In our case, the complete URI will be `http://localhost:23960/UserManagementService.svc/userlist` (note that the port number, here 23960, may vary on your machine).

```
string serviceBaseUrl =  
    "http://localhost:23960/UserManagementService.svc/";  
string getAllUser = "userlist";
```

Ideally, in real-world applications, this URL would be stored in a configuration file.

6. Now that we have the URI, we need to actually make a call to it. For this, we use the `WebClient` class. In the `Reload` Button's `Click` event handler, we first create an instance of this type. Just like any other service calls, REST service calls are asynchronous. Therefore, we need to register an event handler for the `DownloadStringCompleted` event, which will be called whenever the service returns. Finally, we perform the call by using the `DownloadStringAsync` method, passing in the URI as the parameter. This is shown in the following code:

```
private void ReLoadButton_Click(object sender, RoutedEventArgs e)  
{  
    WebClient client = new WebClient();  
    client.DownloadStringCompleted += new  
        DownloadStringCompletedEventHandler  
        (DownloadAllUsersCompleted);  
    client.DownloadStringAsync(new Uri  
        (serviceBaseUrl + getAllUser, UriKind.Absolute));  
}
```

7. When the service call returns, the event handler defined in the previous step will be called automatically. In this event handler, we have access to the result of the call via the `Result` property on the instance of the `DownloadStringCompletedEventArgs` named `e`. The response is plain XML. Each returned `User` instance is serialized before being sent. If errors have occurred, we can see them here as well. This is shown in the following code:

```
void DownloadAllUsersCompleted(object sender,  
    DownloadStringCompletedEventArgs e)  
{  
    ResultTextBox.Text = e.Result;  
}
```

How it works...

Let's first take a look at some particulars of REST. One of the most important principles in REST is the concept of resources. A resource is a container of information. Each resource can be uniquely identified by a URI. One of the best examples of the REST architecture is the World Wide Web itself. A page is a resource and it has a unique URI to access it.

While SOAP mainly uses the HTTP POST verb, RESTful services use GET, POST, PUT, and DELETE. With the default HTTP stack (also known as `BrowserHttpStack`), Silverlight can work only with GET and POST, because of the limitations of the browser networking APIs it uses internally. In Silverlight 3, a second stack was introduced—the `ClientHttpStack` (we'll be looking at the `ClientHttpStack` in a later recipe in this chapter).

Communicating with REST services differs from communicating with SOAP-based services, as REST services don't expose a WSDL file that contains the functionalities of the service. We can't add a reference to these kind of services in Visual Studio, so there will be no proxy generation and no IntelliSense available.

The solution uses the `WebClient` class that is part of the full .NET framework as well. The `WebClient` class has two important ways of requesting data—`DownloadString` and `OpenRead`. `DownloadString` (which we used in this recipe) can be used when we're reading textual information, such as XML returned by a REST service. `OpenRead` can be used when we want to read the result into a stream. The `WebClient` class defines a pair of an asynchronous method and a `Completed` event for both these ways of requesting data. This `Completed` event is fired on the UI thread, which means that to update UI elements in the event handler, we can do so directly and don't have to cross threads.

Instead of using the `WebClient` class, we can also use the `HttpWebRequest` class. This class should be our choice if we need more control over the call to the service. The `WebClient` class uses the `HttpWebRequest` class internally. We looked at working with `HttpWebRequest` in the *Reading XML using HttpWebRequest* recipe in Chapter 7, *Working with Services*.

Calling REST services is possible only asynchronously, because that is the only way Silverlight allows calls to be made. This asynchronous behavior is reflected in both the actual call to the service (`DownloadStringAsync`) and the registration of the event handler (`DownloadStringCompleted`), which is called whenever the service returns.

Communication with REST services can be summarized as a three-step process:

- ▶ Create a URI to which a request needs to be sent
- ▶ Send the request
- ▶ Get in the results and work with them (parsing and so on)

The format of the URI is defined by the service itself. Each URI corresponds to a specific method that will return data. The actual sending of a request is done in the `DownloadStringAsync` method of the `WebClient` class. When the service returns, the callback is invoked and the response is available through the `Result` property of `DownloadStringCompletedEventArgs`.

See also

In the next recipe, we're going to work with the results of the service through the use of LINQ To XML. The asynchronous way of working with services in general is explained in the *Connecting and reading from a standardized service* recipe in *Chapter 7, Working with Services*.

Parsing REST results with LINQ To XML

Applies to Silverlight 3, 4 and 5

We have successfully connected to a REST service from a Silverlight application in the previous recipe. The response from the service is XML. Most of the time, showing pure XML to the end user is not the goal of an application, so we'll want to parse the XML. Silverlight contains several options to work with XML, which include **XmIReader/XmIWriter**, **XmISerializer**, and **LINQ To XML** (also known as **XLinq**). The latter is a preferred way to parse XML.

In this recipe, we'll look at how we can use LINQ To XML to transform XML into real data. The raw user data (originally in XML) will be transformed in `User` objects.

Getting ready

This recipe builds on the code created in the previous recipe, so you can continue using that solution. Alternatively, you can use the starter solution for this recipe located in the `Chapter10/TalkingToSimpleRESTServices_LinqToXml_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter10/TalkingToSimpleRESTServices_LinqToXml_Completed` folder.

How to do it...

In this recipe, we'll transform the plain XML returned by a RESTful service into real, meaningful data. The XML will be parsed using LINQ To XML. Without a doubt, LINQ-To-XML is the easiest and most efficient way for this task. To begin parsing the XML, we'll complete the following steps:

1. Either continue working on the solution created in the previous recipe, or use the provided solution as outlined in the *Getting ready* section.

-
2. The assembly needed to use LINQ To XML in Silverlight applications is not added by default. Therefore, we need to add a reference to the `System.Xml.Linq` assembly in the Silverlight project. The basic features of LINQ, such as the `select` statement, live in the `System.Linq` assembly that is added by default. (Note that the `System.Xml.Linq` assembly is about 120KB in file size.)
 3. As Visual Studio can't create a proxy for a REST service, we don't get types to work with on the client side, although this would be a lot easier. Therefore, we'll manually create a `User` class ourselves in the `TalkingToSimpleRESTServices` Silverlight project that will contain the object representation of the XML data. This is a data-only type. The `User` class is shown in the following code:

```
public class User
{
    public int UserId { get; set; }
    public string UserName { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Department { get; set; }
    public string ReportsTo { get; set; }
}
```

4. Next, in the `DownloadAllUsersCompleted` callback method located in the code-behind of `MainPage.xaml`, we'll need to load the XML into an `XDocument` using the `Parse` method. An `XDocument` is able to load in the entire XML stream given to it. With a query, we search for all `User` descendants of the root node using the `Descendants` method. As we don't want to work with the `XElement` instances in our client code, we read each `User` `XElement` and load its values into a new instance of the `User` class. Note that we can use the `Element` or `Descendants` methods. Both methods have the same result. This is shown in the following code:

```
void DownloadAllUsersCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    XDocument xml = XDocument.Parse(e.Result);
    var users = from results in xml.Descendants("User")
                select new User
    {
        UserId = Int32.Parse(results.Element("UserId")
            .Value.ToString()),
        UserName = results.Descendants("UserName").First().Value,
        FirstName = results.Descendants("FirstName").First().Value,
        LastName = results.Descendants("LastName").First().Value,
        Department = results.Element("Department").Value.ToString(),
        Email = results.Element("Email").Value.ToString(),
        ReportsTo = results.Element("ReportsTo").V    };
}
```

5. We will then need to replace the `ResultTextBox` in `MainPage.xaml` with a `DataGrid` called `UsersDataGrid`. The code for this control is as follows:

```
<UserControl xmlns:data="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls.Data"
    x:Class="TalkingToSimpleRESTServices.MainPage"

    <data:DataGrid x:Name="UsersDataGrid"
        Grid.Row="2"
        AutoGenerateColumns="False"
        Width="600"
        Height="500"
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="3">
        <data:DataGrid.Columns>
            <data:DataGridTextColumn Binding="{Binding UserId}"
                Header="UserId"/>
            <data:DataGridTextColumn Binding="{Binding UserName}"
                Header="User name"/>
            <data:DataGridTextColumn Binding="{Binding FirstName}"
                Header="First name"/>
            <data:DataGridTextColumn Binding="{Binding LastName}"
                Header="Last name"/>
        </data:DataGrid.Columns>
    </data:DataGrid>
```

6. Finally, we can use the data in our application. We can now bind the generic `List<User>` to the `DataGrid` by setting it as the value of the `ItemsSource` property. This is shown in the following code:

```
void DownloadAllUsersCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    UsersDataGrid.ItemsSource = users.ToList();
}
```

7. The following screenshot shows the User instances bound to the DataGrid:

UserId	User name	First name	Last name
1	Gill Cleeren	Gill	Cleeren
2	Kevin Dockx	Kevin	Dockx
3	John Smith	John	Smith
4	An Smith	An	Smith
6	Bill Smith	Bill	Smith
7	Lindsey Smith	Lindsey	Smith

How it works...

When working with data coming from a RESTful service, most of the time it's important to look at the schema of the XML. Here, the data is quite simple as it's created through serialization of an object on the server side. Serialization is the process of converting an object into a stream, so that it can be easily sent over the wire. In our case, we are serializing instances of a class called `User` that is located in the `TalkingToSimpleRestServices.DTO` project. Each property of this class is translated into XML as shown in the following code:

```
<ArrayOfUser>
  <User>
    <Department />
    <Email />
    <FirstName />
    <LastName />
    <ReportsTo />
    <UserId />
    <UserName />
  </User>
</ArrayOfUser>
```

While RESTful services may respond with more complicated XML code, LINQ To XML contains everything needed to parse the data easily. We should always start by loading the entire XML string into an `XDocument` or an `XElement`. `XElement` may even be a better fit here, as we're not using any particularities of the root node. Using the `Descendants` method and passing in the name of the node that we want to retrieve, we get a list of all the `XElement` instances matching the requested pattern. As this is a list, we can perform a query on it. In this query, for each encountered `XElement`, we create a new `User` instance by passing in the retrieved values of the XML.

With this, we have successfully loaded data from a REST service into the types on the client side. This data can now be used in all scenarios we want, for example, data binding.

See also

The previous recipe explains how to get the XML data into the application. In the next recipe, we explore the options to send data from Silverlight to a REST service. In the *Communicating with a REST service using JSON* recipe, we'll look at how we can work with a REST service that returns JSON.

We used very simple data binding here, but if you'd like to explore this topic further, refer to *Chapter 2, An Introduction to Data Binding*, and *Chapter 3, Advanced Data Binding*.

Persisting data using a REST service

Applies to Silverlight 3, 4 and 5

Some REST services accept data that we send to them as well, so this data can then be persisted back into a database. In this recipe, we'll make it possible to add, update, or delete a user in the Computer Inventory application, where we're working on the User Management.

Getting ready

This recipe builds on the code created in the previous two recipes. If you want to follow along with the steps in this recipe, you can also use the starter solution located in the Chapter10/TalkingToSimpleRESTServices_PersistingData_Starter folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the Chapter10/TalkingToSimpleRESTServices_PersistingData_Completed folder.

How to do it...

Persisting data to a REST service is actually the opposite of reading. We'll use the same class, namely the WebClient class. However, instead of downloading data, we'll serialize client-side data and send it back to the service. To begin persisting data to the REST service, we'll complete the following steps:

1. As outlined in the *Getting ready* section, use either the solution from the previous recipe or the provided solution in the sample code.
2. We'll be using the WebClient class that resides in the System.Net namespace. If not yet added, add a reference to this assembly in your Silverlight project. To do so, right-click on the TalkingToSimpleRESTServices project, select **Add reference**, and select the required assembly in the dialog box that appears. Visual Studio 2010 creates this assembly reference automatically.

-
3. In this recipe, we'll use a detail window to add, update, or delete a user. The following is the XAML code for this user control. This code is placed inside a new Silverlight child window. To add a child window to the project, right-click on the Silverlight project node in the **Solution Explorer**, select **Add | New item...**, and select **Silverlight Child Window** in the template selection dialog box. Name this new file `UserDetailEdit`. Such a child window contains out of the box zoom-in or zoom-out effects, when initiated or closed, respectively. Note that we're going to use data binding to show a `User` instance, or to get the changes back into the object when the values have changed. **TwoWay** bindings are used, so that the bound CLR object will update automatically as well. The complete XAML code for this child window can be found in the code bundle. The following code shows the most relevant parts:

```
<Grid x:Name="LayoutRoot" Margin="2">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid Grid.Row="0" x:Name="UserDetailGrid" >
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            ...
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            ...
        </Grid.ColumnDefinitions>
        <TextBlock Text="User ID: " 
            Grid.Row="0"
            Grid.Column="0"
            VerticalAlignment="Top">
        </TextBlock>
        <TextBlock x:Name="UserIdTextBlock"
            Grid.Row="0"
            Grid.Column="1"
            Text="{Binding UserId}"
            VerticalAlignment="Top">
        </TextBlock>
        <TextBlock Text="User name: "
            Grid.Row="1"
            Grid.Column="0"
            VerticalAlignment="Top">
        </TextBlock>
        <TextBox x:Name="UserNameTextBox"
            Grid.Row="1"
            Grid.Column="1"
            Text="{Binding UserName, Mode=TwoWay}">
        </TextBox>
    </Grid>
</Grid>
```

```
    VerticalAlignment="Top">
  </TextBox>
  <!-- Similar code omitted-->
</Grid>
<Button x:Name="DeleteButton"
        Content="Delete"
        Click="DeleteButton_Click"
        Width="75"
        Height="23"
        HorizontalAlignment="Right"
        Margin="0,12,79,0"
        Grid.Row="1" />
<Button x:Name="CancelButton"
        Content="Cancel"
        Click="CancelButton_Click"
        Width="75"
        Height="23"
        HorizontalAlignment="Right"
        Margin="0,12,0,0"
        Grid.Row="1" />
<Button x:Name="SaveButton"
        Content="Save"
        Click="SaveButton_Click"
        Width="75"
        Height="23"
        HorizontalAlignment="Right"
        Margin="0,12,158,0"
        Grid.Row="1" />
</Grid>
```

4. Similar to reading from a REST service, we need a URI to send a request, as dictated by the service itself. Each action (add, update, or delete) has a different address. We'll combine these specific addresses with the base address of the service to get the correct URI based on the required action. This is shown in the following code:

```
string serviceBaseUrl =
  "http://localhost:23960/UserManagementService.svc/";
string getUserId = "user/{0}";
string addUser = "user/add";
string updateUser = "user/update";
string deleteUser = "user/delete";
```

Note that the port number (here 23960) may be different on your machine.

5. Let's now look at the actions required to add a new User. We first need to change the client-side User class in the Silverlight project. The class itself needs to be decorated with a `DataContract` attribute, and the members that we want to send over need a `DataMember` attribute. The updated class is shown in the following code. Note that we define the Namespace to be empty, since our code is not using the namespaces from the returned XML:

```
[DataContract(Name = "User", Namespace = "")]  
public class User  
{  
    [DataMember]  
    public int UserId { get; set; }  
    [DataMember]  
    public string UserName { get; set; }  
    [DataMember]  
    public string FirstName { get; set; }  
    [DataMember]  
    public string LastName { get; set; }  
    [DataMember]  
    public string Email { get; set; }  
    [DataMember]  
    public string Department { get; set; }  
    [DataMember]  
    public string ReportsTo { get; set; }  
}
```

6. Upon constructing the `UserDetailEdit` instance, we can check which action the window is supposed to be performing. This can be either editing an existing User or adding a new User. These actions are reflected in a new enumeration called `EditingModes` that we add to the Silverlight project. This is shown in the following code:

```
public enum EditingModes  
{  
    New,  
    Edit  
}
```

7. This enumeration is now used as a parameter type in the constructor. When we add a User, a new instance is created and is set as the value for the `DataContext` property of the `UserDetailGrid`:

```
private User user;  
private int userId;  
private EditingModes editingMode;  
public UserDetailEdit(int userId, EditingModes editingMode)  
{
```

```
InitializeComponent();
this.userId = userId;
this.editingMode = editingMode;
if (editingMode == EditingModes.New)
{
    user = new User();
    UserDetailGrid.DataContext = user;
    DeleteButton.IsEnabled = false;
}
}
```

8. When the user clicks on the Save button, we need to send the User instance to the RESTful service. However, this can be done only after serializing the object. This can be done through the use of the `DataContractSerializer` type as shown in the following code:

```
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    Uri uri = new Uri(serviceBaseUrl + addUser);
    DataContractSerializer dataContractSerializer = new
        DataContractSerializer(typeof(User));
    MemoryStream memoryStream = new MemoryStream();
    dataContractSerializer.WriteObject(memoryStream, user);
    string xmlData = Encoding.UTF8.GetString(memoryStream.ToArray(),
        0, (int)memoryStream.Length);
}
```

9. Now that we have the XML available, we need to send it. This will be done through the use of the `WebClient`, but instead of using the `DownloadString` method, we'll use the `UploadString` method. It's required to set the content-type. It should be set to `application/xml`, as shown in the following code. Also, in the `UploadStringAsync` method, we're using `POST` as the method for the HTTP request and are adding data:

```
client.UploadStringCompleted += new
    UploadStringCompletedEventHandler(UploadCompleted);
client.Headers[HttpRequestHeader.ContentType] = "application/xml";
client.UploadStringAsync(uri, "POST", xmlData);
```

10. In the `UploadCompleted` event handler for the callback, we can check if the upload went well using the `Error` property of the `UploadStringCompletedEventArgs` event arguments. This is shown in the following code:

```
private void UploadCompleted(object sender,
    UploadStringCompletedEventArgs e)
{
    if (e.Error == null)
```

```

        this.DialogResult = true;
    else
        MessageBox.Show(e.Error.Message);
}

```

- At this point, the child window is ready. The only thing left to do is to call it from MainPage.xaml. To do so, start by adding a new Button called NewUserButton in the StackPanel within MainPage.xaml. This is shown in the following code:

```

<Button x:Name="NewUserButton"
        Content="Add user"
        Click="NewUserButton_Click"
        HorizontalAlignment="Left"
        Margin="3"
        VerticalAlignment="Center">
</Button>

```

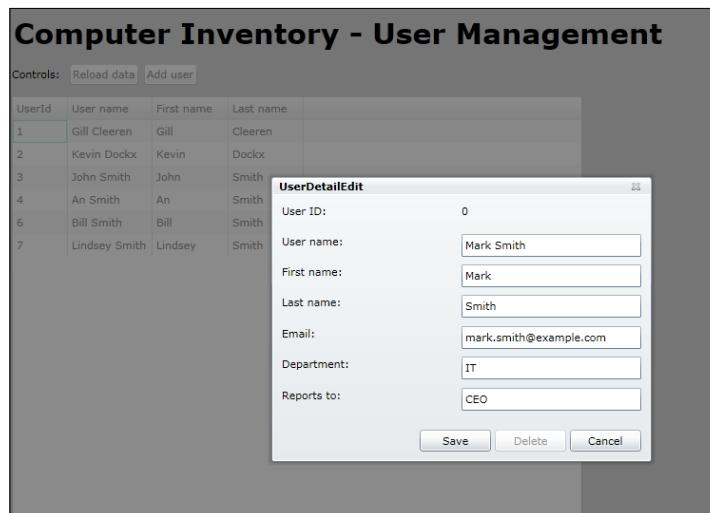
- In the Click event handler, we instantiate the UserDetail child window as shown in the following code:

```

private void NewUserButton_Click(object sender, RoutedEventArgs e)
{
    UserDetailEdit editView = new
        UserDetailEdit(0, EditingModes.New);
    editView.Show();
}

```

With that, we've created all the necessary code to allow the persisting of User instances over the REST service. In the following screenshot, the child window is shown in its New editing mode:



Updating and deleting `User` instances are similar. The sample code contains all the logic for these actions as well.

How it works...

When persisting data to a RESTful service, the first concern is getting the data on the service. Data is almost always available on the client side in the form of objects. We can't just go sending the objects straight away; they have to be serialized first. For the service and the client to understand one another, the XML should be in a correct format. Hence, the `DataContract` and the `DataMember` attributes are used in the `User` class on the client side. The client-side and the server-side objects must have the same names for their properties, otherwise the deserialization will fail.

The process of serialization from and to XML has often been the job of the `XmlSerializer` class and it has been included since .NET 1.0. When WCF arrived, a new serializer called the `DataContractSerializer` was included, in the first place intended for use with WCF. However, as seen in this recipe, it can be used for any serialization purpose.

Since version 3, Silverlight has contained two network stacks—the browser stack used in this sample and the `ClientHttpStack`. The browser stack is named so because Silverlight internally uses the browser networking APIs. Through this stack, only HTTP GET and POST are supported. In the sample code, you can see that we use a POST request to perform an add, an update, or a delete. Another way of using PUT and DELETE is through POST, by passing in the real HTTP value as a custom header. This technique isn't perfectly RESTful either, because the request method and what we want to achieve don't match, which is one of the aspects of the REST principle. This technique is also used in WCF Data Services. The `ClientHttpStack` does allow Silverlight to use real PUT or DELETE messages.

There's more...

For the serialization process, we could have used the `XmlSerializer` class. While this class also does the job and is included in Silverlight, the `DataContractSerializer` is easier to use (as you have more control over the namespace, and so on). The sample code also contains some code where the `XmlSerializer` class is used.

See also

Reading and persisting data using REST services is very similar. Read both the *Reading data from a REST service* and *Parsing REST results with LINQ To XML* recipes in this chapter and notice the link between the two.

Working with the ClientHttp stack

Applies to Silverlight 3, 4 and 5

When communicating with a REST service, Silverlight uses the `BrowserHttpStack` by default. Due to this, Silverlight can't use all the HTTP verbs, such as `PUT` and `DELETE`. Silverlight 3 added a new option, namely the `ClientHttp` stack. This new stack bypasses the browser stack, and performs its communication directly through the operating system.

In this recipe, we'll look at the changes that we need to make to use this networking stack.

Getting ready

To follow along with this recipe, you can use the code created in the previous recipe. Alternatively, you can use the starter solution located in the `Chapter10/TalkingToSimpleRESTServices_ClientHttp_Starter` folder in the code bundle available on the Packt website. The completed solution can be found in the `Chapter10/TalkingToSimpleRESTServices_ClientHttp_Completed` folder.

How to do it...

To make a Silverlight application that talks with REST services, use the `ClientHttp` stack instead of the browser stack. We need to perform a few simple steps. We'll use the application built in the previous recipes (`Computer Inventory`) to use the new stack. Let's take a look at what we need to do:

1. To make an application use the `ClientHttp` stack, and we need to tell Silverlight to do so. The easiest way is telling Silverlight that all traffic for addresses beginning with `http://` has to use this stack. This can be done using the following code:

```
public MainPage()
{
    InitializeComponent();
    HttpWebRequest.RegisterPrefix("http://",
        WebRequestCreator.ClientHttp);
}
```

2. With the previous code executed, all calls will be executed over the `ClientHttp` Stack.

How it works...

The REST protocol specifies that we can identify any resource with a unique URL. This resource can be any information on the web, for example, a user instance in the application. Using REST, we can get this user with the `GET` command, create or update the user using the `PUT` command, use the `POST` command to create a new instance, delete the user using the `DELETE` command, and so on.

Silverlight supports communication with REST services, but as it works by default through the browser stack, it's limited to use only `GET` and `POST`. With Silverlight 3, a new stack was introduced, namely the `ClientHttp` stack.

Working with this new stack requires almost no changes to the existing applications, as the API is identical. The only thing we need to do is let Silverlight know that we want to use this stack. This can be done by saying that all requests starting with `http://` should use the `ClientHttp` stack. This is shown in the following line of code:

```
HttpWebRequest.RegisterPrefix("http://",
    WebRequestCreator.ClientHttp);
```

If we have requests over `HTTPs`s and want these to happen over the client stack as well, we need to register them using the following line of code:

```
HttpWebRequest.RegisterPrefix("https://",
    WebRequestCreator.ClientHttp);
```

We can also be more specific. For example, assume we have an application that communicates with `http://www.snowball.be` and `http://www.packtpub.com`. If we want the REST communication with `http://www.snowball.be` to go over the `ClientHttpStack` and `http://www.packtpub.com` to use the default browser stack, we can specify this using the following code:

```
HttpWebRequest.RegisterPrefix("http://www.snowball.be",
    WebRequestCreator.ClientHttp);
```

Advantages of the ClientHttp stack

Using the `ClientHttp` stack has some advantages over using the browser stack. As already mentioned, it supports more HTTP verbs (`GET`, `POST`, `PUT`, and `DELETE`). It does not support other HTTP verbs, such as `CONNECT`, `TRACE`, and so on. However, the service can be limited in the keywords it supports. It's possible to specify in the client access policy file (`clientaccesspolicy.xml`) which verbs are supported and which aren't. The error messages when using browser stack are limited. With this stack, we have access to only 200 and 404. The `ClientHttp` stack supports all error messages, making it easier to see what's wrong with the service.

Starting with Silverlight 4, it's also possible to perform authentication using the ClientHttp stack (we'll look at this in the *Passing credentials and cross-domain access to Twitter from a trusted Silverlight application* recipe, later in this chapter). In Silverlight 5, there were no changes in this area.

When we download an image with BrowserHttpStack, it's automatically cached by the browser, as it is its default behavior. However, when working with the ClientHttp stack, the browser won't cache it; it simply won't see the image passing by. In Silverlight 3, there was no option to cache using the ClientHttp stack. With Silverlight 4, support for caching was added.

The same goes for cookies. When using the browser stack, all cookies coming in from a site or going out to a site are managed by the browser. Due to this, when we're logged in to a site based on cookies (the way ASP.NET works), the requests made to that same site from a Silverlight application are also authenticated. With the ClientHttp stack, again this won't work. With the ClientHttp stack, we can work with cookies, but this has to be done manually using the `CookieContainer`.

See also

In the previous recipes of this chapter, we looked at the specifics of working with REST services from Silverlight.

Communicating with a REST service using JSON

Applies to Silverlight 3, 4 and 5

When we work with REST services, data is sent over the wire in XML by default. However, REST services can also send back their information in another format, such as **JavaScript Object Notation (JSON)**. This can be the case if the service needs to be accessible from JavaScript code, or if the transferred data is to be very compact.

In this recipe, we'll look at how to communicate from Silverlight with a REST service in the JSON format.

Getting ready

This recipe builds on the code created in the previous recipes, so you can continue using your own code for this recipe. Alternatively, you can also use the provided starter solution located in the `Chapter10/TalkingToSimpleRESTServices_ReadingWithJSON_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter10/TalkingToSimpleRESTServices_ReadingWithJSON_Completed` folder.

How to do it...

Communicating with a REST service using JSON data is a matter of changing the format of the data sent over the wire and parsing this data using a `JsonArray` object. To do this, we have to complete the following steps:

1. Open the solution as outlined in the *Getting ready* section and locate the project containing your services called `TalkingToSimpleRESTServices.Services`. In this project, find the `IUserManagementService` interface and add the following code to it. Notice that the `RequestFormat` and the `ResponseFormat` `NamedParameters` are set to `Json`, as shown in the following code:

```
[OperationContract]
[WebGet(UriTemplate = "userlistjson",
    BodyStyle = WebMessageBodyStyle.Bare,
    RequestFormat = WebMessageFormat.Json),
    ResponseFormat = WebMessageFormat.Json)]
List<DTO.User> GetAllUsersJson();
```

2. We can now implement this method. To do so, add the following code to the `UserManagementService` class:

```
public List<DTO.User> GetAllUsersJson()
{
    List<DTO.User> dtoUserList = new List<DTO.User>();
    List<User> userList = new UserRepository().GetAllUsers();
    foreach (var user in userList)
    {
        DTO.User dtoUser = ConvertUserToDTOUser(user);
        dtoUserList.Add(dtoUser);
    }
    return dtoUserList;
}
```

3. In the Silverlight project, we need to add a reference to `System.Json`.
4. We'll now try to retrieve all users using JSON instead of XML. In the UI, add a new `Button` to the `StackPanel` as shown in the following code:

```
<Button x:Name="JsonButton"
    Content="Load data using JSON"
    Click="JsonButton_Click"
    HorizontalAlignment="Left"
    Margin="3"
    VerticalAlignment="Center">
</Button>
```

5. In the event handler of this Button, we can perform a call to the GetAllUsersJson method using the following code:

```
private void JsonButton_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.OpenReadCompleted += new
        OpenReadCompletedEventHandler(client_OpenReadCompleted);
    client.OpenReadAsync(new Uri(serviceBaseUrl + "userlistjson",
        UriKind.Absolute));
}
```

6. Add the following code to handle the OpenReadCompleted event of our JSON request. In this method, we're parsing the result of the request.

```
void client_OpenReadCompleted(object sender,
    OpenReadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        JSONArray items = (JSONArray)JSONArray.Load(e.Result);
        var query = from user in items
                    select new User
        {
            Department = user["Department"],
            Email = user["Email"],
            FirstName = user["FirstName"],
            LastName = user["LastName"],
            ReportsTo = user["ReportsTo"],
            UserId = user["UserId"],
            UserName = user["UserName"]
        };
        UsersDataGrid.ItemsSource = query.ToList();
    }
}
```

7. Build and run your application. If we place a breakpoint in the returning method, we can see that the data is effectively returned in the JSON format.

How it works...

By setting RequestFormat and ResponseFormat NamedParameters to WebMessageFormat.Json in our OperationContract, we're telling our service that it should use JSON as the data format while transferring data for both requests and responses. Whenever we send or receive data using OperationContract, everything is done using JSON.

To easily parse this result, Silverlight includes classes to easily handle JSON data. They're located in the `System.Json` namespace. By calling `JsonArray.Load` in the response stream, we can load the response into a `JsonArray` object. This represents a collection of `JsonValue`. In this example, each `JsonValue` is a `User`, so all that's left to do is convert these items into `User` objects and set the `ItemsSource` collection of the `DataGrid`.

See also

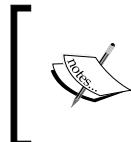
To get data in the XML format rather than the JSON format, have a look at the *Reading data from a REST service* recipe in this chapter.

Using WCF Data Services with Silverlight

Applies to Silverlight 3, 4 and 5

Above our data layer, we may have an entity model that exposes entities (for example, one which is created using the **ADO.NET Entity Framework**) for our application to use. **WCF Data Services** allows exposing these entities over REST-based services. In this recipe, we'll look at how we can use WCF Data Services from Silverlight.

In the previous recipes, we worked on the User Management part of the Computer Inventory application. In this and the following two recipes, we'll work on the Computer Management.



Before being officially released, WCF Data Services were known as **ADO.NET Data Services**. When .NET 4 was released, Microsoft changed the name to WCF Data Services, mainly to show that this technology too is based on the WCF stack.

Getting ready

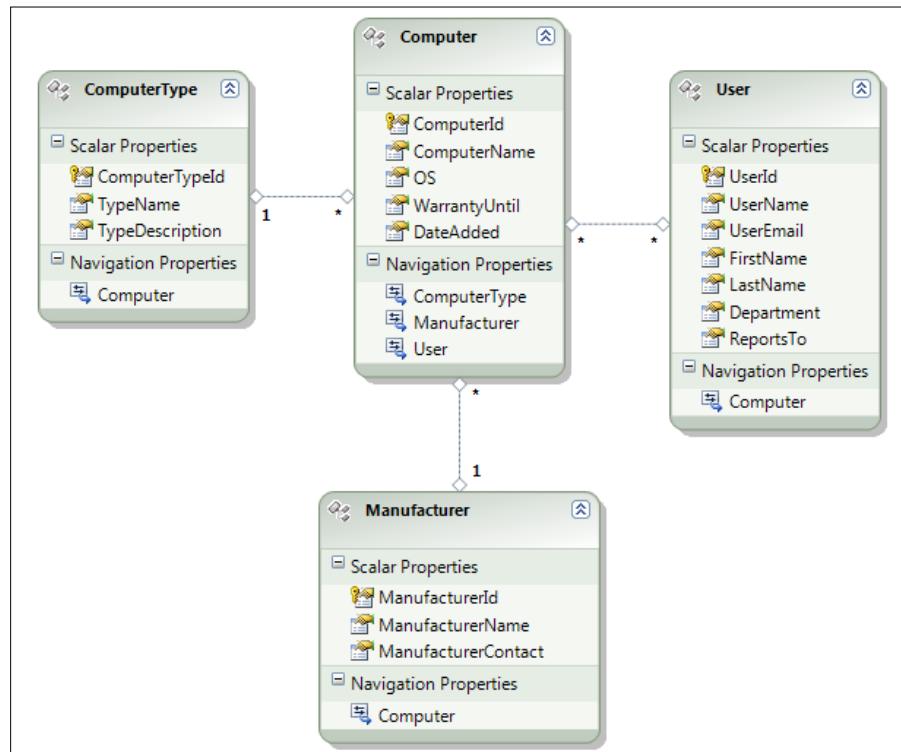
This recipe, along with the following two recipes, uses the same database called `ComputerInventory`, as used in the RESTful services recipes. This database is included as a Microsoft SQL Server Database File (MDF) in the code bundle available on the Packt website. Refer to *Appendix B* for installation instructions.

This recipe starts from an empty Silverlight application. Refer to *Chapter 1, Learning the Nuts and Bolts of Silverlight 5*, for more information on how to do so.

How to do it...

We'll first set up WCF Data Services and then build a model using Entity Framework. In the following recipes, we'll connect to these services from a Silverlight client application. The following are the steps we need to perform to get this working:

1. We'll build the entire application from scratch. Create a new Silverlight solution and select **ASP.NET Web Application** as the type for the hosting website. The latter is needed to create the model and host the service. If you have an existing Silverlight solution to which you want to add a WCF Data Service, you can add the model in the hosting web application.
2. WCF Data Services work on a model, not directly on a database. Add a new Entity Framework Model by right-clicking on the web project, selecting **Add | New Item...**, and selecting **ADO.NET Entity Data Model**. Name the model ComputerInventory.edmx.
3. In the wizard that appears, select **Generate from Database** in the first dialog box. This indicates that we want to start creating the model based on the tables in the database.
4. The next step allows us to configure the connection to the database by clicking on the **New Connection** button. Leave the checkbox checked to allow storing the connection string in the web.config file.
5. The final step in the wizard allows us to select which items from the database we want to make as the part of the model. Select all tables, excluding sysdiagrams. When we click on **Finish**, Visual Studio generates the model as shown in the following image:



6. Next, we create the actual WCF Data Service. To do so, add a WCF Data Service called ComputerInventoryService.svc to your web project. The generated code needs some changes done to it. A link needs to be created between the model and the data service by making the latter inherit from DataService<ComputerInventoryEntities>. The ComputerInventoryEntities type parameter is often referred to as the context or context object representing the model.
7. In the InitializeService method, we need to explicitly allow access-specific entities by using the EntitySetRights enumeration. In the following code, we are saying that all rights are allowed on the specified entities:

```
public static void InitializeService(DataServiceConfiguration config)
{
    config.UseVerboseErrors = true;
    config.SetEntitySetAccessRule("Computer", EntitySetRights.All);
    config.SetEntitySetAccessRule("User", EntitySetRights.All);
    config.SetEntitySetAccessRule("ComputerType",
        EntitySetRights.All);
    config.SetEntitySetAccessRule("Manufacturer",
        EntitySetRights.All);
}
```

8. Go to the Silverlight application and add a service reference to the *.svc file by right-clicking on the Silverlight application and selecting **Add Service Reference....**. Then click on the **Discover** button in the **Add Service Reference** dialog box. Change the namespace to ComputerInventoryService. Visual Studio will now generate a proxy for this service, and a reference to System.Data.Services.Client will be automatically added. You now have typed access to the entities exposed by the service, although we're in the background and using REST to communicate with the service.

How it works...

WCF Data Services is a server-side technology that allows making entities of a model available on the web. Underneath, it uses REST as its communication platform. It's possible to connect to the services using the WebClient class or the HttpWebRequest class. Each entity of the model is exposed as a resource and can be connected to via a unique URI. However, the data source used has to have an IQueryble interface for exposing the entities. If we want updates to be sent to the data, the IUpdatable interface should be implemented as well. A good example of this is the **ADO.NET Entity Framework**, which exposes such a data source through the Entity Model. Note that you can create your own data source and attach WCF Data Services to it as well.

One important thing to understand is that WCF Data Services have nothing to do with the actual data access itself. It works with entities exposed by a model (in this example, the model from Entity Framework).

It's easy to see that WCF Data Services actually use REST under the hood. To get data, we need to send a request to a specifically formed URI, combined with one of the standard HTTP keywords, such as GET, POST, PUT, or DELETE. Sound familiar? Indeed, it is exactly the same way of working as we did with REST in the previous recipes.

The URIs created by WCF Data Services to expose the entities are simple to understand. In the following example, which will retrieve a Computer entity with an ID equal to 1, we can see that the URI is composed of the name of the service, followed by the name of the entity (Computer), and the ID that we want to retrieve: `http://localhost:12345/ComputerInventoryService.svc/Computer(1)`.

The resulting response can be sent in an XML or JSON format. XML, in the form of AtomPub, is the default format and is actually easiest to read. The **Atom Publishing Protocol (AtomPub)** is a protocol based on HTTP that allows creating and publishing web resources. The response sent when invoking the previous URI is shown in the following code:

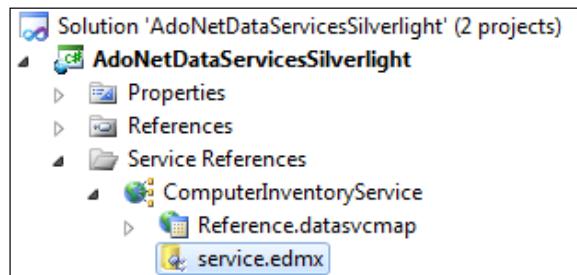
```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<entry
    xml:base="http://localhost:8624/ComputerInventoryService.svc/"
    xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/
        metadata"
    xmlns="http://www.w3.org/2005/Atom">
    <id>http://localhost:8624/ComputerInventoryService.svc/Computer(1)
    </id>
    <title type="text" />
    <updated>2009-07-19T12:37:26Z</updated>
    <author>
        <name />
    </author>
    <link rel="edit" title="Computer" href="Computer(1)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices
        /related/ComputerType"
        type="application/atom+xml;type=entry" title="ComputerType"
        href="Computer(1)/ComputerType" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/
        related/Manufacturer" type="application/atom+xml;type=entry"
        title="Manufacturer" href="Computer(1)/Manufacturer" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices
        /related/User"
        type="application/atom+xml;type=feed" title="User"
        href="Computer(1)/User" />
```

```

<category term="ComputerInventoryModel.Computer" scheme="http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
- <content type="application/xml">
- <m:properties>
<d:ComputerId m:type="Edm.Int32">1</d:ComputerId>
<d:ComputerName>Lenovo W500</d:ComputerName>
<d:OS>Windows 7</d:OS>
<d:WarrantyUntil m:type="Edm.DateTime">2012-07-01T00:00:00</
d:WarrantyUntil>
<d:DateAdded m:type="Edm.DateTime">2009-07-01T00:00:00</
d:DateAdded>
</m:properties>
</content>
</entry>
```

The big difference in working with plain REST services is the existence of the WCF Data Service Client library in Silverlight. It frees us from manually having to create the URLs to request data, and writing XML parsing code to read out the response. It's basically a large wrapper around these tasks, allowing us to work with data on the client as if the service barrier isn't there.

This is achieved through code-generation. It's possible to add a service reference to a `WCF.DataService` file in your Silverlight project. This will result in the creation of client-side data classes and a class derived from `DataServiceContext` that represents the service itself. All these classes are located in the `reference.cs` file. The following screenshot shows where all this generated code is located:



Also, the required assembly—`System.Data.Services.Client.dll`—is added to the Silverlight project. Finally, a client-side version of the server-side entity model (`*.edmx`) is generated that contains the structure of the entity model.

We can write LINQ queries in the Silverlight application that are translated into a URI, to which a request is sent. The response is parsed for us and available as objects of the generated classes. Thus, we have full IntelliSense inside Visual Studio as well, which makes coding a lot easier. We'll look at writing queries to get and update data in the next two recipes.

Locked-down services

WCF Data Services are completely locked down by default. Access is not permitted to the entities automatically. Due to this, in the `InitializeService` method of the `DataService` class, we have to configure this access using the `DataServiceConfiguration` instance. Several options exist to give more or less permissions to the entities, such as `All`, `AllRead`, `None`, and so on. Go to <http://msdn.microsoft.com/en-us/library/system.data.services.entitysetrights.aspx> for a complete overview of this enumeration.

See also

In the next recipe, we'll build further on this recipe by showing how we can read data from services. The *Persisting data using WCF Data Services* recipe will show how we can perform create, update, and delete operations on the data.

Reading data using WCF Data Services

Applies to Silverlight 3, 4 and 5

Let's assume we have decided that WCF Data Services is going to be the technology to get data inside our Silverlight application, which admittedly is a great choice. In the previous recipe, we saw how we can set up Silverlight to use WCF Data Services. However, we didn't actually exchange any data with the service (which is quite ironic for a data service).

In this recipe, we'll perform read operations on the data by building on the code created in the previous recipe. This time, we'll focus on the Computer data in the database.

Getting ready

This recipe continues on the code created in the previous recipe. If you want to follow along, you can continue using your code, or use the provided starter solution located in the `Chapter10/WorkingWithWcfDataServices_Reading_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter10/WorkingWithWcfDataServices_Reading_Completed` folder.

How to do it...

In the previous recipe, we introduced the client library that dramatically reduces the amount of code that we need to write compared to plain REST services. Using this library, we can load data in several formats, such as an entire list, a single object with or without related entities, and so on. We'll build an application that shows a list of computers. The details of each computer can be seen using a detail screen. Perform the following steps to start reading data from WCF Data Services:

- The XAML for the application is similar to the XAML we used in the previous recipes. The application's UI mainly consists of a `DataGrid` with defined columns. The code for this `DataGrid` can be found in the code bundle. Thanks to the client library, we have the ability to write LINQ queries. These LINQ queries are executed using an instance of the context, so creating this context instance should be our first step. Note that the context instance accepts a URI to the `.svc` file of the service as a parameter. After this, we can write a LINQ query, in which we load all `Computer` entities. A little caution though: WCF Data Services wouldn't load the related `Manufacturer` objects by default, although we want to show this information as well in the `DataGrid`. Therefore, we specify this using the `Expand` method as shown in the following:

```
ComputerInventoryEntities context=new ComputerInventoryEntities
    (new Uri("ComputerInventoryService.svc", UriKind.Relative));
public MainPage()
{
    InitializeComponent();
}
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    ComputerLoadStart();
}
private void ComputerLoadStart()
{
    var query = from c in context.Computer.Expand("Manufacturer")
                select c;
}
```

- While the query looks rather normal, do keep in mind that all Silverlight's service requests are carried out asynchronously. Therefore, the query is cast to a `DataServiceQuery<T>` (in this case, the return type `T` is `Computer`). On this instance, the `BeginExecute` method is called, which triggers an asynchronous call to the service. Similar to other asynchronous calls, a callback method is passed in. The query itself is also passed in, so we have access to it in the callback method. This is shown in the following code:

```
private void ComputerLoadStart()
{
    DataServiceQuery<Computer> dsq =
        (DataServiceQuery<Computer>)query;
    dsq.BeginExecute(ComputerLoadCompleted, dsq);
}
```

3. When the service is ready, the ComputerLoadCompleted callback method is invoked. This method receives an IAsyncResult instance as parameter, which contains the DataServiceQuery<T>. By calling the EndExecute method on this instance, we get access to the returned Computer instances. We place these instances in an ObservableCollection called computerCollection for data binding purposes. The collection is bound to the DataGrid using the ItemsSource property as shown in the following code:

```
ObservableCollection<Computer> computerCollection =  
    new ObservableCollection<Computer>();  
private void ComputerLoadCompleted(IAsyncResult asr)  
{  
    DataServiceQuery<Computer> dsq =  
        (DataServiceQuery<Computer>)asr.AsyncState;  
    foreach (var computer in dsq.EndExecute(asr).ToList())  
    {  
        computerCollection.Add(computer);  
    }  
    ComputersDataGrid.ItemsSource = computerCollection;  
}
```

The result is a list of computers as shown in the following screenshot:

The screenshot shows a Silverlight application window titled "Computer Inventory". At the top, there is a control bar with buttons for "Reload data" and "Create new". Below the control bar is a DataGrid displaying three rows of computer data. The columns are labeled "View", "Edit", "ComputerId", "ComputerName", "Manufacturer", and "Date added". The first row has ComputerId 1, ComputerName "Lenovo W500", Manufacturer "Lenovo", and Date added "1/07/2009". The second row has ComputerId 2, ComputerName "XPS M1210", Manufacturer "Dell", and Date added "1/07/2009". The third row has ComputerId 3, ComputerName "Aspire One", Manufacturer "Acer", and Date added "4/07/2009". Each row has a "View" button in the first column and an "Edit" button in the second column.

View	Edit	ComputerId	ComputerName	Manufacturer	Date added	
View	Edit	1	Lenovo W500	Lenovo	1/07/2009	
View	Edit	2	XPS M1210	Dell	1/07/2009	
View	Edit	3	Aspire One	Acer	4/07/2009	

4. Let's now take a look at the detail page. When clicking on a **View** button in the grid, we load a Silverlight Child Window named ComputerDetailView.xaml that features a nice zoom-in effect when opened. The XAML for this window is straightforward and can be found in the code bundle.

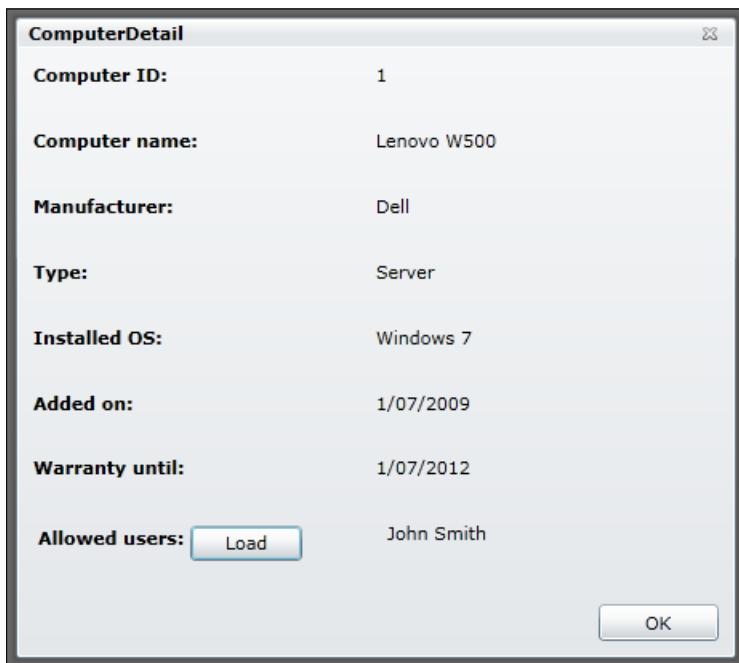
5. To show the details of the selected computer in the `DataGridView`, we pass the context as well as the `computerID` of the selected computer via the constructor. This is shown in the following code:

```
private ComputerInventoryEntities context;
private int computerId;
private Computer computer;
public ComputerDetailView(ComputerInventoryEntities context,
    int computerId)
{
    InitializeComponent();
    this.context = context;
    this.computerId = computerId;
    LoadComputer();
}
```

6. In the `LoadComputer` method, we load the details of the selected computer. However, the computer is already being tracked by the `context` because of the list display, but not all the data we need is loaded (the computer type is omitted in the list). Thus, we need to explicitly tell the `context` that it has to reload the computer using the `OverWriteChanges` of the `MergeOption` enumeration. The following code shows this loading process:

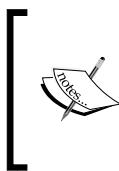
```
private void LoadComputer()
{
    context.MergeOption = MergeOption.OverwriteChanges;
    var query = from c in context.Computer.Expand("ComputerType")
                where c.ComputerId == computerId
                select c;
    DataServiceQuery<Computer> dsq =
        (DataServiceQuery<Computer>)query;
    dsq.BeginExecute(ComputerLoadCompleted, dsq);
}
private void ComputerLoadCompleted(IAsyncResult asr)
{
    DataServiceQuery<Computer> dsq =
        (DataServiceQuery<Computer>)asr.AsyncState;
    computer = dsq.EndExecute(asr).FirstOrDefault<Computer>();
    ComputerDetailGrid.DataContext = computer;
}
```

7. After all the previous code is added, we have successfully created a master-detail implementation based on WCF Data Services. The detail screen is shown in the following image:



How it works...

The most important part of this recipe is the LINQ query. When executing a LINQ query against a WCF Data Service, the query is translated into a format that the service understands—a URI. All the options we specify in the query are translated into the URI. The URI to which a request is sent is `http://127.0.0.1:8624/ComputerInventoryService.svc/Computer()?$expand=Manufacturer`. (This can be seen using **Fiddler2**. More information on this tool can be found in *Appendix C*.)



Note that the `Expand` option instructs the service to retrieve all Computer instances, and expand the results to include the related Manufacturer instances for each Computer instance. This process is called **eager loading**. In this process, we explicitly ask to load the related entities initially. If we omit eager loading, the property will have a null value.

To see what the AtomPub (XML) response of the service looks like, simply copy/paste the previously mentioned URI in your browser or view it in Fiddler2.

The context is the real workhorse in this recipe. It keeps track of all the loaded items (this is called **object tracking**). However, sometimes we need to ask for a complete reload. In the example at hand, we need to do so in the detail screen. We have the `MergeOption` enumeration at our disposal for this. The `OverwriteChanges` explicitly tells the context that it should replace the item loaded in the context.

There's more...

We might know that there are related entities, but not want to load them initially. We can load on demand using the `LoadProperty` method. This method is used in the detail screen of the application. When loading, the allowed users are not retrieved automatically (for example, not to stress the database). By clicking on the `Load` button, we load them asynchronously on demand using the `LoadProperty` method. The result is that the `Computer` entity will have its property filled with the related `User` entities. This is shown in the following code:

```
private void LoadUsersButton_Click(object sender, RoutedEventArgs e)
{
    context.BeginLoadProperty(computer, "User", UsersLoadCompleted,
        null);
}
private void UsersLoadCompleted(IAsyncResult asr)
{
    context.EndLoadProperty(asr);
    // do something with the loaded values here
}
```

See also

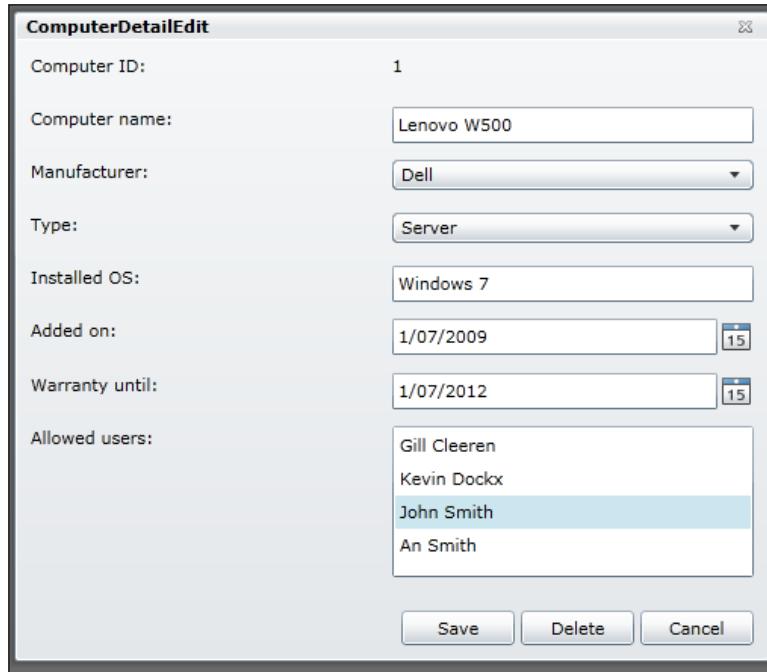
In the *Reading data from ADO.NET Data Services* recipe, we create the ADO.NET Data Service and set up communication with it.

Persisting data using WCF Data Services

Applies to Silverlight 3, 4 and 5

In the previous recipe, we saw how to read data from WCF Data Services. Apart from reading data, we should be able to persist data using these services. In other words, adding, updating, and deleting data to make the **CRUD** story complete (**CRUD: Create, Read, Update, and Delete**, this term is often used to refer to the four basic operations on data).

This recipe will add a new screen to the application built in the previous two recipes to make it possible to create new computers and to update and delete the existing ones. The screen in the following image is similar to the View image, but it has editable fields and some extra buttons:



Getting ready

This recipe builds on the code created in the previous two recipes. This means that you can continue using your own solution to follow along with this recipe. Alternatively, you can use the starter solution located in the `Chapter10/WorkingWithWcfDataServices_Persisting_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter10/WorkingWithWcfDataServices_Persisting_Completed` folder.

How to do it...

We'll follow a small scenario, where we'll create a new computer object, update it, and finally remove it from the database. Along the way, we'll come across the specifics of each operation. In order to do this, we'll need to complete the following steps:

- As this screen is used for both adding new items and editing existing ones, we add an enumeration to the Silverlight application called `EditingModes` to see in which state we are. This is shown in the following code:

```
public enum EditingModes
{
    New,
    Edit
}
```

- The UI contains two `ComboBox` controls that allow the user to select a `Manufacturer` and a `Type`. Also, all `Users` should be loaded in the `ListBox` at the bottom of the screen. Loading data into these controls is similar. The code to load the `Manufacturer` objects is as follows:

```
public ComputerDetailEdit(ComputerInventoryEntities context,
    int computerId, EditingModes editingMode)
{
    InitializeComponent();
    ...
    ManufacturerLoadStart();
}
private void ManufacturerLoadStart()
{
    var query = from m in context.Manufacturer
                select m;
    DataServiceQuery<Manufacturer> dsq =
        (DataServiceQuery<Manufacturer>)query;
    dsq.BeginExecute(ManufacturerLoadCompleted, dsq);
}
private void ManufacturerLoadCompleted(IAsyncResult asr)
{
    DataServiceQuery<Manufacturer> dsq =
        (DataServiceQuery<Manufacturer>)asr.AsyncState;
    ComputerManufacturerComboBox.ItemsSource = dsq.EndExecute(asr);
    ComputerManufacturerComboBox.DisplayMemberPath =
        "ManufacturerName";
}
```

- Let's now look at how we can add an item. We create a new instance of the `Computer` class and set it as the `DataContext` of the main grid—`ComputerDetailGrid`. As this is a new object, the context doesn't know it yet, so we make the context track it using the `AddObject` method. This is shown in the following code:

```
Computer computer = new Computer();
ComputerDetailGrid.DataContext = computer;
context.AddObject("Computer", computer);
```

4. The selected ComputerType and Manufacturer should be linked to the Computer object, so that the context can track this link. Also, every selected user in the listbox should be linked to the computer. It's important that the context knows which links exist between objects. When persisting, it needs to know which relations in the database need to be created. This is shown in the following code:

```
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
    context.SetLink(computer, "ComputerType",
        computer.ComputerType);
    context.SetLink(computer, "Manufacturer",
        computer.Manufacturer);
    foreach (var user in ComputerUsersListBox.SelectedItems)
    {
        context.AddLink(computer, "User", (User)user);
    }
}
```

5. Once the user clicks on the Save button, the actual save operation should start. Again, this is done asynchronously by making use of the BeginSaveChanges method available on the context. We pass in a callback method that will be called when the service returns. This is shown in the following code:

```
context.BeginSaveChanges(SaveChangesOptions.None, new
    AsyncCallback(PersistChanges), null);
```

6. In the callback, we use the EndSaveChanges method, which returns a DataServiceResponse object, containing the response of the server. If errors were encountered, we can retrieve them by looping over this object. This is shown in the following code:

```
private void PersistChanges(IAsyncResult asr)
{
    try
    {
        DataServiceResponse dataServiceResponse =
            (DataServiceResponse)context.EndSaveChanges(asr);
        foreach (OperationResponse operationResponse in
            dataServiceResponse)
        {
            if (operationResponse.Error != null)
            {
                //do something with the error
            }
        }
        if (errorsOccurred)
            MessageBox.Show(builder.ToString());
    }
}
```

```
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

7. When we want to update the instance, the code is quite similar. As shown in the following code, we call the `UpdateObject` method to mark the object as `Modified`. The same callback is used as when adding new items:

```
context.UpdateObject(computer);
context.BeginSaveChanges(SaveChangesOptions.Batch, new
    AsyncCallback(PersistChanges), null);
```

8. Finally, deleting the object is done using the `DeleteObject` method. This is shown in the following line of code:

```
context.DeleteObject(computer);
```

Take a look at the sample code where the full code listing is available.

How it works...

When creating a new instance, we immediately set it as the `DataContext` for the main grid of the user control. This way, all changes done by the user on the textboxes that are bound using the `TwoWay` binding are propagated back into the object. However, as this object is new, it is unknown to the `context`. It's not yet being tracked by the `context`, so we need to add it to the collection of tracked objects.

The `Computer` class has links to other classes, namely the `ComputerType`, the `Manufacturer`, and the `User`. Thus, we need to create links in the `context` using `SetLink` (for links with multiplicity = 1) or `AddLink` (for links with multiplicity > 1). Links also need to be made or recreated when updating and deleted when deleting a `Computer` instance.

Just like all other operations towards services, the actual save operation is asynchronous. That's why we use the `BeginSaveChanges` method and specify the callback method in one go. Saving is actually sending data to one or more specific URLs. In the callback method, we use the `EndSaveChanges` method, which returns a `DataServiceResponse` object. This object contains the responses for all calls made to the service (one for saving the actual object, one for linking, and so on). If an operation fails, we can get the error information from the `DataServiceResponse` object as well.

Updating and deleting are very similar. All changes are done initially on the objects tracked by the `context`. Afterwards, the changes are persisted using exactly the same code used for adding new objects.

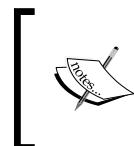
There's more...

When calling the `BeginSaveChanges` method, we have the option to pass along how we want the subsequent operations to be executed through the `SaveChangeOptions` enumeration. We can use the **Batch** option, which creates a unit of work containing all operations. This can be compared to working with a transaction—either all operations work or they all fail. Other options include `None` and `ContinueOnError`. More information on this enumeration can be found at <http://msdn.microsoft.com/en-us/library/system.data.services.client.savechangesoptions.aspx>

Talking to Flickr

Applies to Silverlight 3, 4 and 5

There are quite a few large websites out there that expose (part of) their functionality through services, most of the time through the use of RESTful services. A great example is Flickr (www.flickr.com). Flickr exposes many services that allow searching for pictures, tagging existing pictures, uploading pictures, and so on. We can leverage all the goodness that Flickr provides inside our applications to provide more functionality to our end users.



Flickr is a popular website where people can upload and share images and videos. Apart from viewing this content on the site, Flickr offers a wide range of services for interaction with its content. Currently, Flickr has millions of users sharing several billion images!



One thing that is very important is the open `crossdomain.xml` file that Flickr exposes. It allows connecting from every domain (so also from a Silverlight application running locally). This is why we can connect directly from Silverlight to Flickr. However, most Web 2.0 websites aren't that open, for example, Twitter. Communication with such a service from Silverlight is explained in the following recipe.

Note that not all code for this sample is printed in this book. Refer to the code in the downloadable samples for this.

Getting ready

Most sites that expose public services, such as Flickr, Amazon, Digg, and so on, allow us free access to their services, however, you'll often need to register to get a key/identification. This is then used by the issuing site to track where the call came from. Some services allow only a limited number of calls for a particular key within a certain time span to discourage overuse. For the code in this recipe, you'll need a Flickr API key, which can be obtained for free from <http://www.flickr.com/services/api/keys/>. This key can be pasted in the sample code that can be downloaded for this book.

The recipe uses a `WrapPanel`—a control that's part of the **Silverlight Control Toolkit**. The toolkit is a collection of controls and extensions on Silverlight. This can be obtained from www.codeplex.com/silverlight. See *Appendix D* for more information on the Silverlight Control Toolkit.

To follow along with this recipe, a starter solution has been provided in the `Chapter10/SilverFlickr_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter10/SilverFlickr_Completed` folder.

How to do it...

In this recipe, we'll build a simple application that allows us to search for photos based on a search term that the user can enter. On clicking on one of the results, the details of the photo are shown. For this, the application uses two of the many methods available from Flickr, namely `flickr.photos.search` and `flickr.photos.getinfo`. These methods allow searching for photos matching a search string and getting more information on a photo, respectively. To begin building this application, we'll need to complete the following steps:

1. Open the starter solution as outlined in the *Getting ready* section. This is an empty solution with some assets already in place.
2. As we are going to use REST services, we'll be making use of the `WebClient` class. This class resides in the `System.Net` namespace, which is part of the `System`.`Net` assembly. If you're using Visual Studio 2008 (for building/maintaining a Silverlight 3 project), you need to add a reference to this assembly yourself. Visual Studio 2010 refers to this assembly by default for new projects.
3. The XAML code for the UI of the application is quite easy to understand. A `StackPanel` resides at the top of the page, containing an `Image`, a `TextBox` to enter the search query, and a `Button`. The page also contains a `ScrollViewer` with a `WrapPanel` (part of the Silverlight Control Toolkit, refer to the *Getting ready* section of this recipe) on the left. The XAML code for this is as follows:

```
<Grid x:Name="LayoutRoot"
      Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="50"></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="300"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.Row="0"
               Grid.Column="0"
               HorizontalAlignment="Left">
```

```
    Orientation="Horizontal"
    Grid.ColumnSpan="2">
<Image Source="flickr.png"
    Stretch="None"
    Margin="3 0 0 0" >
</Image>
<TextBox x:Name="SearchTextBox"
    Width="200"
    Height="30"
    Margin="5" >
</TextBox>
<Button x:Name="SearchButton"
    Content="Search Flickr"
    Click="SearchButton_Click"
    HorizontalAlignment="Center"
    VerticalAlignment="Center"
    Margin="5" >
</Button>
</StackPanel>
<ScrollViewer Grid.Row="1"
    Grid.Column="0"
    Background="DarkGray">
<toolkit:WrapPanel x:Name="ResultPanel"
    HorizontalAlignment="Center">
</toolkit:WrapPanel>
</ScrollViewer>
</Grid>
```

4. As mentioned before, Flickr's API is a REST API. Thus, we need to send a request to a specific URI and read out the response being sent back. Let's first take a look at the URI. As defined by Flickr, this URI needs to be in a specific format. As we'll be doing a search, we'll use the `flickr.photos.search` method. It requires two parameters: your personal API key and the search term entered in the search field. This is shown in the following code:

```
string api_key = "123456"; //TODO: replace with your own key
string searchUrl = "http://api.flickr.com/services/rest
    /?method=flickr.photos.search&api_key={0}&text={1}";
```

5. We now have the URI; we can use it to send a request. To send this request, we'll use the `WebClient` class again. In the `Click` event handler of the button, we'll create an instance of this class. We need to register the callback method via `DownloadStringCompleted` and send the request using `DownloadStringAsync`, passing in the URI as a parameter. As with other services, these calls are asynchronous. This is shown in the following code:

```
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted +=
        new DownloadStringCompletedEventHandler
        (client_DownloadStringCompleted);
    client.DownloadStringAsync(new Uri(string.Format(searchUrl,
        api_key, SearchTextBox.Text)));
}
```

6. In the callback, we have access to the result of the call via the `Result` property on the instance of `DownloadStringCompletedEventArgs`. The response is plain XML, formatted by Flickr in a specific format. We'll use LINQ-To-XML to parse this XML code and create a list of `ImageInfo` objects (shown in the following code), a custom type defined to have typed access to our data in the Silverlight application. Note that the `ImageUrl` implementation creates the link to the image as used by Flickr. Add the following class to the Silverlight project:

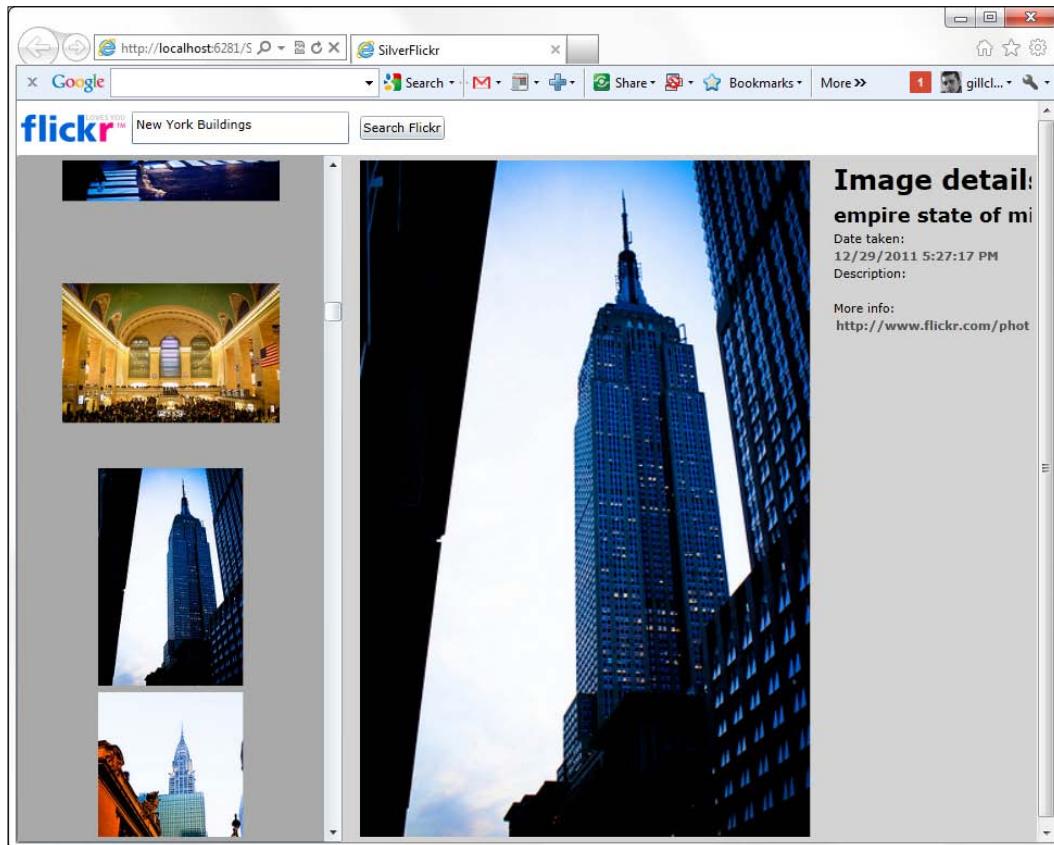
```
public class ImageInfo
{
    public string ImageId { get; set; }
    public string FarmId { get; set; }
    public string ServerId { get; set; }
    public string Secret { get; set; }
    public string ImageUrl
    {
        get
        {
            return string.Format
                ("http://farm{0}.static.flickr.com/{1}/{2}_{3}_m.jpg",
                FarmId, ServerId, ImageId, Secret);
        }
    }
}
```

7. Add a reference to the `System.Xml.Linq` assembly inside the Silverlight project.

8. Finally, each `ImageInfo` instance is used to dynamically create an image and add it to the `WrapPanel`. Every image also gets a click event attached to it, which is used to open the detail page. This is shown in the following code:

```
void client_DownloadStringCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    XDocument xml = XDocument.Parse(e.Result);
    var photos = from results in xml.Descendants("photo")
                 select new ImageInfo
    {
        ImageId = results.Attribute("id").Value.ToString(),
        FarmId = results.Attribute("farm").Value.ToString(),
        ServerId = results.Attribute("server").Value.ToString(),
        Secret = results.Attribute("secret").Value.ToString()
    };
    foreach (var image in photos)
    {
        Image img = new Image();
        BitmapImage bmi = new BitmapImage(new
            Uri(image.ImageUrl, UriKind.Absolute));
        img.Source = bmi;
        img.Width = 200;
        img.Height = 200;
        img.Stretch = Stretch.Uniform;
        img.Tag = image;
        img.Margin = new Thickness(3);
        img.HorizontalAlignment = HorizontalAlignment.Center;
        ResultPanel.Children.Add(img);
    }
}
```

9. At this point, we can search Flickr for images. The following screenshot shows the finished application. Note that this final application includes extra code that allows clicking an image and viewing its details. However, the code for this is very similar and can be found in the code bundle:



How it works...

Communicating with Flickr's REST services is, in fact, no different from communicating with a self-created REST service, as was done in the beginning of this chapter.

The URI is created according to the specifications given by the Flickr API. At <http://www.flickr.com/services/api>, you can find an overview of all the methods exposed by Flickr, varying from searching for pictures and reading out comments to finding pictures based on a location. For this recipe, we use the `flickr.photos.search` and `flickr.photos.getinfo` methods. Both require the API key sent as a parameter, apart from the specific parameters depending on the method.

The format of the XML sent by Flickr's services is fixed. It's safe to build our code around this API, as the format can be considered to be a contract between the service and the client application. The service will always return the response formatted according to this specification. The following is the XML structure used by Flickr:

```
<rsp>
  <photos>
    <photo id="1234567890"
           secret="0987654321"
           server="1234"
           farm="1" />
  </photos>
</rsp>
```

There's more...

Communication with the services exposed by Flickr from Silverlight is possible, because Flickr has a `crossdomain.xml` file in place that allows calls from any domain, as explained in the introduction of the recipe. The following is the complete `crossdomain.xml` (<http://api.flickr.com/crossdomain.xml>) file of Flickr. Refer to the *Configuring cross-domain calls* recipe in Chapter 7, *Working with Services*, for more information on the concept of cross-domain calls.

```
<?xml version="1.0" ?>
  <!DOCTYPE cross-domain-policy (View Source for full doctype...)>
  <cross-domain-policy>
    <allow-access-from domain="*" secure="true" />
    <site-control permitted-cross-domain-policies="master-only" />
  </cross-domain-policy>
```

However, other sites don't open up their API as much as Flickr does. A good example is Twitter (<http://twitter.com/crossdomain.xml>), which allows calls only from particular domains. This can be seen in the following code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<cross-domain-policy xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "http://www.adobe.com/xml/schemas/
  PolicyFile.xsd">
  <allow-access-from domain="twitter.com" />
  <allow-access-from domain="api.twitter.com" />
  <allow-access-from domain="search.twitter.com" />
  <allow-access-from domain="static.twitter.com" />
  <site-control permitted-cross-domain-policies="master-only" />
```

```
<allow-http-request-headers-from domain="*.twitter.com"
    headers="*" secure="true" />
</cross-domain-policy>
```

The consequence of such a `crossdomain.xml` file is that Silverlight can't connect directly with these services. The solution is creating an extra service on the same domain as the Silverlight application, which will in turn call the REST services. Your application then only has to connect with the new service, which shouldn't be a problem. We'll look at this scenario in the following recipe. A second possible solution is building a trusted Silverlight application, which we'll look at in the last recipe of this chapter.

Flickr... more information

The accompanying code for this book also contains the code to create the detail screen. For this, we can use another method, namely `flickr.photos.getinfo`, to retrieve more information about an image based on the photo ID.

Displaying the values is done through the use of data binding. The `DataContext` property of the grid, located in the `Details` portion of the interface, is set to an instance of another type called `ImageDetail`.

One particularity is certainly worth mentioning here - data binding the image is done through the use of a converter. The link to the image is stored as a `Uri` in the instance of `ImageDetail`. However, binding in XAML expects a `BitmapImage`. The conversion of type A to type B is done through the use of a converter—a class that implements the `IValueConverter` interface. This interface has two methods—`Convert` and `ConvertBack`. This is shown in the following code:

```
public class ImageConverter:IValueConverter
{
    public object Convert(object value, Type targetType, object
        parameter, System.Globalization.CultureInfo culture)
    {
        if (value != null)
            return new BitmapImage((Uri)value);
        else
            return "";//can be link to a "NoImage.png" of some kind
    }
    public object ConvertBack(object value, Type targetType, object
        parameter, System.Globalization.CultureInfo culture)
    {
    }
}
```

See also

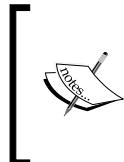
In the *Reading data from a REST service* and *Parsing REST results with LINQ-To-XML* recipes from this chapter, we go deeper into the details of communication with a REST service. The following recipe shows the scenario to connect with services that don't allow cross-domain calls.

For more information on data binding, refer to the recipes in *Chapter 2, An Introduction to Data Binding*, and *Chapter 3, Advanced Data Binding*.

Talking to Twitter over REST

Applies to Silverlight 3, 4 and 5

Like Flickr, Twitter has a great API that allows us to build applications incorporating its functionality.



Twitter is a social networking site where people can post small messages of up to 140 characters, also known as **tweets**. These messages are shared with people that follow you, meaning they're interested in what you're doing. Twitter is often referred to as being a micro-blogging site. Using Twitter is free.

However, as explained in the *There's more...* section of the previous recipe, where we compared the `crossdomain.xml` files of Flickr and Twitter, Twitter is much more locked down. It doesn't allow client-side applications built in Silverlight to make cross-domain calls. In this recipe, which can be generalized for all types of REST services that don't have an open cross-domain file, we'll look at how we can still succeed in communicating with the service.

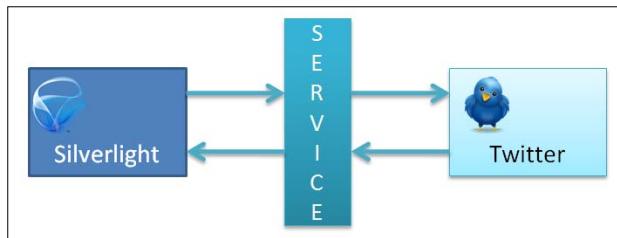
Getting ready

To work with the application built in this sample, you'll need an account on Twitter. Twitter is free and you can register at www.twitter.com. Unlike Flickr, you don't have an API key. In this recipe, we'll start from an empty Silverlight application. Refer to *Chapter 1, Learning the Nuts and Bolts of Silverlight 5*, for more information on this.

A starter solution for this recipe is provided in the `Chapter10/SilverWitter_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter10/SilverWitter_Completed` folder.

How to do it...

The way we architect the application that will work with Twitter is quite important, as we can't call the Twitter services from Silverlight directly. However, services that run on a server don't mind cross-domain restrictions. They can call Twitter's REST services without a problem. The solution for the problem is adding an extra service layer in our architecture. The Silverlight application will communicate with our own services and in turn, these services can talk to Twitter. The following image demonstrates this idea clearly:



To get up and running, we'll need to complete the following steps:

1. Open the starter solution as outlined in the *Getting ready* section, containing an ASP.NET Web Application.
2. Add another ASP.NET web application to the solution called `SilverWitter.Services`.
3. In this web application, add a WCF service called `TwitterService.svc`. Thus, you'll have three projects in your solution: the Silverlight application, the hosting web application, and an extra website containing a WCF service.
4. Silverlight will communicate only with the WCF service and the service will communicate with Twitter. Only the functionality we expose on our own service will be available for the Silverlight application. Let's first define the contract, an interface of our WCF service in the `ITwitterService.svc.cs` file. We want to be able to validate user credentials, get all tweets from the public timeline, get all tweets from a specified user and his/her friends, and finally add a tweet (a small message). Note that this is a WCF service, and not a REST service, although we could create a REST service if we wanted to. The following code shows the contract:

```

[ServiceContract]
public interface ITwitterService
{
    [OperationContract]
    List<TwitterUpdate> GetPublicTimeLine();

    [OperationContract]
    List<TwitterUpdate> GetUserTimeLine(string twitterUser,
        string userName, string userPassword);

    [OperationContract]
    List<TwitterUpdate> GetFriendsTimeLine(string twitterUser,
        string friendName, string userPassword);
}
    
```

```
        string userName, string userPassword);
[OperationContract]
string AddMessage(string message, string userName,
    string userPassword);
[OperationContract]
bool CheckCredentials(string userName, string userPassword);
}
```

5. We used the `TwitterUpdate` class in some of the previous methods. This class should be added to the services project—`SilverWitter.Services`. As instances of this class will be sent over the wire (to the Silverlight application), this class should be attributed with `DataContractAttribute`. Its members have `DataMemberAttribute` applied to them. This class is shown as follows:

```
[DataContract]
public class TwitterUpdate
{
    [DataMember]
    public string Message { get; set; }
    [DataMember]
    public string User { get; set; }
    [DataMember]
    public string Location { get; set; }
}
```

6. In the implementations of these methods, in the `TwitterService.cs` file, we'll write the code to talk with Twitter. Twitter's API is REST-based and can communicate using XML, JSON, RSS, and ATOM. This means that we have to send a request to a particular URI and capture the results sent back by Twitter. This result can then be parsed using LINQ-To-XML and mapped to the CLR objects. Let's take a look at the code that we need to write in the service method implementations. We'll implement the `GetUserTimeLine` method here; the other ones are similar and can be found in the code bundle available on the Packt website. As the service is a REST service, we need to use a specific URL, as defined by Twitter:

```
public List<TwitterUpdate> GetUserTimeLine(string twitterUser,
    string userName, string userPassword)
{
    try
    {
        string userTimeLine =
            "http://twitter.com/statuses/user_timeline/"
            + twitterUser + ".xml";
    }
    catch (Exception)
    {
        return null;
    }
}
```

7. Although we're not writing Silverlight code here to access Twitter (we're writing a WCF service implementation), the concepts are the same. We can use the `WebClient` class to perform the call to the service. One big difference here is that service communication can be done synchronously. Note that Twitter requires that we pass in credentials to access this service method:

```
WebClient client = new WebClient();
client.Credentials = new
    NetworkCredential(userName, userPassword);
ServicePointManager.Expect100Continue = false;
string result = client.DownloadString(
    string.Format(userTimeLine, twitterUser));
```

8. Once the result is available, we'll parse it using LINQ To XML as shown in the following code. On parsing the XML, we are creating a `List<TwitterUpdate>`:

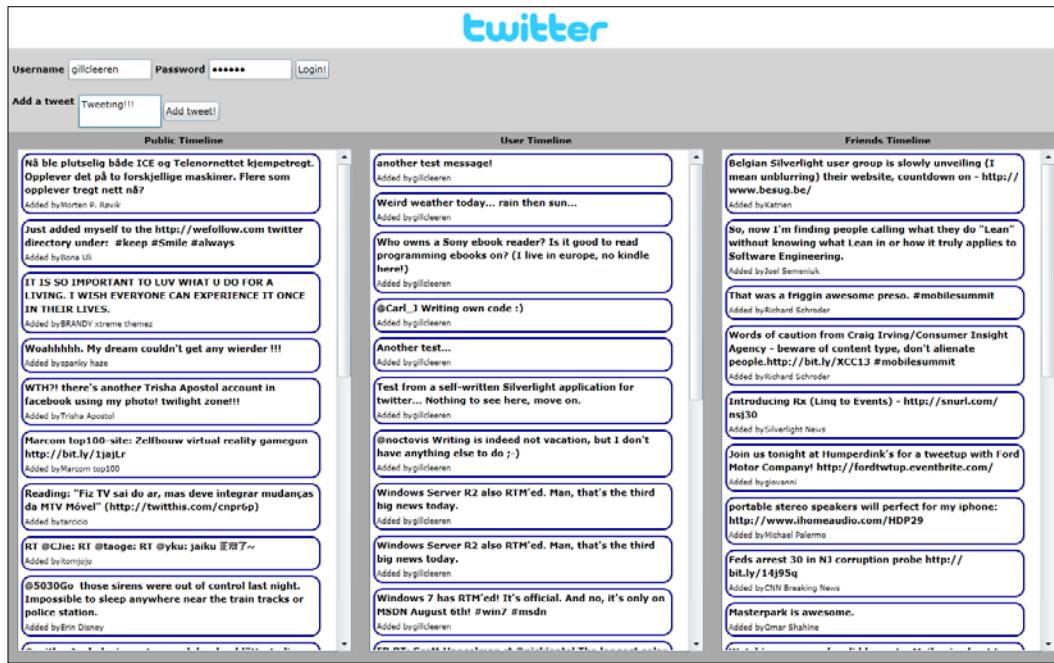
```
XDocument document = XDocument.Parse(result);
List<TwitterUpdate> twitterData =
    (from status in document.Descendants("status")
    select new TwitterUpdate
    {
        Message = status.Element("text").Value.Trim(),
        User = status.Element("user").Element("name").Value.Trim()
    }).ToList();
return twitterData;
```

9. As this service is in another site than the Silverlight application, Silverlight will need to perform a cross-domain call to it. To allow this, we have to add a policy file. Add a new XML file called `clientaccesspolicy.xml` to the services site and insert the following code:

```
<?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>
```

10. After having implemented the methods on the WCF service, let's focus on the Silverlight application. First, in the Silverlight project, add a service reference to the `TwitterService` WCF service. Set the service namespace to `TwitterService`.

11. The UI of the application built in this sample is shown in the following image. The XAML code can be found in the code bundle. When opening the application, the statuses of the public timeline are shown as they don't require any credentials. The user can log in to Twitter, and when authenticated, add a tweet and view his/her tweets and those of his/her friends, as the following image demonstrates:



12. The Silverlight application now talks to our own service. In the following code, we are asynchronously invoking the service to get the time line (status updates) of the user:

```
private void LoadUserTimeLine()
{
    TwitterService.TwitterServiceClient client = new
        SilverWitter.TwitterService.TwitterServiceClient();
    client.GetUserTimeLineCompleted += new EventHandler
        <SilverWitter.TwitterService.GetUserTimeLineCompleted
        EventArgs>(client_GetUserTimeLineCompleted);
    client.GetUserTimeLineAsync(UserNameTextBox.Text,
        UserNameTextBox.Text, PasswordTextBox.Password);
}
void client_GetUserTimeLineCompleted(object sender,
    SilverWitter.TwitterService.GetUserTimeLineCompletedEventArgs e)
{
```

```
if (e.Result != null)
{
    UserTimeLineListBox.ItemsSource = e.Result;
}
```

The other methods are similar and can be found in the sample code.

How it works...

As previously explained, Twitter, along with most Web 2.0-type applications, has a locked-down cross-domain file. Silverlight's cross-domain restrictions prohibit us from directly calling the REST API from Silverlight. Therefore, we need to build a service layer in between the Silverlight application and the REST service. As services themselves don't mind cross-domain restrictions, we can call whatever type of REST services (or other types) we want. Our own service will act as a pass-through for data in both directions.

See also

To understand why Twitter and Flickr need such a different approach, read the previous recipe in this chapter. In the following recipe, we'll see how trusted Silverlight applications can talk directly to Twitter, as they aren't tied to cross-domain restrictions.

Passing credentials and cross-domain access to Twitter from a trusted Silverlight application

Applies to Silverlight 4 and 5

Whenever we need to communicate with a service that is not hosted in the same domain as the Silverlight application, we need to think of cross-domain restrictions. Silverlight will check if a cross-domain policy file is in place at the root of the domain. Silverlight 4 and 5 applications can not only run out-of-browser (a capability added with Silverlight 3 that allows applications to run as a standalone application, instead of in the browser), they can also run as a **Trusted Application**. Such an application runs with elevated permissions. On the agreement of the user, the application is installed, and has more permissions on the local system and other capabilities than the in-browser or regular out-of-browser applications. One of these capabilities is accessing cross-domain services without restrictions, meaning that the service will be accessible from Silverlight even if there's no policy file present. Silverlight 4 only supports out-of-browser trusted applications, Silverlight 5 even allows in-browser applications to run with elevated permissions.

Also with Silverlight 4 came the ability to send credentials to a service, when using a `WebClient` instance.

The combination of these two added features in Silverlight 4 makes it possible to write a standalone Twitter client that does not need the intermediary service layer, like we used in the previous recipe. Instead, we can now directly communicate with Twitter's API from Silverlight, because there are no cross-domain restrictions. To authorize with the services of Twitter, we need to be able to send credentials, which has also become possible. In this recipe, we'll change the **SilverWitter** client to run as a trusted, out-of-browser application.

Getting ready

To follow along with this recipe, you can use your code created with the previous recipe. Alternatively, a starter solution is provided with the samples for the book in the `Chapter10/TrustedSilverWitter_Starter` folder. The finished solution for this recipe can be found in the `Chapter10/TrustedSilverWitter_Completed` folder.

How to do it...

In the previous recipe, we have already built SilverWitter as an in-browser Silverlight application. Due to this, we required a service layer. Silverlight communicates with this service layer, and the service layer in turn communicates with the API of Twitter. If we create the application as a trusted application (that is, with elevated permissions), this service layer becomes obsolete as there are no cross-domain restrictions. We also need to authenticate with Twitter. We'll do so by sending credentials over the service. The following are the steps that we need to follow to create this application:

1. While the UI of the application is similar to the one created for the in-browser version, we need to add a few extra controls. We need to make it possible for the user to install the application. The complete XAML can be found in the code bundle. The following code outlines the changes. We add a `StackPanel` called `InstallPanel`, in which the controls for the installation are placed. All other controls are placed in a `Grid` called `MainGrid`, for which the `Visibility` has been set to `Collapsed` initially. Both the `InstallPanel` and the `MainGrid` are now children of the `LayoutRoot` `Grid`:

```
<Grid x:Name="LayoutRoot">
    <StackPanel x:Name="InstallPanel"
        Orientation="Vertical"
        HorizontalAlignment="Center"
        Margin="10"
        VerticalAlignment="Top">
        <TextBlock x:Name="InstallTextBlock"
            Text="This application needs to be installed before
            it can be used. Click the button below to"
```

```
        install."
        Width="500"
        TextWrapping="Wrap"
        FontWeight="Bold"
        FontSize="20">
    </TextBlock>
    <Button x:Name="InstallButton"
            Click="InstallButton_Click"
            Content="Install"
            Width="100"
            Height="35">
    </Button>
    <TextBlock x:Name="InstallErrorTextBlock"
               Foreground="Red" >
    </TextBlock>
</StackPanel>
<Grid x:Name="MainGrid"
      Visibility="Collapsed">
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/></RowDefinition>
        <RowDefinition Height="100"/></RowDefinition>
        <RowDefinition Height="*"/></RowDefinition>
    </Grid.RowDefinitions>
    ...
</Grid>
</Grid>
```

2. On starting the application, we need to check if we're running the application inside the browser or as a standalone, trusted application. We can do so using the following code where the Boolean `IsRunningOfflineAndElevated` contains `true` if the conditions are met:

```
private bool IsRunningOfflineAndElevated = false;
public MainPage()
{
    InitializeComponent();
    CheckApplicationState();
}
private void CheckApplicationState()
{
    if (Application.Current.IsRunningOutOfBrowser &&
        Application.Current.HasElevatedPermissions)
        IsRunningOfflineAndElevated = true;
    else
        IsRunningOfflineAndElevated = false;
}
```

3. Based on the value of `IsRunningOfflineAndElevated`, we can change the UI. If it is `false`, meaning that we're still in the browser, we display the `InstallPanel`. If it is `true`, meaning that the application is running as a trusted application, the `InstallPanel` is hidden and the real application UI is shown. This check is done using the following code, which is called from the constructor as well:

```
private void ChangeUI()
{
    if (IsRunningOfflineAndElevated)
    {
        MainGrid.Visibility = System.Windows.Visibility.Visible;
        InstallPanel.Visibility = System.Windows.Visibility.Collapsed;
    }
    else
    {
        MainGrid.Visibility = System.Windows.Visibility.Collapsed;
        InstallPanel.Visibility = System.Windows.Visibility.Visible;
    }
}
```

4. If the application is not yet installed, we can perform the installation from code. To do so, we can add the following code in the `Click` event handler of the `InstallButton`, which will check the current state of the application and install it if needed:

```
private void InstallButton_Click(object sender, RoutedEventArgs e)
{
    if (Application.Current.InstallState ==
        InstallState.NotInstalled)
    {
        Application.Current.Install();
    }
    else if (Application.Current.InstallState ==
        InstallState.InstallFailed)
    {
        InstallErrorTextBlock.Text = "This application failed
            to install, please try again";
    }
    else if (Application.Current.InstallState ==
        InstallState.Installed)
    {
        InstallErrorTextBlock.Text = "Application is already
            installed. Please run offline.";
    }
}
```

- To store the results coming from Twitter, we need the `TweetUpdate` class again. However, this class should now be in the Silverlight project. (In the previous recipe, where we had an intermediate service layer, this class was part of the service project.) The code for this class is as follows:

```
public class TweetUpdate
{
    public string Message { get; set; }
    public string User { get; set; }
    public string Location { get; set; }
}
```

 Note that the `DataContractAttribute` as well as the `DataMemberAttribute` are removed. Both these attributes were needed previously, because instances of this class were used in communication with the intermediate service.

- We're now ready to start the communication with Twitter. If the `IsRunningOfflineAndElevated` Boolean variable is `true`, we can perform a call to Twitter to load the public timeline. This call does not require authentication. Note that we're now writing almost the same code we were writing earlier in the service layer, but now inside the Silverlight application itself. The difference is that now the call to the service happens asynchronously. The following code performs the call to Twitter using a `WebClient` instance, uses LINQ-To-XML to parse the XML and create a `List<TweetUpdates>`:

```
public MainPage()
{
    InitializeComponent();
    CheckApplicationState();
    ChangeUI();
    if (IsRunningOfflineAndElevated)
        GetPublicTimeLine();
}

private List<TweetUpdate> publicTimelineTwitterData;
private void GetPublicTimeLine()
{
    string publicTimeLine =
        "http://twitter.com/statuses/public_timeline.xml";
    WebClient client = new WebClient();
    client.DownloadStringCompleted += new
        DownloadStringCompletedEventHandler
        (client_DownloadStringCompleted);
    client.DownloadStringAsync(new Uri(publicTimeLine,
        UriKind.Absolute));
}
```

```
        }
        void client_DownloadStringCompleted(object sender,
            DownloadStringCompletedEventArgs e)
        {
            XDocument document = XDocument.Parse(e.Result);
            publicTimelineTwitterData =
                (from status in document.Descendants("status")
                 select new TweetUpdate
                {
                    Message = status.Element("text").Value.Trim(),
                    User = status.Element("user").Element("name").Value.Trim()
                }).ToList();
            PublicTimeLineListBox.ItemsSource = publicTimelineTwitterData;
        }
    }
```

7. The application also allows the user to log in. When logged in, the user timeline and friends timeline can be loaded. Both these methods of the Twitter API require that we authorize. Starting with Silverlight 4, we can send credentials when using the WebClient class using its Credentials property. However, sending credentials is only possible when using the ClientHttpStack and not the default BrowserHttpStack. Making Silverlight use the ClientHttpStack is done by using the WebRequest.RegisterPrefix and passing in http://. This code makes sure that all requests over http:// are now executed using the ClientHttpStack. The following code shows the code to retrieve the user timeline (the friends timeline is similar, the code for this can be found in the samples):

```
private void LoginButton_Click(object sender, RoutedEventArgs e)
{
    LoadAuthorizedContent();
}
private List<TweetUpdate> userTimelineTwitterData;
private void LoadAuthorizedContent()
{
    if (UserNameTextBox.Text != string.Empty &&
        PasswordTextBox.Password != string.Empty)
    {
        WebRequest.RegisterPrefix("http://",
            System.Net.Browser.WebRequestCreator.ClientHttp);
        string userTimeLine =
            "http://twitter.com/statuses/user_timeline/"
            + UserNameTextBox.Text + ".xml";
        WebClient client = new WebClient();
        client.Credentials = new NetworkCredential(
            UserNameTextBox.Text, PasswordTextBox.Password);
        client.UseDefaultCredentials = false;
```

```
client.DownloadStringCompleted += new  
    DownloadStringCompletedEventHandler  
    (user_DownloadStringCompleted);  
    client.DownloadStringAsync(new Uri(userTimeLine,  
        UriKind.Absolute));  
}  
}  
void user_DownloadStringCompleted(object sender,  
    DownloadStringCompletedEventArgs e)  
{  
    XDocument document = XDocument.Parse(e.Result);  
    userTimelineTwitterData =  
        (from status in document.Descendants("status")  
         select new TweetUpdate  
        {  
            Message = status.Element("text").Value.Trim(),  
            User = status.Element("user").Element("name").Value.Trim()  
        }).ToList();  
    UserTimeLineListBox.ItemsSource = userTimelineTwitterData;  
}
```

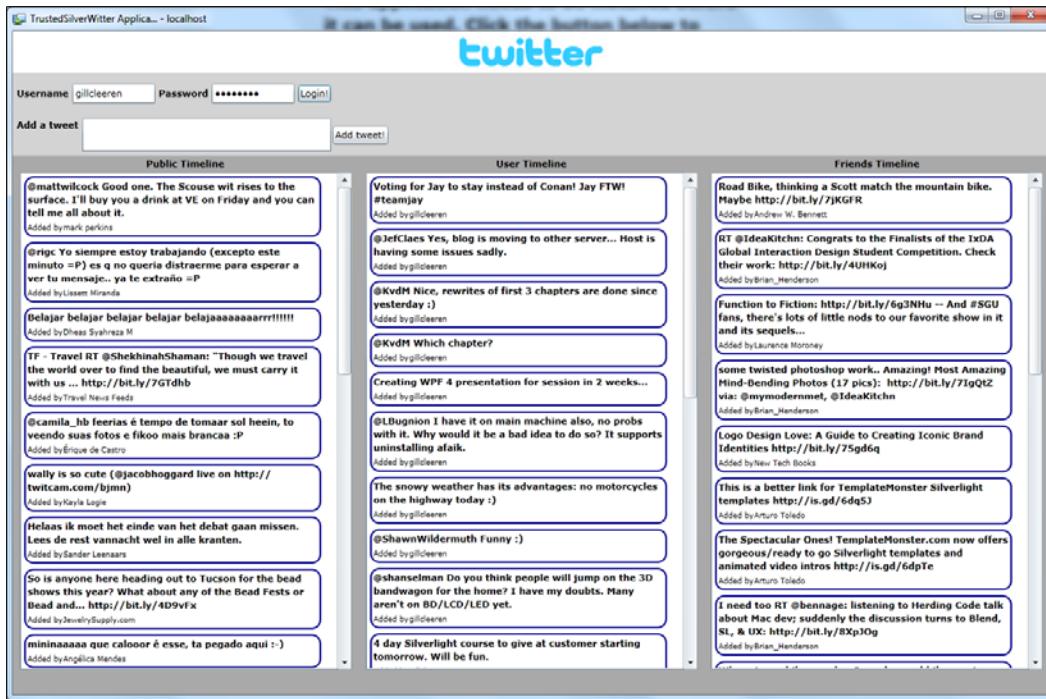
8. To add a message to Twitter from our client, we need to post it using the HTTP POST method. We are using an `HttpWebRequest` for this, and set its method to POST. To this `HttpWebRequest`, we also add the user-entered credentials, because this service method requires authorization as well. Silverlight will then send the message to Twitter:

```
private void AddTweetButton_Click(object sender,  
    RoutedEventArgs e)  
{  
    string uri = @"http://twitter.com/statuses/update.xml";  
    try  
    {  
        string message = AddTweetTextBox.Text.Trim();  
        string parameters = string.Format("status={0}&source={1}",  
            HttpUtility.HtmlEncode(message), "Trusted SilverWitter");  
        WebRequest.RegisterPrefix("http://",  
            System.Net.Browser.WebRequestCreator.ClientHttp);  
        HttpWebRequest request = (HttpWebRequest)  
            WebRequestCreator.ClientHttp.Create  
            (new Uri(uri, UriKind.Absolute));  
        request.Method = "POST";  
        request.Credentials = new  
            NetworkCredential(UserNameTextBox.Text,  
                PasswordTextBox.Password);
```

```
request.ContentType = "application/x-www-form-urlencoded";
request.BeginGetRequestStream(new AsyncCallback(result =>
{
    using (StreamWriter writer =
        new StreamWriter(request.EndGetRequestStream(result)))
    {
        writer.Write(parameters);
    }
    request.BeginGetResponse(response =>
    {
        try
        {
            WebResponse rs = request.EndGetResponse(response);
            Dispatcher.BeginInvoke(LoadAuthorizedContent);
        }
        catch (WebException ex)
        {
            Dispatcher.BeginInvoke(() =>
                HandleError(ex.Message));
        }
    }, request);
}),
null);
}
catch (Exception ex)
{
    Dispatcher.BeginInvoke(() => HandleError(ex.Message));
}
AddTweetTextBox.Text = string.Empty;
}
private void HandleError(string exceptionMessage)
{
    ErrorTextBlock.Text = "An error occurred: " + exceptionMessage;
}
```

9. The code is now ready. However, we still need to configure the application to allow it to run out-of-browser and with elevated permissions. To do so, right-click on the Silverlight project node in the **Solution Explorer** and select **Properties**. In the **Silverlight** tab of the **Properties** window, select the **Enable running application out of the browser** checkbox. Finally, click on the **Out-Of-Browser** settings button on the same tab, and in the resulting dialog, select the **Require elevated trust when running outside the browser** checkbox.

With these steps completed, we have created a standalone Twitter client. Because it runs with elevated permissions, there's no need to add an intermediate service layer between the Silverlight client and Twitter. The running application can be seen in the following image:



How it works...

To build this application, two features added to the platform with the release of Silverlight 4 were put to work: no more cross-domain restrictions when running with elevated permissions and passing credentials using ClientHttp stack. Let's take a look at these in some more detail.

Let's go cross-domain!

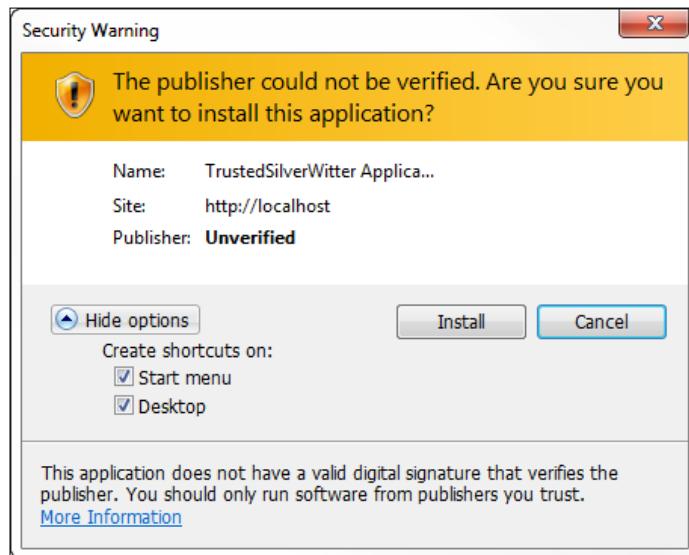
In many recipes in this book, we talked about the cross-domain restrictions that Silverlight has in place. Basically, these come down to Silverlight not allowing us to make requests to services that are not in the same domain as the Silverlight application. Silverlight will make the request only if there's a cross-domain policy file in place that allows the request. Cross-domain restrictions are required for security reasons. We looked at cross-domain issues in depth in *Chapter 7, Working with Services*.

With Silverlight 3, it became possible to create out-of-browser Silverlight applications, allowing us to create standalone Silverlight applications which do not require a browser to be open to run. However, they still run in the sandbox like in-browser applications, meaning these applications do not have extra permissions on the system.

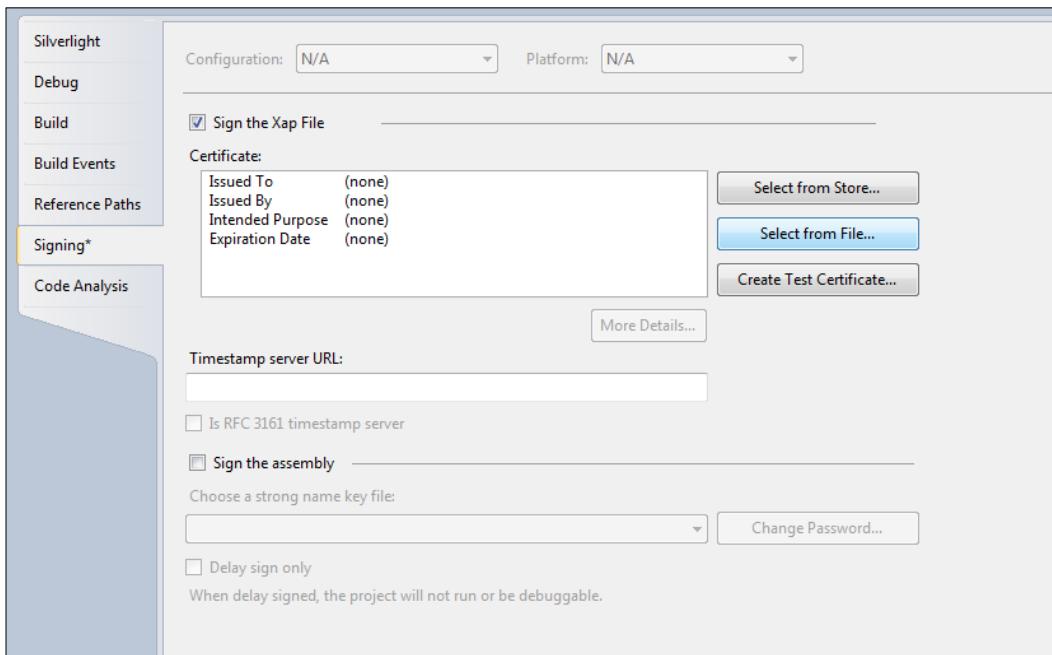
With version 4, it becomes possible to create Trusted Silverlight applications, which are basically out-of-browser applications with elevated permissions. As a result, they have more permissions on the system and can perform some tasks in a different manner. One of these is the ability for this type of applications to perform cross-domain calls without the need of a policy file. Silverlight 5 also makes it possible to run trusted applications in-browser. They have the same permissions and added options, so the recipe could be applied on such an application as well.

For some applications, this is a big plus. Take, for example, our Twitter application. In the in-browser version, which we created in the previous recipe, we had to build a service layer that sits between the Silverlight client and Twitter itself. The reason for this is that Twitter does not expose a policy file, so Silverlight applications can't directly communicate with Twitter's API. With trusted Silverlight applications, the fact that this file isn't there is no problem. When running with elevated permissions, Silverlight will not check for the existence of the file and will perform the service request anyhow.

Applying elevated permissions to Silverlight can be done through Visual Studio. In **Project Properties**, under the **Out-Of-Browser** settings, we can check that the application should request to the user to run with these permissions. On installation, the user will not be prompted with the regular install screen. Instead, the following dialog box shown in the following screenshot, asks the user if he or she fully trusts the application:



Silverlight also gives us the option to sign the XAP file, which results in a more relaxed installation screen being displayed when installing a trusted Silverlight application. To sign the XAP, you need a certificate. Visual Studio 2010 allows signing the XAP from within the IDE itself. To do so, go to the **Project Properties** and select the **Signing** tab (shown as follows) on the left. Here you get the options to use your own certificate, or let Visual Studio create a test certificate for development use.



Pass me those credentials, will you?

Being able to access Twitter without cross-domain restrictions is one thing. We also need to be able to pass credentials to a service when it requires us to. In Silverlight 3, the `WebClient` class already had a `Credentials` property, but this property was not working properly. With Silverlight 4 this changed. It's now possible to pass credentials to a service. One thing that is required is that we use the `ClientHttp` stack.

Passing credentials is very simple and can be done using the following code:

```
WebRequest.RegisterPrefix("http://",
    System.Net.Browser.WebRequestCreator.ClientHttp);
 WebClient client = new WebClient();
 client.Credentials = new NetworkCredential(UserNameTextBox.Text,
    PasswordTextBox.Password);
```

We are specifying that we want the application to use the ClientHttp stack using the `WebRequest.Register` method: basically we're saying for all traffic that goes over `http://`, use the ClientHttp stack.

See also

In the *Working cross-domain from a trusted application* recipe in *Chapter 7, Working with Services*, we looked at working with trusted applications from Silverlight. Working with Twitter was the topic of the previous recipe.

11

Using WCF RIA Services

In this chapter, we will cover the following topics:

- ▶ Setting up a data solution to work with WCF RIA Services
- ▶ Using a WCF RIA Services class library
- ▶ Getting data on the client
- ▶ Using LoadBehavior to control what happens to your data once it's sent to the client
- ▶ Controlling the server-side query from the client
- ▶ Sorting and filtering data on the server
- ▶ Paging through your data
- ▶ Persisting a change set/unit of work
- ▶ Working with concurrency and transactions

Introduction

Microsoft WCF RIA Services is a framework developed to simplify Line of Business RIA development. RIA Services addresses the complexity of building N-tier applications by providing a framework, controls, and services on both your server side (the services, typically .NET-based, hosted in an ASP.NET application, and the application in which your Silverlight client is hosted) and your client side (your Silverlight application).

RIA Services makes it easy to get data from your services to your client. It does this by allowing you to write services linked to a data store (a database, your own classes, an Entity Model, and so on) on your server side, and then regenerates these entities on your client. It also generates the necessary context, methods, and operations on your client to easily talk to your services. In essence, WCF RIA Server is a server-side technology that projects code to a client using WCF as a means of communication between the server and the client. In addition, it makes it easy to add validation and authentication to your services and/or entities.

It has quickly become one of the most-used, popular frameworks for designing LOB applications with Silverlight, as they work nicely together (after all, it was initially conceived to be used in combination with Silverlight). But next to that, it's also used together with other technologies, as WCF RIA Services has SOAP (CRUD) and OData (R) endpoints, so any client, like WPF, WinForms, ASP .NET (WebForms and MVC), Windows Phone, can talk with it and make use of the features it brings. Next to that, there's also a DomainService implementation for Windows Azure (SQL Azure), and a full client-side javascript implementation, RIA/js, to be used with any HTML(5) based client (typically in combination with jQuery) without a line of C# code.

This chapter will teach you how to work with RIA Services, from building a simple service to query data to submitting units of work. More advanced topics are looked at in *Chapter 12, Advanced WCF RIA Services*.

Setting up a data solution to work with WCF RIA Services

Applies to Silverlight 4 and 5

When you want to start working with RIA Services, you'll have to install the SDK and tooling. Besides that, you'll have to think about a few things when starting a new solution to enable WCF RIA Services for that solution. This recipe will guide you through the steps we need to complete to start using WCF RIA Services. Along with that, it will explore some general concepts concerning WCF RIA Services by explaining how we should structure our code.

Getting ready

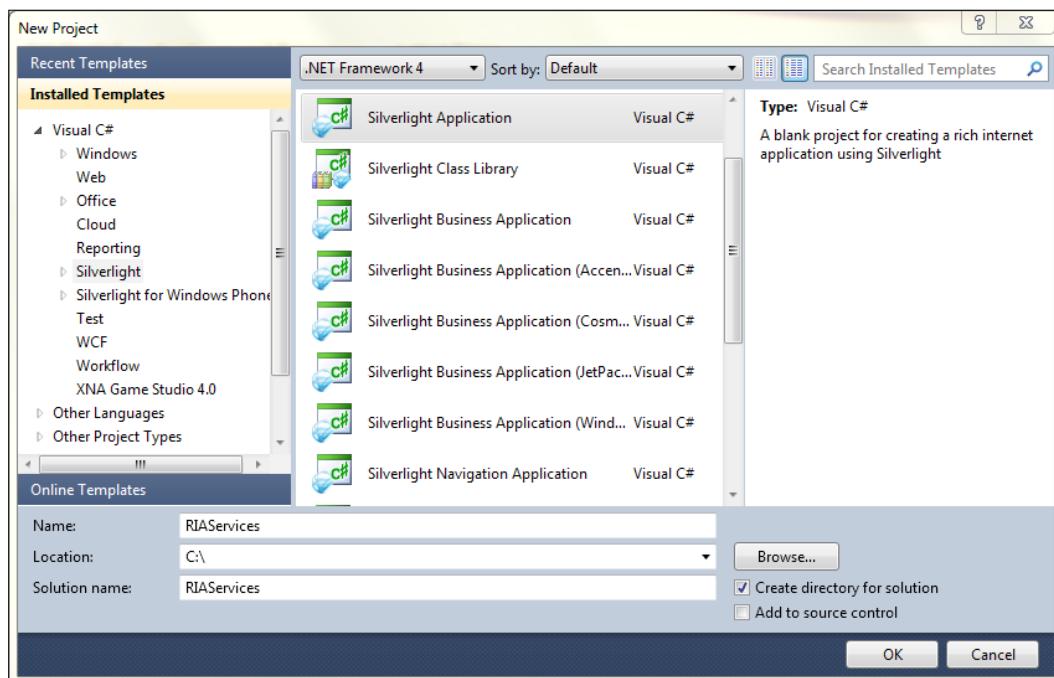
Before you can use RIA Services, the correct SDK has to be installed. WCF RIA Services is distributed together with Silverlight 5 for use with Visual Studio 2010. If you want to download it separately, you can do so at <http://www.microsoft.com/download/en/details.aspx?id=27227>, and at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26939> for the toolkit.

A completed solution can be found in
Chapter 11\Setting_Up_A_Data_Solution_Completed.

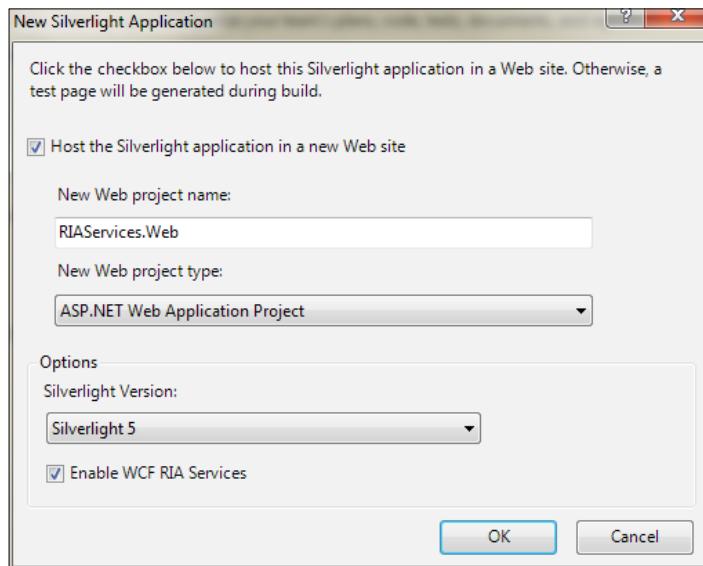
How to do it...

To set up a data solution to work with WCF RIA Services, we'll start with a new solution called **RIAServices**. In order to do this, we need to complete the following steps:

1. Select **File | New Project**, choose a new Silverlight Project, and name the project **RIAServices**, as shown in the following screenshot:



2. After we've done this, we'll be presented with a pop-up screen as shown in the following screenshot:



3. In this screen, we'll need to select the **Enable WCF RIA Services** checkbox and click on the **OK** button. Our WCF RIA Services-enabled solution will be generated for us.

How it works...

The magic happens because we check the **Enable WCF RIA Services** checkbox. By doing this, we tell Visual Studio to form an association between our client (our Silverlight application) and server (our web application) project, so that application logic can be shared between them.

Visual Studio will automatically add references to the necessary assemblies to the client: `System.ComponentModel.DataAnnotations` (used to enable dynamic, data-like annotations to your properties and methods), `System.ServiceModel.DomainServices.Client`, and `System.ServiceModel.DomainServices.Client.Web`.

There's more...

Microsoft WCF RIA Services brings together the **ASP.NET and Silverlight platforms**, thus simplifying the traditional N-tier application pattern. We write our application logic for data access in the middle-tier, typically in our server project, and we can share entities and application logic (data validation, custom methods, and so on) between the tiers easily. This means that we can share entities, code, and validation between our server project and our Silverlight project.

The WCF RIA Services Toolkit

Besides the WCF RIA Services SDK, which is included with Silverlight 5, a WCF RIA Services Toolkit exists. This toolkit contains the classes and namespaces necessary to get WCF RIA Services to work with LINQ to SQL, to create SOAP endpoint, to enable certain collection types for use with MVVM. A few of these will be used in some of the other recipes in this chapter and in *Chapter 12, Advanced WCF RIA Services*.

You can find it at: <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26939>

See also

For a better, reusable solution setup for using domain services, have a look at the next recipe *Using a WCF RIA Services class library*. For querying and submitting data, have a look at the other recipes in this chapter.

Using a WCF RIA Services class library

Applies to Silverlight 4 and 5

Getting started with WCF RIA Services is very easy, as the previous recipe showed you. However, the standard template comes with a few disadvantages: there's no easy way to redistribute your services to other applications, and the services have to be in the exact same place as where your Silverlight application is hosted, so they're part of the same assembly and namespace as your webhost, to name a few.

A WCF RIA Services class library solves this problem. In this recipe, we'll learn how to use a WCF RIA Services library for your domain services.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

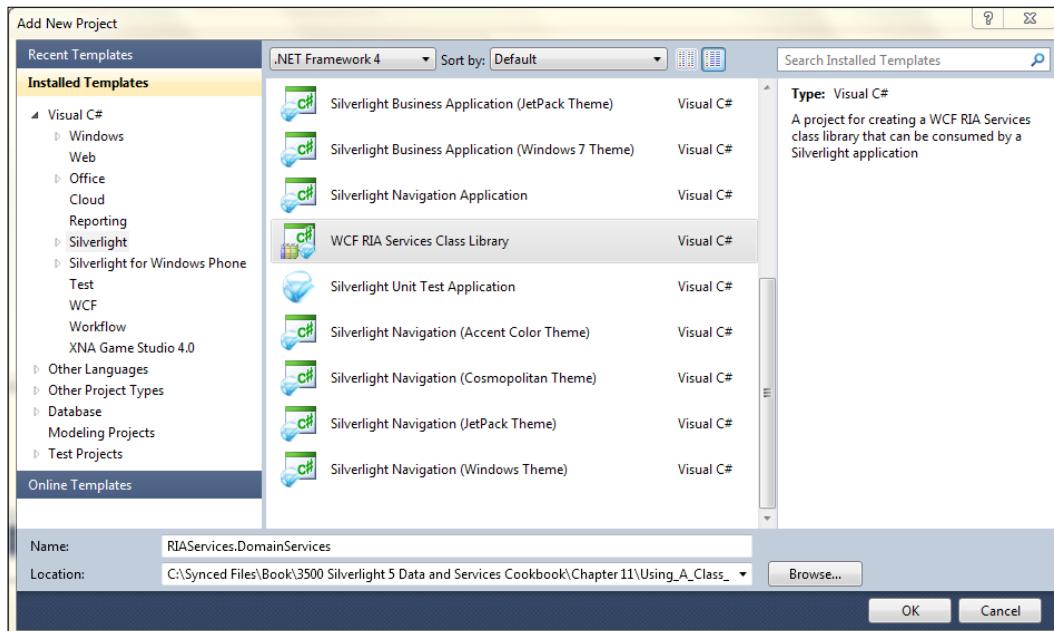
We're starting from a provided started solution, which you can find in `Chapter 11\Using_A_Class_Library_Starter\`.

The completed solution can be found in
`Chapter 11\Using_A_Class_Library_Completed\`.

How to do it...

We're going to enable our solution to use a WCF RIA Services class library, starting from the provided starter solution. In order to do this, we need to complete the following steps:

1. Start by right-clicking the solution, and selecting **Add | New Project**. In the project dialog box, choose **WCF RIA Services Class library**, and name this `RIAServices.DomainServices` as follows:

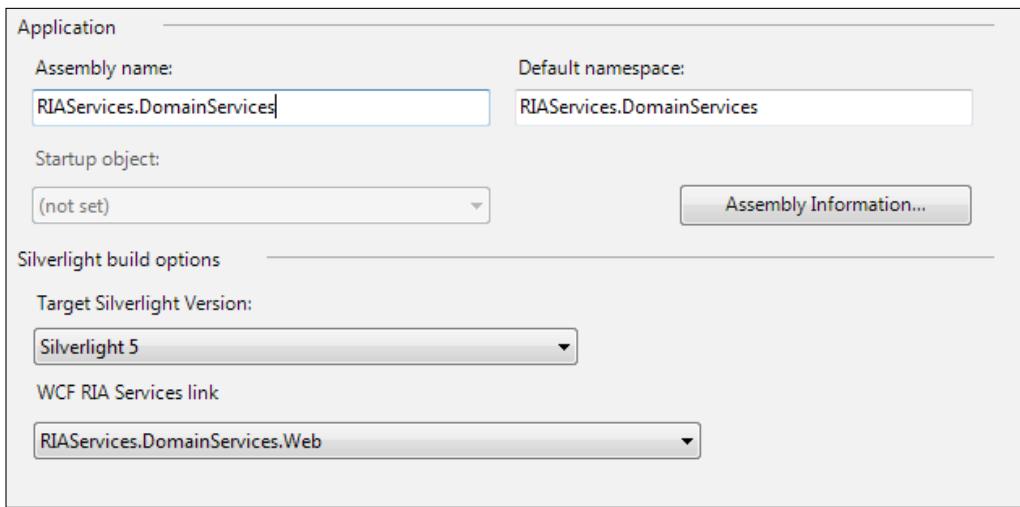


2. Remove `Class1.cs` from both projects.
3. Next, add a reference to `RIAServices.DomainServices` to `RIAServices.Client` (the Silverlight client application), and add references to `System.ComponentModel.DataAnnotations`, `System.ServiceModel`, `DomainServices.Client`, and `System.ServiceModel.DomainServices.Client.Web`.
4. Add a reference to `RIAServices.DomainServices.Web` to `RIAServices.WebHost` (our hosting web project).

With these few steps, we're done; we've now created a **WCF RIA Services Class Library**.

How it works...

When you add a WCF RIA Services class library, for each library two projects are added to your solution: a .NET class library, which will contain your services, and a Silverlight class library, which will contain the code that gets generated on each build. By default, Visual Studio will name the .NET class library "name".Web, and the Silverlight class library "name"—in our case, this means we have a RIAServices.DomainServices.Web project, and a RIAServices.DomainServices project. If you look at the project properties of the RIAServices.DomainServices project, you'll see the **WCF RIA Services link** is set to RIAServices.DomainServices.Web:



This setting ensures that client-side versions of entities, exposed by the domain services in RIAServices.DomainServices.Web and client-side query methods, for each query method that's exposed by those same domain services, are generated in RIAServices.DomainServices. For each domain service, a DomainContext is generated as well.

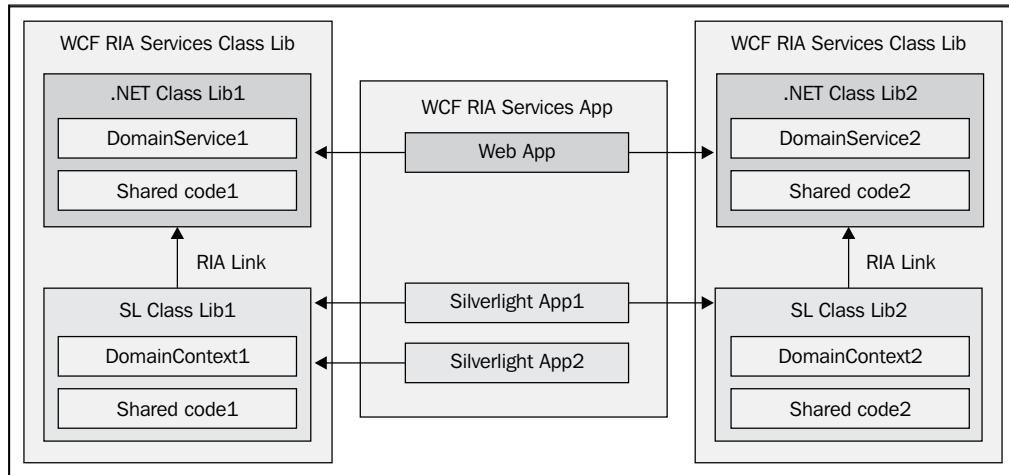
Next, we added a reference to our webhost, RIAServices.DomainServices.Web (so the domain services we'll define are accessible through the webhost).

In our Silverlight project, we do the same, this time adding a reference to the project containing the generated client-side code (RIAServices.DomainServices).

Once these project references are in place, we need to add references to assemblies from the WCF RIA Services SDK: System.ComponentModel.DataAnnotations (used by WCF RIA Services), System.ServiceModel.DomainServices.Client, and System.ServiceModel.DomainServices.Client.Web.

We'll add the SDK assemblies to the webhost, when we're creating our first domain service in the next recipe, *Getting Data on the Client*. With this, everything is in place to use one or more WCF RIA Services Class Libraries.

Organizing your solution like this opens up a range of possibilities when you're working with multiple Silverlight applications, having to reuse the same domain services:



There's more...

We've now got a WCF RIA Services solution up and running using a WCF RIA Services class library. However, you might notice the project naming could be better. Is there anything we can do about that? Also, why is the WCF RIA Services link no longer necessary in the Silverlight client project?

What about the WCF RIA Services link in the Silverlight client?

It is important to notice that the WCF RIA Services link in the project properties of `RIAServices.Client` is empty. Setting this link to a web application tells WCF RIA Services that for the `DomainServices` hosted in the web application, the client side code must be generated in `RIAServices.Client`. In our solution, this is no longer necessary, as the link already exists between `RIAServices.DomainServices.Web` and `RIAServices.DomainServices`.

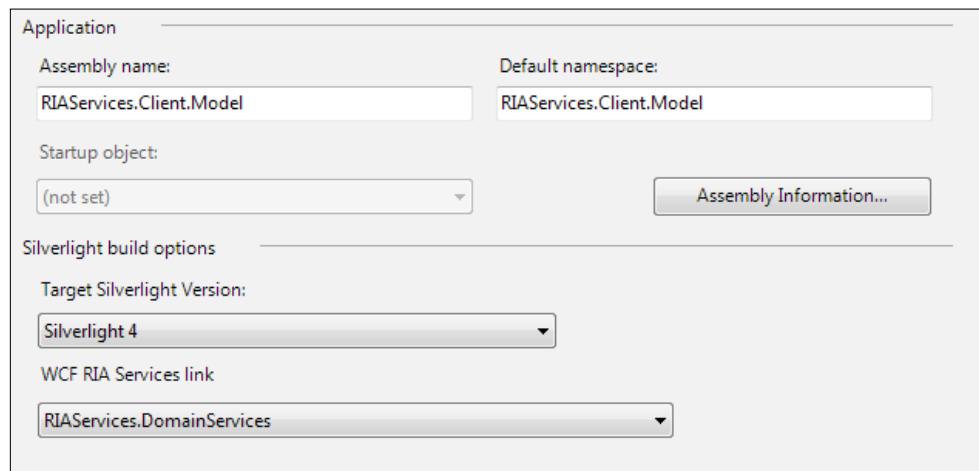
Better naming, no templates

While the templates provided by WCF RIA Services are ok, the naming is often a bit confusing, and not exactly correct, especially if you're working with the MVVM design pattern. In fact, for quite a few MVVM applications, the Silverlight class library named `RIAServices.DomainServices` actually contains the `Model` used by your `ViewModels`. If we could name this accordingly, we'd end up with a better-organized solution.

We can do this quite easily. Instead of using the WCF RIA Services class library template, we can add and name the projects ourselves. In essence, it's just a .NET class library, a Silverlight class library, a WCF RIA Services link, and a few references you'll have to add.

This is how to do it:

1. Add a new Class Library to your solution, naming it `RIAServices.DomainServices`. Remove the default `Class1.cs` file.
2. Add a new Silverlight Class Library to your solution, and name it `RIAServices.Client.Model`. Remove the default `Class1.cs` file.
3. Open the project properties of `RIAServices.Client.Model`, and set the WCF RIA Services link to `RIAServices.DomainServices`. The necessary assemblies will automatically be added:



4. Add a reference to `RIAServices.Client.Model` to `RIAServices.Client`, `System.ComponentModel.DataAnnotations`, `System.ServiceModel.DomainServices.Client`, and `System.ServiceModel.DomainServices.Client.Web`.
5. Add a reference to `RIAServices.DomainServices` to `RIAServicesWebHost` (our hosting web project).

As you can see, we now end up with a better-named, better-structured solution. It's this solution that will be used throughout the next recipes in this chapter.

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*. To learn more about the **MVVM** (**M**odel **V**iew **V**iew**M**odel) design pattern, have a look at *Chapter 6, MVVM*.

Getting data on the client

Applies to Silverlight 4 and 5

In the previous two recipes, we've looked at a few basic RIA Services concepts: the default template and the WCF RIA Services class library template, but we didn't actually fetch or save any data yet.

This recipe will explain what we have to do to design a service using WCF RIA Services to get data from a (database) server to our client (our Silverlight application). It will explain how to create a data store, how to create our first `DomainService` and query method, and how to call this method from our client. It will also explain how this works, thus giving you an insight into the strengths and ease of using WCF RIA Services.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* for more information.

We're starting from a provided started solution, which you can find in `Chapter 11\Getting_Data_On_The_Client_Starter\`.

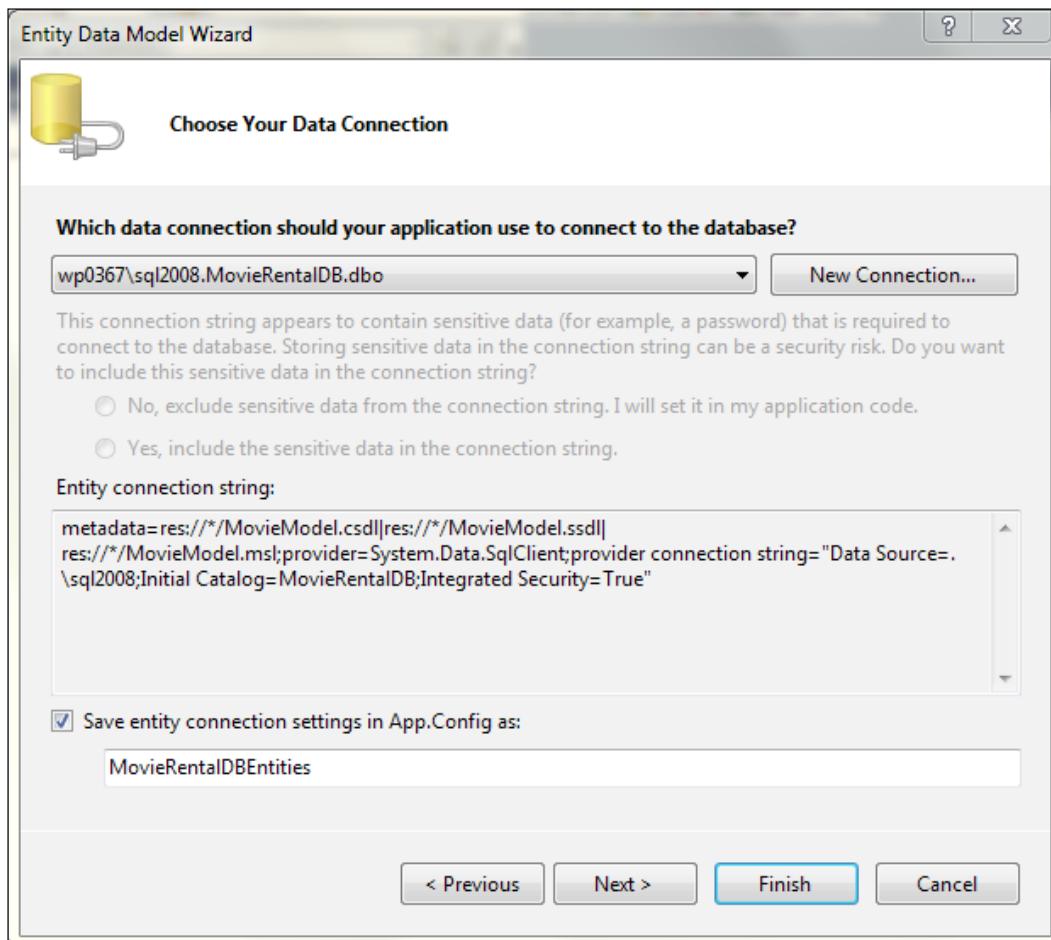
The completed solution can be found in `Chapter 11\Getting_Data_On_The_Client_Completed\`.

How to do it...

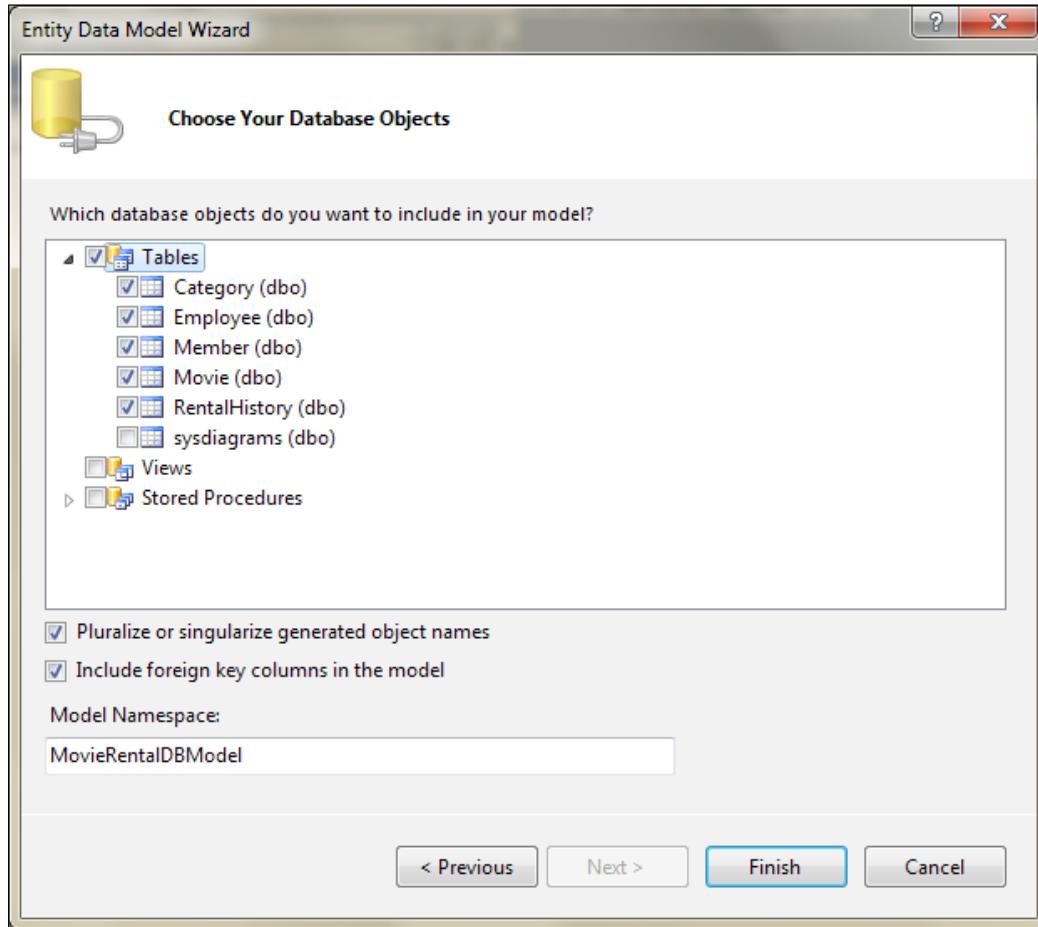
We're going to fetch data from our server to the Silverlight client with WCF RIA Services, starting from the provided starter solution. In order to do this, we need to complete the following steps:

1. Right-click the `RIAServices.DomainServices` project, select **Add New Item**, and add a new ADO .NET Entity Model named `MovieModel.edmx`.

- Follow the **Entity Data Model Wizard**, creating a connection string, which points to the MovieRentalDB database (provided with this recipe), saving the connection string as MovieRentalDBEntities:

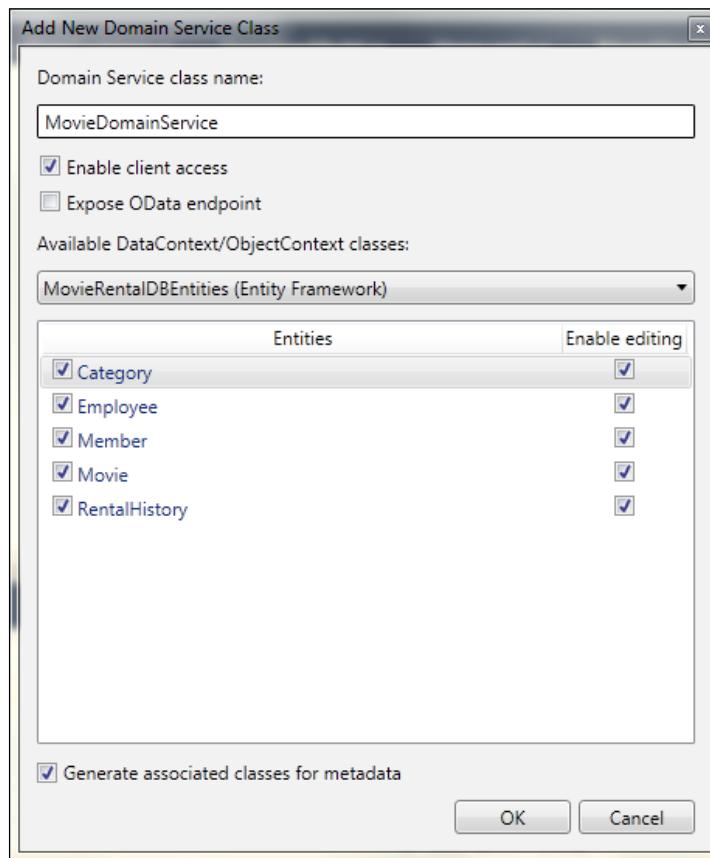


3. In the next screen, select all tables (except sysdiagrams), and make sure you check the **Pluralize or singularize generated object names** and **Include foreign key columns in the model** checkboxes. The Model Namespace should be MovieRentalDBModel. Click **Finish**:



4. Build the solution.
5. Right-click RIAServices.DomainServices, select **Add New Item**, and add a new DomainService, naming it MovieDomainService.

6. In the next screen, select each checkbox, including **Generate associated classes for metadata**, and click **OK**:



7. Rebuild the solution.
8. Locate the configuration tag of the web.config file of RIAServicesWebHost, and replace the code in it with this code:

```

<system.webServer>
    <modules runAllManagedModulesForAllRequests="true">
        <add name="DomainServiceModule"
preCondition="managedHandler"
            type="System.ServiceModel.DomainServices.Hosting.
DomainServiceHttpModule, System.ServiceModel.DomainServices.
Hosting, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856
ad364e35" />
    </modules>
    <validation validateIntegratedModeConfiguration="false" />
</system.webServer>

```

```
<connectionStrings>
    <add name="MovieRentalDBEntities" connectionString="metadata=res://*/MovieModel.csdl|res://*/MovieModel.ssdl|res://*/MovieModel.msl;provider=System.Data.SqlClient;provider connection string="data source=.\sql2008;initial catalog=MovieRentalDB;integrated security=True;multipleactiveresultsets=True;App=EntityFramework"; providerName="System.Data.EntityClient" />
</connectionStrings>
<system.serviceModel>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true" multipleSiteBindingsEnabled="true" />
</system.serviceModel>
<system.web>
    <httpModules>
        <add name="DomainServiceModule" type="System.ServiceModel.DomainServices.Hosting.DomainServiceHttpModule, System.ServiceModel.DomainServices.Hosting, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
    </httpModules>
</system.web>
```

9. Add references to `System.ComponentModel.DataAnnotations`, `System.ServiceModel.DomainServices.EntityFramework`, `System.ServiceModel.DomainServices.Hosting`, and `System.ServiceModel.DomainServices.Server` to `RIAServices.WebHost`.
10. Open `LocalStorageContainer.cs` in `RIAServices.Client`, and add the following code:

```
private static MovieDomainContext _movieContext;
public static MovieDomainContext MovieContext
{
    get
    {
        if (_movieContext == null)
            _movieContext = new MovieDomainContext();

        return _movieContext;
    }
}
```

Open `HomeViewModel.cs`, and add the following properties:

```
public MovieDomainContext MovieContext
{
```

```

get
{
    return LocalStateContainer.MovieContext;
}
}
public EntitySet<Movie> Movies
{
get
{
    return MovieContext.Movies;
}
}
}

```

11. Add the `LoadData()` method to `HomeViewModel.cs`, and call it from the constructor:

```

public HomeViewModel()
{
    LoadData();
}
private void LoadData()
{
    var loadOp = MovieContext.Load<Movie>(MovieContext.
GetMoviesQuery());
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}

```

12. You can now build and run the solution:

Title	Director	Category	Date published
Fear and Loathing in Las Vegas	Terry Gilliam		5/22/1998
Apocalypse Now	Francis Ford Coppola		8/15/1979
Cidade de Deus	Fernando Meirelles		5/7/2003

How it works..

First of all, we add an Entity Model to enable us to query the `ObjectContext` for data through the **Entity Framework ORM**. This is not WCF RIA Services-specific. Entity Framework is the ORM of choice for .NET projects, and is used in a diversity of applications. On the other hand, WCF RIA Services isn't tied to the Entity Framework either, you can use it with LINQ To SQL (via the Toolkit), with DTO's, with Azure.

The `DomainService` class we added next, `MovieDomainService`, is the class that provides a public interface to the mid-tier data: this is our service. When we add a `DomainService` class, code will be generated for all the entities we've selected in step 6, including metadata (see further in this recipe for an explanation on this) if we check the checkbox. Mind you, this is the code that gets generated once, by completing the **Add DomainService** wizard. It does not get updated automatically when you change something in your Entity Model, nor when you build, it just helps you to get started quickly. That said, a `DomainService`, as any service, should not expose any operations that aren't needed, so make sure you don't check/expose entities that aren't required on the client. For that exact reason, quite a few developers are hesitant to use the wizard, as it often creates more code than is needed (and as we know, code cleanup isn't always a priority when a project reaches its deadline). However, for this example, this is a good starting point.

This is how the used parts of our `DomainService` and query look:

```
[EnableClientAccess()]
public class MovieDomainService : LinqToEntitiesDomainService<MovieRen
talDBEntities>
{
    public IQueryable<Movie> GetMovies()
    {
        return this.ObjectContext.Movies;
    }
}
```

When we added the `MovieDomainService` class, we signaled that it should get its data from an Entity Model. Due to this, the `MovieDomainService` class inherits from `LinqToEntitiesDomainService` (a built-in abstract class). We also chose to expose the `Movie` entity. Hence, the **Add DomainService** wizard generated a `GetMovies()` method for us that obtains data from the `MovieModel` Context.

This `DomainService` is typically the starting point for our application logic. In this service, we might want to add new methods, change the parameters or the body of existing methods, or include other application logic.

When we built our solution, a lot of things happened on the client as well, in `RIAServices.Client.Model` (the client part of our WCF RIA Services class library). As the `MovieDomainService` is marked with the `[EnableClientAccess()]` attribute, proxy code will be automatically generated in our Silverlight library on each build. When we click on the **Show All Files** icon in the **Solution Explorer**, we can see this code by opening the `RIAServices.DomainServices.g.cs` file.

If we open the generated file, we see that a client-side version of our server-side entity, `Movie`, has been generated (you'll also notice that it raises `INotifyPropertyChanged` notifications in the setter of its properties, so they are data-bindable, and validation occurs (more on that in the next chapter, *Chapter 12, Advanced WCF RIA Services*)). We also see that a class called `MovieDomainContext`, which inherits from `DomainContext`, has been generated.

In this class, methods are generated to load the data from our database into the `EntityCollections` on the `DomainContext`. In this recipe, this method is called `GetMoviesQuery()`, and it returns an `EntityQuery<Movie>` object. Similar methods will be created for each query method. (A query method is typically a method on our `DomainService` that returns a collection of entities, generally as `IEnumerable<T>` or `IQueryable<T>`). These methods can be used as arguments on the `Load` method of our context. Passing one of these methods will execute the `EntityQuery` returned by it.

To use a metaphor to make this a bit clearer, the `DomainContext` can be seen as the mechanism that moves entities between the `DomainService` and the `EntitySets`. These `EntitySets` are stored in the `EntityContainer`. You could view the `EntityContainer` as the database (on the client), the `EntitySets` as the tables, and the `DomainContext` as the database server.

In the next 2 steps, we did two things: we edited the `web.config` file of our host, which is necessary as we're using a WCF RIA Services class library, the registering of assemblies and definition of the EF (Entity Framework) connection string is put in the `app.config`, but we're referencing this assembly in our `WebHost`, so we need that configuration to be in the configuration file used by the `WebHost`: the `web.config` file. And of course, we also need to add the WCF RIA Services assemblies to the `WebHost`. You'll also have to make sure you update the connection string to your own connection string.

That's it for the server side, on to the client. First of all, we add an instance of the `DomainContext`, `MovieDomainContext`, to a `LocalStorageContainer` class, this enables us to easily use this instance across `ViewModels`. As you can see, `HomeViewModel` receives a property `MovieContext`, which refers to the instance in `LocalStorageContainer`. The other property, `Movies`, is bound to the `DataGrid` in XAML, and refers to the `EntitySet` of `Movies` from the `MovieDomainContext`. This is where the loaded movies from the server will end up in, so once we fetch them from the server, the `EntitySet` will be filled with movies, which will then be reflected in our UI.

Only one thing left, actually fetching the data:

```
private void LoadData()
{
    var loadOp = MovieContext.Load<Movie>(MovieContext.GetMoviesQuery());
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
```

So, what happens when we execute this statement? We want to load `Movies` from the server to our client, so we execute a `MovieContext.Load<Movie>` call. In this call, we define how we want to load these movies by using the `MovieContext.GetMoviesQuery()`. This is an `EntityQuery`, generated on the client (you can find it in `RIAService.DomainServices.g.cs`), which will execute the `GetMovies` method on our `MovieDomainService`. Once it's been executed, the returned `Movie` entities will be available through the `MovieContext.Movies EntitySet`.

What about that `loadOp` variable? When we call `MovieContext.Load<Movie>`, a `LoadOperation<Movie>` is returned, which we store in the `loadOp` variable. This variable can be used to execute code after the movies have been fetched from the server (asynchronously, as every service call in Silverlight). It contains the loaded Entities, but it also contains information on errors during the load (if any), and can be used to cancel the load operation (for example, if it takes too long).

There's more...

WCF RIA Services offers you quite a few options on how to control what's sent over the wire. Have a look at the following sections for more information on that.

Some details on code generation

As you've noticed, a lot of code gets automatically generated on our client when we use WCF RIA Services. Working with RIA Services almost feels like we're not using services to get our data, seeing that we've got a client-side representation of all our entities and our context.

RIA Services automatically generates this client-side `EntityQuery`, returning methods for every query method that will have the same name as the server-side method, postfixed with `Query`. Server-side methods whose names are prefixed with `Get`, `Fetch`, `Find`, `Query`, `Retrieve`, or `Select`, and which return an `IEnumerable<T>` or `IQueryable<T>` are `Query` methods that will result in a client-side `EntityQuery` returning method. We can also mark these methods with the `[Query]` attribute to specifically tell the code generator that a method is a `Query` method.

An important fact to notice is that method overloading is not possible due to automatic generation of Load methods for our query methods.

Aren't there some properties missing?

As you might have noticed, the Category and RentalHistory of a Movie isn't visible in our DataGrid, it's not sent to the client. Why is this? And how can we make sure they are?

1. Create a new query method in MovieDomainService.cs:

```
public IQueryable<Movie> GetMoviesWithCategoryAndRentalHistory()
{
    return this.ObjectContext.Movies.Include("Category") .
Include("RentalHistories");
}
```

2. Open MovieDomainService.metadata.cs, and locate the MovieMetaData class. In this class, add the Include attribute to the Category and RentalHistories properties:

```
[Include]
public Category Category { get; set; }

[Include]
public EntityCollection<RentalHistory> RentalHistories { get; set;
}
```

3. Open HomeView.xaml, and change the DataGrid declaration to include a DetailTemplate:

```
<sdk:DataGrid.RowDetailsTemplate>
    <DataTemplate>
        <ItemsControl ItemsSource="{Binding RentalHistories}" Margin="40,0,0,0">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="100"/>
                            <ColumnDefinition Width="100"/>
                            <ColumnDefinition Width="100"/>
                            <ColumnDefinition Width="100"/>
                            <ColumnDefinition Width="100"/>

```

```
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="30"></
RowDefinition>
</Grid.RowDefinitions>
<TextBlock Text="Out: "
    HorizontalAlignment="Left"></
TextBlock>
<TextBlock Text="Returned: "
    Grid.Column="2"
    HorizontalAlignment="Left"></
TextBlock>
<TextBlock Text="{Binding DateOut,
StringFormat=d}"
    HorizontalAlignment="Left"
    Grid.Column="1"></TextBlock>
<TextBlock Text="{Binding DateReturned,
StringFormat=d}"
    HorizontalAlignment="Left"
    Grid.Column="3"></TextBlock>
</Grid>
</DataTemplate>
</ItemsControl.ItemTemplate>
</ItemsControl>
</DataTemplate>
</sdk:DataGrid.RowDetailsTemplate>
```

4. Build the solution.

5. Open `HomeViewModel.cs`, and change the `LoadData()` method, so it now calls the query we've created in step 1:

```
private void LoadData()
{
    var loadOp = MovieContext.Load<Movie>
        (MovieContext.
GetMoviesWithCategoryAndRentalHistoryQuery());
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
```

If we now build and run our solution, we can see the `Category` and `RentalHistory` objects, belonging to a specific movie, are sent to the client as well.

RIA SERVICES			
MOVIE RENTAL MANAGEMENT SYSTEM			
MOVIE OVERVIEW			
Title	Director	Category	Date published
Fear and Loathing in Las Vegas	Terry Gilliam	Adventure	5/22/1998
Out:	4/20/2011	Returned:	4/22/2011
Apocalypse Now	Francis Ford Coppola	Drama	8/15/1979
Out:	4/20/2011	Returned:	4/22/2011
Out:	6/18/2011	Returned:	6/19/2011
Cidade de Deus	Fernando Meirelles	Drama	5/7/2003
Out:	6/12/2011	Returned:	6/13/2011

By default, WCF RIA Services sends the simple properties of an Entity to the client when that Entity is requested, but this can be controlled through the metadata classes. For each Entity, a metadata class was generated when we created our `DomainService` through the wizard (you can always create these manually as well, of course). On this metadata class, we can use the `[Include]` attribute to decorate non-simple properties, like `EntityCollections`. We did this on the non-simple `Category` type property, and on the `EntityCollection<RentalHistory>` type property, this tells WCF RIA Services to send these over the wire as well when a `Movie` Entity is requested.

With that, WCF RIA Services knows what to do, but of course we're not finished yet, if we want to send the `Category` and `RentalHistories` of a `Movie` to the client, we need to fetch them from our database, that's what happens in the `GetMoviesWithCategoryAndRentalHistories` method. In this method, we use Entity Framework to fetch the `Movies`, including their respective `Category` and `RentalHistories`. These two steps are all it takes to ensure the data you need is sent to the client.

Excluding some properties

We've learned how to include EntityCollections with a service call, but what about excluding properties (as said, WCF RIA Services will send simple properties over the wire by default)? An Entity might have some properties you don't want to send over the wire—take, for example, a password property for a user in a limited user management view, or imagine you're writing an application, which will be used over slow network connections, where every byte counts.

Well, this is possible as well, by using the `[Exclude]` attribute. Much like the `[Include]` attribute, which tells WCF RIA Services to include a certain property, the `[Exclude]` attribute tells it to ignore it. It will not be sent to the client when the service method is executed.

If that still doesn't fit the requirements, another approach could be to write your own DTO classes to send over the wire, instead of the Entities from your Entity Model. Yet another approach could be to create subclasses of entities, straight on your Entity Model, and send those over the wire.

Explaining all these options in detail would take us a bit too far, but they are valid as well, as usual, there's more than one way to solve a specific problem.

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Using LoadBehavior to control what happens to your data once it's sent to the client

Applies to Silverlight 4 and 5

Fetching data from the server is one thing, but as we're working in a disconnected environment, we need a way to control what happens to entities when they arrive on the client, what if a version of that same Entity already exists on the client? Should it be ignored? Merged? Overwritten?

In this recipe, we'll learn how to deal with these situations.

Getting ready

Before getting started, you have to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

We're starting from a provided starter solution, which you can find in Chapter 11\Using_LoadBehavior_Starter\.

The completed solution can be found in Chapter 11\Using_LoadBehavior_Completed\.

How to do it...

We're going to use the `LoadBehavior` parameter to decide what should happen with data once it's been sent to the client, starting from the provided starter solution. In order to do this, we need to complete the following steps:

1. Open `Home.xaml`, and locate the `StackPanel` named `ContentStackPanel`. In this `StackPanel`, add the following xaml before the closing tag:

```
<StackPanel Orientation="Horizontal" Margin="0,10,0,0">
    <Button Height="30"
        Width="150"
        Margin="0,0,10,0"
        Content="Keep current"
        Command="{Binding
LoadWithKeepCurrentLoadBehaviorCommand}"></Button>
    <Button Height="30"
        Width="150"
        Margin="0,0,10,0"
        Content="Merge into current"
        Command="{Binding
LoadWithMergeIntoCurrentLoadBehaviorCommand}"></Button>
    <Button Height="30"
        Width="150"
        Margin="0,0,10,0"
        Content="Refresh current"
        Command="{Binding
LoadWithRefreshCurrentLoadBehaviorCommand}"></Button>
</StackPanel>
```

2. Open `HomeViewModel.cs`, and add the following command definitions to the class:

```
public ICommand LoadWithKeepCurrentLoadBehaviorCommand
{ get; private set; }
public ICommand LoadWithMergeIntoCurrentLoadBehaviorCommand
{ get; private set; }
public ICommand LoadWithRefreshCurrentLoadBehaviorCommand
{ get; private set; }
```

3. Add the following using statements to `HomeViewModel.cs`:

```
using System.Windows.Input;
using GalaSoft.MvvmLight.Command;
```

4. Add the `InstantiateCommands()` method to this class, and call it from the constructor:

```
public HomeViewModel()
{
    InstantiateCommands();
}

private void InstantiateCommands()
{
    LoadWithKeepCurrentLoadBehaviorCommand = new RelayCommand(() 
        => LoadWithKeepCurrentLoadBehavior());
    LoadWithMergeIntoCurrentLoadBehaviorCommand = new 
    RelayCommand(() 
        => LoadWithMergeIntoCurrentLoadBehavior());
    LoadWithRefreshCurrentLoadBehaviorCommand = new 
    RelayCommand(() 
        => LoadWithRefreshCurrentLoadBehavior());
}
```

5. Implement `LoadWithKeepCurrentLoadBehavior()` to execute loading of the data with the `Keep Current LoadBehavior`:

```
private void LoadWithKeepCurrentLoadBehavior()
{
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesWithCategoryAndRentalHistoryQuery(),
        LoadBehavior.KeepCurrent, false);
    loadOp.Completed += (send, args) =>
    {
        // handle errors etc
    };
}
```

6. Implement `LoadWithMergeIntoCurrentLoadBehavior()` to execute loading of the data with the `Merge Into Current LoadBehavior`:

```
private void LoadWithMergeIntoCurrentLoadBehavior()
{
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesWithCategoryAndRentalHistoryQuery(),
        LoadBehavior.MergeIntoCurrent, false);
    loadOp.Completed += (send, args) =>
    {
        // handle errors etc
    };
}
```

7. Implement `LoadWithRefreshCurrentLoadBehavior()` to execute loading of the data with the Refresh Current LoadBehavior:

```
private void LoadWithRefreshCurrentLoadBehavior()
{
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesWithCategoryAndRentalHistoryQuery(),
        LoadBehavior.RefreshCurrent, false);
    loadOp.Completed += (send, args) =>
    {
        // handle errors etc
    };
}
```

8. You can now build and run your solution (note: ensure you've updated the `web.config` file in `RIAServices.WebHost` with the correct connection string). To test the different load behaviors, try them before/after updating items in both the `DataGrid` and your database.

The screenshot shows a user interface for a movie rental system. At the top, there's a green header bar with the text "RIA SERVICES". Below it, a white section titled "MOVIE RENTAL MANAGEMENT SYSTEM" contains the heading "MOVIE OVERVIEW". Three buttons are displayed horizontally: "Keep current", "Merge into current" (which is highlighted with a green border), and "Refresh current". Below these buttons is a DataGrid with the following columns: Title, Director, Category, and Date published. The DataGrid displays movie information, including rental details like Out and Returned dates. One specific row for the movie "Apocalypse Now Edited" is highlighted with a green background, showing its director as Francis Ford Coppola and its category as Drama. The DataGrid also lists other movies such as "Fear and Loathing in Las Vegas" and "Cidade de Deus".

Title	Director	Category	Date published
Fear and Loathing in Las Vegas	Terry Gilliam	Adventure	5/22/1998
Out: 4/20/2011	Returned: 4/22/2011		
Apocalypse Now Edited	Francis Ford Coppola	Drama	8/15/1979
Out: 4/20/2011	Returned: 4/22/2011		
Out: 6/18/2011	Returned: 6/19/2011		
Cidade de Deus	Fernando Meirelles	Drama	5/7/2003
Out: 6/12/2011	Returned: 6/13/2011		

How it works...

We've started out by defining three buttons in XAML, binding their `Command` property to the commands we defined on our `ViewModel`. These will be executed when we click the respective buttons. Each command represents a specific `LoadBehavior`. These `LoadBehaviors` can be added as a parameter to the `Load<>` method on the `DomainContext`:

```
MovieContext.GetMoviesWithCategoryAndRentalHistoryQuery() ,  
LoadBehavior.KeepCurrent, false);
```

If you don't specify a `LoadBehavior`, the default behavior (`LoadBehavior.KeepCurrent`) is used.

Load behaviors are used to control what happens with the entities you're loading once they reach the client, and the same entities (by their `Key` value) are already available on the client.

If you've specified the `LoadBehavior` to be `LoadBehavior.KeepCurrent`, server-side changes to the entities that already were available on the client will not be reflected in these entities, the currently available versions on the client will keep on being used.

If the `LoadBehavior` is set to `LoadBehavior.RefreshCurrent`, the entities that are fetched from the server will be used, no matter what. This ensures you'll always have the latest version of an entity, but it also implies that you'll lose client-side changes on an entity if you haven't persisted to the server before executing a new `Load`.

If the `LoadBehavior` is set to `LoadBehavior.MergeIntoCurrent`, the entity you've just loaded from the server will be compared to the corresponding entity on the client on a property-level. Properties that have been edited on the client will keep their value, while the other properties will be set to the new values from the server.

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Controlling the server-side query from the client

Applies to Silverlight 4 and 5

Controlling the data that's fetched is a very common requirement, instead of fetching all entities of a certain type, you might only want to fetch those that have been changed in the last week. Or you might want to fetch a specific employee from an employee database, instead of fetching all employees from the server.

In the next recipe, *Sorting and filtering data on the server*, we'll look into doing this by writing extra query methods on our `DomainService`. In this recipe, we'll learn how to do this (for the most common requirements) straight from the client, without having to write extra code on the server.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* for more information.

We're starting from a provided starter solution, which you can find in Chapter 11\Controlling_Query_Starter\.

The completed solution can be found in Chapter 11\Controlling_Query_Completed\.

How to do it...

We're going to write code to control what's returned from the server, on the client. To achieve this, we'll complete the following steps:

1. Open Home.xaml, and locate the StackPanel named ContentStackPanel. In this StackPanel, add the following xaml before the closing tag:

```
<StackPanel Orientation="Horizontal"
    Margin="0,10,0,0">
    <Button Content="Order movies"
        Width="150"
        Height="30"
        Command="{Binding LoadOrderedMoviesCommand}"></Button>
    <Button Content="Filter movies"
        Width="150"
        Height="30"
        Margin="10,0,0,0"
        Command="{Binding LoadFilteredMoviesCommand}"></
    Button>
</StackPanel>
```

2. Open HomeViewModel.cs, and add the following command definitions to the class:

```
public ICommand LoadOrderedMoviesCommand
{    get;    private set; }

public ICommand LoadFilteredMoviesCommand
{    get;    private set; }
```

3. Add the following using statements to HomeViewModel.cs:

```
using System.Windows.Input;
using GalaSoft.MvvmLight.Command;
```

4. Instantiate the commands, and call the `InstantiateCommands()` method from the constructor:

```
public HomeViewModel()
{
    InstantiateCommands();
}
private void InstantiateCommands()
{
    LoadOrderedMoviesCommand = new RelayCommand(() =>
LoadOrderedMovies());
    LoadFilteredMoviesCommand = new RelayCommand(() =>
LoadFilteredMovies());
}
```

5. Implement the commands as follows:

```
private void LoadFilteredMovies()
{
    // clear the context for this demo
    MovieContext.Movies.Clear();
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesQuery().Where<Movie>(m => m.Director
== "Francis Ford Coppola"));
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
private void LoadOrderedMovies()
{
    // clear the context for this demo
    MovieContext.Movies.Clear();
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesQuery().OrderBy(m => m.Title));
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
```

6. You can now build and run the application (note: ensure you've updated the web.config file in RIA Services .WebHost with the correct connection string). Clicking the **Order** button will return the list of movies, ordered by Title, and clicking the **Filter** button will return a filtered collection.

Title	Director	Category	Date published
Apocalypse Now	Francis Ford Coppola		8/15/1979
Cidade de Deus	Fernando Meirelles		5/7/2003
Fear and Loathing in Las Vegas	Terry Gilliam		5/22/1998

How it works...

In steps 1 through 4, we've added the necessary XAML code for our buttons, binding them to commands we defined on our ViewModel, so these commands will be executed once the button is clicked.

The magic happens in step 5:

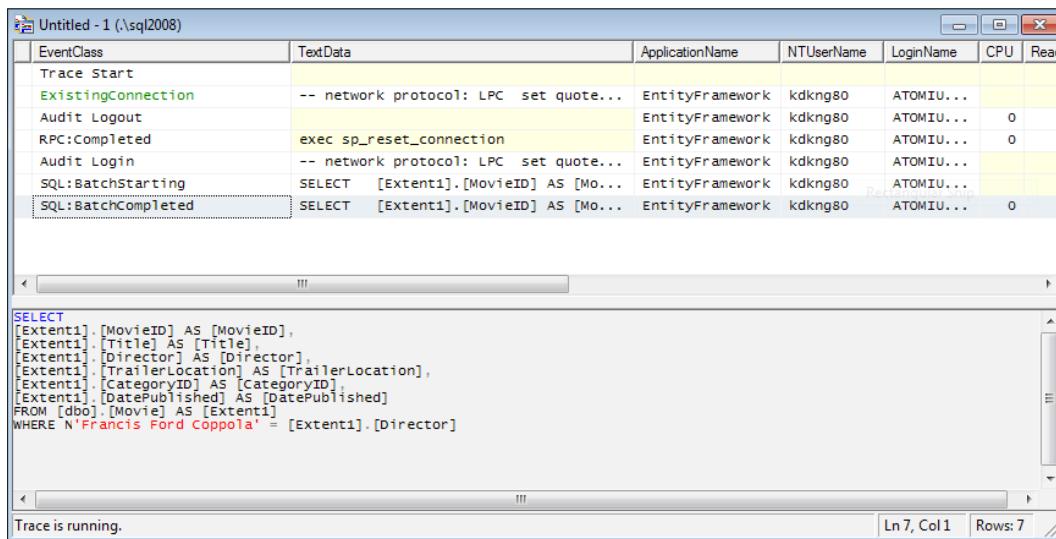
```
MovieContext.Load<Movie>(
    MovieContext.GetMoviesQuery().Where<Movie>(m => m.Director ==
    "Francis Ford Coppola"));
```

Here, we define we want to load movies, using the `GetMoviesQuery()` method. This (auto-generated) method, returning an `EntityQuery<Movie>`, will start loading Movies through the `GetMovie` method on our `MovieDomainService`.

The special part is the `.Where<Movie>` clause we add to the `EntityQuery<Movie>` object that's returned by `GetMoviesQuery()`. In this clause, we define a filter, we only want to load movies directed by **Francis Ford Coppola**.

The special thing about this is that unlike one would expect, this Where clause is reflected straight down to the query that's executed on the database. We're on the client, adding a Where clause to an `EntityQuery`, and this Where clause is executed on the server. We're not fetching all Movies and then filtering them on the client, we're actually only fetching the movies that correspond to the filter.

This can be seen by firing up **SQL Profiler** and looking at the query that's executed against our DB:



It's WCF RIA Services that takes care of this - it combines the server-side method, `GetMovies` (which returns `this.ObjectContext.Movies`), with the `EntityQuery` extension (the `Where` clause) we've defined on the client.

In the other command we use the same technique, this time sorting the movies by title.

There's more...

Not everything is controllable from the client in this way, only the most commonly used LINQ operators are supported. If more control is needed, you'll have to write server-side code on the `DomainService` (have a look at the next recipe, *Sorting and filtering data on the server*, to learn how to do this).

All `EntityQuery` extensions are defined in the `System.ComponentModel.`
`DomainServices.Client` namespace (so you'll have to import this if you want to use them), and these are the ones that are supported:

- ▶ `OrderBy`, `ThenBy`, `OrderByDescending`, `ThenByDescending`: Used to sort data
- ▶ `Where`: Used to filter data
- ▶ `Skip`, `Take`: Used to skip over entities from the result set, or to select a specific amount of entities
- ▶ `Select`: Used to apply a selector to the `EntityQuery` result

See also

Have a look at the next recipe, *Sorting and filtering data on the server*, to learn how to sort and filter without relying on the client. To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Sorting and filtering data on the server

Applies to Silverlight 4 and 5

In almost every application, you'll run into having to enable sorting and/or filtering, for example, instead of fetching all employees from an employee database, you might only want to fetch one specific (set of) employees, order by date of birth.

In this recipe, you'll learn how to achieve this by extending your domain service with extra query operations.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

We're starting from a provided started solution, which you can find in `Chapter 11\Sorting_Filtering_Server_Starter\`.

The completed solution can be found in
`Chapter 11\Sorting_Filtering_Server_Completed\`.

How to do it...

We're going to sort and filter the data that's being returned to our Silverlight application. To achieve this, we'll complete the following steps:

1. Open `Home.xaml` and locate the `StackPanel` named `ContentStackPanel`. In this `StackPanel`, add the following xaml before the closing tag:

```
<StackPanel Orientation="Horizontal"
            Margin="0,10,0,0">
    <Button Content="Order movies"
           Width="150"
           Height="30"
           Command="{Binding LoadOrderedMoviesCommand}"></Button>
    <Button Content="Filter movies"
```

```
        Width="150"
        Height="30"
        Margin="10,0,0,0"
        Command="{Binding LoadFilteredMoviesCommand}"></
Button>
</StackPanel>
```

2. Open MovieDomainService.cs in RIAServices.DomainServices, and add the following two methods:

```
public IQueryable<Movie> GetMoviesOrderedByTitle()
{
    return this.ObjectContext.Movies.OrderBy(m => m.Title);
}
public IQueryable<Movie> GetMoviesFiltered(string filter)
{
    return this.ObjectContext.Movies.Where(m => m.Director ==
filter);
}
```

3. Build your solution.

4. Open HomeViewModel.cs, and add the following command definitions to the class:

```
public ICommand LoadOrderedMoviesCommand
{    get;    private set; }

public ICommand LoadFilteredMoviesCommand
{    get;    private set; }
```

5. Add the following using statements to HomeViewModel.cs:

```
using System.Windows.Input;
using GalaSoft.MvvmLight.Command;
```

6. Instantiate the commands, and call the InstantiateCommands() method from the constructor:

```
public HomeViewModel()
{
    InstantiateCommands();
}
private void InstantiateCommands()
{
    LoadOrderedMoviesCommand = new RelayCommand(() =>
LoadOrderedMovies());
    LoadFilteredMoviesCommand = new RelayCommand(() =>
LoadFilteredMovies());
}
```

7. Implement the commands as follows:

```
private void LoadFilteredMovies()
{
    // clear movies for this example
    MovieContext.Movies.Clear();
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesFilteredQuery("Francis Ford
Coppola"));
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
private void LoadOrderedMovies()
{
    // clear movies for this example
    MovieContext.Movies.Clear();
    var loadOp = MovieContext.Load<Movie>(
        MovieContext.GetMoviesOrderedByTitleQuery());
    loadOp.Completed += (send, args) =>
    {
        // handle errors
    };
}
```

8. You can now build and run the application (note: ensure you've updated the web.config file in RIA Services.WebHost with the correct connection string). Clicking the **Order** button will return the list of movies, ordered by Title, and clicking the **Filter** button will return a filtered collection, each by calling their respective server-side method:

The screenshot shows a user interface titled "RIA SERVICES" at the top. Below it, a section titled "MOVIE RENTAL MANAGEMENT SYSTEM" contains "MOVIE OVERVIEW". There are two buttons: "Order movies" and "Filter movies". A table displays movie details with columns: Title, Director, Category, and Date published. The data row shows "Apocalypse Now" directed by "Francis Ford Coppola" with the date "8/15/1979".

Title	Director	Category	Date published
Apocalypse Now	Francis Ford Coppola		8/15/1979

How it works...

In the first few steps, we've added the necessary XAML code for our buttons, binding them to commands we define on our `ViewModel`, so these commands will be executed once the button is clicked.

In step 2, we're in our `MovieDomainService`, defining two extra query methods:

```
public IQueryable<Movie> GetMoviesOrderedByTitle()
{
    return this.ObjectContext.Movies.OrderBy(m => m.Title);
}

public IQueryable<Movie> GetMoviesFiltered(string filter)
{
    return this.ObjectContext.Movies.Where(m => m.Director == filter);
}
```

The first one will return all movies, ordered by `Title`. The second one is a method which accepts a string parameter, `filter`, which is then applied to the `Director` field of each `Movie`. When we build the solution (step 3), the client-side methods `GetMoviesOrderedByTitleQuery` and `GetMoviesFilteredQuery` are generated.

In step 5, we load movies using these methods, passing in a filter as parameter for the `GetMoviesFilteredQuery` method.

So, when you click the **Order movies** button, the server-side query method `GetMoviesOrderedByTitle` will be executed, which will then be reflected in our UI.

When you click the **Filter movies** button, the same thing happens with the `GetMoviesFiltered(string filter)` query method.

There's more...

Not all types can be passed in as parameters for a Query method, only simple types (like `string`, `int`) are supported. Even though there is `ComplexObject` type support in WCF RIA Services (complex objects must derive from the `ComplexObject` base class), they are only supported for `Invoke` and `NamedUpdate` methods. This is an intentional restriction, in place due to the fact that the parameters passed in to the Query method must be representable in the GET request URI (used behind the scenes by WCF RIA Services).

See also

Have a look at the previous recipe, *Controlling the server-side query from the client*, to learn how to sort and filter without having to write extra query methods on your domain services. To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Paging through your data

Applies to Silverlight 4 and 5

If there's one requirement that returns in every business application that works with large amounts of data, it must be paging. Without paging, you'd have to load all the data to your client, resulting in massive hits on your database, on the wire, and on memory usage on the client.

In this recipe, you'll learn how to go about implementing server-side paging using WCF RIA Services.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

Also, ensure you have the latest version of the Toolkit installed, which can be found here:
<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26939>.

If you do not want to install the Toolkit, the assembly used by this recipe from the Toolkit is included in the \Dependencies folder.

We're starting from a provided started solution, which you can find in
Chapter 11\Paging_Data_Starter\.

The completed solution can be found in Chapter 11\Paging_Data_Completed\.

How to do it...

We're starting out with the starter solution, and we're going to add paging capabilities to our DataGrid. To achieve this, complete the following steps:

1. Open HomeViewModel.cs, and add the following properties:

```
private EntityCollectionPagerAndSorter<Movie> ecpsMovies;  
public DomainCollectionView Movies
```

```
{  
    get  
    {  
        return ecpsMovies.View;  
    }  
}
```

2. Locate the `LoadData()` method, and implement it as such:

```
private void LoadData()  
{  
    ecpsMovies = new EntityCollectionPagerAndSorter<Movie>(MovieCo  
ntext,  
        MovieContext.GetMoviesQuery(), MovieContext.Movies, 2);  
    ecpsMovies.OnQueryCompleted += (send, args) =>  
    {  
        // handle operations that need to be  
        // executed after a page has been loaded  
    };  
    //// move to first page when sort descriptors change  
    INotifyCollectionChanged notifyingSortDescriptions =  
        (INotifyCollectionChanged)this.Movies.SortDescriptions;  
    notifyingSortDescriptions.CollectionChanged +=  
        (sender, e) => ecpsMovies.View.MoveToFirstPage();  
}
```

3. Open `Home.xaml`, locate the `DataGridView`, and bind its `ItemsSource` and `IsEnabled` properties as such:

```
<sdk:DataGridView ItemsSource="{Binding Movies}"  
    Grid.Row="1"  
    IsReadOnly="True"  
    IsEnabled="{Binding Movies.IsDone}"  
    AutoGenerateColumns="False"  
    HorizontalAlignment="Left"  
    VerticalAlignment="Top"  
    Margin="0,10,0,0">
```

4. Add a `DataPager` to `Home.xaml` as such:

```
<sdk:DataPager Grid.Row="2"  
    Source="{Binding Movies}"  
    IsEnabled="{Binding Movies.IsDone}"  
    PageSize="2" />
```

5. You can now build and run your application (note: ensure you've updated the `web.config` file in `RIAServices.WebHost` with the correct connection string), and page through the data.

The screenshot shows a web application interface titled "RIA SERVICES". Below it is the title "MOVIE RENTAL MANAGEMENT SYSTEM". Underneath that is a section titled "MOVIE OVERVIEW". A table displays two rows of movie information:

Title	Director	Date published
Apocalypse Now	Francis Ford Coppola	8/15/1979
Cidade de Deus	Fernando Meirelles	5/7/2003

At the bottom of the page is a navigation bar with icons for back, forward, and search, followed by the text "Page 1 of 2".

How it works...

This recipe uses the `DomainCollectionView` to achieve server-side paging, which can be found in the `Microsoft.Windows.Data.DomainServices` assembly, which is part of the WCF RIA Services Toolkit. This is a collection type, specifically designed to achieve our scenario.

It takes a bit of setting up, which is why most of the logic has been placed in a reusable helper class, `EntityCollectionPagerAndSorter`. Have a look at the constructor of this helper class, which is called in step 2:

```
public EntityCollectionPagerAndSorter(DomainContext context,
    EntityQuery<T> query,
    EntitySet<T> items, int pageSize)
{
    this.context = context;
    this.query = query;
    query.IncludeTotalCount = true;
    this.source = new EntityList<T>(items);
    this.loader = new DomainCollectionViewLoader<T>(this.OnLoad,
    this.OnLoadCompleted);
    this.view = new DomainCollectionView<T>(this.loader, this.
    source);
    using (this.view.DeferRefresh())
```

```
        {
            this.view.PageSize = pageSize;
            this.view.MoveToFirstPage();
        }
    }
```

As you can see, a DomainCollectionView gets initialized with a source and with a loader.

The source is typically an `EntityList<T>` over an `EntitySet<T>` (in our example, it's an `EntityList<Movie>`, as we pass in `MovieDomainContext.Movies`), and it's this source that defines the source entities for our view.

The loader takes care of loading the data, allowing us to pass in a callback, which defines what should happen when data must be loaded (`OnLoad`), and a callback which defines what should happen when a load operation is completed (`OnLoadCompleted`).

If we look a bit further down in this helper class, we can see how these are implemented:

```
private LoadOperation<T> OnLoad()
{
    IsDone = false;
    var loadOp = this.context.Load(this.query.SortAndPageBy(this.view));
    return loadOp;
}
private void OnLoadCompleted(LoadOperation<T> op)
{
    IsDone = true;
    if (op.HasError)
    {
        op.MarkErrorAsHandled();
        view.PageSize = 0;
    }
    else if (!op.IsCanceled)
    {
        this.source.Source = op.Entities;
        if (op.TotalEntityCount != -1) this.View.
SetTotalItemCount(op.TotalEntityCount);
        if (OnQueryCompleted != null)
        {
            OnQueryCompleted.Invoke(this, new LoadOpEventArgs<T>()
{ LoadOperation = op });
        }
    }
}
```

When we load data, we define that we are loading by setting `IsDone` to `false` (which is there to easily bind to the `Enabled` property of our `DataGridView` and `DataPager`), and the query we passed in [step 2](#) (`MovieDomainContext.GetMoviesQuery()`) is executed, using the `SortAndPageBy` extension method to ensure only the correctly sorted and paged part of all `Movies` is returned (WCF RIA Services handles this for us by extending the `EntityQuery`, much in the same way as is explained in the recipe [Controlling the server-side query from the client](#)).

When the data has been loaded, `IsDone` is set to `true`, the source collection is set to the loaded entities, and the `TotalItemCount` is set via `TotalEntityCount`.

Finally, the `OnQueryCompleted` event is invoked, allowing us to execute additional logic when a page has been loaded.

Now, how does this loading get triggered? The following code in the constructor takes care of triggering the initial load, and subsequent loads are triggered automatically by binding the `DomainCollectionView` (the `Movies` property in `HomeViewModel.cs`) to the `DataPager`.

```
using (this.view.DeferRefresh())
{
    this.view.PageSize = pageSize;
    this.view.MoveToFirstPage();
}
```

If we look back at `HomeViewModel`, we can see how two properties are defined:

```
private EntityCollectionPagerAndSorter<Movie> ecpsMovies;
public DomainCollectionView Movies
{
    get
    {
        return ecpsMovies.View;
    }
}
```

These represent a `Movies` property, used to bind to in `HomeView.xaml`, and an instance of our helper class, which is initialized in the constructor, as explained previously:

```
ecpsMovies = new EntityCollectionPagerAndSorter<Movie>(MovieContext,
    MovieContext.GetMoviesQuery(), MovieContext.Movies, 2);
```

We pass in the context we're using, the query that will be used for loading data, the items that will be used as the source, and the size of 1 page.

On to Home.xaml: as we can see in the last few steps, we bind the `DataGridView` and `DataPager` to the `Movies DomainCollectionView`, and their `IsEnabled` properties to the `IsDone` property.

To sum it all up, this is what happens when you trigger the refreshing of a `DomainCollectionView` (in the constructor or by paging):

- ▶ The Loader is executed and loads the data
- ▶ Once this is done, the `Source` property gets updated
- ▶ This in turn notifies the View that it has updated
- ▶ The control of your View reflects these changes

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Persisting a change set/unit of work

Applies to Silverlight 4 and 5

Getting data on the client is only one part of the story, in most applications, you'll also have to persist data back to server. More often than not, you'll need to persist a set of entities all at once (for example, if you're creating a master-detail list, you'll have to ensure the master record is available before saving the detail records).

WCF RIA Services supports these scenarios out of the box. Entities support change tracking, and the `DomainContext` tracks these changes through a `ChangeSet`, which can be persisted to the server when a save is required.

In this recipe, we'll learn how to achieve this.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

We're starting from a provided started solution, which you can find in `Chapter 11\Persisting_Data_Starter\`.

The completed solution can be found in `Chapter 11\Persisting_Data_Completed\`.

How to do it...

We're going to start from our starter solution, and learn how to persist a change set back to the server. To achieve this, we'll complete the following steps:

1. Open `HomeViewModel.cs`, and locate the constructor. Change it so it calls the `InstantiateCommands()` method, and implement that method as such:

```
public HomeViewModel()
{
    LoadData();
    InstantiateCommands();
}

private void InstantiateCommands()
{
    SaveChangesCommand = new RelayCommand(() => SaveChanges());
    RemoveMovieCommand = new RelayCommand<Movie>((m) =>
RemoveMovie(m));
    AddMovieCommand = new RelayCommand(() => AddMovie());
}
```

2. Implement the `AddMovie()` method:

```
private void AddMovie()
{
    MovieContext.Movies.Add(new Movie()
    {
        // fill out category with default value for example
        CategoryID = 1,
    });
}
```

3. Implement the `RemoveMovie(Movie movie)` method:

```
private void RemoveMovie(Movie movie)
{
    if (movie != null)
        MovieContext.Movies.Remove(movie);
}
```

4. Implement the `SaveChanges()` method:

```
private void SaveChanges()
{
    // make sure the elements aren't in edit mode anymore
    // (SL Datagrid workaround)
```

```
MovieContext.Movies.ToList().ForEach(m => (m as  
IEditableObject).EndEdit());  
if (!MovieContext.IsSubmitting)  
{  
    MovieContext.SubmitChanges();  
}  
}
```

5. You can now build and run the application (note: ensure you've updated the web.config file in RIA Services.WebHost with the correct connection string). You can add new Movies, remove a Movie, update a movie through the DataGrid, and persist the changes to the server by pushing the **Save changes** button:

Title	Director	Date published
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998
Apocalypse Now	Francis Ford Coppola	8/15/1979
Cidade de Deus	Fernando Meirelles	5/7/2003
New movie	New director	1/1/0001

[Add new movie](#) [Remove selected movie](#) [Save changes](#)

How it works...

In step 1, we instantiate our commands. These commands are bound to the buttons on Home.xaml, so they get executed when clicking the corresponding button, as you can see in this piece of XAML code from Home.xaml:

```
<Button Content="Add new movie"  
       Height="30"  
       Width="200"  
       Margin="0,0,10,0"  
       Command="{Binding AddMovieCommand}"  
></Button>
```

```
<Button Content="Remove selected movie"
    Height="30"
    Width="200"
    Margin="0,0,10,0"
    Command="{Binding RemoveMovieCommand}"
    CommandParameter="{Binding SelectedItem,
ElementName=grdMovies}"></Button>
<Button Command="{Binding SaveChangesCommand}"
    Height="30"
    Width="200"
    HorizontalAlignment="Left"
    Content="Save changes"></Button>
```

When adding or removing entities, we do this on the `EntitySets`, which are stored in the `EntityContainer`.

Let's re-use a metaphor we've explained in the recipe *Getting data on the client* to make this a bit clearer. The `DomainContext` can be seen as the mechanism that moves entities between the `DomainService` and the `EntitySets`. These `EntitySets` are stored in the `EntityContainer`. You could view the `EntityContainer` as the database (on the client), the `EntitySets` as the tables, and the `DomainContext` as the database server.

This results in the following code for adding a `Movie`, from step 2:

```
MovieContext.Movies.Add(new Movie())
{
    // fill out category with default value for example
    CategoryID = 1,
});
```

As you see, we add a `Movie` to the `Movies EntitySet`. As the `DataGridView` is bound to this `EntitySet`, this is immediately reflected on-screen (note: we fill out the `CategoryID` with a default value for this example).

The same mechanism is used to remove a `Movie` (step 3), and ensures we can immediately see it reflected in on-screen:

```
if (movie != null)
    MovieContext.Movies.Remove(movie);
```

Next up is submitting the data to the server. When you add, remove, or update an Entity, this is automatically saved in a `ChangeSet`. You can always ask for this `ChangeSet` through the `EntityContainer`:

```
MovieContext.EntityContainer.GetChanges()
```

To submit the data to the server, we call the `SubmitChanges()` method on `MovieDomainContext` (first checking that we aren't submitting already):

```
if (! (MovieContext .IsSubmitting) )  
{  
    MovieContext .SubmitChanges () ;  
}
```

Client-side validation occurs, and once we reach the server, the `Submit` method (accepting a `ChangeSet`) on our `DomainService` will be executed. These methods are already implemented by the class we derive from, in our case a `LinqToEntitiesDomainService`. If needed, you can override these methods. The explanation that follows, essentially happens behind the scenes.

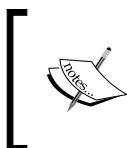
The `Submit` method on the server will look at the `ChangeSet`, and perform the operations indicated by this `ChangeSet`, by invoking the corresponding domain operations.

First of all, `Submit` will call the `AuthorizeChangeSet` method, which will check if the user has the necessary authority to execute it. Next up is the `ValidateChangeSet` method, which will validate the complete set of changes, server-side (have a look at *Chapter 12, Advanced WCF RIA Services*, for more information on validation). If it's valid, the `ExecuteChangeSet` method will be called. This method will invoke the corresponding `DomainOperationEntry` for each operation in the `ChangeSet`.

This means that when a `Movie` has been updated, the `UpdateMovie` method will be executed to execute this update on the `ObjectContext`. When a `Movie` has been added, the `InsertMovie` method will be executed, which will add the `Movie` to the `ObjectContext`. And finally, when a `Movie` has been deleted, the `DeleteMovie` method will be called, removing the `Movie` from the `ObjectContext`.

After this has been executed, our client-side `ChangeSet` has essentially been rebuilt on the server and our Entity Framework `ObjectContext` now has the exact same changes we've executed on the client.

One last step remains, the `ObjectContext` is persisted by calling the `PersistChangeSet` method. It's this step that saves the changes to the database.



There's actually another method that could be called, `ResolveConflicts`. This is called in case the persisting `ChangeSet` fails due to concurrency errors. For more information on this, refer to recipe *Working with concurrency and transactions*.

After all these steps, the `ChangeSet` has been persisted to the database, and there are no more pending changes on our client.

There's more...

When you try and save data, there's always the possibility that the save fails, expected or unexpected. For example, a server-side validation rule might fail, the database could be offline, you could get a concurrency problem.

When you submit changes, the `SubmitChanges` method on your `DomainContext` returns a `SubmitOperation` object. With this object, you can check if something went wrong. Have a look at the following code:

```
private void SaveChanges()
{
    // make sure the elements aren't in edit mode anymore
    // (SL Datagrid workaround)
    MovieContext.Movies.ToList().ForEach(m => (m as
    IEditableObject).EndEdit());
    if (!(MovieContext.IsSubmitting))
    {
        var submitOp = MovieContext.SubmitChanges();
        submitOp.Completed += (send, args) =>
        {
            if (submitOp.HasError)
            {
                // show error, mark as handled
                MessageBox.Show(submitOp.Error.Message);
                submitOp.MarkErrorAsHandled();
            }
        };
    }
}
```

Through the `SubmitOperation`, we can check if there are errors, and solve them accordingly. Next to that, a `SubmitOperation` can also be used to cancel the running operation, by calling `SubmitOperation.Cancel()`.

This code has been implemented in the solution that you can find at [Chapter 11\Persisting_Data_With_Submit_Operation\](#).



By default, validation will occur when the bound value is set. Validation will also occur on the client before submitting to the server, resulting in a failed submit. Next to that, you could always force validation before submitting to the server by using the `Validator.TryValidateObject` method on your object. Nevertheless, that wouldn't solve this problem in case of server-only validations or unexpected errors. Have a look at *Chapter 12, Advanced WCF RIA Services* to learn more about validation.

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

Working with concurrency and transactions

Applies to Silverlight 4 and 5

Handling concurrency is something that comes up sooner or later in any multi-user environment. What should happen when two users are working on the same set of data and a conflict occurs?

As far as transactions are concerned, your changes are automatically wrapped in a transaction when using a `LinqToEntityDomainService`. However, when you're not using one of those, or when you want to have some more control, you can write your own code for this.

In this recipe, we're going to learn how to work with concurrency and transactions with WCF RIA Services.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to recipe *Setting up a data solution to work with WCF RIA Services* for more information.

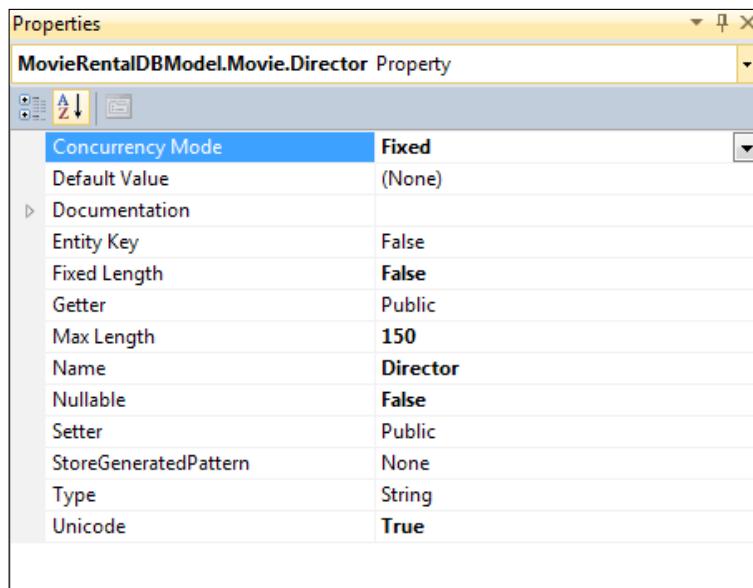
We're starting from a provided started solution, which you can find in `Chapter 11\Concurrency_And_Transactions_Starter\`.

The completed solution can be found in `Chapter 11\Concurrency_And_Transactions_Completed\`.

How to do it...

We're going to start from the starter solution and implement a way to handle concurrency problems and gain more control over transactions. To achieve this, complete the following steps:

1. Open the starter solution, build it, start it up twice, and wait until both DataGrids are filled with data (note: ensure you've updated the `web.config` file in `RIAServices.WebHost` with the correct connection string). In the first running application, change the `Director` and submit the changes. Now do the same in the second application, for the same record. You'll notice the changes are submitted successfully, even though this could result in a concurrency error. In the second application, you're making changes to data that has been changed by someone else in the meantime.
2. Go back to Visual Studio, and open `MovieModel.edmx`. In the designer, locate the `Director` property on the `Movie` entity, and set its **Concurrency Mode** to **Fixed**:

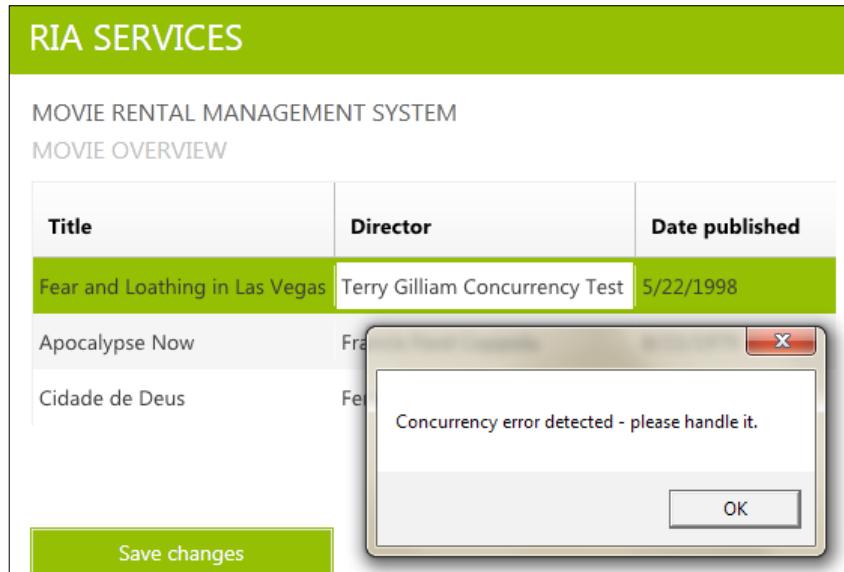


3. Open `HomeViewModel.cs`, and implement the `SaveChanges()` method as follows:

```
private void SaveChanges()
{
    // make sure the elements aren't in edit mode anymore
    // (SL Datagrid workaround)
    MovieContext.Movies.ToList().ForEach(m => (m as
    IEditableObject).EndEdit());
}
```

```
if (! (MovieContext.IsSubmitting))
{
    var submitOp = MovieContext.SubmitChanges();
    submitOp.Completed += (send, args) =>
    {
        if (submitOp.HasError)
        {
            bool hasConcurrencyConflict = false;
            // show error, mark as handled
            foreach (var entityInError in submitOp.
EntitiesInError)
            {
                if (entityInError.EntityConflict != null)
                {
                    hasConcurrencyConflict = true;
                    break;
                }
            }
            if (hasConcurrencyConflict)
            {
                // concurrency error
                MessageBox.Show("Concurrency error detected
- please handle it.");
            }
            else
            {
                // other error
                MessageBox.Show(submitOp.Error.Message);
            }
            submitOp.MarkErrorAsHandled();
        }
    };
}
}
```

4. Build the application, and do the same as in step 1: start it up twice, and wait until both DataGrids are filled with data. In the first running application, change the Director and submit the changes. Now do the same in the second application, for the same record. This time, you'll notice you get notified of a concurrency problem:



5. Add a reference to `System.Transactions` to `RIAServices.DomainServices`.
 6. Open `MovieDomainService.cs`, and override the `PersistChangeSet()` method as such:

```
protected override bool PersistChangeSet()
{
    bool resultFromPersist = false;
    using (var transScope = new TransactionScope(TransactionScopeOption.Required,
        new TransactionOptions
        { IsolationLevel = System.Transactions.IsolationLevel.
          ReadCommitted }))
    {
        resultFromPersist = base.PersistChangeSet();
        if (!this.ChangeSet.HasError)
        {
```

```
        transScope.Complete();
    }
}
return resultFromPersist;
}
```

7. You can now build and run your application.

How it works...

We'll start out with the concurrency part of this recipe. What we want to achieve is a `ConcurrencyCheck` or `TimeStamp` attribute on our properties, which in turn results in a `RoundTripOriginal` attribute.

The `RoundTripOriginal` attribute ensures the original value is sent over the wire together with the current value of the property. It's then up to the DAL to check for concurrency, which is achieved by setting the concurrency mode to fixed on our Entity Model. If this checks out, the data access layer can update or delete the data. Else, a conflict occurs and an `OptimisticConcurrencyException` is thrown. Information about the conflict is stored in the `EntityConflict` property of the entity that no longer has current data (which is what we look at when we handle the `SubmitOperations' Completed` event).

If you look at `RIAServices.DomainServices.g.cs` (the generated code) in `RIAServices.Client.Model` and navigate to the `Director` property on the `Movie` entity, you'll notice that it now has the `ConcurrencyCheck` and `RoundTripOriginal` attributes:

```
[ConcurrencyCheck()]
[DataMember()]
[Required()]
[RoundtripOriginal()]
[StringLength(150)]
public string Director
{
    get
    {
        return this._director;
    }
    set
    {
        if ((this._director != value))
        {
            this.OnDirectorChanging(value);
            this.RaiseDataMemberChanging("Director");
            this.ValidateProperty("Director", value);
        }
    }
}
```

```
        this._director = value;
        this.RaiseDataMemberChanged("Director");
        this.OnDirectorChanged();
    }
}
```

This happens because we set the concurrency mode to **Fixed**.

If you're not working with a `LinqToEntitiesDomainService`, you can add the `RoundTripOriginal` attribute to the metadata of the property you want to check for concurrency (in `MovieDomainService.metadata.cs`):

```
[RoundTripOriginal]
public string Director { get; set; }
```

You can also add it on the entity class itself. This will ensure all properties are checked for concurrency.

```
[RoundtripOriginal]
public partial class Movie
```

Important to know is properties with the `Key` attribute automatically get this `RoundTripOriginal` attribute as well.

As far as handling a concurrency error is concerned, in this recipe, we let it boil up to the client, where you can catch it once the `SubmitOperation` is completed, as shown in step 3. You can now handle it in various ways, you could reload the latest version of the `Movie` on the client, discarding the changes made by the current user, you could keep his changes, load the correct version and submit it again. This depends on your applications' requirements.

Another way to handle the concurrency conflict is by overriding the `ResolveConflicts` method on your `DomainService`:

```
protected override bool ResolveConflicts(IEnumerable<System.Data.
Objects.ObjectStateEntry> conflicts)
{
    // resolve the conflicts...

    return base.ResolveConflicts(conflicts);
}
```

This method is executed by default when a conflict (concurrency error) happens on persisting the changes to the database. As you're working server-side in this case, you can handle this the way you would handle it when using the Entity Framework, by calling the `Refresh` method on your `ObjectContext`, passing in how it should refresh it (`ClientWins` or `StoreWins`), and resubmitting the `ChangeSet`.

On to the second part of the recipe: transactions. In most cases, you shouldn't really worry about this when using WCF RIA Services, and let SQL Server handle this. By default, a unit of work (`ChangeSet`) is wrapped in a transaction when submitting it, while using a `LinqToEntitiesDomainService`.

However, in some cases, you might need more control over this, or you might be working with another datastore/DTO's. That's where you can use the technique described in this recipe. Override the `PersistChangeSet()` method, and handle the transaction yourself.

See also

To learn more techniques concerning WCF RIA Services, have a look at the other recipes in this chapter or in *Chapter 12, Advanced WCF RIA Services*.

12

Advanced WCF RIA Services

In this chapter, we will cover the following topics:

- ▶ Tracking a user's identity – default Windows authentication
- ▶ Tracking a user's identity – a custom authentication service
- ▶ Integrating Windows Identity Foundation with WCF RIA Services
- ▶ Controlling a user's access to services and service methods
- ▶ Validating data: using data annotations
- ▶ Validating data: writing a custom validator
- ▶ Validating data: server-side validation with client-side feedback
- ▶ Validating data: triggering validation when needed
- ▶ Validating data: using the ValidationContext
- ▶ Handling errors on the server
- ▶ Using SQL Azure with WCF RIA Services
- ▶ Exposing WCF RIA Domain Services as OData endpoints
- ▶ Exposing WCF RIA Domain Services for other technologies

Introduction

Microsoft WCF RIA Services is a framework developed to simplify Line of Business RIA development. RIA Services addresses the complexity of building N-tier applications by providing a **framework**, **controls**, and **services** on both your server side (the services, typically .NET based, hosted in an ASP.NET application, and the application in which your Silverlight client is hosted) and your client side (your Silverlight application).

RIA Services makes it easy to get data from your services to your client. It does this by allowing you to write services linked to a data store, such as a database, your own classes, an entity model, and so on, on your server side, and then regenerates these entities on your client. It also generates the necessary **context**, **methods**, and **operations** on your client to easily talk to your services. So, in essence, *WCF RIA Server is a server-side technology that projects code to a client using WCF as a means of communication between the server and the client*. In addition, it makes it easy to add **validation** and **authentication** to your services and/or entities.

It has quickly become one of the most-used, popular frameworks for designing LOB applications with Silverlight, as they work nicely together: after all, it was initially conceived to be used in combination with Silverlight. But besides that, it's also used together with other technologies, as WCF RIA Services has **SOAP** (CRUD) and **OData** (R) endpoints – so any client such as **WPF**, **WinForms**, **ASP.NET (WebForms and MVC)**, or **Windows Phone**, can talk with it and make use of the features it brings. Besides that, there's also a DomainService implementation for **Windows Azure (SQL Azure)**, and a full client-side JavaScript implementation, RIA/js, to be used with any HTML5-based client (typically in combination with jQuery) without a line of C# code.

This chapter will teach you some of the more advanced techniques concerning WCF RIA Services, from authentication over validation to exposing domain services for consuming in other technologies. If you want to learn the basics, have a look at *Chapter 11, Using WCF RIA Services*.

Tracking a user's identity – default Windows authentication

Applies to Silverlight 4, 5

In quite a few applications, knowing which user is using your application is important. You might want to enable certain features if a certain user is logged in, or limit the data you're offering depending on the logged-in user's identity. In this recipe, you'll learn how easy it is to get to know which user is using your application. You'll learn how to verify and track a user's identity in your Silverlight application.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services* for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Tracking_Identity_Starter.

The complete solution can be found in Chapter 12\Tracking_Identity_Completed.

How to do it...

We're going to track a user's identity with WCF RIA Services. To achieve this, we'll open the starter solution and complete the following steps:

1. We'll use **Windows Authentication** for this recipe. Check the web.config file in the WebHost application and add an authentication tag to system.web:

```
<authentication mode="Windows"></authentication>
```

2. Add a new **authentication domain service**,

MovieAuthenticationDomainService, to RIAServices.DomainServices. Use the **Authentication Domain Service** template for this.

3. Build your application, open App.xaml.cs, and add the following code to the Application_Startup method:

```
((WebAuthenticationService)WebContext.Current.Authentication).  
DomainContext =  
    new MovieAuthenticationDomainContext();  
  
// This will enable you to bind controls in XAML files to  
WebContext.Current  
// properties  
this.Resources.Add("WebContext", WebContext.Current);  
// Windows authentication: the user is available through  
HttpContext on the server,  
// but we still need to load it from the server to our SL app  
  
WebContext.Current.Authentication.LoadUser();
```

4. Add the following using statements to App.xaml.cs:

```
using RIAServices.Client.Model;  
using System.ServiceModel.DomainServices.Client.  
ApplicationServices;  
using RIAServices.DomainServices;
```

5. Open App.xaml, and add the following **xmns imports**:

```
xmlns:app="clr-namespace:RIAServices.Client.  
Model;assembly=RIAServices.Client.Model"  
xmlns:appsvc="clr-namespace:System.ServiceModel.DomainServices.  
Client.ApplicationServices;assembly=System.ServiceModel.  
DomainServices.Client.Web"
```

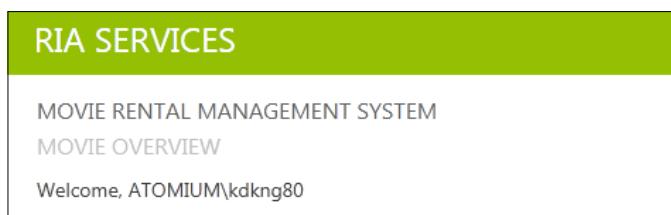
6. Add the following code to App.xaml:

```
<Application.ApplicationLifetimeObjects>  
    <app:WebContext>  
        <app:WebContext.Authentication>  
            <appsvc:WindowsAuthentication />  
        </app:WebContext.Authentication>  
    </app:WebContext>  
</Application.ApplicationLifetimeObjects>
```

7. Add the following XAML code to Home.xaml:

```
<Grid Grid.Row="1">  
    <TextBlock Margin="0,10"  
        Text="{Binding Source={StaticResource WebContext}, Path=User.  
Name, StringFormat='Welcome, {0}'}"></TextBlock>  
</Grid>
```

8. You can now build and run your application. You'll notice the current user's credentials are displayed on-screen:



How it works...

We start out by defining that we'll use Windows authentication in the web.config file. With Windows authentication, we rely on IIS to authenticate the clients. How this happens depends on how you configure IIS: typically, you'd set it to use integrated Windows authentication. After successfully authenticating, the Windows token is passed to the ASP.NET worker process, making it available to our (server-side) application.

In step 2, an authentication domain service is added, and nothing is changed: by default, WCF RIA Services contains an implementation for using Windows authentication with WCF RIA Services, meaning you don't have to change anything to use your authentication domain service: it will work out of the box. Typically, you log in by passing your credentials to the service – in the case of Windows authentication, IIS provides these credentials, as explained in the first paragraph.

So what's happening in step 3? The credentials are available at our server side, but we want them to be visible in our Silverlight application: we're using the `WebContext` to load the currently authenticated user. When this load operation completes, we'll know who's signed in on the client. The `WebContext` is also added to the resource dictionary, so we can easily bind our UI to it.

In step 6, we're defining the authentication mode on the `WebContext`, in this case, Windows authentication: this ensures `WebContext.Current` is configured to use Windows authentication. All that's left now is showing the logged in user in our UI, which is achieved through step 7.

To sum it up: when the hosting web application starts, the current Windows credentials are available at the server, provided by IIS. In our Silverlight application, the current `WebContext` is defined as one that uses Windows authentication. When the Silverlight application starts, we state that we're going to use our `MovieAuthenticationDomainContext`, add the current `WebContext` to the resource dictionary, and load the current user from the server. Once this load operation completes, the current user is shown in our UI.

See also

For more on authentication, have a look at the *Tracking a user's identity – a custom authentication service* and *Integrating Windows Identity Foundation with WCF RIA Services* recipes. For more advanced recipes on WCF RIA Services, have a look at the rest of this chapter.

Tracking a user's identity – a custom authentication service

Applies to Silverlight 4, 5

Windows authentication, as we've used in the previous recipe, is very suitable for internal environments. However, if you're publishing a public Silverlight application, you typically require the user to enter his or her credentials. Besides that, the `User` object itself might be different: it might have extra properties; it might be an `Employee` object from your database.

In this recipe, we'll learn how we can achieve this kind of authentication.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services* for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Tracking_Identity_Custom_Starter.

The complete solution can be found in Chapter 12\Tracking_Identity_Custom_Completed.

How to do it...

We're starting from the completed solution of the previous recipe, or from the provided starter solution, and we're going to *make sure our application can use a custom user object and custom authentication logic*. To enable this, we'll complete the following steps:

1. Open the web.config file in RIAServices.WebHost, and change the authentication mode to Forms:

```
<authentication mode="Forms"></authentication>
```
2. Add a reference to System.Web to RIAServices.DomainServices.
3. Locate the MovieAuthenticationDomainService, and add the following using statements:

```
using System.ServiceModel.DomainServices.EntityFramework;
using System.Web.Security;
using System.Runtime.Serialization;
```
4. Change the class itself; implement it as follows:

```
public class MovieAuthenticationDomainService : LinqToEntitiesDoma
inService<MovieRentalDBEntities>, IAuthentication<Employee>
{
    private static Employee DefaultUser =
        new Employee() { Name = string.Empty };

    public Employee GetUser()
    {
        if ((this.ServiceContext != null) &&
            (this.ServiceContext.User != null) &&
            this.ServiceContext.User.Identity.IsAuthenticated)
        {
```

```
        return this.GetUserByName(this.ServiceContext.  
User.Identity.Name);  
    }  
  
    return MovieAuthenticationDomainService.DefaultUser;  
}  
  
public Employee GetUserByName(string userName)  
{  
    return this.ObjectContext.Employees.FirstOrDefault(e  
=> e.UserName == userName);  
}  
  
public Employee Login(string userName, string password,  
bool isPersistent, string customData)  
{  
    if (this.ValidateUser(userName, password))  
    {  
        FormsAuthentication.SetAuthCookie(userName,  
isPersistent);  
        return this.GetUserByName(userName);  
    }  
    return null;  
}  
  
private bool ValidateUser(string userName, string  
password)  
{  
    // validate username/pw  
    var user = this.ObjectContext.Employees.  
FirstOrDefault(e => e.UserName == userName && e.Password ==  
password);  
    return (! (user == null));  
}  
  
public Employee Logout()  
{  
    FormsAuthentication.SignOut();  
    return MovieAuthenticationDomainService.DefaultUser;  
}  
  
public void UpdateUser(Employee user)  
{  
    throw new NotImplementedException();  
}
```

```
        }  
    }
```

5. In the same file, add the following code for our Employee (IUser) class:

```
public partial class Employee : IUser  
{  
    [DataMember]  
    [Key]  
    public string Name  
    {  
        get  
        {  
            return this.UserName; }  
        set  
        {  
            this.UserName = value; }  
    }  
    [DataMember]  
    public IEnumerable<string> Roles  
    {  
        get { return null; }  
        set { //  
        }  
    }  
}
```

6. Open WebContext.cs in RIAServices.Client.Model, and change it as follows:

```
/// <summary>  
/// Gets a user representing the authenticated identity.  
/// </summary>  
public new Employee User  
{  
    get  
    {  
        return ((Employee)(base.User));  
    }  
}
```

7. Open App.xaml.cs, remove the loading of a user, and change it to logging in:

```
private void Application_Startup(object sender,  
StartupEventArgs e)  
{  
    ((WebAuthenticationService)WebContext.Current.Authentication).  
DomainContext =  
    new MovieAuthenticationDomainContext();  
    // This will enable you to bind controls in XAML files to  
    WebContext.Current
```

```
// properties  
this.Resources.Add("WebContext", WebContext.Current);  
// Forms authentication: we need to log in  
WebContext.Current.Authentication.Login("KevinDockx",  
"password");  
this.RootVisual = new MainPage();  
}
```

8. Open App.xaml, and change the **Authentication of the WebContext** to FormsAuthentication:

```
<Application.ApplicationLifetimeObjects>  
    <app:WebContext>  
        <app:WebContext.Authentication>  
            <apps:FormsAuthentication />  
        </app:WebContext.Authentication>  
    </app:WebContext>  
</Application.ApplicationLifetimeObjects>
```

9. You can now build and run your application. You'll notice the current user's credentials are displayed on-screen:

RIA SERVICES

MOVIE RENTAL MANAGEMENT SYSTEM

MOVIE OVERVIEW

Welcome, KevinDockx

How it works...

In this recipe, we're using Forms authentication to log in to our application, and we want our authenticated user to be an Employee, not just a User object. First, we must of course change the authentication mode to Forms on our server, which is done in step 1.

Then, we're moving on to the authentication itself: by default, WCF RIA Services allows for Forms authentication *which works out of the box if you're using the default ASP.NET Membership tables*: nothing has to be changed to use the authentication service itself. However, for a lot of applications, this isn't feasible: they often have their own user tables. In this example, we're assuming we have a table of employees, and we want to validate our users against that table. The user object that is returned should then of course be an Employee, not just a default user.

In step 4, a lot of that logic is added: the class is defined as a class that implements the `IAuthentication` interface, typed to the `Employee` user. This interface definition looks as follows:

```
[AuthenticationService]
public interface IAuthentication<T> where T : global::System.
ServiceModel.DomainServices.Server.ApplicationServices.IUser
{
    T GetUser();
    T Login(string userName, string password, bool isPersistent,
string
    T Logout();
    void UpdateUser(T user);
}
```

As you can see, we need to implement a few methods. `UpdateUser` is only necessary if you want to allow the user object to be updated through this authentication service – we can skip that for this recipe.

In the `Login` method, most of the logic happens: *this is where we check whether or not the passed-in credentials are valid*. If they are, the logged-in user (an `Employee`) is returned. In the `ValidateUser` method, called from `Login`, we check if the provided credentials are valid. If they are, we set the **authentication cookie** (remember: this is actually ASP.NET's Forms authentication we're leveraging), and we return the corresponding `Employee`. If they're not valid, we simply return null.

The `GetUser` method is used to fetch the current authenticated user, or a default user object when we're not authenticated. We do this by checking the `ServiceContext` for an authenticated user, and returning the matching `Employee` by its `Name` key.

All that's left for this class is the `Logout` method: as with the `Login` method, we're leveraging Forms authentication; so we simply need to call `FormsAuthentication.SignOut`, and return a default user.

Besides all this authentication logic, the interface also defines that the type we pass in (`Employee`) should implement the `IUser` interface. This is what happens in step 5: our `Employee` class (a table in our database, available through our entity framework model – which is why we also defined our authentication service to be a `LinqToEntitiesDomainService<MovieRentalDBEntities>` service) implements this `IUser` interface, which defines a `Name` and `Roles` property. We're not using roles, so that can be left empty, and the `Name` of course refers to our employee `UserName`, so we implement that property as such. Note the `[Key]` attribute on `Name`: it should be unique, as it's used to identify the user.

All that's left to do on the server is ensuring our `WebContext` works with the `Employee` class instead of the regular `User` class, which is done in step 6.

On to the client. This logic is more or less the same as with Windows authentication, as described in the recipe *Tracking a user's identity - default Windows authentication*, with two notable changes: we're using Forms authentication now, so we need to define that that is what the `WebContext` will use, in `App.xaml`. This is done in step 8. The other change can be found in step 7: instead of just loading the user, as we did with Windows authentication, we now need to log in by calling the `Login` method on our authentication service. This will execute the server-side login method we just implemented, and the logged-in `Employee` will be loaded in our context when this method is completed and the credentials are valid.

See also

For more on authentication, have a look at the *Tracking a user's identity – default Windows authentication* and *Integrating Windows Identity Foundation with WCF RIA Services* recipes. For more advanced recipes on WCF RIA Services, have a look at the rest of this chapter.

Integrating Windows Identity Foundation with WCF RIA Services

Applies to Silverlight 4, 5

Besides using Windows and Forms authentication, there's another way to authenticate/authorize a user with WCF RIA Services: through **Windows Identity Foundation (WIF)**. This consists of a set of classes with which you can build identity-aware applications. It allows us to externalize authentication and authorization: the application relies on an identity provider for this.

In this recipe, we'll learn how to use WIF with a WCF RIA Services solution, using an external Security Token Service (the identity provider), which will take care of authentication/authorization for us. Then, we'll see how we can change the business logic in a domain service depending on the available claims.

Getting ready

This recipe relies on quite a few prerequisites. You'll have to have at least Windows Vista SP2 (Windows 7 or 2008 server advised), and IIS 7.0 (or better).

First of all, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from *Chapter 11, Using WCF RIA Services*, for more information.

You'll also need to install the Windows Identity Foundation Runtime, which can be found at <http://www.microsoft.com/download/en/details.aspx?id=17331>. Choose the correct version, depending on your system.

Next, you'll need to install the Windows Identity Foundation SDK, which can be found at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=4451>. Choose the SDK version for .NET 4.0.

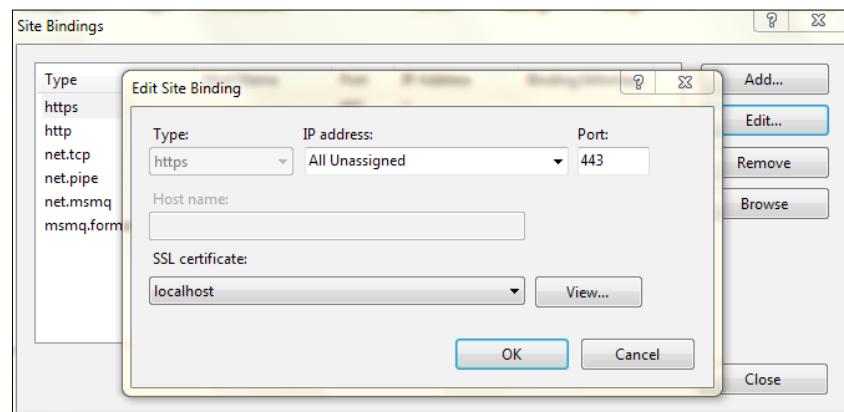
After this, you'll need to install the Windows Identity Foundation Developer Training Kit (April 2011), which can be found at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=14347>. This kit contains sample implementations of, amongst others, a Security Token Service, which is used in this recipe. But, more importantly, it also contains the necessary certificates, which you'll have to register to get up and running.

Note: these certificates and scripts are also included in this recipe, in the \Setup folder of the sample solutions. If you do not want to install the Developer Training Kit, you can use these to set up your system by running the `SetupCertificates.cmd` file, for which you will need administrator permissions. However, as the Developer Training Kit includes a lot of extra samples, guidelines, and best practices, it's a good idea to completely install it.

You can check if the certificates are installed correctly in IIS: in the main overview, select **Server Certificates**, and ensure the **IdentityTKStsCert** and **localhost** certificates are available and valid:

Name	Issued To	Issued By	Expiration Date	Certificate Hash
WP0367.ATOMIUM.local	ESPSEC01-CA		9/02/2012 9:06:08	87862BF038EAF2AEAE...
DefaultApplicationCertificate	DefaultApplicationCertificate		16/11/2011 18:05:43	49372ACA17C9A01A9...
IdentityTKStsCert	IdentityTKStsCert		1/01/2036 4:00:00	40A1D2622BFBDAC80...
localhost	localhost		1/01/2036 4:00:00	308EFDEE6453FFF68C...

Also, ensure the site binding on your default website is using the **localhost** certificate for **https** (check this in IIS: select the **Default website** (or whichever site you're using to host this solution) | **SSL Settings** | **Bindings...** (on the right-hand side)):



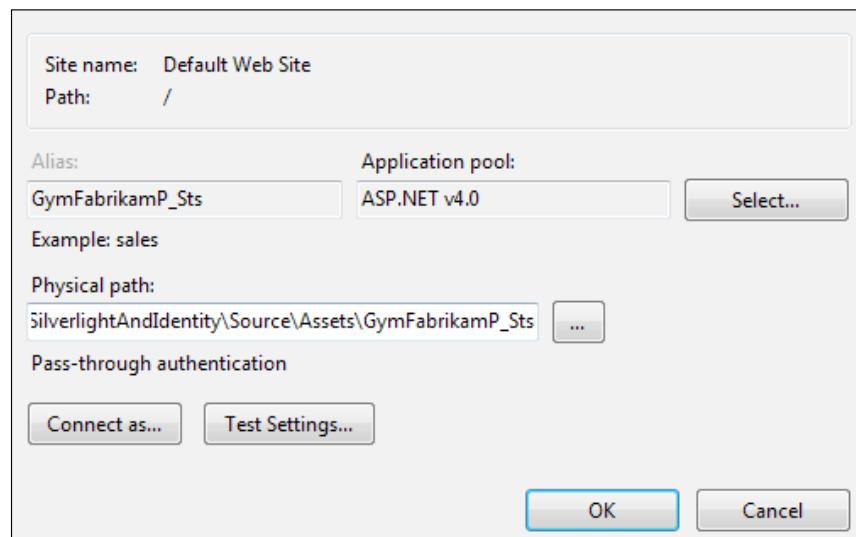
We're starting from a provided starter solution, which you can find in `Chapter 12\Integrating_WIF_Starter`.

The complete solution can be found in `Chapter 12\Integrating_WIF_Completed`.

How to do it...

We're going to set up our starter solution, which is Windows Identity Foundation-enabled, for authentication/authorization. In order to do this, we need to complete the following steps:

1. First of all, we're using the *provided STS from the Windows Identity Foundation Training Kit*. In case you haven't installed it, the STS has been provided for you with this solution, but you'll have to add it to IIS manually. Add a new application in IIS in the default website, naming it `GymFabrikamP__STS`, and point it to the location of the provided website. Do this before opening the starter solution.



2. Open `MovieDomainService.cs` in `RIAServices.DomainServices`, and add the following attribute to the class definition:

```
[RequiresAuthentication]
[EnableClientAccess()]
public class MovieDomainService : LinqToEntitiesDomainService<MovieRentalDBEntities>
```

3. Locate the `GetMovies()` method, and implement it as follows:

```
public IQueryable<Movie> GetMovies()
{
    // get authenticated user
```

```

        var currentClaimsIdentity = (Microsoft.IdentityModel.Claims.
ClaimsIdentity)ServiceContext.User.Identity;
        // get gender claim
        var genderClaim = currentClaimsIdentity.Claims.
FirstOrDefault(c => c.ClaimType == genderClaimType);
        if (genderClaim != null && genderClaim.Value == "male")
        {
            return this.ObjectContext.Movies.Where(m => m.CategoryID
== 2);
        }
        else
        {
            return this.ObjectContext.Movies.Where(m => m.CategoryID
== 4);
        }
    }
}

```

4. (Optional) Check the connection string in `web.config` to ensure it uses an SQL Server Authentication login that is available on your system and linked to a `MovieRentalDB` user.
5. You can now build and run your application. When the application starts, you'll have to provide your credentials. Depending on these credentials, you'll get different results in the DataGrid:

The screenshot shows a Silverlight application window titled "RIA SERVICES". Below the title bar, there's a green header bar with the text "MOVIE RENTAL MANAGEMENT SYSTEM" and "MOVIE OVERVIEW". The main content area contains a DataGrid with four columns: "Title", "Director", "Date published", and "Trailer location". There are two rows of data:

Title	Director	Date published	Trailer location
Apocalypse Now	Francis Ford Coppola	8/15/1979	
Cidade de Deus	Fernando Meirelles	5/7/2003	

How it works...

As you might have noticed, we're starting from a solution that already has most of the WIF-parts in place: this is because this complete process is already explained in the recipe *Integrating Windows Identity Foundation in Silverlight* from *Chapter 9, Talking to WCF and ASMX Services—One Step Beyond*: you can refer to that recipe for more information on how to set up passive federation using WIF, and how it works exactly. This recipe focuses on the integration with WCF RIA Services.

In this recipe, we're using WIF to get authenticated through passive federation. Once a token has been received, it's available through the service context, which you can access from each domain service. Through this service context, we can get the authenticated user, and we get access to the user's claims, which means we can easily change our business logic depending on what the claims of the authenticated user are.

This is what's done in this recipe: in step 3, we've added business logic that will execute a different query depending on the gender claim of the logged in user.

First of all, we ensure our service is only accessible to authenticated users, by adding the `RequiresAuthentication` attribute to the service class definition (step 2). Then, in the `GetMovies()` method, we get the claims from the currently authenticated user, and find the `Gender` claim. Depending on the value of that claim, we execute a different query.

As a result of this, you'll see different results when you log in as John versus when you log in as Mary.

See also

For more on authentication, have a look at the *Tracking a user's identity – default Windows authentication* and *Tracking a user's identity: a custom authentication service* recipes. For more advanced recipes on WCF RIA Services, have a look at the rest of this chapter.

To learn much more about how WIF and passive federation works, have a look at *Integrating Windows Identity Foundation in Silverlight* in Chapter 9, *Talking to WCF and ASMX Services–One Step Beyond*.

Controlling a user's access to services and service methods

Applies to Silverlight 4, 5

In the previous recipes, we've learned how to track a user's identity. However, most applications that need a user's identity need it to do something with it, for example, restricting access to a certain service or a service method so that only an authenticated user can access it. This will make sure that our service can't be freely used by anyone. Only the users we allow in will be able to use it.

In this recipe, you'll learn how to restrict access to a service or service method to authenticated users.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, Using WCF RIA Services, for more information.

We're starting from a provided starter solution, which you can find in [Chapter 12\Controlling_Access_Starter](#).

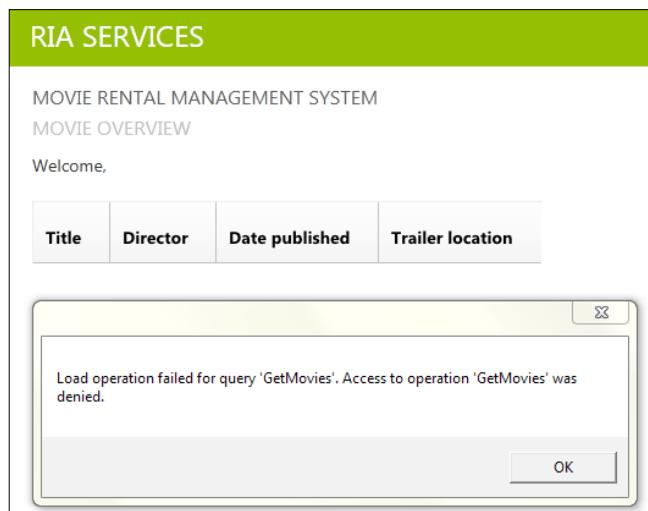
The complete solution can be found in [Chapter 12\Controlling_Access_Completed](#).

How to do it...

We're starting from the provided starter solution, which includes an example authentication service. We're going to write code to limit access to authenticated users only. To achieve this, we'll perform the following steps:

1. Open MovieDomainService.cs in RIAServices.DomainServices.
2. Locate the GetMovies() method, and add the following attribute:

```
[RequiresAuthentication()]
public IQueryable<Movie> GetMovies()
```
3. You can now build and run the solution. When the application starts, we're providing credentials to log in (you can find this code in App.xaml.cs, in the Application_Startup method), after which we are authenticated, enabling us to access the GetMovies() method. If you comment out the code to log in in Application_Startup (or provide invalid credentials), you'll notice we cannot access the GetMovies() method anymore.



How it works...

Restricting access with WCF RIA Services is something you get out of the box. The `RequiresAuthentication` attribute is all you need: by applying this attribute to a method, as we did in step 2, we're ensuring the service method can only be accessed by an authenticated user.

Besides applying this attribute to a method, you can also apply it to the domain service, which restricts access to any method in the domain service to authenticated users:

```
[RequiresAuthentication()]
public class MovieDomainService : LinqToEntitiesDomainService<MovieRen
talDBEntities>
```

There's more...

Besides restricting access to authenticated users, applications often require a more fine-grained approach: restricting access to a certain role. Support for this comes through the `RequiresRole` attribute, which can be applied to a service method or the complete domain service, pretty much the same as with the `RequiresAuthentication` attribute.

The provided example authentication service provides the logged in user with an example role: `DummyRole`. To restrict access to the users with this role, we'll add a custom role provider, which will provide the current user's role(s) via the authentication service. To achieve this, complete the following steps:

1. Add a reference to `System.Web.ApplicationServices` and `System.Configuration` to `RIAServices.DomainServices`.
2. Add a new class to `RIAServices.DomainServices: CustomRoleProvider`.
3. Let this class inherit the `RoleProvider` class (from `System.Web.Security`, which must be imported via a `using` statement), and implement the `GetRolesForUser` method (the other ones aren't necessary for this example), as follows:

```
public class CustomRoleProvider : RoleProvider
{
    public override string[] GetRolesForUser(string username)
    {
        MovieAuthenticationDomainService authService =
            new MovieAuthenticationDomainService();
        return authService.GetUserByName(username).Roles.
            ToArray();
    }

    // ... other methods, not implemented
}
```

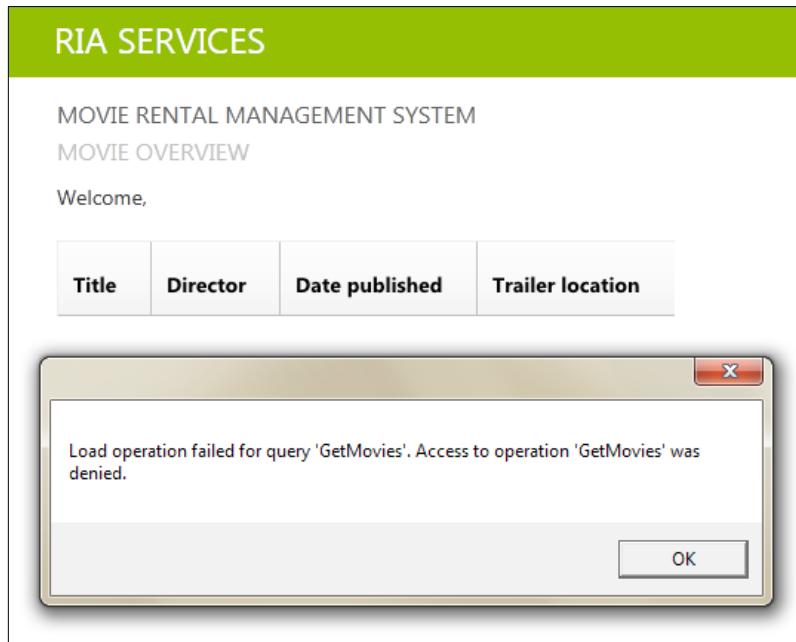
4. Locate the GetMovies() method in MovieDomainService, and apply the RequiresRole attribute as follows:

```
[RequiresRole("DummyRole")]
public IQueryable<Movie> GetMovies()
{
    return this.ObjectContext.Movies;
}
```

5. Open web.config in RIAServices.WebHost, and add the roleManager tag to the system.web tag, as follows:

```
<roleManager defaultProvider="CustomRoleProvider" enabled="true"
cacheRolesInCookie="true" >
    <providers>
        <add name="CustomRoleProvider" type="RIAServices.
DomainServices.CustomRoleProvider, RIAServices.DomainServices"/>
    </providers>
</roleManager>
```

6. You can now build and run the application. If you're not logged in as a user with the DummyRole role, you won't have access to the GetMovies() method (you can check this by providing invalid credentials):



The complete solution can be found at
Chapter 12\Controlling_Access_Roles_Completed.

See also

For more on authentication with WCF RIA Services, have a look at the *Tracking a user's identity – default Windows authentication*, *Tracking a user's identity: a custom authentication service*, and *Integrating Windows Identity Foundation with WCF RIA Services* recipes.

Validating data: using data annotations

Applies to Silverlight 4, 5

Validation of your data before persisting it is a requirement for almost every application. By using validation on your Entities, you make sure that no invalid data is persisted to your data store. When you don't implement validation, there's a risk that a user will input wrongly formatted or plain incorrect data on the screen and even persist this data to your data store. This is something you should definitely avoid.

In this recipe, you'll learn how to validate data using data annotation attributes that you can use on your classes and properties.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services*, for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Validation_Annotations_Starter.

The complete solution can be found in
Chapter 12\Validation_Annotations _Completed.

How to do it...

We're going to start from the provided starter solution, and add validation to our Movie entity. To achieve this, we'll complete the following steps:

1. Open MovieDomainService.metadata.cs from the RIAServices project, and locate the MovieMetadata class.
2. Add the following attributes:

```
[Required(ErrorMessage="You must provide a director for this
movie.")]
[StringLength(50, MinimumLength = 3,
ErrorMessage = "You must input between 3 and 50 characters.")]
public string Director { get; set; }
```

```
[Required(ErrorMessage = "You must provide a title for this movie.")]
public string Title { get; set; }

[RegularExpression(@"^http\://[a-zA-Z0-9\-\.\.]+\.[a-zA-Z]{2,3}(/S*)?")]
ErrorMessage="You must input a valid URL."]
public string TrailerLocation { get; set; }
```

3. You can now build and run your application. When you input an invalid value in one of the fields, for example: an empty title, you'll see the correct error message:

The screenshot shows a web application interface titled 'RIA SERVICES' and 'MOVIE RENTAL MANAGEMENT SYSTEM'. Under 'MOVIE OVERVIEW', there is a table with columns 'Title', 'Director', 'Date published', and 'Trailer location'. The first row contains 'Fear and Loathing in Las Vegas', 'Terry Gilliam', and '5/22/1998'. The second row contains 'Apocalypse Now', 'Francis Ford Coppola', and '8/15/1979'. In the third row, the 'Title' field is empty, highlighted with a red border. A red box to its right contains the validation error message: 'You must provide a title for this movie.' Below the table is a button labeled 'Save changes'.

How it works...

One of the biggest advantages of WCF RIA Services is that you only need to write validation logic once: at the server (your set of changes is automatically validated on the client and on the server when you submit them from the client). This logic is typically attributed to properties/entities in the metadata classes, which describe the entities and their properties. This is what we do in step 2: we add built-in validation attributes to some of the properties of the Movie entity.

Once you build the solution, the client-side code is generated. If you look at the generated code (in `RIAServices.Client.Model`), you'll see that the attributes we've added on the server are propagated to the client. For example, this is the generated code for the `Director` property of the `Movie` entity:

```
/// <summary>
/// Gets or sets the 'Director' value.
/// </summary>
```

```

[DataMember()]
[Required(ErrorMessage="You must provide a director for this
movie.")]
[StringLength(50, ErrorMessage="You must input between 3 and 50
characters.", MinimumLength=3)]
public string Director
{
    get
    {
        return this._director;
    }
    set
    {
        if ((this._director != value))
        {
            this.OnDirectorChanging(value);
            this.RaiseDataMemberChanging("Director");
            this.ValidateProperty("Director", value);
            this._director = value;
            this.RaiseDataMemberChanged("Director");
            this.OnDirectorChanged();
        }
    }
}

```

Besides the attributes being propagated, you can also notice that the validation is triggered in the setter of this property: `this.ValidateProperty` is called on the new value.

What happens behind the screens is a bit more complicated. When you look at the definition of the `Movie` class, you'll see it inherits from `Entity`. This `Entity` class implements the `INotifyDataErrorInfo` interface, which enables various validation scenarios for us. It defines a `HasErrors` property, a `GetErrors` method, and an `ErrorsChanged` event. It *enables data entity classes to implement custom validation rules and expose validation results to the user interface*. The interface also supports **custom error objects, multiple errors per property, cross-property errors** (errors that affect multiple properties), and **entity-level errors** (errors that affect multiple properties, or affect the complete entity without affecting a particular property) – you'll learn about those in the other validation recipes in this chapter.

Thanks to the attributes being propagated, we actually get the validation logic that we wrote on the server on our client: the validation is triggered when you're inputting a new value for a certain property, in the property setter. Besides that, validation is also triggered on submit, both on the client and on the server, ensuring no invalid values can be inputted. Even if you were to have another client application reusing the same domain service, the validation would still be triggered on the server.

As far as the UI is concerned, we didn't have to change anything: the validation errors are automatically shown. This is because the Silverlight binding engine provides built-in support for this interface: you simply need to bind a control to a property of an entity that implements it (like our Movie entity), or the entity itself, and ensure the `ValidatesOnNotifyDataErrors` property on your Data Binding is set to `true` (which is the default value).

The binding engine calls `GetErrors` to retrieve any initial errors for a bound property or entity. It also handles the `ErrorsChanged` event to monitor for updates. Whenever the errors change for a bound property or entity, the binding engine calls `GetErrors` to retrieve the updated errors. The binding engine uses the validation results to update the `ValidationErrors` collection for that binding. First, the binding engine removes any existing errors for the bound property that originate from `INotifyDataErrorInfo` validation. Then, if the new value is not valid, the binding engine adds a new error for each error that is returned by the `GetErrors` method. These errors are then shown in your UI (provided it's available in the template of the control you're using).

There's more...

In this recipe, we've used just a few of the possible data annotations. **`DataTypeAttribute`**, **`RangeAttribute`**, **`RegularExpressionAttribute`**, **`RequiredAttribute`**, **`StringLengthAttribute`**, and **`CustomValidationAttribute`** are all the data annotations at our disposal.

For all these attributes, named parameters are possible to further customize the way validation should occur. **`ErrorMessage`**, **`ErrorMessageResourceName`**, and **`ErrorMessageResourceType`** are available on all attributes, but many more are available depending on the attribute you use. You can check these named parameters by looking at the IntelliSense tooltip you get on the attribute constructor.

See also

Have a look at the other validation recipes in this chapter for more validation scenarios.

Validating data: writing a custom validator

Applies to Silverlight 4, 5

Validation of your data before persisting it is a requirement for almost every application. By using validation on your Entities, you make sure that no invalid data is persisted to your data store. With WCF RIA Services, there are various ways to do validation, each tailored to a specific scenario.

In this recipe, you'll learn how to validate data with custom validators: on a single property, cross-field and on the Entity level. We'll also look into a way to reuse these validators by implementing them as attributes.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services*, for more information.

We're starting from a provided starter solution (or, alternatively, the completed solution of the previous recipe), which you can find in Chapter 12\Validation_Custom_Validator_Starter.

The complete solution can be found in Chapter 12\Validation_Custom_Validator_Completed.

How to do it...

We're starting from the provided starter solution (or the completed solution from the previous recipe), and we're going to add custom validation logic to our application. To achieve this, we'll complete the following steps:

1. Add a new folder, `Validators`, to `RIAServices.DomainServices`.
2. Add a new class to this folder, `MovieValidators`, and name the file `MovieValidators.shared.cs`.
3. Implement this class as follows:

```
public static class MovieValidators
{
    public static ValidationResult PublishedDateMustBeInThePast (
        DateTime publishedDate,
        ValidationContext validationContext)
    {
        if (publishedDate > DateTime.Now)
        {
            return new ValidationResult(
                "You cannot input a date that's in the future.",
                new[] { validationContext.MemberName });
        }

        return ValidationResult.Success;
    }
}
```

4. Build the application.

5. Open MovieDomainService.metadata.cs, and add the following using statement:


```
using RIAServices.DomainServices.Validators;
```
6. Locate the metadata class for the Movie entity (MovieMetadata), and add the following attribute to DatePublished:


```
[CustomValidation(typeof(MovieValidators),  
"PublishedDateMustBeInThePast")]  
public DateTime DatePublished { get; set; }
```
7. Build and run the application, and try to input a date that's in the future. You'll notice the validator is triggered.

The screenshot shows a web application interface titled "RIA SERVICES" and "MOVIE RENTAL MANAGEMENT SYSTEM". Under "MOVIE OVERVIEW", there is a table with four columns: "Title", "Director", "Date published", and "Trailer location". The table contains three rows of movie data. The third row, for "Cidade de Deus", has its "Date published" field highlighted with a red border. To the right of this field is a red rectangular callout box containing the text "You cannot input a date that's in the future." Below the table is a button labeled "Save changes".

Title	Director	Date published	Trailer location
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998	
Apocalypse Now	Francis Ford Coppola	8/15/1979	
Cidade de Deus	Fernando Meirelles	5/7/2015	You cannot input a date that's in the future.

8. Open MovieValidators.shared.cs, and add the following code, which we will use for cross-field validation:

```
public static ValidationResult TrailerAndDirector(string trailerOrDirector,  
    ValidationContext validationContext)  
{  
    Movie movie = (Movie)validationContext.ObjectInstance;  
  
    // inputted value: trailer or director?  
    string trailer = validationContext.MemberName ==  
        "TrailerLocation"  
        ? (trailerOrDirector ?? "") : (movie.TrailerLocation ??  
        "");  
    string director = validationContext.MemberName == "Director"  
        ? (trailerOrDirector ?? "") : (movie.Director ?? "");
```

```

        if (trailer.Length > 0 && director.Length == 0)
    {
        return new ValidationResult("If a trailer is available,
the director must be filled out.",
            new[] { "TrailerLocation", "Director" });
    }

    return ValidationResult.Success;
}

```

9. Open MovieDomainService.metadata.cs, and locate the metadata class for the Movie entity. Add the following attributes to Director and TrailerLocation:

```
[CustomValidation(typeof(MovieValidators), "TrailerAndDirector")]
public string Director { get; set; }
```

```
[CustomValidation(typeof(MovieValidators), "TrailerAndDirector")]
public string TrailerLocation { get; set; }
```

10. Build and run the application, and try to input a location for the trailer, while not inputting a value for the director (and the other way around): you'll notice our cross-field custom validator is triggered.

RIA SERVICES

MOVIE RENTAL MANAGEMENT SYSTEM

MOVIE OVERVIEW

Title	Director	Date published	Trailer location
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998	
Apocalypse Now	Francis Ford Coppola	8/15/1979	
Cidade de Deus	 	5/7/2003	http://www.trailers.com/trailer.wmv

! 1 Error

If a trailer is available, the director must be filled out.

[Save changes](#)

11. Open MovieValidators.shared.cs, and add the following code:

```
public static ValidationResult RecentMovieCheck(Movie movie)
{
    if (movie.Director.Length > 0 && movie.Title.Length > 0
        && movie.DatePublished > new DateTime(2010, 1, 1)
        && string.IsNullOrWhiteSpace(movie.
    TrailerLocation)
    )
    {
        return new ValidationResult("If you know the director and
the title, and the movie is recent, you must fill out a location
for the trailer.");
    }
    return ValidationResult.Success;
}
```

12. Open MovieDomainService.metadata.cs, and locate the class for the Movie entity. Add the following attribute to the Movie entity:

```
[CustomValidation(typeof(MovieValidators), "RecentMovieCheck")]
public partial class Movie
```

13. Build and run the application, and try to input a director, title, and recent date (after 1/1/2010), while not inputting a value for the trailer: you'll notice our entity-level custom validator is triggered.

RIA SERVICES

MOVIE RENTAL MANAGEMENT SYSTEM

MOVIE OVERVIEW

Title	Director	Date published	Trailer location	
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998		
Apocalypse Now	Francis Ford Coppola	8/15/1979		
Cidade de Deus	Fernando Meirelles	5/7/2011		

1 Error

If you know the director and the title, and the movie is recent, you must fill out a location for the trailer.

Save changes

How it works...

In this recipe, we're using custom validators to solve various validation scenarios. In the second step, we've created a new file, `MovieValidators.shared.cs`. By ending this filename with `shared.cs`, we let WCF RIA Services know this file should be copied to the client (which happens on each build): this is essential, as this is what allows us to *write the validation just once, but have it available on the client as well as on the server*.

From step 3 on, we're writing a custom validator for the `PublishedDate` property, to ensure it isn't in the future. Have a look at the following code:

```
public static class MovieValidators
{
    public static ValidationResult PublishedDateMustBeInThePast(
        DateTime publishedDate,
        ValidationContext validationContext)
```

Important to notice is that the `MovieValidators` class must be `static`. Besides that, the validator itself should also have a specific signature: it should be `static`, return a `ValidationResult`, and accept a parameter of the type you're going to validate and a `ValidationContext`.

The validation logic itself is pretty straightforward:

```
if (publishedDate > DateTime.Now)
{
    return new ValidationResult(
        "You cannot input a date that's in the future.",
        new[] { validationContext.MemberName });
}

return ValidationResult.Success;
```

We check if the inputted date isn't in the future. If it isn't, we return a successful `ValidationResult`. If it is, we return a `ValidationResult` in which we input an error message, and we pass in the `MemberName` from the `validationContext`: this ensures the validation error will be associated with the `MemberName` we're applying the validator to, in this case: `PublishedDate`. Thanks to this association, the binding engine knows where to show the validation error.

Next up is step 6: applying this custom validator to the `DatePublished` property; this is done by decorating this property with the `CustomValidation` attribute, passing in the `MovieValidators` type and the method we want to use for validation: `PublishedDateMustBeInThePast`.

That's all that needs to be done. When we now build our solution, all the necessary code gets generated: if you look at the generated code file on the client, you'll see the `CustomValidation` attribute is propagated there, as well as the shared code file: this means the validation will be triggered from the client as well. If we now try to input a date that's in the future, the validation will occur in the `PublishedDate` property setter, and the error message will be shown (for more information on how the binding engine handles this, have a look at the previous recipe, *Validating data: using data annotations*).

Of course, not all validation scenarios can be handled with a simple property validator. In the rest of this recipe, we'll look into two other types of validation: cross-field validation and entity-level validation.

The general logic is the same: write a validator in a shared class, use the `CustomValidation` attribute, and the validation will be available on and triggered from both client and server. We'll look into the differences.

In step 8, we wrote a `TrailerAndDirector` validator, which implements the validation rule stating that the `Director` must be filled out when a location for the trailer is available. This is a cross-field validator: important to notice here is that when the validation fails, we pass in both the `TrailerLocation` and `Director` member names (so the error is attributed to both properties). The other thing you need to take care of is that the validator must be set, through the `CustomValidation` attribute, on both the `Director` and `TrailerLocation` properties, which is done in step 9.

Next up is an entity-level validator. Instead of passing in a property, we get the complete entity as a parameter in our validator. Important to know is that you don't need to pass in member names in the `ValidationResult`: the error is attributed to the complete entity, not to a specific property (or list of properties) of that entity. Also, the `CustomValidation` attribute is now applied to the `Movie` entity itself, not to a specific property in the metadata class.

Note that the validation engine first triggers property-level validation, and only when all those are OK, is entity-level validation triggered. There's a bit of overlap between cross-field validation on the property level and entity-level validation: both are typically used for cross-field validation. The choice mostly depends on when you want the validation to occur, and how complicated it is. For example: once you've got validation spanning 3 fields, cross-level validation on the property level quickly gets very complicated, both for the developer and for the end user, while still being straightforward on the entity level.

There's more...

Validators can be reused quite easily, but there's an even more reusable way of adding validation to your applications: by using custom validation attributes instead of custom validation methods.

For example, we could take our date validation, and put it in a custom attribute, which can then be reused on other DateTime fields. To achieve this, we need to do two things: first, we'll add a new file to the Validators folder, `DateMustBeInThePastAttribute.shared.cs`, and implement it as such:

```
public class DateMustBeInThePastAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        DateTime date = (DateTime)value;

        if (date > DateTime.Now)
        {
            return new ValidationResult(
                "You cannot input a date that's in the future.",
                new[] { validationContext.MemberName });
        }

        return ValidationResult.Success;
    }
}
```

As you can see, we're inheriting from `ValidationAttribute`, and implement one method: `IsValid`.

To use this attribute, we can simply decorate the property on which we need this validation to occur with it:

```
[DateMustBeInThePast()]
public DateTime DatePublished { get; set; }
```

You can find the complete solution in
`Chapter 12\Validation_Custom_Validator_Attribute_Completed`.

See also

For more information on the general validation mechanism (how it works together with the binding engine), have a look at the *Validating data: using data annotations* recipe.

For other validation scenarios, have a look at the other validation-related recipes in this chapter.

Validating data: server-side validation with client-side feedback

Applies to Silverlight 4, 5

Validation of your data before persisting it is a requirement for almost every application. By using validation on your Entities, you make sure that no invalid data is persisted to your data store. With WCF RIA Services, there are various ways to do validation, each tailored to a specific scenario.

In this recipe, you'll learn how to validate data in which the validator should execute different code on server versus client, which is typically used when you want direct client-side feedback for cases where the validation logic has to execute a call to your server. A "username must be unique" validator is a very common scenario for such a validator.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, Using WCF RIA Services, for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Validation_Feedback_Starter.

The complete solution can be found in Chapter 12\Validation_Feedback_Completed.

How to do it...

We're starting from the provided starter solution, and we're going to add a custom validator to check for a unique title to our application: this validator should execute different code depending on where it is executed (client versus server). To achieve this, we'll complete the following steps:

1. Open MovieDomainService.cs in RIAServices.DomainServices, and add the following Invoke method:

```
[Invoke]
public bool TitleAlreadyExists(string title, int currentMovieId)
{
    // not a new movie
    if (currentMovieId > 0)
    {
        return this.ObjectContext.Movies
            .Where(m => m.MovieID != currentMovieId && m.Title.
ToLower() == title.ToLower()).Count() > 0;
```

```
        }
    else
    {
        return this.ObjectContext.Movies
            .Where(m => m.Title.ToLower() == title.ToLower()) .
Count() > 0;
    }
}
```

2. Open MovieValidators.shared.cs, and add the following using statement, including the compiler directive:

```
#if SILVERLIGHT
using System.ServiceModel.DomainServices.Client;
#endif
```

3. Add the following validation method:

```
public static ValidationResult TitleMustBeUnique(string title,
    ValidationContext validationContext)
{
    ValidationResult errorResult = new ValidationResult(
        "Title must be unique",
        new string[] { validationContext.MemberName });

    Movie movie = (Movie)validationContext.ObjectInstance;

    int currentID = -1;
    if (movie.MovieID > 0)
    {
        currentID = movie.MovieID;
    }

#if !SILVERLIGHT

    MovieDomainService movieDS = new MovieDomainService();
    if (movieDS.TitleAlreadyExists(title, currentID))
    {
        return errorResult;
    }

#else

    MovieDomainContext context = new MovieDomainContext();

    InvokeOperation<bool> ioTitleExists = context.
TitleAlreadyExists(title, currentID);
```

```

        ioTitleExists.Completed += (send, args) =>
    {
        if (!ioTitleExists.HasError && ioTitleExists.
Value)
        {
            Entity entity = (Entity)validationContext.
ObjectInstance;
            entity.ValidationErrors.Add(errorResult);
        }
    };
#endif

        return ValidationResult.Success;
}

```

4. Locate the `Title` property in the `MovieMetadata` class in `MovieDomainService.metadata.cs`, and add the following attribute:

```
[CustomValidation(typeof(MovieValidators), "TitleMustBeUnique")]
public string Title { get; set; }
```

5. You can now build and run your application. You'll notice an error if you try to input a title that's already in use.

Title	Director	Date published	Trailer location
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998	
Cidade de Deus	Title must be unique 3/15/1979		
Cidade de Deus	Fernando Meirelles	5/7/2003	

Save changes

How it works...

With WCF RIA Services, validation logic can be shared between the client and the server. But in some cases, the logic that needs to be executed is different depending on the side you're on. When we want to check for a unique Title from the client, we need to contact the server *asynchronously* from the client through the domain context when this validation occurs (fetching all the titles to the client for this type of validation would of course be very bad practice). But when we're on the server, we don't need to do a call to the server through the domain context anymore – as we're already there. A validator like this can be written through the use of compiler directives.

We start out by defining an `Invoke` method that will check if the `Title` is unique: this is the method that will be called when the validator is executed on the client. In the validator itself, we need to be able to make a distinction depending on the environment we're in: that's why we use the `SILVERLIGHT` compiler directive: this compiler directive tells the compiler that this part of the code should be taken into account when we're building against the Silverlight CLR (when we're on the client), and the other part otherwise (when we're on the server). Related to our code, this means that when we're building against the Silverlight CLR, the validator will call the `Invoke` method on `MovieDomainContext`. When we're building against the regular .NET CLR, the `TitleAlreadyExists` method can simply be called through the `MovieDomainService`.

All that's left is applying the custom validator to the `Title` attribute, which is done by adding the `CustomValidation` attribute in step 4.

When we run the application and change the `Title` property, the validation will occur: as we're on the client, the `Invoke` method will be invoked for our validation. If we're on the server, validation will occur through `MovieDomainService`.

There's more...

As you might have noticed, this validator actually creates a dependency between itself and the data layer. If you want a more loosely coupled solution for situations like this, have a look at the *Validating data: using the ValidationContext recipe*.

See also

For more information on the general validation mechanism (how it works together with the binding engine), have a look at the *Validating data: using data annotations* recipe. For more information on using shared code for validation, have a look at recipe *Validating data: writing a custom validator*.

For other validation scenarios, have a look at the other validation-related recipes in this chapter.

Validating data: triggering validation when needed

Applies to Silverlight 4, 5

Validation of your data before persisting it is a requirement for almost every application. By using validation on your Entities, you make sure that no invalid data is persisted to your data store. With WCF RIA Services, you've got the option to write validation logic once, and it will be executed both on the client and the server. A lot of this is done automatically: in property setters, on the client before submitting to the server, on the server before submitting to the data store, ... but sometimes, you want to trigger validation exactly when you need it: this is what this recipe is about.

In this recipe, you'll learn to manually trigger validation.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, Using WCF RIA Services, for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Validation_Trigger_Starter.

The complete solution can be found in Chapter 12\Validation_Trigger_Completed.

How to do it...

We're starting from the starter solution, and we're going to manually trigger validation from code, on a button click. To achieve this, we'll complete the following steps:

1. Open Home.xaml in RIAServices.Client, and replace the DataGridTextColumns for Title and Director with this code:

```
<sdk:DataGridTemplateColumn Header="Title">
    <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <TextBox Text="{Binding Title, Mode=TwoWay}"
                    IsEnabled="False"
                    BorderThickness="0"
                    Background="Transparent" />
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
    <sdk:DataGridTemplateColumn.CellEditingTemplate>
```

```

<DataTemplate>
    <TextBox Text="{Binding Title, Mode=TwoWay}"
             />
</DataTemplate>
</sdk:DataGridTemplateColumn.CellEditingTemplate>
</sdk:DataGridTemplateColumn>
<sdk:DataGridTemplateColumn Header="Director">
    <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <TextBox Text="{Binding Director, Mode=TwoWay}"
                     IsEnabled="False"
                     BorderThickness="0"
                     Background="Transparent" />
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
    <sdk:DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
            <TextBox Text="{Binding Director, Mode=TwoWay}" />
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellEditingTemplate>
</sdk:DataGridTemplateColumn>

```

2. Open `HomeViewModel.cs`, and add the following using statements:

```

using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

```

3. Locate the `Validate()` method, and implement it as such:

```

private void Validate()
{
    foreach (var movie in MovieContext.Movies)
    {
        List<ValidationResult> vResults = new
List<ValidationResult>();

        if (!Validator.TryValidateObject(movie,
                new ValidationContext(movie, null, null),
vResults))
        {
            vResults.ForEach(vr => movie.ValidationErrors.
Add(vr));
        }
    }
}

```

4. You can now build and run your application. When you press the **Validate** button, you'll notice the expected validation errors appear on the fields of the newly added movie.

The screenshot shows a user interface for a 'MOVIE RENTAL MANAGEMENT SYSTEM'. At the top, a green bar displays 'RIA SERVICES'. Below it, a section titled 'MOVIE OVERVIEW' contains a table with four columns: 'Title', 'Director', 'Date published', and 'Trailer location'. The table has three rows of data: 'Fear and Loathing in Las Vegas' (Director: Terry Gilliam, Date: 5/22/1998), 'Apocalypse Now' (Director: Francis Ford Coppola, Date: 8/15/1979), and 'Cidade de Deus' (Director: Fernando Meirelles, Date: 5/7/2003). A fourth row is partially visible. In the 'Director' column of this fourth row, there is a red rectangular highlight around the input field, and a yellow border highlights the entire row. A red callout box with white text appears next to the input field, stating 'You must provide a director for this movie.' At the bottom of the table are two buttons: 'Validate' and 'Save changes'.

How it works...

WCF RIA Services executes validation logic automatically on different occasions: it will trigger validation when setting a new property value. It will also trigger validation when you're submitting your data to the server (on the client), and before the data is saved to your data store (on the **server**). Sometimes, that might not be enough: through the `Validator` class, we can easily trigger validation from code any time we need it.

We'll start out by explaining how to trigger validation manually, which is done in step 3: the `Validator` class has a method, `TryValidateObject`, which expects an entity to validate, a `ValidationContext`, and an output parameter, which will contain the validation results. We run through all the movies on our context, and validate all of them. If the validation fails, we manually add the validation results to the movies' `ValidationErrors` collection. These validation results include the correct member names from our validators, which (together with the binding engine) ensure the errors are visible in the UI.

Important to notice is that we need to manually add these errors: unlike what is the case with regular validation, they aren't automatically added when validating like this.

`TryValidateObject` has one overload, allowing you to request validation of all the properties as well as the object. Two other methods of interest exist: `TryValidateProperty`, which validates a property, not a complete object, and `TryValidateValue`, which allows you to pass in a list of attributes to be used as validation rules.

You might notice other methods exist on the `Validator` class: `ValidateObject`, `ValidateProperty`, and `ValidateValue`. However, these methods will throw `ValidationExceptions` when validation fails, while the `TryValidate`-methods work with a list of `ValidationResults`: the latter is better suited for our scenario, as this works together nicely with the `INotifyDataErrorInfo` interface (for more information on this, have a look at the recipe *Validating data: using data annotations*).

There's one other thing: in the first step, we've changed the `TextColumns` of the `DataGridView` to `TemplateColumns`, using a disabled `TextBox` as `CellTemplate`. This is done for cosmetic reasons: by default, a `TextColumn` uses a `TextBlock` to display the bound value when you're not in edit mode. But, the default `TextBlock` template doesn't have a way to show validation errors. This would mean the validation errors wouldn't be visible on rows that aren't in edit mode. By changing the `CellTemplate` to a `TextBox`, we get to see the errors as expected, as a `TextBox` does have a template that includes a way to show these validation errors. This is purely cosmetic, but it ensures a better user experience. You can still template the `TextBox` to your liking, of course.

There's more...

When you're validating manually, you need to add the validation results to the `ValidationErrors` collection on your entity (when needed). This is quite repetitive code, but there's a way to make this somewhat easier: by writing an extension on the `Entity` class to validate the entity, automatically adding the validation results to the error collection.

To achieve this, complete the following steps (starting from the completed solution from this recipe):

1. Add a new class, `EntityExtensions`, to the Silverlight client project, and implement it as such:

```
public static class EntityExtensions
{
    public static bool TryValidate(this Entity entity, bool
validateAllProperties = true,
                                bool applyValidationErrorsOnObject = true, bool
leaveExistingValidationErrorsWhenValid = false)
    {
        var valResults = new List<ValidationResult>();
        var validationResult = false;

        if (Validator.TryValidateObject(entity, new
ValidationContext(entity, null, null),
                                    valResults, validateAllProperties))
    {
        if (!leaveExistingValidationErrorsWhenValid)
```

```
        {
            entity.ValidationErrors.Clear();
        }
        validationResult = true;
    }
    else
    {
        // not valid. Apply errors on object?
        if (applyValidationErrorsOnObject)
        {
            entity.ValidationErrors.Clear();

            foreach (var valResult in valResults)
            {
                entity.ValidationErrors.Add(valResult);
            }
        }
    }

    return validationResult;
}
}
```

2. Add the following using statements to this class:

```
using System.ServiceModel.DomainServices.Client;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
```

3. Locate the Validate() method in HomeViewModel.cs, and change it to this:

```
private void Validate()
{
    foreach (var movie in MovieContext.Movies)
    {
        movie.TryValidate(true, true, false);
    }
}
```

4. You can now build and run the application. It behaves the same as the one we completed previously in this recipe, but now we've got cleaner and reusable code.

The complete solution can be found in
Chapter 12\Validation_Trigger_Extension_Completed.

See also

For more information on the general validation mechanism (how it works together with the binding engine), have a look at the *Validating data: using data annotations* recipe. For more information on using shared code for validation, have a look at the recipe *Validating data: writing a custom validator*.

For other validation scenarios, have a look at the other validation-related recipes in this chapter.

Validating data: using the ValidationContext

Applies to Silverlight 4, 5

Validation of your data before persisting it is a requirement for almost every application. By using validation on your entities, you make sure that no invalid data is persisted to your data store. With WCF RIA Services, you've got the option to write validation logic once, and it will be executed both on the client and the server. In some cases, you will need additional data in your validators: a user role, a value depending on in which company location you are. To enable these scenarios, you can use the ValidationContext, which is explained in this recipe.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services* for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Validation_ValidationContext_Starter.

The complete solution can be found in Chapter 12\Validation_ValidationContext_Completed.

How to do it...

We're going to write a custom validator, which will check if a Movie entity has a recent PublishedDate. We'll use the ValidationContext to provide us with a RecentDateValue to check the published date against. To achieve this, we'll complete the following steps:

1. Open MovieDomainService.cs, in RIAServices.DomainServices, override the Initialize method, and implement it as such:

```
public override void Initialize(DomainServiceContext context)
{
```

```
Dictionary<object, object> myCustomItems = new  
Dictionary<object, object>()  
{  
    {"RecentDateValue",  
        new DateTime(2010,1,1)}  
};  
  
this.ValidationContext = new ValidationContext(this,  
context, myCustomItems);  
base.Initialize(context);  
}
```

2. Open HomeViewModel.cs in the client application, and add the following code to the constructor:

```
Dictionary<object, object> myCustomItems = new  
Dictionary<object, object>()  
{  
    {"RecentDateValue",  
        new DateTime(2010,1,1)}  
};  
  
MovieContext.ValidationContext = new ValidationContext(this.  
MovieContext, null, myCustomItems);
```

3. Open MovieValidators.shared.cs, and add a new validator:

```
public static ValidationResult PublishedDateMustBeRecent(  
    DateTime publishedDate,  
    ValidationContext validationContext)  
{  
    // get date to compare to  
    if (validationContext.Items.ContainsKey("RecentDateValue"))  
    {  
        DateTime RecentDateValue = (DateTime)validationContext.  
Items["RecentDateValue"];  
        if (publishedDate < RecentDateValue)  
        {  
            return new ValidationResult(  
                "You cannot input a date that isn't recent."  
                , new[] { validationContext.MemberName });  
        }  
    }  
    return ValidationResult.Success;  
}
```

4. Open `MovieDomainService.metadata.cs`, locate the `DatePublished` property in the metadata for the `Movie` entity, and decorate it with the following attribute:

```
[CustomValidation(typeof(MovieValidators),  
"PublishedDateMustBeRecent")]
```

5. You can now build and run your application. If you input a date before the `RecentDateValue` value, you'll see the expected validation error:

Title	Director	Date published	Trailer location
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998	
Apocalypse Now	Francis Ford Coppola	8/15/1979	
Cidade de Deus	Fernando Meirelles	<input type="text" value="5/7/2009"/>	You cannot input a date that isn't recent.

How it works...

As you might have noticed in one of the previous recipes, when you're writing a custom property validator, a `ValidationContext` parameter is available: WCF RIA Services provides this to every validator. A `ValidationContext` has an `Items` property, a `Dictionary<object, object>`: this can be used as a property bag, so any state we might need can be inputted in this bag.

Let's have a look at our validator: we're going to check if the movie we're inputting has a recent date, which we'll check against a `RecentDateValue` value. This `RecentDateValue` will be made available through the `ValidationContext`: this is something that might change, and we don't want to rewrite our validators each time; you might want to put this value to compare to in a database or configuration file, and fetch it from there.

Validation occurs both on the client as well as on the server side, so we need to provide the `ValidationContext` on both sides. On the client, this is done through the `DomainContext` instance we're using: in our case, the `MovieDomainContext`. This is what happens in step 2: we create a new `Dictionary<object, object>`, adding our `RecentDateValue` property, and set the `ValidationContext` on the `MovieDomainContext` to a new instance, constructing it with our dictionary as its `Items` property.

On to the server, to provide a custom `ValidationContext` to our domain service, the ideal place is the `Initialize` method, as this is always called before any validation occurs. This is what happens in step 1: just as on the client, we create a custom `ValidationContext` and provide it to our domain service.

Note that, besides passing it into the `ValidationContext` constructor, you can also add items to the `Item` dictionary at a later point in time. This is particularly useful when you're on the client: a domain context's lifetime can be quite long, often it's kept alive until the application is exited, while a domain service lifetime is short-lived: every time you invoke it, a new instance is created.

On to the validator itself: through the `ValidationContext` parameter, we now have access to our `RecentDateValue` value: we simply fetch it from the `Items` dictionary, and use it to validate the `PublishedDate`.

That's all there is to it: when the `PublishedDate` property is validated, our custom validator will fire, and it will have access to the `RecentDateValue` value through the `Items` property of the `ValidationContext`.

There's more...

You might have noticed there's another parameter we can pass into the constructor of a `ValidationContext` object: an `IServiceProvider` implementation. This interface defines one method: `GetService`.

How would this come in handy? Well, as we've seen in the recipe *Validating data: server-side validation with client-side feedback*, validators sometimes require access to your data layer. However, in the recipe *Validating data: server-side validation with client-side feedback*, we've actually created a dependency between our validator and our data layer: not an ideal situation.

Knowing this, you could refactor that validator using a custom `ValidationContext`, in which you pass in a service provider. This in turn accesses the database to check for a unique title. Like that, you remove the dependency to the data access layer from the validator, which is especially useful if you want to reuse your validators and achieve better separation of concerns.

Note that you can add `IServiceProvider` implementations at any time through `ValidationContext.ServiceContainer.AddService`, on the server. On the client, this isn't available: you can only add them during initialization of the `ValidationContext`. Besides that, when you're on the server, it's important to know that the `DomainServiceContext` itself (which you get as a parameter in the `Initialize` method of a domain service) also implements `IServiceProvider` – so you can get easy access to its services.

Why is my Validation.Items collection sometimes empty?

If you run the complete application without setting breakpoints, everything will seem to work as it should (and it does). But when you put a breakpoint in the new validator, you'll notice that sometimes the `Items` collection is empty, even though we provide a `ValidationContext` with an `Items` dictionary which contains the `RecentDateValue` Key/Value pair. Why is this?

Validation can be triggered at different moments, and by different controls. In this example, we're using a `DataGrid`, which triggers its own validation (for example: when you start editing a property). But the `DataGrid` control doesn't have any knowledge of WCF RIA Services. So when validation is triggered by the `DataGrid`, it will execute the validators without the custom `ValidationContext` – therefore, the `Items` collection is empty (the same applies for services you provide) in those cases.

Luckily, validation is also triggered in the property setters (amongst other places), which does provide the correct `ValidationContext`, ensuring our validator works as expected.

See also

For more information on the general validation mechanism (how it works together with the binding engine), have a look at the *Validating data: using data annotations* recipe.

For other validation scenarios, have a look at the other validation-related recipes in this chapter.

Handling errors on the server

Applies to Silverlight 4, 5

Keeping track of what happens to your services, especially if something goes wrong, is essential for each business application. In this recipe, you'll learn how you can log errors when they happen on the server, and where you should write the code to do this.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, *Using WCF RIA Services*, for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Handling_Errors_Starter.

The complete solution can be found in Chapter 12\Handling_Errors_Completed.

How to do it...

We're starting from the provided starter solution, and we'll write code to log errors on the server. To achieve this, complete the following steps:

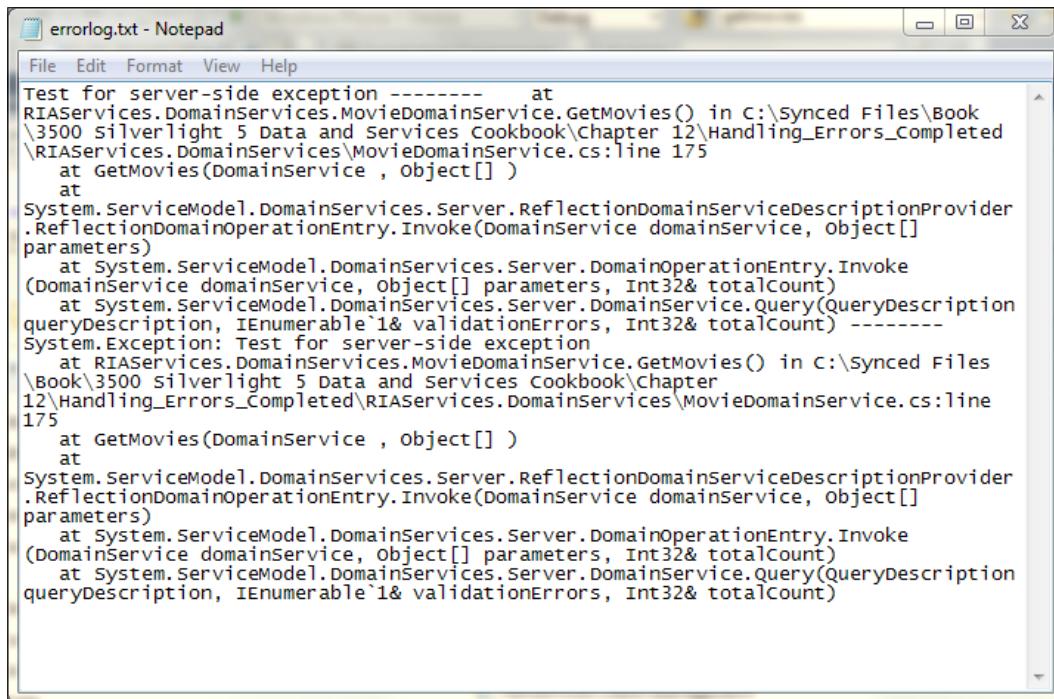
1. Open MovieDomainService.cs, and override the OnError method as such:

```
protected override void OnError(DomainServiceErrorInfo  
errorInfo)  
{  
    string fullError = errorInfo.Error.Message + " -----  
" + errorInfo.Error.StackTrace  
    + " ----- " + errorInfo.Error.ToString();  
  
    if (errorInfo.Error.InnerException != null)  
    {  
        fullError += " ----- " + errorInfo.Error.  
InnerException.ToString();  
    }  
  
    try  
    {  
        System.IO.StreamWriter StreamWriterError =  
            new System.IO.StreamWriter(System.Web.  
HttpContext.Current.Server.MapPath("/errorlog.txt"), true);  
        StreamWriterError.WriteLine(fullError);  
        StreamWriterError.Close();  
    }  
    catch (Exception)  
    {  
        // eat  
    }  
  
    base.OnError(errorInfo);  
}
```

2. Add a reference to System.Web to RIAServices.DomainServices.
3. Locate the GetMovies() method, and change it so it throws an exception:

```
public IQueryable<Movie> GetMovies()  
{  
    throw new Exception("Test for server-side exception");  
  
    return this.ObjectContext.Movies;  
}
```

4. You can now build and run your application. If you set a breakpoint in the `OnError` method, you'll notice the error is caught and written to an `errorlog.txt` file on the server (note: ensure the account you're running the (development server) on has the necessary rights to write to this file on disk).



```
errorlog.txt - Notepad
File Edit Format View Help
Test for server-side exception ----- at
RIAServices.DomainServices.MovieDomainService.GetMovies() in C:\Synced Files\Book
\3500 Silverlight 5 Data and Services Cookbook\Chapter 12\Handling_Errors_Completed
\RIAServices.DomainServices\MovieDomainService.cs:line 175
at GetMovies(DomainService , Object[])
at
System.ServiceModel.DomainServices.Server.ReflectionDomainServiceDescriptionProvider
.ReflectionDomainOperationEntry.Invoke(DomainService domainService, Object[])
parameters)
at System.ServiceModel.DomainServices.Server.DomainOperationEntry.Invoke
(DomainService domainService, Object[] parameters, Int32& totalCount)
at System.ServiceModel.DomainServices.Server.DomainService.Query(QueryDescription
queryDescription, IEnumerable`1& validationErrors, Int32& totalCount) -----
System.Exception: Test for server-side exception
at RIAServices.DomainServices.MovieDomainService.GetMovies() in C:\Synced Files
\Book\3500 silverlight 5 Data and Services Cookbook\Chapter
12\Handling_Errors_Completed\RIAServices.DomainServices\MovieDomainService.cs:line
175
at GetMovies(DomainService , Object[])
at
System.ServiceModel.DomainServices.Server.ReflectionDomainServiceDescriptionProvider
.ReflectionDomainOperationEntry.Invoke(DomainService domainService, Object[])
parameters)
at System.ServiceModel.DomainServices.Server.DomainOperationEntry.Invoke
(DomainService domainService, Object[] parameters, Int32& totalCount)
at System.ServiceModel.DomainService.Query(QueryDescription
queryDescription, IEnumerable`1& validationErrors, Int32& totalCount)
```

How it works...

When an error happens when you load or submit data with WCF RIA Services, you get notified of this on the client: the load or submit operation's `HasErrors` property will be true, and you can handle the error accordingly, depending on your application's requirements.

But besides that, you'll also want to get an overview of these errors on the server. This is what we're doing in this recipe.

Each domain service has an overridable (virtual) method, `OnError`, in which you'll end up whenever an error happens: when fetching data fails, when your business logic throws an exception, ... In the first step, we're overriding this method. This method has a parameter, `errorInfo`, of type `DomainServiceErrorInfo`. This parameter contains an `Error` property, which holds the exception that occurred.

In the third step, we're simulating an error by throwing an exception when loading the `Movie` entities: this will ensure we end up in the `OnError` method.

All that's left now is logging the error: in this recipe, we're writing all the errors to a file, `errorlog.txt`, which you can find in the `RIAServices.WebHost` directory. Of course, you can log errors any way you want: you can log to the event log, you can use a logging component from, for example, Enterprise Library, or you can write your own custom logging logic.

There's more...

You'll also notice the error bubbles up to the client. In most cases, this is what we want: we want to notify the client something went wrong.

Typically, you'd check for errors (through the `HasErrors` property), and show a message accordingly: the `Error` property contains the exception that happened on the server.

See also

For more advanced scenarios for use with WCF RIA Services, have a look at the other recipes in this chapter.

Using SQL Azure with WCF RIA Services

Applies to Silverlight 4, 5

One part of Microsoft's offering in the Azure cloud is **SQL Azure**: a cloud-hosted database service. But can you use WCF RIA Services in combination with a cloud-hosted SQL Azure database? Well, you can, and what's more: it's very, very easy to get this going – actually, it just requires a few simple steps, which you'll learn about in this recipe.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, Using WCF RIA Services, for more information.

Besides that, this recipe also requires you to have access to an SQL Azure environment. As these are paid subscriptions, we cannot provide you with one for testing purposes for this recipe.

We're starting from a provided starter solution, which you can find in Chapter 12\SQL_Azure_Starter.

The complete solution can be found in Chapter 12\SQL_Azure_Completed.

How to do it...

To enable our WCF RIA Services solution to use SQL Azure, we'll modify the provided starter solution. To achieve this, we'll complete the following steps:

1. First of all, you'll need to execute the provided script on your SQL Azure environment, to create the tables and data used by this recipe. Log in to your SQL Azure environment, and execute the `MovieRentalDB_TablesAndData.sql` script, provided in the Chapter 12 directory.
2. Locate the `web.config` file in `RIAServices.WebHost`.
3. Locate the `MovieRentalDBEntities` connection string, and change it to this:

```
<connectionStrings>
    <add name="MovieRentalDBEntities"
        connectionString="metadata=res://*/MovieModel.
        csdl|res://*/MovieModel.ssdl|res://*/MovieModel.
        msdl;provider=System.Data.SqlClient;provider connection strin
        g=INSERTYOURCONNECTIONSTRINGHERE" providerName="System.Data.
        EntityClient"/>
</connectionStrings>
```

4. You can build and run your solution. It will now use the SQL Azure database at the provided location:



The screenshot shows a web-based application interface titled "RIA SERVICES". A green header bar contains the text "MOVIE RENTAL MANAGEMENT SYSTEM". Below this, a sub-header "MOVIE OVERVIEW" is displayed. A table grid lists three movies with columns for Title, Director, Date published, and Trailer location. The first row shows "Fear and Loathing in Las Vegas" directed by Terry Gilliam on 5/22/1998. The second row shows "Apocalypse Now" directed by Francis Ford Coppola on 8/15/1979. The third row shows "Cidade de Deus" directed by Fernando Meirelles on 5/7/2003. At the bottom left of the grid is a button labeled "Save changes".

Title	Director	Date published	Trailer location
Fear and Loathing in Las Vegas	Terry Gilliam	5/22/1998	
Apocalypse Now	Francis Ford Coppola	8/15/1979	
Cidade de Deus	Fernando Meirelles	5/7/2003	

How it works...

As you've noticed, it's very easy to use SQL Azure as a backend data store instead of SQL Server: it actually hasn't got a lot to do with WCF RIA Services at all, since you don't need to change anything on that side: you just need to make sure you use a connection string that refers to an SQL Azure database instead of SQL Server.

There's more...

SQL Azure extends the capabilities you get with SQL Server to the cloud: it's a cloud-hosted, relational database service (full name: Microsoft SQL Azure Database). It should feel very familiar to anyone who's used Transact SQL before: it supports tables, views, stored procedures, index handling, and so on. However, not everything you're used to from SQL Server is supported in the cloud: before migrating, have a good look at the supported Transact SQL statements, which you can find at: <http://msdn.microsoft.com/en-us/library/windowsazure/ee336250.aspx>.

See also

Have a look the other recipes in this chapter for more advanced WCF RIA Services scenarios.

Exposing WCF RIA Domain Services as OData endpoints

Applies to Silverlight 4, 5

An application isn't an island that's standing on its own: often, data from an application is required by other applications. WCF RIA Services allows you to expose your domain services to other applications/clients (for example: a WPF, Windows Forms, ASP.NET, Windows Phone, ... client), as **OData (Open Data Protocol) endpoints** (read-only) and **SOAP or JSON/REST endpoints** (read and persist).

In this recipe, you'll learn how to expose a domain service as an OData endpoint.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from *Chapter 11, Using WCF RIA Services*, for more information.

We're starting from a provided starter solution, which you can find in Chapter 12\Exposing_OData_Starter.

The complete solution can be found in Chapter 12\Exposing_OData_Completed.

How to do it...

To expose an existing domain service as an OData endpoint, we need to make some changes to the web.config file. You can start from the provided starter solution, but any solution with one or more domain services will do. Complete the following steps:

1. Add a reference to System.ServiceModel.DomainServices.Hosting.OData to RIA Services.DomainServices and RIA Services.WebHost.
2. Locate the web.config file in RIA Services.WebHost.
3. Add the following tags to the configuration tag:

```
<configSections>
    <sectionGroup name="system.serviceModel">
        <section name="domainServices" type="System.ServiceModel.
DomainServices.Hosting.DomainServicesSection, System.
ServiceModel.DomainServices.Hosting, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" allowDefiniti
on="MachineToApplication" requirePermission="false" />
    </sectionGroup>
</configSections>
```

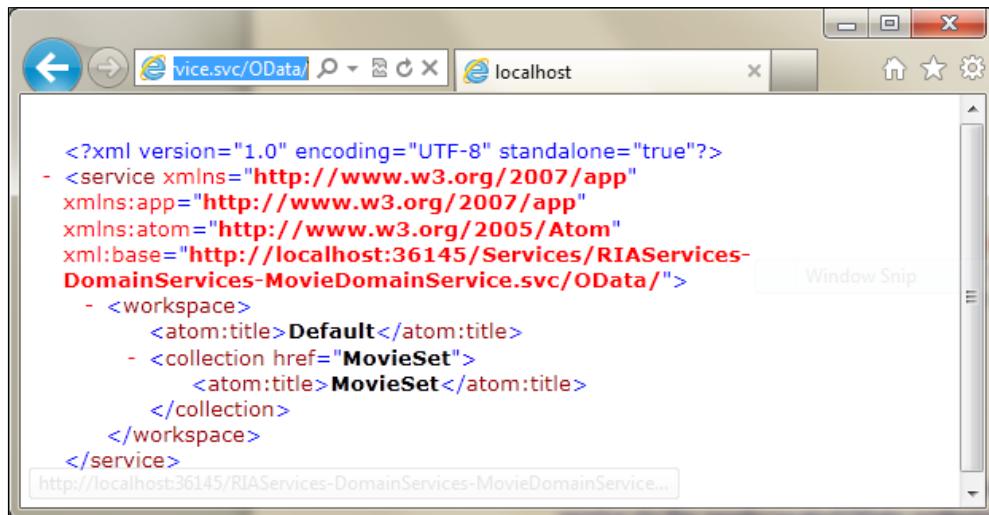
4. Add the following tags to the system.ServiceModel tag:

```
<domainServices>
    <endpoints>
        <add name="OData" type="System.ServiceModel.
DomainServices.Hosting.ODataEndpointFactory, System.
ServiceModel.DomainServices.Hosting.OData, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
    </endpoints>
</domainServices>
```

5. Open MovieDomainService.cs in RIA Services.DomainServices, and add the [Query(IsDefault=true)] attribute to each query method you wish to expose through the OData endpoint. For example: to the GetMovies() method:

```
[Query(IsDefault = true)]
public IQueryable<Movie> GetMovies()
```

6. That's all there is to it: from now on, *other clients can access your domain service via the newly created OData endpoint*. For this example, you can point your browser to `http://localhost:36145/RIAServices-DomainServices-MovieDomainService.svc/OData/`, which is where the endpoint resides:



The screenshot shows a Microsoft Internet Explorer window with the address bar containing `MovieDomainService.svc/OData/`. The page content displays an XML document representing the OData service metadata. The XML includes namespaces for `http://www.w3.org/2007/app` and `http://www.w3.org/2005/Atom`, and defines a workspace named `Default` containing a collection named `MovieSet`.

```

<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <service xmlns="http://www.w3.org/2007/app"
  xmlns:app="http://www.w3.org/2007/app"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xml:base="http://localhost:36145/Services/RIAServices-
  DomainServices-MovieDomainService.svc/OData/">
  - <workspace>
    <atom:title>Default</atom:title>
    - <collection href="MovieSet">
      <atom:title>MovieSet</atom:title>
    </collection>
  </workspace>
</service>

```

http://localhost:36145/RIAServices-DomainServices-MovieDomainService...

How it works...

WCF RIA Services provides built-in support for OData. The first thing we need is some extra configuration in the `web.config` file, which we've done in steps 3 and 4. We add a new endpoint, `OData`, which will work for all domain services in the host.

Next, we need to specify which queries we want to expose, by adding the `[Query(IsDefault=true)]` attribute, as specified in step 5.

To access this endpoint, you need to point your browser to `host/namespace-type/OData/`, in our case: `http://localhost:36145/RIAServices-DomainServices-MovieDomainService.svc/OData/`. The domain service resides in the `RIAServices-DomainServices` namespace, and is named `MovieDomainService`, so "namespace-type" boils down to `RIAServices-DomainServices-MovieDomainService.svc`.

Note that you can let WCF RIA Services create these endpoints automatically through the Add Domain Service wizard, by checking the **Expose as OData endpoint** checkbox.

See also

Have a look at the *Exposing WCF RIA Domain Services for other technologies* recipe for another way to expose your domain service.

Exposing WCF RIA Domain Services for other technologies

Applies to Silverlight 4, 5

WCF RIA Services allows you to expose your domain services to other applications/clients (for example: a WPF, Windows Forms, ASP.NET, Windows Phone, ... client), as OData endpoints (read-only) or SOAP or JSON/REST endpoints (read and persist). This is very useful, as an application isn't an island that's standing on its own: often, data from an application is required by other applications.

In this recipe, you'll learn how to expose a domain service as a SOAP endpoint or JSON /REST endpoint.

Getting ready

Before getting started, you've got to make sure the correct SDK and assemblies to enable WCF RIA Services are available. Please refer to the recipe *Setting up a data solution to work with WCF RIA Services* from Chapter 11, Using WCF RIA Services, for more information.

For this solution, you also need to have installed the WCF RIA Services Toolkit, which you can find at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26939>.

We're starting from a provided starter solution, which you can find in Chapter 12\Exposing_Other_Starter.

The complete solution can be found in Chapter 12\Exposing_Other_Completed.

How to do it...

To expose an existing domain service as a SOAP endpoint, we need to make some changes to the web.config file. You can start from the provided starter solution, but any solution with one or more domain services will do. Complete the following steps:

1. Add a reference to Microsoft.ServiceModel.DomainServices.Hosting (from the Toolkit) to RIAServices.DomainServices and RIAServices.WebHost.
2. Locate the web.config file in RIAServices.WebHost.
3. Add the following tags to the configuration tag:

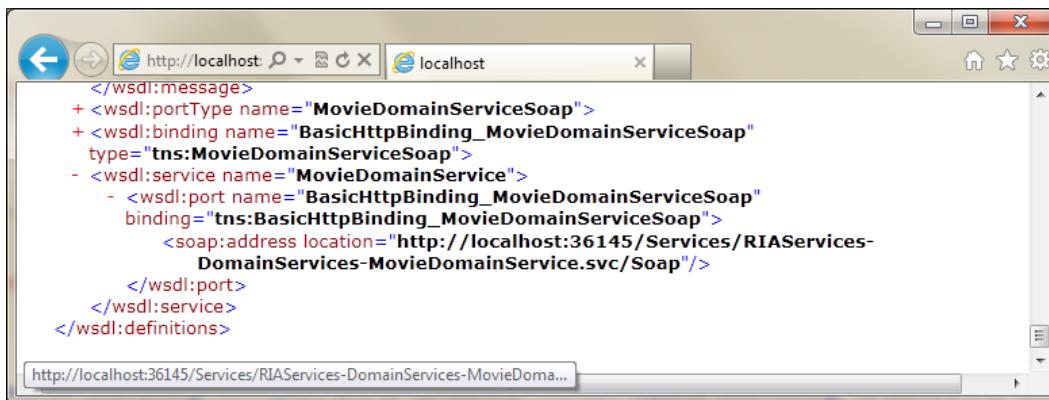
```
<configSections>
    <sectionGroup name="system.serviceModel">
```

```
<section name="domainServices" type="System.ServiceModel.  
DomainServices.Hosting.DomainServicesSection, System.  
ServiceModel.DomainServices.Hosting, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=31bf3856ad364e35" allowDefinition="MachineToApplication" requirePermission="false" />  
</sectionGroup>  
</configSections>
```

4. Add the following tags to the system.ServiceModel tag:

```
<domainServices>  
    <endpoints>  
        <add name="soap"  
            type="Microsoft.ServiceModel.DomainServices.  
Hosting.SoapXmlEndpointFactory, Microsoft.  
ServiceModel.DomainServices.Hosting, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />  
    </endpoints>  
</domainServices>
```

5. That's all there is to it: from now on, other clients can access your domain service via the newly created SOAP endpoint. For this example, you can point your browser to <http://localhost:36145/Services/RIAServices-DomainServices-MovieDomainService.svc?wsdl> (the WSDL), you'll notice the SOAP endpoint location:



How it works...

WCF RIA Services provides support for SOAP endpoints through the WCF RIA Services Toolkit. In the first step, we added a reference to the assembly from the toolkit containing the endpoint factories. The next thing we need is some extra configuration in the `web.config` file, which we've done in steps 3 and 4. We add a new endpoint, SOAP, which will work for all domain services in the host. The `SoapXMLEndpointFactory` will be used to instantiate this.

If you look at the WSDL, by pointing your browser to `http://localhost:36145/Services/RIAServices-DomainServices-MovieDomainService.svc?wsdl`, you'll notice a SOAP endpoint at the bottom of the WSDL. This is the address you'd use in other clients to access the domain service through a SOAP endpoint.

There's more...

Besides exposing a domain service as a SOAP endpoint, there's also built-in support for JSON/REST endpoints in the toolkit. To enable a JSON endpoint instead of (or as well as) a SOAP endpoint, all you need to do is change the tag (or add one) in `system.ServiceModel` in the `web.config` file to this:

```
<add name="Json" type="Microsoft.ServiceModel.DomainServices.Hosting.JsonEndpointFactory, Microsoft.ServiceModel.DomainServices.Hosting, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
```

See also

Have a look at the *Exposing WCF RIA Domain Services as OData endpoints* recipe for another way to expose your domain service.

13

Windows Phone 7

In this chapter, we will cover the following topics:

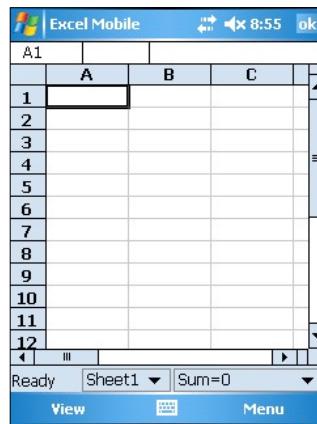
- ▶ Getting our environment ready to start building Windows Phone 7 applications
- ▶ Building your first data-driven Windows Phone 7 application
- ▶ Getting data on your Windows phone 7 using WCF and ASMX
- ▶ Accessing REST services from Windows Phone 7 using XML
- ▶ Accessing REST services from Windows Phone 7 using JSON
- ▶ Working with push notifications using the cloud
- ▶ Storing data in a local SQL CE database
- ▶ Using the background download service

Introduction

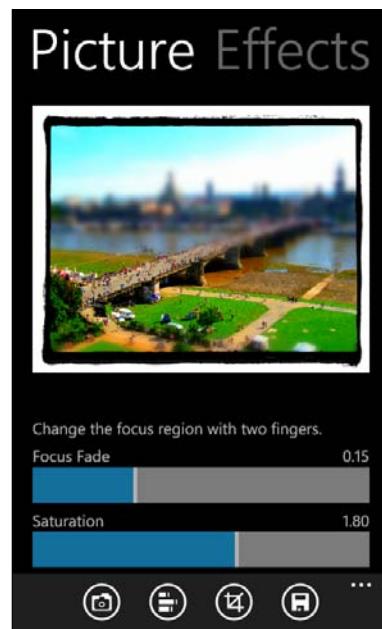
Windows Phone 7 is Microsoft's new mobile platform, replacing the mobile versions known as Windows Mobile (WM) 5, 6, and 6.5. The development platform for this new version is Silverlight or XNA. Silverlight is of course the choice if we want to build line-of-business applications.

For many years, mobile developers have been developing applications for the Windows Mobile platform using the .NET Compact Framework. This framework was actually a trimmed down version of the regular .NET framework, aimed at developing applications for the mobile versions of Windows. While the technology had a lot of strengths, Microsoft acknowledged that it needed something really impressive in the mobile space to compete with the growing competition. One of the areas where WM was running behind was, for example, touch-based input using just your fingers.

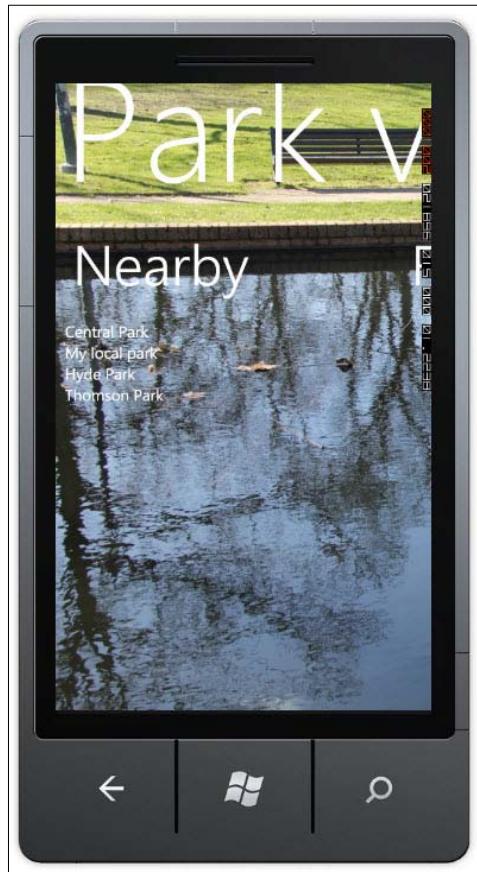
The following screenshot shows a typical Windows Mobile application:



Also, the user experience that could be achieved, as well as performance-related issues, was sometimes a pain for developers. So, in the beginning of 2010, Microsoft introduced **Windows Phone 7** (note that the name is not Windows Mobile 7). WP7 is a **complete break** with the past—it was written from the bottom up and had no resemblance anymore to any previous versions. By looking at some typical WP7 screens shown below, it's easy to see that the new version features a modern and very clean interface.



(Screenshot of Pictures Lab by Schulte Software Development <http://bit.ly/PicLab>)



The phone experience revolves around the **Start Screen**. The goal of this screen is to allow users to have all their relevant information in the blink of an eye. The preceding two screenshots show the typical application interface, with some remarkable items, such as the emphasis on **typography** and easy horizontally and vertically **scrollable** (through the use of touch) applications. (Yes, this is of course difficult to see in a printed version of a book!)

Another break with the past is the development platform. Previously, we had just one choice: .NET Compact Framework. With WP7, we can now create applications with **Silverlight** or **XNA**. Silverlight for WP7 is a subset (and in some areas a superset) of the regular Silverlight: it contains all the usual features, such as data binding, threading, LINQ, Collections, and so on. A specific control set for WP7 is included as well, which is focused on touch input. On top of the Silverlight foundation, APIs for phone-specific functions are added that enable making calls, sending text messages, or taking a picture. XNA should be your choice if you want to build games for the platform (note that you can build games in Silverlight as well though).

It's of course not the goal of this chapter (and in general this book) to teach you everything there is to know about developing for WP7; there are plenty of great resources on that available. Therefore, in this chapter, we'll focus on data and services-related topics, which are what we'll need when building business applications for WP7.

In the very first recipe, we'll build up a basic WP7 application that will guide you through the steps of building a real-life application. That application also forms the foundation for all other recipes in this chapter. Just like other Silverlight applications, WP7 apps need to **access live data** somewhere. This is again done by talking to services. Since we're working with Silverlight code here, most of the things we learned in the Silverlight-specific chapters will apply here as well. We'll see how to communicate with **WCF services** and **REST services** using XML and JSON. We'll also look at how we can integrate server-initiated messages or so-called **push notifications**. These notifications are a way of pushing data to a device and in this way, alerting the user about new updates of data for example.

Before we dive into the recipes, let's take a look at the scenario for this chapter.

The sample scenario for this chapter

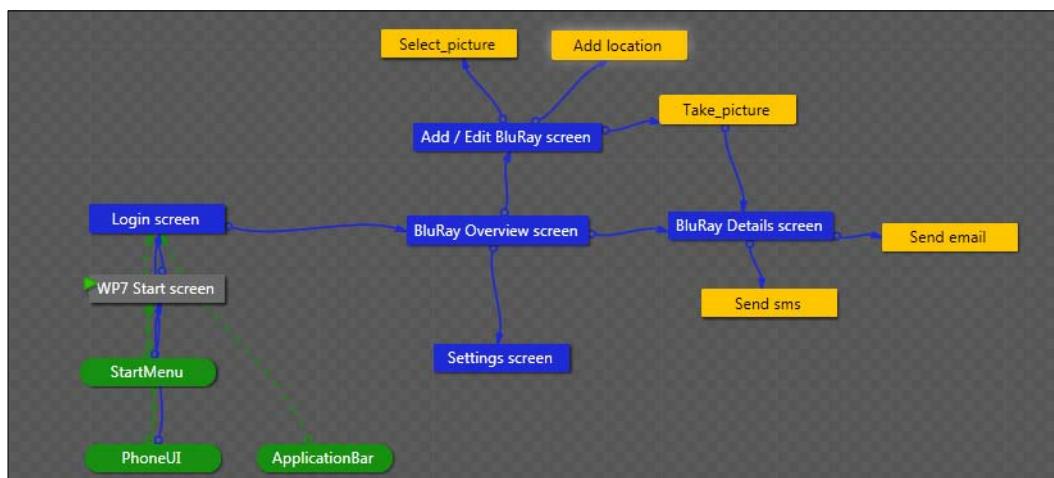
All recipes we'll be looking at in this chapter are centered on the same application called BluRay Collector. Being a movie-fan, I like to buy the occasional Blu-ray disc when I'm on the road. I am, however, really bad at thinking about creating a list with the Blu-rays I already have in my collection, let alone keeping that list up-to-date! For this particular problem, I started thinking about a Windows Phone 7 application and so BluRay Collector was born!

Before building the application, I spent some time thinking about the requirements of the application. Here's a shortlist of requirements that we'll try to cover in our implementation:

- ▶ Multiuser application: the user needs to log in within the application
- ▶ Store the collection on the server (optionally, this can be hosted in Windows Azure)
- ▶ Manage the list of Blu-rays already owned, favorites, and a wish list
- ▶ When buying a Blu-ray, allow the location to be saved
- ▶ Send e-mails or text messages about a Blu-ray
- ▶ Receive notifications about new Blu-rays being released without the application being open
- ▶ Save data locally and send to the central (cloud) store when online
- ▶ Cache data locally

During this chapter, we'll be building several of the preceding requirements into a final application. Note that some items of the preceding list aren't scripted in one of the recipes, simply because they aren't relevant in the context of this book. However, the final application included in the code download contains all code to fulfill these requirements. Take a look in the Chapter13/BluRayCollector_Final folder in the downloads for the source code.

Before writing the code for the application, it's a good practice to build a prototype of the application, so that we get a high-level view of what we are going to build. For this, **SketchFlow**, part of Expression Blend, can be used (note that SketchFlow is only included in Expression Studio Ultimate and not in other versions of Expression Blend. More info on this can be found at http://www.microsoft.com/expression/products/sketchflow_overview.aspx). Using SketchFlow, we can easily create mockups of the screens we are going to build as well as a map that shows the way these screens can be navigated to. In the following diagram, we see the map created by SketchFlow, showing the navigation within the application.



The very first screen is a **Login** screen, allowing the user to identify himself/herself within the application. Once authenticated, the user arrives on the **Blu-Ray Overview** screen. This screen is a Panorama page. A **Panorama** is a Windows Phone 7-specific control giving really rich experiences, similar to the Hubs inside Windows Phone 7 itself (a Hub inside WP7 is a central location where related information is gathered, such as Office, Music, or the Pictures Hub). On this page, which serves as the landing page of the application, the user can see the list of Blu-rays they currently have, their favorites, and their wish list. When clicking on an individual item, a detail page opens (**Blu-Ray Details** screen), allowing the user to see the details of the disc, including an image. The detail page also gives the user a chance to send a text message or a mail about the item to anyone. Finally, the detail page also allows the user to go into edit mode so that they can edit the currently selected disc. When the user wants to add a new disc, they can do so using a button on the **Blu-Ray Overview** screen. The overview screen also allows the user to go to the **Settings** page.

Now that we have an idea about what needs to be built, we can start implementing the functionalities. First, we'll show you what you need to install for WP7 development. The first recipe of this chapter will show how to build the general structure of the application to get acquainted with WP7 development. The following recipes will then focus on the data and services-related issues that this application needs to overcome.

Getting our environment ready to start building Windows Phone 7 applications

Just like Silverlight, Windows Phone 7 development requires some add-ons being installed inside Visual Studio 2010. This recipe walks you through getting your environment ready to build WP7 applications.

How to do it...

The best place to start for all things related to WP7 development is the App Hub (<http://create.msdn.com/en-US/>). From here, you can download the WP7 SDK via http://create.msdn.com/en-us/home/getting_started. The SDK will install the necessary tools as well as the emulator to test your WP7 applications on your PC.

At the time of writing, the newest version of the SDK is 7.1. This version allows us to build applications for Windows Phone 7.5 (also known as **Windows 7 Phone Mango**) and 7.0. This chapter contains recipes that are specific for WP7.5. Where this is the case, this will be clearly indicated. The 7.1 SDK release is fully compatible with the 7.0 version. You should target the 7.1 SDK for every new project.

Building your first data-driven Windows Phone 7 application

As mentioned in the introduction, this first WP7 recipe will guide us through the development of the general structure of the Blu-ray collector application. With WP7, we have the choice between Silverlight and XNA as our development platform. Since the latter is aimed at developing games, the choice for Silverlight to build our app with is quite logical. In this recipe, we'll be creating a few screens for our application, use navigation between the screens and in general, look at how our Silverlight knowledge can be leveraged on WP7. The data we'll be using is hardcoded in the phone application. We'll be looking at how we can access it over a service in the coming recipes. Later recipes will therefore add to the result of this first recipe.

Getting ready

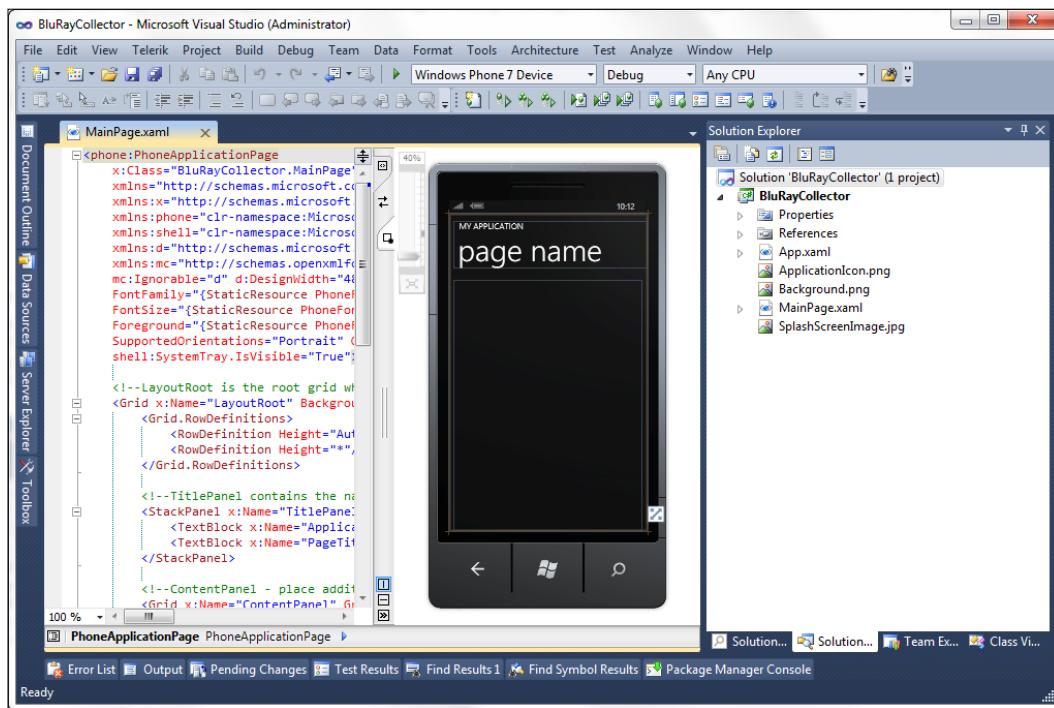
This recipe starts from scratch so there's no specific starting solution. The finished solution for this recipe can be found in the `Chapter13/BluRayCollector_General_Completed` folder.

To build Windows Phone 7 applications, the Windows Phone 7 SDK needs to be installed as outlined earlier in this chapter.

How to do it...

Since this recipe is aimed at getting up-to-speed with WP7 development, we are going to start from scratch. After completing the following steps, you'll have developed a full WP7 application:

1. Open Visual Studio 2010 with the Phone developer tools installed and create a new Windows Phone 7 application from the **New Project Template** window. Note that several options exist, including **Windows Phone Panorama application** and **Windows Phone Pivot application**. For this recipe, we'll start with the regular **Windows Phone Application**. Name the application BluRayCollector. Visual Studio will now offer us by default a split view, showing on one side the preview of the phone application and on the other, the XAML view, as shown in the following screenshot:



2. Looking at the SketchFlow map shown in the scenario outline, we can see that the very first screen should be the Login page, allowing the user to log in to the application. Add a new **Windows Phone Portrait Page** and call it `Login.xaml`.

3. To make this page the startup page for the application, open the WMAppManifest.xml file under the **Properties** item in the **Solution Explorer**. In this file, change the NavigationPage to Login.xaml, as follows:

```
<Tasks>
    <DefaultTask Name ="_default"
        NavigationPage="Login.xaml"/>
</Tasks>
```

4. Now, let's create the layout of the **Login** page. The following XAML code shows the content of the Grid called ContentPanel. The code we are writing here is similar (and in most places identical) to Silverlight code.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Source="/Assets/bluray.png" Margin="0 360 0 0"></Image>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="70"/></RowDefinition>
            <RowDefinition Height="30"/></RowDefinition>
            <RowDefinition Height="80"/></RowDefinition>
            <RowDefinition Height="30"/></RowDefinition>
            <RowDefinition Height="80"/></RowDefinition>
            <RowDefinition Height="70"/></RowDefinition>
            <RowDefinition Height="80"/></RowDefinition>
            <RowDefinition Height="29"/></RowDefinition>
            <RowDefinition Height="*"/></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock Text="Login to your collection"
            FontSize="40" Margin="3">
        </TextBlock>
        <TextBlock Text="User name" Grid.Row="1"
            Style="{StaticResource LabelTextBlockStyle}">
        </TextBlock>
        <TextBox Grid.Row="2" Height="72"
            Text="{Binding UserName, Mode=TwoWay}">
        </TextBox>
        <TextBlock Text="Password" Grid.Row="3"
            Style="{StaticResource LabelTextBlockStyle}">
        </TextBlock>
        <PasswordBox Grid.Row="4" Height="72"
            Password="{Binding Password, Mode=TwoWay}">
        </PasswordBox>
        <CheckBox Content="Remember me?" Name="RememberCheckBox"
            Grid.Row="5">
        </CheckBox>
        <Button Content="Login" HorizontalAlignment="Center"
```

```
Grid.Row="6" Name="LoginButton"
    Click="LoginButton_Click">
</Button>
<TextBlock Name="ResultTextBlock" Foreground="Red"
    Grid.Row="7">
</TextBlock>
</Grid>
</Grid>
```

In the Assets folder for this chapter, the bluray.png file can be found. Place it inside a new folder called Assets inside the WP7 project. Set the Build Action to Content via the **Properties** window. This is required so that Visual Studio will not compile the file but include it directly in the output. The following screenshot shows the result of this XAML code with the image added as well:



5. For now, we won't be adding any real check for the username and password, so we'll just navigate to the Blu-Ray overview screen, which is a Panorama page (see the SketchFlow map for the navigation scheme) when clicking on the **Login** button. Before we can navigate to the page, let's first create it. To do so, add a **Windows Phone Panorama Page** to the application and call it `BluRayOverview.xaml`. In the Login page, within the Visual Studio designer, double-click on the **LoginButton**. Add the following code to the generated event handler to navigate to the overview page:

```
private void LoginButton_Click
    (object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(
        new Uri("/BluRayOverview.xaml", UriKind.Relative));
}
```

A Panorama page contains a Panorama control automatically. Such a control is recognizable by the fact that we can do **horizontal flick** gestures to move from one item to another. The following code shows the Panorama control code. Each `PanoramaItem` instance is a **content control**, meaning that we can place whatever content we want in there.

```
<Grid x:Name="LayoutRoot">
    <controls:Panorama Title="my application">
        <!--Panorama item one-->
        <controls:PanoramaItem Header="item1">
            <Grid/>
        </controls:PanoramaItem>
        <!--Panorama item two-->
        <controls:PanoramaItem Header="item2">
            <Grid/>
        </controls:PanoramaItem>
    </controls:Panorama>
</Grid>
```

6. In the Panorama control, we'll use three items: the first one shows all of our Blu-rays, the second one shows our favorites, and the third one shows the wish list. Since we aren't accessing a service here yet, we are going to be binding to two hardcoded lists of Blu-ray instances. The following is the code for this class; add this class to the WP7 project in a file called `BluRay.cs`:

```
public class BluRay
{
    public int BluRayId { get; set; }
    public string MovieName { get; set; }
    public int MovieLength { get; set; }
    public string MovieImageUrl { get; set; }
```

```
    public bool IsFavorite { get; set; }
    public bool IsBadMovie { get; set; }
    public string MovieDescription { get; set; }
    public string MovieType { get; set; }
}
```

7. In the code-behind of BluRayOverview.xaml, we need to bind three times to an ObservableCollection<BluRay>. Each of these will keep track of a list of Blu-rays. In the code-behind, we are creating three private ObservableCollection<BluRay> objects and a public property for each of them, as follows:

```
private ObservableCollection<BluRay> _allBluRays;

public ObservableCollection<BluRay> AllBluRays
{
    get { return _allBluRays; }
    set { _allBluRays = value; }
}

private ObservableCollection<BluRay> _favoriteBluRays;

public ObservableCollection<BluRay> FavoriteBluRays
{
    get { return _favoriteBluRays; }
    set { _favoriteBluRays = value; }
}

private ObservableCollection<BluRay> _wishlist;
```

8. From the constructor, we are calling a method called InitializeData(). This method generates the hardcoded data to use for now in our UI. The following code shows adding one movie, the rest is omitted:

```
public BluRayOverview()
{
    InitializeComponent();
    InitializeData();
}
```

```
private void InitializeData()
{
    _allBluRays = new ObservableCollection<BluRay>();

    BluRay bluRay = new BluRay();
    bluRay.BluRayId = 1;
    bluRay.IsFavorite = true;
    bluRay.MovieName = "Ice Age";
    bluRay.MovieType = "Animation";
    bluRay.MovieImageUrl = "iceage.jpg";
    bluRay.MovieDescription = "Some description";
    bluRay.IsBadMovie = false;

    _allBluRays.Add(bluRay);
    ...

    _favoriteBluRays = _allBluRays.Where(b =>
    b.IsFavorite == true).ToObservableCollection<BluRay>();

    _wishlist = new ObservableCollection<BluRay>();
    bluRay = new BluRay();
    bluRay.BluRayId = 101;
    bluRay.IsFavorite = true;
    bluRay.MovieName = "Ice Age 4";
    bluRay.MovieType = "Animation";
    bluRay.MovieImageUrl = "iceage4.jpg";
    bluRay.MovieDescription = "Some description";
    bluRay.IsBadMovie = false;

    _wishlist.Add(bluRay);
    ...
}
```

9. The `ToObservableCollection<T>` used above is an extension method. We need to add it ourselves. To do so, add a new class named `CollectionExtensions` to the project. Add the following code to this class. Note that the class is static, which is required to create extension methods.

```
public static class CollectionExtensions
{
    public static ObservableCollection<T>
        ToObservableCollection<T>(this IEnumerable<T> collection)
    {
        var c = new ObservableCollection<T>();
        foreach (var item in collection)
```

```
        c.Add(item);
    return c;
}
}
```

10. To data-bind to the public properties from the code-behind in the UI, we need to set the `DataContext` from the constructor. This line basically sets the current instance of the `Page` as the `DataContext`:

```
this.DataContext = this;
```

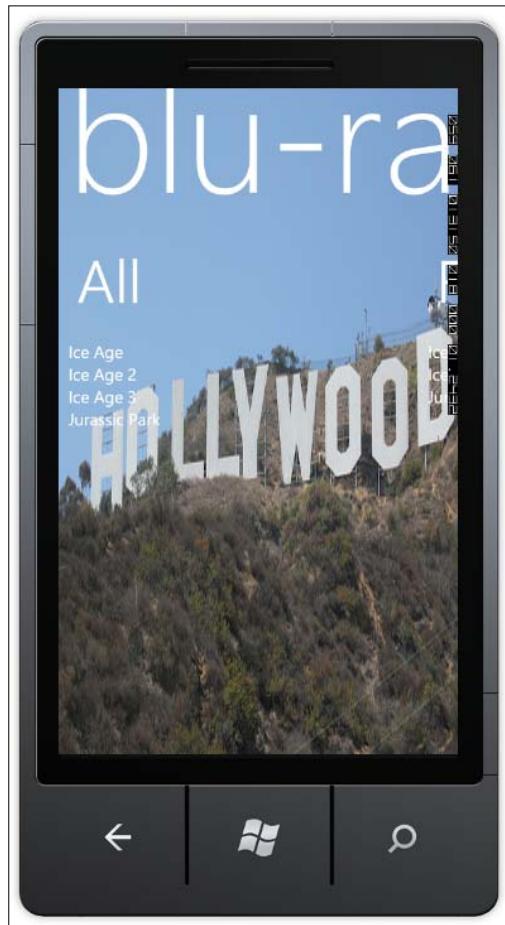
11. With the data exposed, we can now continue our work on the `Panorama` control. In each `PanoramaItem`, we'll add a `ListBox` control. Each of these can bind to one of the just-created `ObservableCollection<BluRay>` objects. We'll also specify through the `DisplayMemberPath` that we want to display the `MovieName` in the `ListBox`. The following code snippet shows how our `Panorama` control is now created:

```
<controls:Panorama Title="blu-ray collector">
    <controls:PanoramaItem Header="All">
        <Grid>
            <ListBox ItemsSource="{Binding AllBluRays}"
                    DisplayMemberPath="MovieName"></ListBox>
        </Grid>
    </controls:PanoramaItem>
    <controls:PanoramaItem Header="Favorites">
        <Grid>
            <ListBox ItemsSource="{Binding FavoriteBluRays}"
                    DisplayMemberPath="MovieName"></ListBox>
        </Grid>
    </controls:PanoramaItem>
    <controls:PanoramaItem Header="Wishlist">
        <Grid>
            <ListBox ItemsSource="{Binding Wishlist}"
                    DisplayMemberPath="MovieName"></ListBox>
        </Grid>
    </controls:PanoramaItem>
</controls:Panorama>
```

12. A nice touch of the `Panorama` control is that we can specify a background. Automatically, this background pans horizontally at a different speed than the actual content of the control (the `ListBoxes`), creating some sort of 3D effect. The following code sets the `Background` property of the control to an image inside the `Assets` folder. You can use any image you want to create this effect.

```
<controls:Panorama.Background>
    <ImageBrush ImageSource="/Assets/Hollywood.jpg">
    </ImageBrush>
</controls:Panorama.Background>
```

The resulting Panorama control is shown in the following screenshot:



13. The next thing we want to add is the detail page of a Blu-ray. This page can be accessed by tapping on an item in either one of the ListBoxes in the Panorama control. Start by adding a new Windows Phone Portrait Page and name it `BluRayDetails.xaml`. For spacing reasons, we won't show the XAML here, you can find it in the finished solution for this recipe.
14. To access the details page from the overview, we'll add an event handler on the `SelectionChanged` event for each of the `ListBox` instances in the Panorama control, as follows:

```
<controls:PanoramaItem Header="All">
    <Grid>
        <ListBox Name="AllBluRayListBox"
            ItemsSource="{Binding AllBluRays}">
```

```

        DisplayMemberPath="MovieName"
        SelectionChanged=
            "AllBluRayListBox_SelectionChanged">
    </ListBox>
</Grid>
</controls:PanoramaItem>
```

15. In the event handler for the `SelectionChanged` event, we can use the `NavigationService.Navigate()` method to navigate to the details page. One other thing we need to do is pass the selected Blu-ray. Since only one single page is in-scope at any time in a WP7 application, we need to solve the issue of passing objects between pages. One thing we can do is use **Isolated Storage** (you can read more on Isolated Storage in *Chapter 5, Working with Local Data*). In the following code, we are saving the selected Blu-ray to Isolated Storage using `IsolatedStorageSettings`. The key is the `BluRayId`; the value is the `BluRay` instance itself. After storing the instance, we navigate to the details page, passing the ID via the URI string:

```

private void AllBluRayListBox_SelectionChanged
    (object sender, SelectionChangedEventArgs e)
{
    if (AllBluRayListBox.SelectedItem != null)
    {
        BluRay selectedBluRay =
            (BluRay)AllBluRayListBox.SelectedItem;
        IsolatedStorageSettings.ApplicationSettings
            [selectedBluRay.BluRayId.ToString()] =
                selectedBluRay;
        NavigationService.Navigate(
            new Uri("/BluRayDetails.xaml?id=" +
            selectedBluRay.BluRayId.ToString(),
            UriKind.Relative));
    }
}
```

The preceding code requires a `using` statement for the `System.IO.IsolatedStorage` namespace.

16. In the `BluRayDetails` page, we can capture the passed-in ID in the `OnNavigatedTo()` method. This method gets called automatically by the navigation framework upon navigating to the page, similar to what happens in Silverlight itself. Using the `NavigationContext`, we can access the ID. With this ID in hand, we can access the Isolated Storage again, retrieving the selected Blu-ray. The following code shows how to do this:

```

private BluRay _selectedBluRay;
private string _selectedBluRayId = string.Empty;
```

```
protected override void OnNavigatedTo
    (System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    if (NavigationContext.QueryString.TryGetValue("id",
        out _selectedBluRayId))
    {
        _selectedBluRay =
            IsolatedStorageSettings.ApplicationSettings
                [_selectedBluRayId] as BluRay;
        this.DataContext = _selectedBluRay;
    }
}
```

Since we are using Isolated Storage classes here again, we need to add a `using` statement for `System.IO.IsolatedStorage`.

17. One last thing we'd like to add is a way to allow the user to navigate to a page where he or she can add a new Blu-ray or navigate to the settings page. These actions should be available from the BluRayOverview page. We can make this possible through the use of an **Application Bar**. An Application Bar can be compared to a menu, allowing the user to perform actions. To add such a menu, add the following code to the overview page. This Application Bar has two buttons, each with an associated event handler.

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar BackgroundColor="#FF000000" >
        <shell:ApplicationBarIconButton
            IconUri="/Assets/Icons/appbar.add.rest.png"
            Text="Add Blu-Ray" x:Name="AddBluRayButton"
            Click="AddBluRayButton_Click">
        </shell:ApplicationBarIconButton>
        <shell:ApplicationBarIconButton
            IconUri="/Assets/Icons/appbar.feature.settings.rest.png"
            Text="Settings" x:Name="SettingsButton"
            Click="SettingsButton_Click">
        </shell:ApplicationBarIconButton>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

18. Add two new Windows Phone Portrait Pages to the project, namely `Settings.xaml` and `BluRayAdd.xaml`.

19. In the event handlers of the `ApplicationBarIconButton`s, navigate to the newly created pages. For now, we'll leave these pages blank. The following code performs the navigation:

```
private void AddBluRayButton_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(
        new Uri("/BluRayAdd.xaml", UriKind.Relative));
}

private void SettingsButton_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(
        new Uri("/Settings.xaml", UriKind.Relative));
}
```

With these steps completed, we have successfully created our first data-driven Windows Phone 7 application!

How it works...

The fact that the development platform for Windows Phone 7 is nothing more than Silverlight is great news for Silverlight developers. In one go, they have become mobile application developers as well, since most of the skills gathered learning Silverlight can be leveraged with Windows Phone 7. However, since it's a mobile platform, not everything that regular Silverlight supports will work in WP7. The following list outlines the most important concepts we are covering in this book and explains how they relate to WP7 development:

- ▶ **Development environment:** Visual Studio and Blend can be used to develop WP7 applications.
- ▶ **Data binding:** Most data-binding features available in Silverlight work in the exact same way as in WP7.
- ▶ **DataGrid:** The DataGrid control doesn't exist in WP7.
- ▶ **Local data/Isolated Storage:** Isolated Storage works in the same way as it does in Silverlight. Mango—local database support.
- ▶ **MVVM:** The principles of MVVM can be applied in WP7 development as well.
- ▶ **Services—ASMX/WCF:** Works in the same way, although not everything is supported. See further recipes of this chapter.
- ▶ **Services—REST:** Works in the same way.
- ▶ **WCF Data Services:** Supported.
- ▶ **WCF RIA Services:** Not supported at this point.

In this recipe, we built a small, data-driven WP7 application. Of course, not everything was covered deeply here. The following sections will give you some more information on WP7-specific topics. In the following recipes of this chapter, we'll take a more in-depth look at the services and data-related topics.

Application model, a device, and the emulator

Windows Phone 7 applications, just like Silverlight applications, are compiled into a XAP file, including all resources required for the application, such as images. After building the application from Visual Studio, the application is ready to be deployed. This deployment can be done from Visual Studio directly onto a device or on the emulator.

To deploy on a real device, the device needs to be **unlocked** to become a developer device. To unlock it, you must register on the **AppHub**, the central location for all WP7 developer-related material (<http://create.msdn.com/en-US/>) as a WP7 developer. This registration costs (at the time of writing) \$99 and not only unlocks the device but also allows you to publish your applications on the marketplace where you can sell them (and perhaps make a good living out of your apps!).

If you don't have a device or don't want to pay for the unlocking process, you can still do WP7 development: when installing the tools, an emulator is installed that you can use to deploy and test your apps. Both ways of working are supported from Visual Studio 2010 and both also support things like setting and hitting breakpoints.

Pages, navigation, and orientation

With Silverlight 3, the **navigation application template** was introduced. This new template allows navigation more easily using the concepts of pages instead of just UserControls. This page-based model also made its way into WP7, being the way to create new **pages** (also known as screens) in your application.

WP7 supports two orientations—**portrait** (vertical) and **landscape** (horizontal). A page has the `SupportedOrientations` property, which specifies which orientations are supported by a page. A page can support both, although we as developers are responsible for re-ordering content when going from one orientation to the other.

When we want to go from one page to another, we again fall back on the Navigation Application concepts. With this model, navigation is done using the `NavigationService`, which exposes a `Navigate()` method. This method accepts the name of the XAML file to navigate to. If we want to pass parameters from one page to another, we can use the `NavigationContext` to capture the parameters from the query string using the following code:

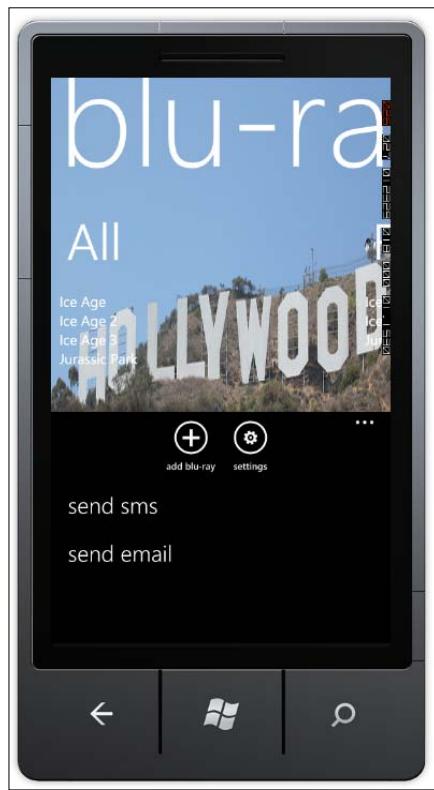
```
NavigationContext.QueryString.TryGetValue("id", out _selectedBluRayId)
```

Application Bar

If one word describes the WP7 interface, *minimalistic* should do the trick. Nowhere in the OS do we find ourselves confronted with a huge list of options or large menus that we need to scroll through. A place where this shines is the Application Bar. This type of menu sits at the bottom of the screen (more generally, it always stays on the side of the hardware buttons if going to landscape mode) and allows the developer to add a maximum of four icon buttons. If we do need more actions to be available, there's an optional extra menu that can pop up if needed. The following code shows an Application Bar with an extra menu:

```
<shell:ApplicationBar BackgroundColor="#FF000000" >
    <shell:ApplicationBarIconButton
        IconUri="Assets/add.png"
        Text="Add Blu-Ray" x:Name="AddBluRayButton"
        Click="AddBluRayButton_Click">
    </shell:ApplicationBarIconButton>
    <shell:ApplicationBarIconButton
        IconUri="Assets/settings.png"
        Text="Settings" x:Name="SettingsButton"
        Click="SettingsButton_Click">
    </shell:ApplicationBarIconButton>
    <shell:ApplicationBar.MenuItems>
        <shell:ApplicationBarMenuItem Text="Send SMS"
            x:Name="SMSMenuItem" Click="SMSMenuItem_Click">
        </shell:ApplicationBarMenuItem>
        <shell:ApplicationBarMenuItem Text="Send Email"
            x:Name="EmailMenuItem" Click="EmailMenuItem_Click">
        </shell:ApplicationBarMenuItem>
    </shell:ApplicationBar.MenuItems>
</shell:ApplicationBar>
```

In the following screenshot, we see the opened menu in the Application Bar:



The Application Bar can be defined in resources (for example in the App.xaml file) and can then be referenced from the page where it's used. This is handy if more than one page uses the same Application bar. Alternatively, the page itself can define its own instance.

Isolated Storage

Just like Silverlight, Windows Phone 7 offers Isolated Storage as a means to allow applications to store data. This data is persistent and can be retrieved after the application has been shut down. When placing information in this store, we are effectively placing the information on the internal storage of the device.

In Silverlight, isolated storage is quota-based. By default, an application gets just 1 MB of storage per user. In the case of Windows Phone 7, there are no quotas and thus each application can store as much as it wants. This holds a potential risk, in that badly written applications might end up storing useless information which they neglect to clean up. This results in a lot of lost storage on the device.

Code-wise, there's little difference with Silverlight. We can create files, read files, and use the `ApplicationSettings` to have access to the internal dictionary to store arbitrary objects.

Panorama and Pivot

Windows Phone 7 has so-called **hubs**. A hub is a central location for specific content. For example, there's a Pictures hub, a Game hub, the Office hub, and the People hub. The Picture hub gathers pictures from the device, from friends, from Facebook, and so on.

This typical interface element was opened for developers as well in the form of the **Panorama** control. This control can be compared to a Tab control. However, the content of the different tabs can be flipped through horizontally. Each `tab`, or `PanoramaItem`, can contain whatever contents we want. In the sample from this recipe, we placed in each item a `ListBox`. A `Panorama` is wider than a screen; it can be compared to a canvas that we are sliding over with the phone, only bringing part of the canvas into the view at each moment. The `Panorama` can have a background image, which scrolls at a different speed than the actual content, giving a 3D/depth effect.

Besides the `Panorama` control, a similar control called the **Pivot** was added. This control works more or less in the same way—we can scroll horizontally as well—the `Pivot` is more aimed at displaying data. In the WP7 OS, the mail inbox uses a `Pivot`: several data views are placed next to one another (all mails, unread mails, high priority mails, and so on), again allowing us scroll through them.

A question that often pops up is "Which of these two similar controls should I use and when should I use them?". While there's no rule set in stone as to which control you should use and when, more often than not, the `Panorama` is used as a navigational element. It serves as a starting place/element to navigate to other pages in our application. The `Pivot` is more aimed at displaying data or providing different views of the same data.

See also

Isolated Storage for Silverlight is deeply covered in *Chapter 5, Working with local data*. Since Isolated Storage in WP7 is similar to the Silverlight implementation, you can take a look there for more coverage on the topic. We used data-binding concepts in this chapter as well (although we didn't spend time explaining them again here). If you want to learn more about data binding in WP7 or Silverlight, take a look at *Chapter 2, An Introduction to Data Binding* and *Chapter 3, Advanced Data Binding*.

Getting data on your Windows phone 7 using WCF

In the previous recipe, we used some hardcoded data in our application to hide the details of working with services in a WP7 application. The focus of this recipe is accessing remote data using WCF (Windows Communication Foundation).

Getting ready

In this recipe, we are continuing with the scenario of the previous recipe but we are going to refactor it to use real data coming from a WCF service. Therefore, you can continue using the code from the previous recipe or use the starting solution provided with the code of the book, located in `Chapter13/BluRayCollector_WCF_Starter`. The completed solution is located in the `Chapter13/BluRayCollector_WCF_Completed` folder.

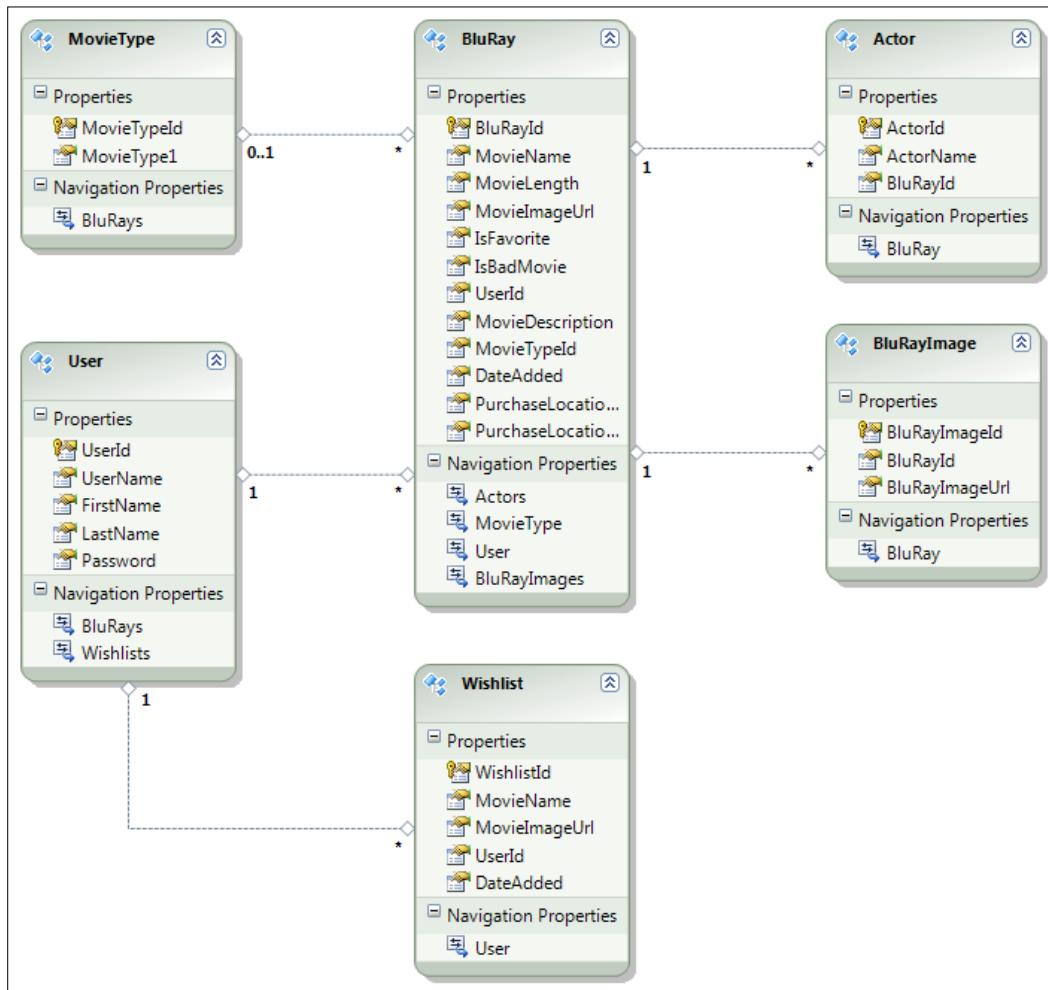
This recipe uses a SQL Server database called `BluRayCollector`. This database needs to be added to your SQL Server instance (this can be either a SQL Server Express or a full version of SQL Server). Refer to the *Appendix* for more information on how to attach this database to your server. For your convenience, the database script (`*.sql`) is also provided with the downloads.

How to do it...

The goal of this recipe is accessing a service to get data within a WP7 application. There are several ways of doing this. In this very recipe, we'll take a look at working with WCF. We'll soon notice that accessing a WCF service from WP7 is not very different from accessing a WCF service from Silverlight. To get this working, you will need to complete the following steps:

1. Start by adding a new web project to the solution. Select the **ASP.NET Empty Web Application** project template and name it `BluRayCollector.Services`. We'll refer to this project as the services project from here on.

2. Add an **ADO.NET Entity Model** to the services project and name it BluRayModel.edmx. In the wizard, select to generate the model from the database and select all tables to be added to the model (apart from the sysdiagrams, should it be shown). Make sure you check the **Pluralize or singularize generated object names** in the wizard. In the following screenshot, we can see the generated entity model:



3. Now add a new **WCF service** that uses the `basicHttpBinding`—a regular WCF service will do fine here—and name it `BluRayService`. Automatically, Visual Studio generates the `BluRayService` class and the `IBlurayService` interface.

-
4. In the service code we need to write, we can return entities coming from Entity Framework. While this is a good choice, there's an important side-effect that we need to consider: the amount of returned data. The EF entities contain quite a lot of data overhead, so it's better to create our own Data Transfer Objects (DTO) and return those to the WP7 device. Let's first create these DTOs. Create a new class library in the solution called `BluRayCollector.Services.DTO`.
 5. In this new class library, create a new class called `BluRay`. The following is the code for the class. Note that it's almost the same class that we used in the previous recipe apart from some new attributes that are being used on the class and its members here.

```
[DataContract]
public class BluRay
{
    [DataMember]
    public int BluRayId { get; set; }

    [DataMember]
    public string MovieName { get; set; }

    [DataMember]
    public int MovieLength { get; set; }

    [DataMember]
    public string MovieImageUrl { get; set; }

    [DataMember]
    public bool IsFavorite { get; set; }

    [DataMember]
    public bool IsBadMovie { get; set; }

    [DataMember]
    public string MovieDescription { get; set; }

    [DataMember]
    public string MovieType { get; set; }

    [DataMember]
    public int MovieTypeId { get; set; }

    [DataMember]
    public MovieType BluRayMovieType { get; set; }

    [DataMember]
    public byte[] ImageBytes { get; set; }
```

```
[DataMember]
public int UserId { get; set; }

[DataMember]
public string PurchaseLocationLatitude { get; set; }

[DataMember]
public string PurchaseLocationLongitude { get; set; }
}
```

The other classes aren't shown here for spacing reasons. Take a look at the sample code of this recipe for the other classes. Add to your project the classes MovieType, User, and Wishlist. Note that the DataContract and DataMember attributes require a using statement for the System.Runtime.Serialization namespace. We also need to add an assembly reference to the System.Runtime.Serialization assembly.

6. Add a reference from the services project to the DTO project so that we can use these DTOs from within the service code.
7. We're now ready to build the service code. In the IBluRayService interface, add a method stub called GetAllBluRaysForUser() as shown in the following code. Note that this method has the OperationContract applied to it. This method is available to be called from clients.

```
[ServiceContract]
public interface IBluRayService
{
    [OperationContract]
    List<DTO.BluRay> GetAllBluRaysForUser(int userId);
}
```

8. With the interface method added, we need to add an implementation in the BluRayService class. Add the following code:

```
public List<DTO.BluRay> GetAllBluRaysForUser(int userId)
{
    BluRayCollectorEntities context =
        new BluRayCollectorEntities();

    var result = from b in context.BluRays
                where b.UserId == userId
                select b;

    return ConvertIntoBluRayDtoList(result.ToList());
}
```

This method returns a `List` of DTO `BluRay` objects. We use **LINQ-To-Entities** code to retrieve all `BluRays` from a specific user from the database. Instead of returning these entities directly, we do a conversion into our DTO objects using the `ConvertIntoBluRayDtoList()` method, as shown in the following code:

```
private List<DTO.BluRay> ConvertIntoBluRayDtoList
    (List<BluRay> list)
{
    List<DTO.BluRay> bluRays = new List<DTO.BluRay>();

    foreach (var item in list)
    {
        DTO.BluRay bluRay = ConvertIntoSingleBluRayDto(item);
        bluRays.Add(bluRay);
    }
    return bluRays;
}

private DTO.BluRay ConvertIntoSingleBluRayDto(BluRay item)
{
    DTO.BluRay bluRay = new DTO.BluRay();
    bluRay.BluRayId = item.BluRayId;
    bluRay.MovieName = item.MovieName;
    bluRay.MovieLength = (int)item.MovieLength;
    bluRay.MovieImageUrl = item.MovieImageUrl;
    bluRay.IsFavorite = (bool)item.IsFavorite;
    bluRay.IsBadMovie = (bool)item.IsBadMovie;
    bluRay.MovieDescription = item.MovieDescription;
    bluRay.UserId = item.UserId;
    bluRay.PurchaseLocationLatitude =
        item.PurchaseLocationLatitude;
    bluRay.PurchaseLocationLongitude =
        item.PurchaseLocationLongitude;
    return bluRay;
}
```

9. With the service code ready, build it. Now go to the WP7 project and add a service reference to the services project. Set the namespace to `BluRayService`. Just as with regular Silverlight, Visual Studio creates a service proxy.
10. In the `BluRayOverview` page, we'll now use the proxy to load in the Blu-rays. An important aspect is that we want to limit the amount of data being sent, so once the list is loaded, we'll cache it in the Isolated Storage. This can easily be seen in the following code snippet. First we check for the list already being cached on the client. If it's not present, we use a service call to get new data. In the callback method, the data is bound and also stored in the Isolated Storage.

```

protected override void OnNavigatedTo
    (System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    BluRayService.BluRayServiceClient proxy =
        new BluRayService.BluRayServiceClient();
    if (!IsolatedStorageSettings.ApplicationSettings.Contains
        ("AllBluRays"))
    {
        proxy.GetAllBluRaysForUserCompleted +=
            new EventHandler<BluRayService.GetAllBluRaysForUserCompletedEventArgs>
                (proxy_GetAddressCompleted);
        proxy.GetAllBluRaysForUserAsync(1);
    }
    else
    {
        AllBluRayListBox.ItemsSource =
            IsolatedStorageSettings.ApplicationSettings
                ["AllBluRays"]
                as ObservableCollection<BluRayService.BluRay>;
    }
}
void proxy_GetAddressCompleted
    (object sender,
     BluRayService.GetAllBluRaysForUserCompletedEventArgs e)
{
    AllBluRayListBox.ItemsSource = e.Result;

    IsolatedStorageSettings.ApplicationSettings["AllBluRays"] =
        e.Result;
}

```

This code is placed in the `OnNavigatedTo()` method, since we want it to execute when the page is being navigated to.

11. We also need to update the properties that were previously using the `BluRay` class. Instead, they now need to use the `BluRayService.BluRay` class. This is shown in the following code:

```

private ObservableCollection<BluRayService.BluRay>
    _allBluRays;
public ObservableCollection<BluRayService.BluRay> AllBluRays
{
    get { return _allBluRays; }
    set { _allBluRays = value; }
}

```

```
        }
        private ObservableCollection<BluRayService.BluRay>
            _favoriteBluRays;
        public ObservableCollection<BluRayService.BluRay>
            FavoriteBluRays
        {
            get { return _favoriteBluRays; }
            set { _favoriteBluRays = value; }
        }
        private ObservableCollection<BluRayService.BluRay> _wishlist;

        public ObservableCollection<BluRayService.BluRay> Wishlist
        {
            get { return _wishlist; }
            set { _wishlist = value; }
        }
```

12. Also the AllBluRayListBox_SelectionChanged() needs to be updated.
Change the following line of code as follows:

```
var selectedBluRay =
    AllBluRayListBox.SelectedItem as BluRayService.BluRay;
```

13. Finally, update the BluRayDetails.xaml.cs to also make use of the
BluRayService.BluRay class.

With this code, the data used in the application is now real data coming from the service.
In the finished application, quite a few more service methods are being used to get, create,
update, and delete data from the WP7 application. Take a look at the finished sample for this!

How it works...

As you may have noticed, working with WCF from WP7 is not very different from working with Silverlight. The steps that we need to follow are the same as we followed when connecting to a service from a Silverlight application:

1. Create a WCF Service that uses basicHttpBinding.
2. Implement the service methods.
3. Create a reference from the WP7 application to the service. Visual Studio creates a proxy class for us.
4. Instantiate the proxy and let the WP7 application perform asynchronous communication with the service.

ASMX services

In this recipe, we worked with WCF services. Would things work with ASMX web services as well? The answer is definitely a yes. Just like Silverlight, WP7 can communicate with ASMX services without any restriction. This type of services defaults to `basicHttpBinding` so there's no change needed in the configuration.

WCF services

For WCF, the story is twofold. On one hand, WCF services can be connected from WP7 applications. However, on the other hand, WCF services can use different types of bindings and these aren't all supported from the phone.

In the preceding steps, we added a **regular WCF service** to the services project and still we were able to connect to this service. Normally, we would expect that this generates a service that uses `wsHttpBinding` and in that case, Silverlight wouldn't be able to communicate with it. Let's look at the generated config code (the generated code is as follows):

```
<system.serviceModel>
    <behaviors>
        <serviceBehaviors>
            <behavior name="">
                <serviceMetadata httpGetEnabled="true" />
                <serviceDebug
                    includeExceptionDetailInFaults="false" /> </behavior>
            </serviceBehaviors>
        </behaviors>
        <serviceHostingEnvironment
            multipleSiteBindingsEnabled="true" />
    </system.serviceModel>
```

As we can see, no service configuration code was added. In .NET 4.0, when we create a new WCF service, it uses the default, being a `basicHttpBinding` for HTTP traffic. The WP7 application can connect to a service with this type of binding without problems. Should we have used a service already configured to use another type of binding, this would not have worked. In this case, we would need to add a new service endpoint that uses the `basicHttpBinding`.

Data transfer

An important aspect to consider in the case of mobile applications is the **amount of transferred data**. Data connections on mobile devices aren't always fast and data subscriptions in many countries are still quite expensive. Therefore, when we build applications for the mobile platform, we need to think of ways to limit the amount of transferred data.

In this recipe, the service-side uses Entity Framework-generated entities. While we could transfer these to the client, we choose not to do so. There are two reasons for this:

- ▶ When transferring the EF entities, there's an overhead created with the amount of generated data when serializing the entities.
- ▶ The entities may (and probably will) contain properties that we aren't using on the client side. Transferring these to the client results in useless data being sent over the wireless connection, costing more in data usage and also causing the application to become slower. Also, the default entity may even contain data that we don't want to expose (for example, a Customer may have a Bank Account Number as a field; we shouldn't send that down to the device).

The solution is building DTOs, classes that merely perform data transfer and have no further goal (notice that there aren't any methods in these classes). On the server side, after executing the query, we then need to convert the objects, but the overhead caused by this translation is minor to the benefit we get in just transferring clean classes with only the properties we need on the client. The DTOs give us much more control over how the data is sent and what data is being sent.

Credentials

If we need to send credentials to a WCF service (preferably in combination with HTTPS!), we can do so from a WP7 application using the following code:

```
proxy.ClientCredentials.UserName.UserName = "gill";  
proxy.ClientCredentials.UserName.Password = "1212";
```

What works and what doesn't work

Throughout the book, we have been using WCF services from Silverlight that are far from regular services. Not everything, however, is supported from WP7 in the WCF area. The following list provides an overview of things that aren't working in the current release of WP7:

- ▶ WCF RIA Services
- ▶ Authentication over NTLM
- ▶ Custom bindings
- ▶ ChannelFactory.Create is protected
- ▶ Duplex communication
- ▶ RSS and Atom (we can use the plain XML parsing for this though)

See also

As already mentioned, working with WCF (and ASMX services) from a WP7 application is similar to Silverlight. Therefore, all concepts explained in Chapters 7 through 9 are relevant here as well.

Accessing REST services from Windows Phone 7 using XML

In *Chapter 10, Talking to REST and WCF Data Services*, we looked extensively at working with REST services. REST services return a clear text response to their clients, in XML or in JSON. Both have the advantage over SOAP messages (used when we work with ASMX and WCF) of having a smaller footprint when it comes to data usage, which is a real plus in the mobile world. In this recipe, we'll rewrite the WCF service so that it uses REST over XML instead of sending SOAP messages. Of course the client needs to be updated as well to read and parse these XML messages, so we will be writing some WP7 code as well!

Getting ready

If you have been working through the other recipes in this chapter, you can keep using your own code. Alternatively, you can use the starting solution located in the `Chapter13/BluRayCollector_XML_Starter`. The completed solution is located in the `Chapter13/BluRayCollector_XML_Completed` folder. This recipe, just like almost all others in this chapter, uses the `BluRayCollector` database.

How to do it...

By default, a WCF service communicates using the SOAP protocol. We can, however, configure it to work as a REST service, so that it returns XML or JSON responses. We covered this in *Chapter 10, Talking to REST and WCF Data Services* but for the sake of simplicity we'll go over the required steps here very quickly. After updating the service, the WP7 client requires changes as well. It can no longer use a generated proxy—we can't add a service reference to a REST service—so we'll need to introduce the `WebClient` or the `HttpWebRequest`. Follow these steps to make the WP7 application read out an XML response, coming from a REST service:

1. The service methods within the `IBluRayService` interface need to be attributed with a `WebGet` attribute. To this attribute, we pass several parameters. The `UriTemplate` defines what the URI pattern should be for clients requesting the service. In this case, we have set this to `getallbluraysforuser/{userid}`. Secondly, the `BodyStyle` is set to `WebMessageBodyStyle.Bare` and finally the `ResponseFormat` is set to `WebMessageFormat.Xml`. This causes the service to respond in XML format. The following code shows the required changes:

```
[OperationContract]
[WebGet(UriTemplate = "getallbluraysforuser/{userid}",
    BodyStyle = WebMessageBodyStyle.Bare,
    ResponseFormat = WebMessageFormat.Xml)]
List<DTO.BluRay> GetAllBluraysForUser(string userid);
```

2. The service implementation code itself doesn't need to be changed.
3. The configuration code, however, needs to be changed. We need to specify that we are using the `webHttpBinding` for this service. The following code should now be the only `system.serviceModel` code in the `web.config` file:

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="webBehavior">
        <webHttp/>
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <services>
    <service
      name="BluRayCollector.Services.RESTBluRayService">
      <endpoint address=""
                binding="webHttpBinding"
                behaviorConfiguration="webBehavior"
                contract="BluRayCollector.Services.IRESTBluRayService"/>
    </service>
  </services>
</system.serviceModel>
```

4. We're ready on the server side; let's now look at the changes we need to do on the client within the WP7 application. We can start by removing the service reference.
5. Next, in the code of the `BluRayOverview` page, in the `OnNavigatedTo()` method, we need to remove the use of the proxy class. Instead, we need to contact the REST service. We learned in *Chapter 10, Talking to REST and WCF Data Services*, that we can do so using either the `WebClient` or the `HttpWebRequest`. We'll use the first one here (the next recipe uses the latter). The following code shows how to perform the service call:

```
private string baseUrl =
  "http://localhost:2956/RESTBluRayService.svc/";

private string allBluRaysForUser = "getallbluraysforuser/{0}";

protected override void OnNavigatedTo
  (System.Windows.Navigation.NavigationEventArgs e)
{
  base.OnNavigatedTo(e);

  if (!IsolatedStorageSettings.ApplicationSettings
    .Contains("AllBluRays"))
```

```

{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += 
        new DownloadStringCompletedEventHandler
        (client_DownloadStringCompleted);
    client.DownloadStringAsync(
        new Uri(
            baseUrl + string.Format(allBluRaysForUser, "1")));
}
else
{
    AllBluRayListBox.ItemsSource =
        IsolatedStorageSettings.ApplicationSettings
        ["AllBluRays"] as ObservableCollection<BluRay>;
}
}
}

```

Note that we are also here checking if the list of BluRays has already been downloaded to the client. If not, we will use the `WebClient` class to communicate in an asynchronous manner with the service.

6. The previous code specified a callback method, called when the service call completes. In this method, we are using LINQ To XML to parse the service response:

```

void client_DownloadStringCompleted
    (object sender, DownloadStringCompletedEventArgs e)
{
    if (e.Error == null)
    {
        string responseString = e.Result;

        XDocument xml = XDocument.Parse(e.Result);
        var bluRays = from results in xml.Descendants("BluRay")
                      select new BluRay
        {
            BluRayId = Int32.Parse(results.Element("BluRayId")
                .Value.ToString()),
            IsBadMovie = Boolean.Parse(results.Descendants
                ("IsBadMovie").First().Value),
            IsFavorite = Boolean.Parse(results.Descendants
                ("IsFavorite").First().Value),
            MovieDescription = results.Descendants
                ("MovieDescription").First().Value,
            MovieLength = Int32.Parse(results.Element
                ("MovieLength").Value.ToString()),
            MovieName = results.Element("MovieName")
        };
    }
}

```

```
        .Value.ToString(),
        MovieType = results.Element("MovieType")
        .Value.ToString()
    };
    AllBluRays = bluRays.ToList()
    .ToObservableCollection<BluRay>();
    AllBluRayListBox.ItemsSource = AllBluRays;
}
}
```

7. The preceding code requires the `BluRay` class within the Silverlight code. In WCF, this class was generated when adding the service reference. With REST, we don't have that advantage. Therefore, we need to add this class ourselves. Create a new class, name it `BluRay`, and add the following code:

```
public class BluRay
{
    public int BluRayId { get; set; }
    public string MovieName { get; set; }
    public int MovieLength { get; set; }
    public string MovieImageUrl { get; set; }
    public bool IsFavorite { get; set; }
    public bool IsBadMovie { get; set; }
    public string MovieDescription { get; set; }
    public string MovieType { get; set; }
}
```

With this code in place, the application works now using XML over a REST service.

How it works...

When a WP7 application needs to work with a REST service that sends XML responses, it can do so in the exact same way as a Silverlight application. We can either use the `WebClient` class or the `HttpWebRequest` class. In both cases, the call will be asynchronously executed so that the WP7 application remains responsive in the meantime.

One limitation in regular Silverlight applications is that the service either needs to be in the same domain or a client access policy file needs to be in the root of the domain. This, however, is not required for WP7 applications. They can call any service, whether or not a policy file is in place.

This results in the fact that WP7 applications can communicate with the REST API from for example Twitter, Facebook, and Google without problems. For WP7 applications, we don't need to include an intermediate service layer, which is required for regular, in-browser Silverlight applications.

See also

We covered working with REST services extensively in *Chapter 10, Talking to REST and WCF Data Services*, as well as how to parse the XML using LINQ to XML.

Accessing REST services from Windows Phone 7 using JSON

In the previous recipe, we looked at how we could connect to a REST service that returned XML. The REST service itself was a WCF service that was configured to act as a REST service. XML is not the only format often used in REST scenarios; often JSON is used. In this recipe, we'll look at how we can connect and communicate from a WP7 application with a REST service that returns JSON.

Getting ready

If you have been working through the recipes of this chapter, you can keep using your own code. Alternatively, you can use the starting solution located in the `Chapter13/BluRayCollector_JSON_Starter` folder. The completed solution is located in the `Chapter13/BluRayCollector_JSON_Completed` folder. This recipe, just like almost all others in this chapter, uses the `BluRayCollector` database.

How to do it...

For this recipe, we need to make changes both on the server side and of course on the client side. Within the service, we need to make changes so that the service acts as a REST service and configure it so it returns a JSON response. On the client, we need to make sure that we can connect with the service and read out the JSON-formatted response. Take a look at the following steps to get this scenario working:

1. First things first! Let's change the service so that it will send a response in the JSON format. In the previous recipe, we looked at what we needed to do to make the WCF service act as a REST service that communicates using XML. Not much needs to change to use JSON. In the `IBluRayService` interface, we need to specify that we want to use JSON as follows:

```
[OperationContract]
[WebGet(UriTemplate = "getallbluraysforuser/{userid}",
BodyStyle = WebMessageBodyStyle.Bare,
ResponseFormat = WebMessageFormat.Json)]
List<DTO.BluRay> GetAllBluraysForUser(string userid);
```

2. The configuration code is the same as for the REST service that uses XML. Again we use the `webHttpBinding`, as shown in the following code. This code needs to be in the `web.config` of the services project.

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="webBehavior">
        <webHttp/>
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <services>
    <service
      name="BluRayCollector.Services.RESTBluRayService">
      <endpoint address=""
                binding="webHttpBinding"
                behaviorConfiguration="webBehavior"
                contract=
                "BluRayCollector.Services.IRESTBluRayService"/>
    </service>
  </services>
</system.serviceModel>
```

3. For the server side, we are finished. It's possible to test that the service does indeed return a JSON response. To do so, build the project, right-click on the `*.svc` file in the **Solution Explorer**, and select **View in browser**. By appending the URL with `/getallbluraysforuser/1`, we get a response in the form of a JSON-formatted text file.
4. Let's now focus our attention on the client side. We need to use an asynchronous pattern in order not to lock the UI while making the request. In the `BluRayOverview.xaml` in the `OnNavigatedTo()` method, we'll make the request to the JSON-enabled REST service. The following code creates the async call:

```
private string baseUrl = "http://localhost:2956/
RESTBluRayService.svc/";
private string allBluraysForUser = "getallbluraysforuser/{0}";

protected override void OnNavigatedTo
  (System.Windows.Navigation.NavigationEventArgs e)
{
  base.OnNavigatedTo(e);

  if (!IsolatedStorageSettings.ApplicationSettings
    .Contains("AllBlurays"))
  {
```

```

var webRequest = (HttpWebRequest)WebRequest
    .Create(baseUrl +
        string.Format(allBluRaysForUser, "1"));
webRequest.BeginGetResponse(
    new AsyncCallback(getAllBluRaysForUserIdCallback),
    webRequest);
}
else
{
    AllBluRayListBox.ItemsSource =
        IsolatedStorageSettings.ApplicationSettings
        ["AllBluRays"] as ObservableCollection<BluRay>;
}
}

```

As in previous recipes, we are first performing a check to see if the list of Blu-rays has already been loaded. If not, things get interesting: we can make the call to the JSON service. Using the `HttpWebRequest` class (which we know from Silverlight), we make an async call, passing in the request to have access to it in the callback, `getAllBluRaysForUserIdCallback`.

5. We now need to write the callback method, which will be invoked when the call is ready. The following snippet contains the code for the callback. First, we access the response object and we read it out using a `StreamReader` instance.

```

void getAllBluRaysForUserIdCallback
    (IAsyncResult asynchronousResult)
{
    WebResponse response =
        ((HttpWebRequest)asynchronousResult.AsyncState)
        .EndGetResponse(asynchronousResult);

    using (StreamReader streamReader =
        new StreamReader(response.GetResponseStream()))
    {
        string responseString = streamReader.ReadToEnd();
        streamReader.Close();
    }
}

```

6. Having the response as a JSON string isn't very useful. Instead, we need a `List<BluRay>` object, so the next thing we need to do is parse the JSON string. In WP7, there's no LINQ To JSON support, but the `DataContractJsonSerializer` is supported. This class lives in the `System.Runtime.Serialization.Json` namespace, so add a reference to the `System.Runtime.Serialization` assembly.

7. We're now ready to parse the response. The following code converts the string into the List<BluRay>:

```
using (MemoryStream memoryStream =
    new MemoryStream(
        Encoding.Unicode.GetBytes(responseString)))
{
    DataContractJsonSerializer jsonSerializer =
    new DataContractJsonSerializer(typeof(List<BluRay>));
    AllBluRays = ((List<BluRay>) jsonSerializer.ReadObject
    (memoryStream)).ToObservableCollection<BluRay>();
    Dispatcher.BeginInvoke(() =>
    AllBluRayListBox.ItemsSource = AllBluRays);
}
```

The fact that we need to use the `Dispatcher.BeginInvoke()` here is simple: the code was executing on a thread different than the UI thread.

8. The `BluRay` class needs to be updated. Note the use of the `DataMember` attribute. It can be used to perform a translation from the JSON property to the CLR property.

```
[DataContract]
public class BluRay
{
    [DataMember(Name = "BluRayId")]
    public int BluRayId { get; set; }
    [DataMember(Name = "MovieName")]
    public string MovieName { get; set; }
    [DataMember(Name = "MovieLength")]
    public int MovieLength { get; set; }
    [DataMember(Name = "MovieImageUrl")]
    public string MovieImageUrl { get; set; }
    [DataMember(Name = "IsFavorite")]
    public bool IsFavorite { get; set; }
    [DataMember(Name = "IsBadMovie")]
    public bool IsBadMovie { get; set; }
    [DataMember(Name = "MovieDescription")]
    public string MovieDescription { get; set; }
    [DataMember(Name = "MovieType")]
    public string MovieType { get; set; }
}
```

With this code in place, the application works in the same way as it did before, now using JSON to communicate with the service layer.

How it works...

Communicating with a REST service that returns JSON is similar to communicating with a service that returns XML. The biggest difference is of course in the way the response must be parsed.

In WP7, to parse JSON-formatted text, we need to use the `DataContractJsonSerializer`. This class is included in the `System.Runtime.Serialization.Json` namespace. In Silverlight 5, we have the choice between this class and all classes living in the `System.Json` namespace. In WP7, however, the latter isn't included so we can't use it.

To get the JSON response, we used here the `HttpWebRequest` class. This way, we have full control over the response being sent. In the callback method, which is invoked when the service call is complete, we get access to the JSON response. This response is read out using a `StreamReader`. At that point, we just have a string; we need more than that!

This is where the `DataContractJsonSerializer` comes in. In its constructor, we pass the type we are targeting to create, in this case, `List<BluRay>`. Using the `ReadObject` method, passing in a `MemoryStream` containing the response string, we convert the string into the required `List<Bluray>`. This `BluRay` class is attributed using the `DataContract` attribute. More importantly, its members are attributed with the `DataMember` attribute. This attribute accepts a `Name`. The value of the `Name` parameter is the name of the property in the JSON-string. So, if the `MovieName` was called `pictureName` in the JSON-string, we could perform the conversion as follows:

```
[DataMember(Name = "pictureName")]
public string MovieName { get; set; }
```

In our sample, the names are the same in both cases, so no conversion was needed.

Why JSON?

One might ask: why would I use JSON? Remember that we discussed in the *Getting data on your Windows phone 7 using WCF* recipe that data on a mobile device is still expensive today. Well, when comparing all service types, a REST service that uses JSON is the cheapest one when it comes to exchanged data. JSON is very compact and adds almost no overhead to the response. Compared to other technologies, it's sometimes ten times more compact. This of course has a huge effect on the amount of data being transferred and the cost associated with it, not to mention the speed of downloading. The following code shows a sample JSON-string and the same data in an XML response. It's easy to see that JSON is more compact!

The JSON response:

```
{  
    "BluRayId":1,  
    "BluRayMovieType":null,  
    "ImageBytes":null,  
    "IsBadMovie":false,  
    "IsFavorite":true,  
    "MovieDescription":"blablabla",  
    "MovieImageUrl":"gillcleeren_jurassicpark.jpg",  
    "MovieLength":135,  
    "MovieName":"Jurassic Park 3",  
    "MovieType":null,  
    "MovieTypeId":0,  
    "PurchaseLocationLatitude":"7.01270642551253",  
    "PurchaseLocationLongitude":"-13.2422042979124",  
    "UserId":1  
}
```

The XML response:

```
<BluRay>  
    <BluRayId>1</BluRayId>  
    <BluRayMovieType i:nil="true"  
        xmlns:a="http://schemas.datacontract.org/2004/07/BluRayCollector.  
        Services.DTO" />  
    <ImageBytes i:nil="true" />  
    <IsBadMovie>false</IsBadMovie>  
    <IsFavorite>true</IsFavorite>  
    <MovieDescription>blablabla</MovieDescription>  
    <MovieImageUrl>gillcleeren_jurassicpark.jpg</MovieImageUrl>  
    <MovieLength>135</MovieLength>  
    <MovieName>Jurassic Park 3</MovieName>  
    <MovieType i:nil="true" />  
    <MovieTypeId>0</MovieTypeId>  
    <PurchaseLocationLatitude>7.01270642551253  
    </PurchaseLocationLatitude>  
    <PurchaseLocationLongitude>-13.2422042979124  
    </PurchaseLocationLongitude>  
    <UserId>1</UserId>  
</BluRay>
```

See also

In the previous recipe, *Accessing REST services from Windows Phone 7 using XML*, we looked at communicating using XML. Besides that, we looked at JSON from Silverlight as well. Take a look at the *Communicating with a REST service using JSON* recipe in *Chapter 10, Talking to REST and WCF Data Services*, for more information on this topic.

Working with push notifications using the cloud

In the previous recipes, we saw that the model of the WP7 OS is such that only one application can be running at any given moment (apart from a few built-in applications). The big advantage of this model is the performance benefit. No longer can an application that's running in the background leak memory and in the end, lock up the entire device. With the single running application model, all these are *memories of the past*. Of course, it's not all good. On some occasions, we may want to receive updates from an application, say a stock application. To receive these updates, the application must be running. On the other hand, we want to use the phone for other applications in the meantime as well. This seems, at first hand, an impossible thing to do.

There is a solution, however, in the form of **push notifications**. As the word says, a push notification is a message being pushed to the device, wherever it may be, over a network connection from the **cloud**. The push service is a service hosted by Microsoft Windows Azure, Microsoft's cloud offering. In this recipe, we'll take a look at how we can work with these notifications and what other advantages they offer.

Getting ready

This recipe builds on an already existing WCF service that returns a list of Blu-rays, like we had at the end of the *Getting data on your Windows phone 7: using WCF* recipe. You can use the code from that recipe or, alternatively, use the starting solution located in the in Chapter13/ BluRayCollector_Push_Notifications_Starter. The completed solution is located in the Chapter13/ BluRayCollector_Push_Notifications_Completed folder. This recipe, just like almost all others in this chapter, uses the BluRayCollector database.

How to do it...

This recipe will guide you through the entire process of getting up and running with WP7 push notifications. We'll need to create server-side code and of course make changes in the WP7 application. To make things easier, in the following steps, we'll look at setting up toast notifications. A toast notification appears on top of any other running application at the top of the screen, as shown in the following screenshot:



Complete the following steps to understand notifications in a WP7 application:

1. We'll kick off with the "service" part. Why the quotes, you may ask. The notifications process involves several parties:
 - ❑ The WP7 application.
 - ❑ A service (created by us) that wants to send updates to the device(s) that are registered to receive messages. This can be any type of application though; in our case, this will be a webpage.
 - ❑ A service from Microsoft in the cloud. Our service/application will ask this service to send specific content to the devices.

The service, as mentioned, in our case will be a web page that we can use to trigger the sending of messages. In real-life scenarios, this is probably a web service as well, which can be called by any type of application. The web page we are using here is to be hosted in a new project. Add a new **ASP.NET Empty Web Application** to the solution and name it `BluRayCollector.PushNotifications.Web`. Also, add a blank page to this site called `Default.aspx`.

2. The UI for this ASP.NET WebForms page is very trivial. Take a look in the code samples of the book for the full markup code for this. The following screenshot shows what the page looks like:



As can be seen, the page needs a URL and some text for the **Title** and **Content** fields, in case of the toast notification type.

3. The data that will be sent to the device is XML with a **specific format**. The URL is a unique URL, linked to a specific application on a specific device. When we click on the **Send single push notification** button, the following code will be executed:

```
private void SendToastMessage()
{
    string toastXml =
        "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
        "<wp:Notification xmlns:wp=\"WPNotification\">" +
        "<wp:Toast>" +
        "    <wp:Text1>{0}</wp:Text1>" +
        "    <wp:Text2>{1}</wp:Text2>" +
        "</wp:Toast>" +
        "</wp:Notification>";

    string toastMessage =
        string.Format(toastXml, txtTitle.Text, txtContent.Text);

    byte[] bytes = new UTF8Encoding().GetBytes(toastMessage);

    SendMessage(
        new Uri(txtUrl.Text), bytes, MessageType.Toast);
}
```

4. First, the XML is shown. This XML can't be changed, it should be exactly this piece of text. We then paste in the values added by the user. Combined, this string can't exceed a maximum value of 1024 bytes. After that, we call the `SendMessage()` method, passing in the URI to which we want to send the message, a byte array containing the message, and an enumeration value.
5. Next, add the enumeration `MessageType` as shown in the following code:

```
public enum MessageType
{
    Tile,
    Toast
}
```

Up next is the `SendMessage()` method we just used. This method starts by creating an `HttpWebRequest` to the Uri.

```
private void SendMessage
    (Uri uri, byte[] bytes, MessageType messageType)
{
    HttpWebRequest notificationRequest =
        (HttpWebRequest)WebRequest.Create(uri);
}
```

6. This request needs to be configured correctly, as shown in the following code. We specify that it uses a POST method, uses the text/XML application type, and then we add the correct headers. Finally, we specify the content length.

```
notificationRequest.Method = WebRequestMethods.Http.Post;
notificationRequest.Headers = new WebHeaderCollection();
notificationRequest.ContentType = "text/xml";

notificationRequest.Headers.Add("X-MessageID", Guid.NewGuid()
    .ToString());

switch (messageType)
{
    case MessageType.Tile:
        notificationRequest.Headers
            .Add("X-WindowsPhone-Target", "token");
        notificationRequest.Headers
            .Add("X-NotificationClass", "1");
        break;
    case MessageType.Toast:
        notificationRequest.Headers
            .Add("X-WindowsPhone-Target", "toast");
        notificationRequest.Headers
            .Add("X-NotificationClass", "2");
        break;
}
```

```
//case raw not implemented here  
}  
  
notificationRequest.ContentLength = bytes.Length;
```

7. With the request configured, we can write the XML message to the request stream. The following code shows that we are writing the bytes parameter, passed in to the SendMessage () method earlier, to the request stream.

```
using (Stream requestStream =  
    notificationRequest.GetRequestStream())  
{  
    requestStream.Write(bytes, 0, bytes.Length);  
}
```

8. In the next step, we fire off the request and read out the response. This is done in the following code:

```
try  
{  
    HttpWebResponse response =  
        (HttpWebResponse)notificationRequest.GetResponse();  
  
    var notificationStatus =  
        response.Headers["X-NotificationStatus"];  
    var subscriptionStatus =  
        response.Headers["X-SubscriptionStatus"];  
    var deviceConnectionStatus =  
        response.Headers["X-DeviceConnectionStatus"];  
  
    lblStatus.Text =  
        string.Format("NotificationStatus: {0}<br/>  
        SubscriptionStatus: {1}<br/>DeviceConnectionStatus: {2}",  
        notificationStatus, subscriptionStatus,  
        deviceConnectionStatus);  
}  
catch (Exception ex)  
{  
    lblStatus.Text = "Sending the message failed: "  
        + ex.Message;  
}
```

We can see if everything went well by looking at the response headers. In this case, we are showing the result in the UI; in a real-life scenario, we could, based on these headers, take a specific measure such as trying to send again or logging the error.

9. The web page is ready. Let's turn our attention to the WP7 application. If it doesn't exist yet, add a new page called `Settings.xaml`.
10. From the `BluRayOverview.xaml`, we must be able to navigate to this page. Using the **SettingsButton** in the application bar on this page, we can navigate to the `Settings.xaml` using the following code:

```
private void SettingsButton_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(
        new Uri("/Settings.xaml", UriKind.Relative));
}
```

11. The `Settings.xaml` page has just a single button called **CreateChannelButton**, so for the XAML markup code, take a look at the code samples of the book.
12. In the `click` event handler of this button, we call the `CreateNotificationChannel()` method as shown in the following code:

```
private void CreateChannelButton_Click
    (object sender, RoutedEventArgs e)
{
    CreateNotificationChannel();
}
```

13. In this `CreateNotificationChannel()` method, we work with the `HttpNotificationChannel`, which represents a connection between the Microsoft push service and the WP7 application. It's possible that the channel already exists; the `Find()` method helps us to look for this channel. If it exists, we need to check if the `ChannelUri` method isn't null. If it is, we need to unbind the toast subscription from the `HttpNotificationChannel` using the `UnbindToShellToast()` method and call the `CreateNotificationChannel()` method again. If the `ChannelUri` isn't null, we set the `ChannelUri` property and call two methods we'll add in the next steps. If the channel doesn't exist yet, we create it and open it. We also call the two above mentioned methods. The code for this rather complex method is as follows:

```
private void CreateNotificationChannel()
{
    HttpNotificationChannel httpChannel = null;
    string channelName = "BluRayCollectorChannel";

    try
    {
        httpChannel = HttpNotificationChannel.Find(channelName);
        if (httpChannel != null)
        {
```

```
        if (httpChannel.ChannelUri == null)
        {
            httpChannel.UnbindToShellToast();
            httpChannel.Close();
            CreateNotificationChannel();
            return;
        }
        else
        {
            ChannelUri = httpChannel.ChannelUri;
            RegisterForChannelEvents(httpChannel);
            RegisterForNotifications(httpChannel);
        }
    }
    else
    {
        httpChannel =
            new HttpNotificationChannel(channelName);
        RegisterForChannelEvents(httpChannel);

        httpChannel.Open();
        RegisterForNotifications(httpChannel);
    }
}
catch (Exception ex)
{
    Debug.WriteLine("Creating the channel failed: "
        + ex.ToString());
}
```

14. The code in the previous step used the `RegisterForChannelEvents()` method, which is shown next. This method registers two event handlers on the `HttpNotificationChannel` created earlier.

```
private void RegisterForChannelEvents
    (HttpNotificationChannel httpChannel)
{
    httpChannel.ChannelUriUpdated +=
        new EventHandler<NotificationChannelUriEventArgs>
        (httpChannel_ChannelUriUpdated);
    httpChannel.ErrorOccurred +=
        new EventHandler<NotificationChannelErrorEventArgs>
        (httpChannel_ErrorOccurred);
}
```

15. The other method we used, `RegisterForNotifications()` is shown next. It calls the `BindToShellToast()` method on the channel, effectively creating the toast subscription on the channel.

```
private static void RegisterForNotifications
    (HttpNotificationChannel httpChannel)
{
    try
    {
        httpChannel.BindToShellToast();
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Binding to the shell failed: "
            + ex.ToString());
    }
}
```

16. Finally, the last step shows some easy methods and properties referenced earlier. The `OnChannelUriChanged()` method is invoked when the URI of the channel changes. This happens also when the channel is created and opened and is assigned a URI. At this point, the application is registered to receive notifications.

```
private Uri _channelUri;

void httpChannel_ErrorOccurred
    (object sender, NotificationChannelErrorEventArgs e)
{
    Debug.WriteLine(e.Message);
}

void httpChannel_ChannelUriUpdated
    (object sender, NotificationChannelUriEventArgs e)
{
    ChannelUri = e.ChannelUri;
}

public Uri ChannelUri
{
    get
    {
        return _channelUri;
    }
    set
    {
        _channelUri = value;
```

```
        OnChannelUriChanged(value);
    }
}

private void OnChannelUriChanged(Uri value)
{
    Debug.WriteLine(value.ToString());
    Dispatcher.BeginInvoke(() =>
    {
        ResultTextBlock.Text = "Channel created";
    });
}
```

We have successfully created both sides of the application, but we aren't done yet—we still need to run the application to test it! Follow these steps to test that the application is working correctly.

1. Make sure you have an Internet connection! It's quite logical that you won't be able to access the cloud service without it.
2. Set the WP7 application as the startup project.
3. Place a breakpoint in the `OnChannelUriChanged (Settings.xaml.cs)` method on the following line:

```
ResultTextBlock.Text = "Channel created";
```

4. Run the application, navigate to the Settings page and press the button. After a few seconds, assuming you have implemented everything correctly, the breakpoint will be hit.
5. In debug mode, take a look at the value of the `ChannelUri` property. This is of the form:

```
http://db3.notify.live.net/throttledthirdparty/01.00/
AEuIbRo4qZFS6tX6kQMNqrFAGAAAAADAQAAAQUZm52OjIzOEQ2NDJDRkI5MEVFMEQ
```

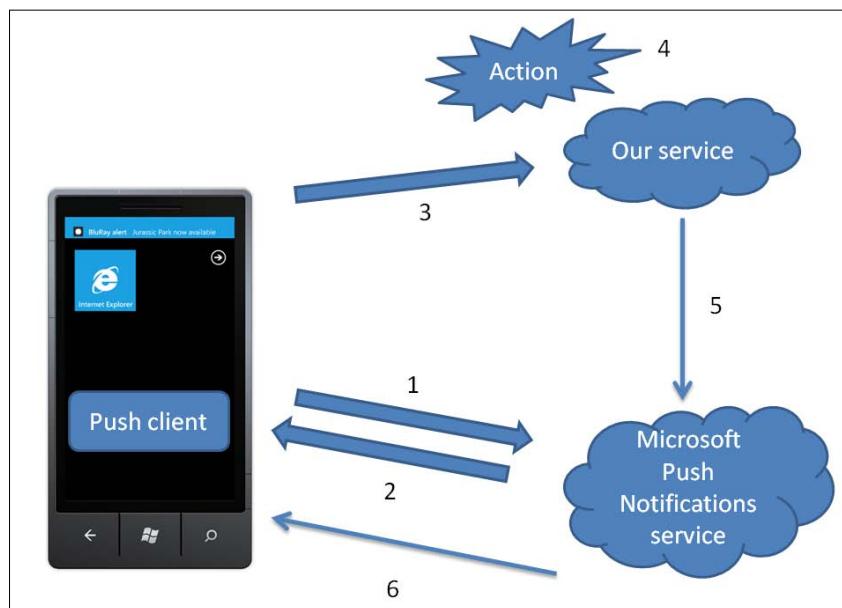
Copy this link from Visual Studio.

6. Keep the emulator open and press the Windows button so the WP7 start screen is displayed. If you are hitting the breakpoint, you can remove it.
7. Within the **Solution Explorer**, right-click on the `BluRayCollector.PushNotifications.Web` project and select **Debug | Start new instance**.
8. In the default page, paste the link in the **URL** field. Enter a value for the **Title** and **Content** fields and press the **Send Single Push Notification** button.
9. After a few seconds, you'll hear a sound from the WP7 emulator as well as see a message, the toast message, being displayed at the top of the screen.

How it works...

Windows Phone 7 offers a notification system in the form of push notifications that are sent over the air to the device. The current model of the WP7 OS is such that at each point in time, only one application can be running. However, there are real-life scenarios imaginable in which we'd want to receive updates from a certain installed application, but we can't (for practical reasons) keep the application open at all times. Imagine that we want to be notified about a new Blu-ray being released while we are in the middle of playing a game. Because of the one-running-application model, our BluRayCollector application can't be running while we are playing a game and in normal circumstances, wouldn't be able to receive this update.

Notifications, more specifically toast notifications, can be a life-saver here. Even while the application isn't running, we may receive updates from a service. That service is linked to an application on the device: when clicking on the message, which is shown at the top of the screen, we are taken to the related application. But wait, how does the device know which is the related application? Let's take a look at the architecture for push notifications, shown in the following figure:



The numbers in the image are explained in the following bullets:

1. We install an application that uses notifications. The application registers a so-called **channel** with a Microsoft cloud-based service, called the **Microsoft Push Notifications Service**.
2. From that service, we get back a URI for the channel. This URI is a **unique identifier** for a particular application on a particular device.

3. The WP7 application then sends this URI to a service that we create. In our sample, we have done this manually by copying the URL into the webpage. In a real-life application, the web page would be a service that is ready to accept incoming calls from new WP7 registrations. It is the responsibility of our service to keep a list of registered clients.
4. At some point, our service needs to notify all clients about an event/action that occurs. This event/action may occur in another system, which then triggers the service.
5. To notify all clients, it sends a notification to the Microsoft Push Notifications service in a specific XML format that is predefined by Microsoft, including the URIs to notify.
6. Microsoft's cloud-based service sends the data to the WP7 device.

Apart from allowing us to create a better experience in that we have the ability to notify a user about an event even when the application isn't running, push notifications also bring down battery usage and data usage: there's no need to make continuous calls from the device to check for a specific server message.

There's more...

The message type discussed above is toast notifications. That is actually only one of the three possible types of notifications. The two others are tile notifications and raw messages.

Tile notifications refer to updates that are reflected in the tile of the application if it's pinned to the WP7 start screen. If there's no tile for the particular application, the message is suppressed and no further action happens.

To be able to receive tile notifications, the application has to register that it wants to do so. Instead of using the `BindToShellToast()` method, we use the similar `BindToShellTile()` method on the `HttpNotificationChannel`, as follows:

```
httpChannel.BindToShellTile();
```

On the service-side, sending tile notifications is entirely similar to toast notifications, apart from a few things. When creating the message, two headers are different as shown in the following code:

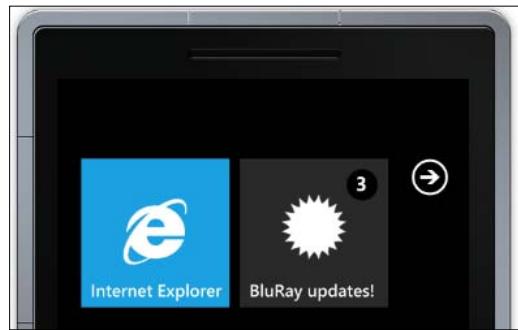
```
switch (messageType)
{
    case MessageType.Tile:
        notificationRequest.Headers
            .Add("X-WindowsPhone-Target", "token");
        notificationRequest.Headers
            .Add("X-NotificationClass", "1");
        break;
```

```
case MessageType.Toast:  
    notificationRequest.Headers  
        .Add("X-WindowsPhone-Target", "toast");  
    notificationRequest.Headers  
        .Add("X-NotificationClass", "2");  
    break;  
//case raw not implemented here  
}
```

Also, the XML is somewhat different. We can notice a wp:Count element here as well as a wp:Title, whereas in the toast message, we had wp:Text1 and wp:Text2. The XML is as shown in the following code:

```
string tileXml =  
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +  
    "<wp:Notification xmlns:wp=\"WPNotification\">" +  
    "<wp:Tile>" +  
    "    "<wp:Count>{0}</wp:Count>" +  
    "    "<wp:Title>{1}</wp:Title>" +  
    "</wp:Tile>" +  
    "</wp:Notification>";
```

The value of the wp:Count field will be shown on a specific place in the tile, while the wp:Title replaces the name of the application in the tile. The following screenshot shows how the tile looks after receiving this message:



The third type of notification is raw messages. While the two previous types are meant to be received when the application isn't currently running in the foreground, raw messages are the opposite—they can be received by an application only when that application is currently running. This enables a sort-of duplex messaging: there's no call from the client required to receive a message from the notification service.



Toast notifications can also, just like raw messages, be received while the application is running.



Storing data in a local SQL CE database

Isolated Storage is a good way of storing data locally on the device. With the release of Windows Phone Mango (7.5), an option was added to support a **client-side database**. This database is a **SQL CE** (Compact Edition) database. SQL CE databases are file-based and in the case of WP7, the database file is stored as a physical file inside the Isolated Storage. As developers, we can talk to the database using LINQ to SQL.

In this recipe, we'll introduce the local database to store the downloaded list of Blu-rays.

Getting ready

This recipe uses a starter solution, which can be found in the `Chapter13/BluRayCollector_Database_Starter`. The finished solution can be found in the `Chapter13/BluRayCollector_Push_Notifications_Completed` folder.

How to do it...

Working with the local database requires a few specific steps related to setting up the database. Once the setup is done, we can change our existing code to interact with the database instead of the Isolated Storage. This is done by writing LINQ to SQL queries. Follow along with these steps to get this to work:

1. Open the starter solution as outlined in the *Getting ready* section for this recipe.
2. Microsoft recommends using the **code-first approach**. This means that we describe the model in code instead of starting with a database model. With code-first, we don't really have to worry about database queries—we just write our code using the classes from our model.

To apply this approach, we need to write a context class first. This class can be understood as a class representing the database from code. To add the context, create a new class named `BluRayDataContext`. This class should inherit from `DataContext`. The initial code for this class is shown in the following code snippet:

```
using System.Data.Linq;
public class BluRayDataContext : DataContext
{
    ...
}
```

The `DataContext` class lives in the `System.Data.Linq` namespace. This namespace requires that we add a reference to the `System.Data.Linq` assembly in the WP7 project.

3. Next, we need to create a connection string, containing the location of the database. Add the following line to the `BluRayDataContext` class:

```
public static string DBConnectionString =
    "Data Source=isostore:/BluRayCollector.sdf";
```

Note that the file containing the database is an `*.sdf` file. This is the typical file extension for SQL CE.

4. Our custom context needs to pass the connection string to the base class. This can be done using the following code where we do exactly that from the constructor:

```
public BluRayDataContext(string connectionString) :
    base(connectionString) { }
```

5. Each entity in our model is represented by a `Table<T>`. In our case, we have just a `BluRay` table. This is represented by the following code:

```
public Table<BluRay> BluRays;
```

6. With the context ready (and thus the model outlined), we can move on to describing the entities. In our case, we need to describe from code just the `BluRay` class. To indicate that this class is used as a mapping between our model and the database, we need to attribute it with the `Table` attribute. To perform this step, add a new class named `BluRay` and add the following code:

```
using System.Data.Linq.Mapping;
using System.ComponentModel;
[Table]
public class BluRay :
    INotifyPropertyChanged, INotifyPropertyChanging
{
    ...
}
```

7. Notice that this class implements two interfaces. `INotifyPropertyChanging` is used to notify clients that the value is about to change while `INotifyPropertyChanged` is used to notify clients after the value has changed. For this code to work, make sure to add the two `using` statements indicated here as well. Also, a reference to the `mscorlib.Extensions` assembly needs to be added to the project.

8. In this class, we now need to describe all its properties as well as how these properties should be mapped inside the database. Remember that a database will be created based on the model we are describing here, so all information regarding keys, database types, and relations is to be added in this class. Add the following code to the BluRay class. To save space, not all properties have been written out; the code is similar though.

```
[Column(IsPrimaryKey = true, IsDbGenerated = true,
DbType = "INT NOT NULL Identity", CanBeNull = false,
AutoSync = AutoSync.OnInsert)]
public int BluRayId
{
    get
    {
        return _bluRayId;
    }
    set
    {
        if (_bluRayId != value)
        {
            NotifyPropertyChanging("BluRayId");
            _bluRayId = value;
            NotifyPropertyChanged("BluRayId");
        }
    }
}

private string _movieName;

[Column]
public string MovieName
{
    get
    {
        return _movieName;
    }
    set
    {
        if (_movieName != value)
        {
            NotifyPropertyChanging("MovieName");
            _movieName = value;
            NotifyPropertyChanged("MovieName");
        }
    }
}
```

```
}

private int _movieLength;

[Column]
public int MovieLength { ... }

private string _movieImageUrl;

[Column]
public string MovieImageUrl { ... }

private bool _isFavorite;

[Column]
public bool IsFavorite { ... }

private bool _isBadMovie;

[Column]
public bool IsBadMovie { ... }

private string _movieDescription;

[Column]
public string MovieDescription { ... }

private string _movieType;

[Column]
public string MovieType { ... }

public event PropertyChangedEventHandler PropertyChanged;

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

```
public event PropertyChangingEventHandler PropertyChanging;

private void NotifyPropertyChanging(string propertyName)
{
    if (PropertyChanging != null)
    {
        PropertyChanging(this,
            new PropertyChangingEventArgs(propertyName));
    }
}
```

This class also contains the implementation for the two implemented interfaces.

9. We're now ready to start using the database. In the `App.xaml.cs` file, we're going to add code that creates the database file for our application to use if it doesn't exist yet. Based on the model, the `*.sdf` file will be generated. In the constructor of the `App` class, add the following code:

```
using (BluRayDataContext db =
    new BluRayDataContext(BluRayDataContext.DBConnectionString))
{
    if (db.DatabaseExists() == false)
    {
        db.CreateDatabase();
    }
}
```

10. We now know for sure that the database file was created. We can now safely work with the context as our intermediary to talk with the database itself. Change the code in the `BluRayOverview.xaml.cs` file as follows:

```
private BluRayDataContext _bluRayDataContext;

public BluRayOverview()
{
    InitializeComponent();

    _bluRayDataContext =
        new BluRayDataContext
            (BluRayDataContext.DBConnectionString);

    this.DataContext = this;
}
```

11. In the OnNavigatedTo method, we only want to get the list of Blu-rays if the database table is still empty. We can write a simple LINQ query here and then use the Count() method. Add the following code to do so:

```
var localBluRays = from BluRay b in _bluRayDataContext.BluRays
                    select b;
if (localBluRays.Count() > 0)
{
    //DB is locally filled already
    AllBluRayListBox.ItemsSource = localBluRays;
}
else
{
    proxy.GetAllBluRaysForUserCompleted += 
        new EventHandler<
            BluRayService.GetAllBluRaysForUserCompletedEventArgs>
            (proxy_GetAllBluRaysForUserCompleted);
    proxy.GetAllBluRaysForUserAsync(1);
}
```

12. Initially, the list will need to be downloaded, so the GetAllBluRaysForUserAsync will be triggered. In this callback, we will loop over the result and insert new BluRay instances in the Table<BluRay> property of the context. Finally, we call the SubmitChanges() method which will commit the added instances. This is done as follows:

```
void proxy_GetAllBluRaysForUserCompleted(object sender, BluRayService.GetAllBluRaysForUserCompletedEventArgs e)
{
    AllBluRayListBox.ItemsSource = e.Result;
    foreach (var serviceBluRay in e.Result)
    {
        var localBluRay = new BluRay()
        {
            BluRayId = serviceBluRay.BluRayId,
            IsBadMovie = serviceBluRay.IsBadMovie,
            IsFavorite = serviceBluRay.IsFavorite,
            MovieDescription = serviceBluRay.MovieDescription,
            MovieImageUrl = serviceBluRay.MovieImageUrl,
            MovieLength = serviceBluRay.MovieLength,
            MovieName = serviceBluRay.MovieName,
            MovieType = serviceBluRay.MovieType
        };
        _bluRayDataContext.BluRays.InsertOnSubmit(localBluRay);
    }
    _bluRayDataContext.SubmitChanges();
}
```

With this code added, the sample now uses the SQL CE database instead of isolated storage to store the downloaded information.

How it works...

Having a local database at our disposal within a WP7 application opens up a lot of new ways of working with data locally on the device. Isolated Storage allows us to work with any type of data (such as raw data that we can store as XML but also binary data such as images). Custom libraries, such as the Sterling database (which we used in *Chapter 5, Working with Local Data*, for Silverlight but available for WP7 as well) allow us to interact with data in an easier way. With the addition of a SQL CE database, we can work with data in a relational way. This client-side database is a perfect addition for applications that work together with services as well: it can be used to store downloaded data to work in offline mode as well.

The database context

To work with the local database, Microsoft recommends using a code-first approach. As mentioned before, this basically frees us from thinking in relational ways about our data. We as developers can write a model in code and don't have to worry about how this will be stored in the database. All this is handled for us.

The context class is the start of this approach. It's a class that describes the model. Each entity inside this model is a table in the database that will be created for us. All these entities are described as properties of the context. The type of these properties should be `Table<T>`.

The context also contains the link to the physical database file in the form of a connection string. The connection string we used here is as follows:

```
public static string DBConnectionString =
    "Data Source=isostore:/BluRayCollector.sdf";
```

The file type is `*.sdf`, which is the default file type for a SQL CE database. Such a database is file-based.

The model classes

The entities themselves used in the context are classes as well. The class is attributed with the `Table` attribute. This is an indication that this class is used for mappings between the model in code and the actual database. This is reflected in attributes on the properties of the class. Properties will have a `Column` attribute applied to them that means *this property should become a field in the database table*. To this attribute, we can optionally pass more information such as specifying that the property is to be the primary key, database type, and so on.

Interaction with the database

Once the model is declared, we can have the database file created for us. In the sample, we placed code in the constructor of the app that uses the `CreateDatabase()` method to generate the database file based on the context. Of course, this only has to be done if the database hasn't been generated yet. If needed, it's possible to package an initial database with your application and copy it to isolated storage from code. This can be useful if your application has a lot of initial settings that need to be packaged in the database that comes with your application.

Once the database is created, we can write LINQ to SQL queries. In the code from this recipe, we wrote a regular LINQ query that retrieves all Blu-Rays. Inserting (as well as updating and deleting) instances is done using the context as well. In the sample code, we used the `InsertOnSubmit()` method to add instances to the `Table<BluRay>`. The `SubmitChanges()` method commits the changes to the database.

Save often

It's a good idea to commit changes using the `SubmitChanges()` method often. It might be tempting to do all editing of the data *in-memory* without committing and then perform the commit when the application exits. The latter isn't such a good idea. In WP7, applications have a limited amount of time when exiting to perform tasks before being killed off. If we were have to do a lot of commits to perform in that timeframe, we might just hit that time limit. If that happens, there's no guarantee that the data is still intact inside the database!

See also

In *Chapter 5, Working with Local Data*, in the *Using the Sterling Isolated Storage* database recipe, we used the Sterling database for Silverlight. This is also applicable to WP7 development.

Using the background transfer service

Downloading files on a mobile device can be time-consuming. A slow Wi-Fi connection or bad 3G coverage are all still too common. While downloading a file, the application doing the download needs to remain the active application because of the way the application lifecycle on WP7 works.

With the release of Windows Phone 7.5, a service named the `BackgroundTransferService` was added. This service, managed by the operating system itself, can be consumed by a WP7 application to **upload and download files in the background**, even after the application that started the transfer has been shut down.

In this recipe, we'll extend the `BluRayCollector` application once more to download and play a movie trailer.

Getting ready

For this recipe, a starter solution is available with the downloads of the book. It can be found in the Chapter13/BluRayCollector_BackgroundService_Starter folder. The finished solution for this recipe can be found in the BluRayCollector_BackgroundService_Completed folder.

How to do it...

The BluRayCollector application is extended with a new feature: allowing the user to view the most popular movie trailer. Since movie trailers tend to get quite large, the download may take some time. Instead of forcing the user to leave the BluRayCollector application open and wait for the download to finish, we are going to use the BackgroundTransferService class to download the file. In the following steps, we are using this class:

1. Open the starter solution as outlined in the *Getting ready* section.
2. In the `BluRayOverview.xaml` page, which contains the Panorama control, we start by adding a new `PanoramaItem`. On this `PanoramaItem`, we add a `Button` to initiate the download, a `TextBlock` to show the download status and another `Button` to start playing the trailer after the download is finished. The latter is not enabled initially, since nothing can be played yet. The code for this is as follows:

```
<controls:PanoramaItem Header="Popular Trailer">
    <StackPanel>
        <Button Name="DownloadTrailerButton"
            Click="DownloadTrailerButton_Click"
            Content="Download trailer"
            HorizontalAlignment="Center"
            VerticalAlignment="Top"
            Height="80">
        </Button>
        <TextBlock Name="StatusTextBlock"
            Text="Waiting..."
            HorizontalAlignment="Center">
        </TextBlock>
        <Button Name="StartPlayingButton"
            IsEnabled="False"
            Click="StartPlayingButton_Click"
            Content="Start playing"
            HorizontalAlignment="Center"
            VerticalAlignment="Top"
            Height="80">
        </Button>
    </StackPanel>
</controls:PanoramaItem>
```

3. Switch to the code-behind, BluRayOverview.xaml.cs. When clicking the **Download** button, we want to initiate the transfer. This is done by creating a `BackgroundTransferRequest` instance. To the constructor of this class, we pass two `Uri` instances. The first one is the `Uri` pointing to the file we want to download. The second one indicates where the downloaded file should be placed. Downloads done using a `BackgroundTransferRequest` will always be placed in a subdirectory of the Isolated Storage named `shared/transfers`. We can then pass the `BackgroundTransferRequest` to the `BackgroundTransferService`, which will take care of downloading the file. This is shown in the following code:

```
private void DownloadTrailerButton_Click(object sender,
    RoutedEventArgs e)
{
    using (IsolatedStorageFile isolatedStorageFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isolatedStorageFile.FileExists(downloadLocation))
        {
            isolatedStorageFile.DeleteFile(downloadLocation);
        }
    }
    backgroundTransferRequest = new BackgroundTransferRequest
        (trailerUri, downloadLocationUri);

    BackgroundTransferService.Add(backgroundTransferRequest);
}
```

Note that the preceding code checks if the file is already downloaded. If so, we delete it in this scenario. An alternative might be using the already downloaded file and thus skipping the download.

4. On the instance of the `BackgroundTransferRequest`, we register for events raised by the object. In this case, we register for the `TransferProgressChanged` and the `TransferStatusChanged` events. We can also pass preferences as to when the download should be allowed or disallowed (for example we can specify to only allow the download if a Wi-Fi is available). The following code should be inserted before adding the request to the service:

```
backgroundTransferRequest.TransferPreferences =
    TransferPreferences.AllowCellularAndBattery;
backgroundTransferRequest.TransferProgressChanged
    += new EventHandler<BackgroundTransferEventArgs>
        (backgroundTransferRequest_TransferProgressChanged);
backgroundTransferRequest.TransferStatusChanged +=
    new EventHandler<BackgroundTransferEventArgs>
        (backgroundTransferRequest_TransferStatusChanged);
```

5. The handler of the `TransferProgressChanged` event will be called whenever progress is made in the download of the file. In this case, we handle this by updating the `StatusTextBlock`. This is shown in the following code:

```
void backgroundTransferRequest_TransferProgressChanged
    (object sender, BackgroundTransferEventArgs e)
{
    string progressText = string.Empty;
    if (backgroundTransferRequest != null)
    {
        progressText = String.Format
            ("Downloaded {0} bytes of {1} bytes",
            backgroundTransferRequest.BytesReceived,
            backgroundTransferRequest.TotalBytesToReceive);
    }
    StatusTextBlock.Text = progressText;
}
```

The `BackgroundTransferEventArgs` have quite a few properties available that contain information on the download in progress. Here we are using the `BytesReceived` (reflecting how many bytes were already downloaded) and the `TotalBytesToReceive` (indicating the total size of the download at hand).

6. The `TransferStatusChanged` event will be called whenever the status of the transfer is updated. This happens quite a few times. The `TransferStatus` enumeration has statuses including `None`, `Transferring`, `Waiting`, `Completed`, and so on. In our code, we want to react to the `Completed` status, since then, the `StartPlayingButton` can be enabled. When the status is `Completed`, however, we still need to check the `StatusCode` value. The latter should be 200 or 206, which both indicate a successful transfer. If an error occurred, we display it in the `StatusTextBlock`. We also call the `RemoveRequest()` method, which will remove `BackgroundTransferRequest` from the download queue. Finally, we set the `StartPlayingButton` enabled. This can be seen in the following code:

```
void backgroundTransferRequest_TransferStatusChanged
    (object sender, BackgroundTransferEventArgs e)
{
    string statusText = string.Empty;

    if (backgroundTransferRequest.TransferStatus
        == TransferStatus.Completed)
    {
        if (backgroundTransferRequest.StatusCode == 200
            || backgroundTransferRequest.StatusCode == 206)
        {
            statusText = String.Format("Status: {0}",
                backgroundTransferRequest.TransferStatus);
        }
        RemoveRequest();
        StartPlayingButton.Enabled = true;
    }
}
```

```
        RemoveRequest (backgroundTransferRequest) ;
    }
    else
    {
        statusText = String.Format ("Status: {0}",
        backgroundTransferRequest.TransferError.Message) ;
    }
}
StartPlayingButton.IsEnabled =
(backgroundTransferRequest.TransferStatus
== TransferStatus.Completed);
StatusTextBlock.Text = statusText;
}
```

7. The code for the RemoveRequest () method is shown next. Note that we as the developer are responsible for removing requests from the service. The fact that a download has completed does not remove it automatically.

```
private void RemoveRequest
(BackgroundTransferRequest backgroundTransferRequest)
{
    BackgroundTransferService.
    Remove (backgroundTransferRequest) ;
}
```

8. When the movie trailer is downloaded, we can start playing it. This is done using the following code, which is called when the StartPlayingButton is clicked:

```
private void StartPlayingButton_Click
(object sender, RoutedEventArgs e)
{
    MediaPlayerLauncher mediaPlayerLauncher =
    new MediaPlayerLauncher()
    {
        Media = downloadLocationUri,
        Location = MediaLocationType.Data
    };
    mediaPlayerLauncher.Show() ;
}
```

If we had exited the application while it was downloading the file, the download would have continued in the background. The file is transferred to the `shared/transfers` subdirectory of the isolated storage and will at some point be available for the application to use.

How it works...

Starting with Windows Phone 7.5 (SDK 7.1), a new service was added to the Windows Phone OS: the `BackgroundTransferService`. This service can be seen as a queue for uploads and downloads, managed and owned by the OS itself. Applications can add requests to the service in the form of a `BackgroundTransferRequest`. Whereas before, a download was to be done directly from application code (thus forcing the user to keep the application open while the download was taking place), it is now done outside of the application, entirely managed by the `BackgroundTransferService`. This results in the fact that transfers can continue, even after the application has been closed. Even after a reboot of the Windows Phone OS, the transfers will be resumed.

Files that are downloaded are placed inside a subdirectory named `shared/transfers` of the Isolated Storage. When files are uploaded, they should be in that same directory.

Restrictions

Of course, transferring large files on a mobile OS can be dangerous. Without the necessary precautions, the user may be faced with huge bills because he or she transferred too much data. Also, transferring files uses a lot more power of the device.

To prevent background transfers from undermining the overall user experience, several restrictions are built in. The following is a list of some restrictions; visit [http://msdn.microsoft.com/en-us/library/hh202955\(v=VS.92\).aspx](http://msdn.microsoft.com/en-us/library/hh202955(v=VS.92).aspx) for the entire list:

- ▶ A file that is to be uploaded should not be larger than 5 MB.
- ▶ If on cellular, a file to be transferred can't be larger than 20 MB.
- ▶ An application may only have five outstanding requests in the queue. In code, we should check that we aren't passing this limit.
- ▶ All applications together on the device shouldn't have more than 500 outstanding transfers.

Appendix

Sometimes you'll have to complete tasks that apply to various recipes in this book, or you'll run into tasks that are too generic for a specific recipe (for example, creating a REST service with the help of WCF). Besides that, some topics are very interesting, but don't really fit in the recipe format, or are only of interest to some readers (for example, using NuGet). All that info and more can be found in this appendix, as a reference.

Creating a REST service from WCF

Using WCF services, it is possible to expose a REST endpoint. This way, we can easily create a WCF services project in which some services have a SOAP endpoint while others have a REST endpoint.

In .NET 4.0, there are several ways to create REST services. In this example, we'll use a manual approach based on the `WebHttpBinding`. This process is quite simple: we need to make a configuration change to the configuration code of the service and apply an attribute on the service methods.

We'll create a small sample service project and expose the service as a REST endpoint. To do so, create a new empty ASP.NET web application in Visual Studio 2010 and name it `OfficeSupplies`. Within the created web application, add a new WCF service called `TonerService`. This triggers the creation of both the service interface, `ITonerService.cs` and the service implementation, `TonerService.cs`.

To allow our service to communicate using REST, we'll first create a REST endpoint. In the `web.config`, we'll start by adding the `webHttp` behavior:

```
<system.serviceModel>
<behaviors>
    <endpointBehaviors>
        <behavior name="webBehavior">
```

```
<webHttp/>
  </behavior>
</endpointBehaviors>
</behaviors>
</system.serviceModel>
```

Next, we can add a new endpoint, which uses this behavior:

```
<system.serviceModel>

  <services>
    <service name="OfficeSupplies.TonerService">
      <endpoint
        address=""
        behaviorConfiguration="webBehavior"
        binding="webHttpBinding"
        bindingConfiguration=""
        contract="OfficeSupplies.ITonerService"/>
    </service>
  </services>
</system.serviceModel>
```

Finally, we have to apply an attribute, the `WebGetAttribute`, on the service methods we want to expose over REST. Using the `UriTemplate` on this attribute, we can use the same endpoint for several service methods. In the following sample, we can see that `IsTonerAvailable` has its `UriTemplate` set to `Toner`:

```
[OperationContract]
[WebGet(UriTemplate = "toner",
       BodyStyle = WebMessageBodyStyle.Bare,
       RequestFormat = WebMessageFormat.Xml)]
bool IsTonerAvailable();
```

This service method can thus be accessed using `http://localhost:123/TonerService.svc/Toner`.

If we want to pass in a parameter, we can do so using a query expression. If we want to check the toner for a specific color, we can pass the color using the following method:

```
[OperationContract]
[WebGet(UriTemplate = "toner/{color}",
       BodyStyle = WebMessageBodyStyle.Bare,
       RequestFormat = WebMessageFormat.Xml)]
bool IsColorAvailable(string color);
```

To invoke this service, we can perform a call to `http://localhost:123/TonerService.svc/Toner/Red`.

Installing an SQL Server database

In some samples of the book, we use an SQL Server database. To install an SQL Server database, we can either attach the SQL Server MDF file to SQL Server or execute an SQL query file, which contains the commands to create the database, its tables, and columns.

Attaching an MDF file

The simplest way to install a database is attaching the database into the SQL Server instance. To do so, follow the next steps:

1. Open SQL Server Management Studio and connect to the database engine (most of the time, this instance is called localhost).
2. Within the **Object Explorer**, right-click on the **Databases** node and select **Attach**.
3. In the **Attach Databases** dialog, click on the **Add** button and browse to the location of the MDF file that contains the database. Click **OK**; SQL Server will now attach the database to the SQL Server instance.

The database can now be consulted from within SQL Server Management Studio and it can also be used in our own applications.

Executing a query file

If attaching the database is not an option (for example, on an externally hosted database server), we can use a query file that contains all the commands needed to recreate the database and its tables. To do so, follow the next steps:

1. Open SQL Server Management Studio and connect to the database engine (mostly localhost).
2. Within Management Studio, select **File | Open File** and select the query file (*.sql) containing the SQL commands.
3. Click on **Execute**.

SQL Server Management Studio will now execute the commands, resulting in the database being created.

Working with Fiddler

Fiddler is a web debugging proxy tool. Being a proxy, it redirects all traffic going from your computer to the Internet. This way, we can inspect the contents of the packages being sent over the wire. Fiddler also works for inspecting traffic to the local web server (ASP.NET Development Server or local IIS).

While developing with Silverlight in combination with services, Fiddler is a valuable tool. Without a debugging proxy, it's not possible to see what data Silverlight is sending to the service or to see the response coming back from the service. In this book, we show the usage of Fiddler several times. In your daily development life, Fiddler can help you track bugs when building Silverlight applications that work with services.

Setting up Fiddler is easy. To install it, download a free copy at <http://www.fiddler2.com/fiddler2/>. Once installed, if you're using Internet Explorer, you can start it by selecting **Tools | Fiddler2**.

Fiddler will open its main window, showing all the traffic going in and out from your machine.

Local traffic

By default, Fiddler will not capture local traffic (traffic going to your ASP.NET Development Server or local IIS) because Internet Explorer and .NET in general don't send requests for `http://localhost` through any proxy. It is however, important to be able to do so, mainly in development stages.

There are several ways to work around this problem. The easiest solution is replacing `http://localhost` in your request with `http://<ComputerName>` (for example, `http://Dev01` where Dev01 is the name of your computer). Another solution is replacing `http://localhost` with `http://ipv4.fiddler` (for IPv4 requests) or `http://ipv6.fiddler` (for IPv6 requests). If your request contains a port number (for example, `http://localhost:123/MyService.svc`), this port number should remain in the changed URL (for example, `http://ipv4.fiddler:123/MyService.svc`).

Working with the Silverlight control toolkit

The **Silverlight control toolkit** is a collection of controls and utilities that is not part of the default installation of Silverlight. It's hosted on **CodePlex** (Microsoft's open source hosting website: <http://www.codeplex.com>) as a project owned by Microsoft itself and managed by the Silverlight product team. It has an out-of-band release cycle (meaning that releases of the toolkit are not necessary). Being hosted on CodePlex also means that you get all the source code for the controls, including many unit tests.

To download the toolkit, go to <http://www.codeplex.com/Silverlight> where you can find the latest release. After installation, the new controls are added to the Toolbox in Visual Studio.

Once installed, the controls can be used just like regular controls. On adding one of these controls to your XAML, Visual Studio will add an XML namespace mapping (`xmlns:controlsToolkit="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"`) within the XAML and create a reference to the toolkit assembly, `System.Windows.Controls.Toolkit`.

Working with WIF

Some recipes use **Windows Identity Foundation**, and require the installation of the Windows Identity Foundation training kit. This kit contains a set of hands-on labs, documents, and references that will help you to learn how to take advantage of Microsoft's latest identity and access control developer's products and services.

You can download it at this link (it's the April 2011 version you require):

<http://www.microsoft.com/download/en/details.aspx?id=14347>.

Installing the WCF RIA Services Toolkit

For some recipes (for example, the ones in which you expose a domain service via SOAP), the WCF RIA Services Toolkit is required. This toolkit is a collection of classes not included in the default WCF RIA Services install (which is included with a Silverlight 5 install).

The toolkit enables Entity Framework 4.1 support, Linq to SQL support, Windows Azure Table Storage support, SOAP/JSON endpoints, T4 Code Generation, ViewModel (MVVM) support, and ASP.NET WebForms controls. It also includes the jQuery client for RIA Services (RIA/JS).

You can download the toolkit from this link: <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26939>, or you can get it as NuGet packages: [http://nuget.org>List/Search?packageType=Packages&searchCategory>All+Categories&searchTerm=riaservices&sortOrder=package-download-count&pageSize=10](http://nuget.org/List/Search?packageType=Packages&searchCategory>All+Categories&searchTerm=riaservices&sortOrder=package-download-count&pageSize=10).

Installing and using NuGet

NuGet is a developer-focused package-management system for the .NET platform. Its intent is simplifying the process of incorporating third-party libraries in your .NET applications.

When you're using one of these libraries, it's often quite a challenge to integrate it into a project: you need to find the correct version online, unblock it, add the necessary references to your projects, and often, you'll also need to add references to other external dependencies.

NuGet takes care of this for you, right from Visual Studio.

To install it (as a Visual Studio extension), you can surf to <http://www.nuget.org/>.

Index

A

Access Control 274
ActivateEngine() method 197, 198
Activate() method 197
Active Directory Federation Services. *See ADFS*
Add DomainService wizard 466
AddNewPerson() method 213, 218
Add Person button 213, 218
Add Person button tag 232
AddPersonExecution() method 240
AddPerson method 216
Add Service Reference dialog box 311
ADFS 381
ADO.NET Data Services 411
ADO.NET Entity Framework 411
AJAX 325
AlreadyRead property 162
AlternatingRowBackground property 142
Amazon 426
Ancestor RelativeSource binding
 about 8, 119
 using 120-123
 working 123
App Hub 562
AppHub 574
application
 requisites 560
application bar 572
ApplicationExit event 201
application services 302
ApplicationSettings property 186
Application_Startup method 505, 518
App.xaml.cs 505
App.xaml file 18, 90

ASMX 302
ASMX/WCF service 573
ASMX webservice 244, 249
ASP.NET 7, 38, 325
ASP.NET Application Services 325
ASP.NET authentication
 using, in Silverlight 325-331
ASP.NET Membership API 325, 332
ASP.NET platform 454
ASP.NET Web Application Project option 17
ASP.NET (WebForms and MVC) 504
ASP.NET Web Site option 17
AsyncResult property 270
AtomPub 414
Atom Publishing Protocol. *See AtomPub*
attributes
 used, for validating data input 101-103
authentication 374
authentication cookie 512
AuthenticationService 332
authorization 374
Author property 167
AutoGenerateColumns property 142, 145,
 161
automatic synchronization
 about 60
 for collections 66
 for single object 65
Azure Storage 274

B

BackgroundTransferRequest instance 618
background transfer service
 BackgroundTransferRequest instance 618
 BackgroundTransferService class 617

RemoveRequest() method 619, 620
 TransferProgessChanged event 619
 TransferStatusChanged event 618, 619
 using 616
BackgroundTransferService class 617
Base Class Library. *See* **BCL**
Batch option 426
BCL 8
BeginAcceptTcpClient method 290, 297
BeginGetRequestStream method 268-270
BeginGetResponse method 271
BeginningEdit event 148-150
binary encoding 317
BinaryFormatter 200
binaryMessageEncoding element 317
binary XML
 used, for optimizing performance 315-317
binding
 about 38, 353
 concepts 38
BindingBase class 93, 96
BindingBase properties
 about 93
 converters, replacing with 93-96
BindingBase property 79
Binding class 96
Binding expression 92
binding markup extension 19
BindingSource control 38
BindingValidationError event 99, 103
BindToShellTile() method 607
BindToShellToast() method 604, 607
Bing
 about 302, 315
 URL 312
Bing API 312
Bing.com service
 invoking 312-315
BitmapImage class 92, 343
blob storage 286
BluRay class 580, 590, 594, 595
BluRayCollector application 616, 617
BluRayDataContext class 610
Button tag 233

C

caching mechanism 184
CallbackContract attribute 354
callback method 270
CanExecute method 232
CanUserReorderColumns property 142
CanUserResizeColumns property 142
Castle 228
CellEditEnded event 148-150
cellEditing variable 148
CellTemplate property 165
change-aware collection type
 building 127-130
channel 606
ChannelFactory
 about 383
 used, for calling WCF service from Silverlight 383-386
ChannelUri method 602
ChannelUri property 602
check out 31
ChildWindow class 329
classic web services 301
Cleanup() method 223
Click event 53, 72
click event handler 602
client
 about 355
 data, acquiring on 460-468
 server-side query, controlling from 476-480
ClientBin directory 18
ClientHttpStack
 advantages 407
 working with 406, 407
client-side database 609
Closed event 330
cloud
 database, accessing 274-278
 service, accessing 279-284
 Silverlight application, running from 284-286
 used, for working with push notifications 597-599
CloudBerry Explorer 286, 287
cloud computing 274
CLR 7, 32

CLR feature 265
code generation 468
CodePlex 626
CollectionChanged event 129
collections
 about 66
 binding, to UI elements 56-59
ColumnWidth property 142
ComboBox control 10, 160
COM interop 264
commanding 213
Command property 231
commands
 used, to pass events to ViewModel 229-234
Common Language Runtime. *See* **CLR**
composition keyword 227
Computer Inventory application 388
Computer Management 388
concurrency
 about 496
 working with 497-502
concurrency issues 495
ConnectAsync method 298, 351
Contains() method 187
content control 566
ConvertBack method 90, 91
converter on steroids 213
ConverterParameter 92
Converter property 90
converters
 about 88
 replacing, with BindingBase properties 93-96
 used, for displaying images based on URL 92,
 93
 used, for hooking, into data binding properties
 88-91
 working 91
converters, data binding and DataContext
 combining, into custom DataTemplate
 130-137
ConvertIntoBluRayDtoList() method 582
Convert method 90-92
cookies
 comparing, with isolated storage 176
Count() method 614
CreateNotificationChannel() method 602
CreatePersonVM() method 223

Create, Read, Update, and Delete. *See* **CRUD**
CreateTableDefinition() method 196
credentials
 passing 449
 passing, to Twitter from trusted Silverlight
 application 439-447
cross-domain access
 about 447-449
 passing, to Twitter from trusted Silverlight application 439-447
cross-domain calls
 configuring 254-261
 making, from trusted Silverlight application
 262-265
cross-domain restrictions 262
crossdomain.xml files 434
CRUD 150, 421
culture parameter 90
CurrencyConverter class 89
currentPerson property 238
custom columns
 using, in DataGridView 160-165
custom DataTemplate
 converters, data binding, and DataContext
 combining 130-137
custom markup extensions
 creating 124-127
customUserNamePasswordValidatorType attribute 373
CustomValidation attribute 530
custom validator
 used, for validating data 524-530

D

data

 acquiring, on client 460-468
 binding, to UI element 53-55
 caching, between Silverlight
 applications 190, 191
 controlling, LoadBehavior used 472-476
 customizing, templates used 109-115
 deleting, in DataGridView 146-151
 displaying, in customized DataGridView 140-145
 displaying, in Silverlight applications 40-47
 filtering, in DataGridView 156-160
 filtering, on server 481-484

getting on Windows Phone 7, WCF
 used 578-584

grouping, in DataGrid 151-155

inserting, in DataGrid 146-151

loading 202

locating 184

obtaining, from UI element 68-72

paging, in DataGrid 156-160

paging through 485-489

persisting, REST service used 399-405

persisting, standardized service
 used 250-254

persisting, WCF Data Services used 422-425

reading, from isolated storage 176-184

reading, from REST service 389-395

reading, WCF Data Services used 416-420

saving 202

server-side validation, with client side
 feedback 532-535

sorting, in DataGrid 151-155

sorting, on server 481-484

storing, in isolated storage 176-184

storing, in local SQL CE database 609-615

updating, in DataGrid 146-151

validating 522-535

validating, custom validator used 524-531

validating, data annotations used 521-524

validating, IDataErrorInfo used 104-108

validating, INotifyDataErrorInfo used 104-108

validating, ValidationContext used 541-544

validation, triggering 536, 538

data annotations

 about 101, 171

 CustomValidationAttribute 174

 DataTypeAttribute 174

 RangeAttribute 174

 RegularExpressionAttribute 174

 RequiredAttribute 174

 StringLengthAttribute 174

 used, for validating data 521-524

 uses 104

database

 accessing, in cloud 274-278

 creating 201

database driver 201

DatabaseService class 197

data binding

 about 37

 creating, from Expression Blend 5 81-83

 different modes, used for persisting
 data 73-77

 process 38, 39

data binding engine 87

data binding expressions

 debugging, in Visual Studio 77-80

data binding process

 hooking into 88-91

data bound input

 about 97

 validating 97-99

DataContext 68

DataContext binding statement 225

DataContext class 610

DataContract attribute 12, 311, 595

data-driven Silverlight 5 application

 creating, Visual Studio 2010 used 10-17

data-driven Windows Phone 7 application

 application bar 575

 application bar, menu 575, 576

 ASMX/WCF services 573

 building 562-564

 content control 566

 data binding 573

 DataContext, setting 569

 DataGrid 573

 deploying, on real device 574

 development environment 573

 hubs 577

 InitializeData() method 567

 isolated storage 576, 577

 landscape (horizontal) 574

 local data/isolated storage 573

 Login button 566

 Login page 564

 MVVM 573

 Navigate() method 575

 navigation application template 574

 NavigationService.Navigate() method 571

 OnNavigatedTo() method 571

 Panorama control 566, 569, 577

 Pivot 577

 portrait (vertical) 574

 properties window 565

REST services 573
SelectionChanged event 570
System.IO.IsolatedStorage namespace 571
ToObservableCollection<T> 568

WCF data services 573
WCF RIA services 573
Windows Phone Portrait Page, adding 563,
 572

data encryption
ensuring 363-368

DataGridView
about 146, 180
creating, steps 141-143
custom columns, using 160-165
data, deleting 146-151
data, displaying 140-145
data, filtering 156-160
data, grouping 151-155
data, inserting 146-151
data, paging 156-160
data, sorting 151-155
data, updating 146-151
master-detail, implementing 167-170
validating 171-173

DataGridViewCheckBoxColumn 152, 161

DataGridView control 545

DataGridViewTemplateColumn 152, 161

DataGridViewTextColumn 161

DataGridViewTextColumns 152

data input
validating, attributes used 101-103

DataLoader.GeneratePeople() 218

DataMember attribute 595

DataPager control 156-158

data solution
setting up, for WCF RIA Services 452-454

DataTable 109

data templates 88

Data Transfer Objects (DTO) 580

DateConverter 92

DatePicker control 168

DatePublished property 529, 543

DeactivateEngine() method 201

declared faults 324

DeleteFile() method 183

DeleteObject() method 189

dependency injection 228

dependency properties 47

deserialization 176

Developer Training Kit 375

DialogMessage type 241

DialogResult property 329, 330

Digg 426

Director attribute 527

DisplayMemberPath property 109, 111

DomainService class 466

DownloadStringAsync method 393

DownloadStringCompleted event 393

DoWork method 12

duplex communication
using, over HTTP 346-353
using, with WCF net.tcp binding 355-362

Duplex WCF communication 362

DutchDictionary class 124

dynamic bindings
creating 49-52

E

eager loading 420

element bindings
about 55
without bindings 56

ElementStyle property 164

elevated permissions
Silverlight applications, running with 363

Employee class 304

EmployeeFault class 322

Employee (IUser) class 510

EmployeeRepository class 304

EndGetRequestStream method 268

EndSaveChanges method 424

Entity class 539

Entity Framework ORM 466

entity set
persisting 490-494

environment
setting up, for developing Silverlight applica-
 tions 9

e.Result property 311

errorInfo parameter 547

Error property 547

ErrorsChanged event 524

events
passing to ViewModel, commands used 229-234

export keyword 227

Expose as OData endpoint checkbox 552

Expression Blend 5
about 20
data binding, creating from 81-83
integrating with Visual Studio 2010 20-28
source control, using 29-31
URL, for downloading 9
used, for generating sample data 84-86

Expression Suite 20

F

Facebook 388

FailedRules dictionary 108

FallbackValue property 95, 96

Federation Authentication 381

FetchItems() method 272

Fiddler
about 319, 625
local traffic 626
URL, for installing 626

Fiddler2 420

files
uploading, to WCF service 332-338

filtering 159

FinalSource property 80

Find() method 602

FireValidation method 107, 108

Flash 260

Flickr
about 261, 388, 426
additional information 433
application, building 427-432

Flush()method 202

FontSize property 164

FontWeight property 164

ForeGround property 164

G

GetAllEmployees method 308

GetAllUsers method 392

GetCallbackChannel method 354

GetErrors method 108

Get latest 31

GetMovies() method 466, 517, 518, 546

GetMoviesQuery() method 467

GetNewsAsXml method 266

GetObject() method 188

GetRolesForUser method 519

GetTranslation() method 124

GetUser method 512

GetUserStoreForApplication() method 180, 183, 191

GetUserStoreForSite() method 191

GridLinesVisibility property 142

H

HasErrors method 108

HasErrors property 547, 548

HeadersVisibility property 142

Home.xaml 506

hooks 38

HorizontalGridLinesBrush property 142

Hosted Service 274

HTTP
duplex communication, using over 346-353

HttpNotificationChannel 607

HTTPS 346

HttpsTransport element 374

HttpsTransport security 368

HttpWebRequest
used, for reading XML 265-270

HttpWebRequest class 266, 394, 590, 593, 595

hubs 577

I

IAsyncResult parameter 268

IBluRayService interface 581, 591

ICommand implementation 214

IDataErrorInfo 104

IDataErrorInfo interface 109

IdentityTKStsCert certificate 514

IIS 7.5 34

Image control 92

ImageDownload class 339

Image property 315

ImageRequest class 315

images
displaying, as stream from WCF service 338-343

images, based on URL
displaying, with converters 92, 93

ImageUpload class 333

implicit data templates
about 115
using 115-119
working 119

import keyword 228

ImportMany. *See* **import keyword**

InactivityTimeout 353

InEditMode property 136

InitializeComponent() method 50

InitializeData() method 567

InitializeOwner method 41

INotifyCollectionChanged interface 64, 127, 145, 150

INotifyDataErrorInfo interface 104-109

INotifyDataErrorInfo validation 524

INotifyPropertyChanged interface 212, 213, 218, 610

Insert method 129

InsertOnSubmit() 616

InstantiateCommand() method 230

InstantiateCommands() method 474, 478, 491

Integrated Development Environment (IDE)
11

IntelliSense 77

Internet Information Services (IIS) 356

Internet Information Services (IIS 7) 34

IsLoggedInAsync method 328

IsLoggedInCompleted event 328

IsLoggedIn method 328

isolated storage 571
about 175, 176
comparing, with cookies 176
data, caching between Silverlight applications 190, 191
data, reading from 176-184
data, storing in 176-184
size limit 185, 186

IsolatedStorageFile class 176, 180, 183

IsolatedStorageFileStream class 181-184

IsolatedStorageSettings class 176
about 186
working with 186-189

IsoStoreSettingsHelper class 187

IsReadOnly property 142

IsSelected property 120

IStockService interface 348

IsTonerAvailable 624

IsWinningCodeAsync method 250

IsWinningCodeCompleted event 250

Items property 543

ItemsSource property 19, 60, 142, 145

ItemTemplate property 60

ITonerService.cs 623

IUser interface 512

IValueConverter interface 89-93

J

JavaScript Object Notation. *See* **JSON**

JSON
about 192, 388, 408
used for accessing REST services, from Windows Phone 7 591-594
used, for communicating with REST service 408, 410

JSON/REST endpoints 550

L

landscape (horizontal) 574

LangnaugeList property 163

LanguageHelper class 163

Language property 153

LastAddedDate 213

LastName property 103, 108

LinqToEntitiesDomainService<MovieRentalDB Entities> service 512

LINQ to SQL 455

LINQ-To-XML
REST results, parsing with 395-399

ListBox 109, 160

LoadAllCities() method 179

LoadBehavior
used, for controlling data 472-476

LoadCitiesFromService() method 180

LoadCitiesFromXml() method 180, 182

LoadData() method 216, 465

LoadEmployees method 308
LoadingRow event 140
local cloud 284
local database 278
local development cloud 274
localhost certificate 514
locked-down services 416
login page 564

M

Managed Extensibility Framework. *See* **MEF**
markup extension 42, 123
MarkupExtension class 127
mashups 261
master-detail
 implementing, in **DataGridView** 167, 170
master-detail implementation 167
MDF file, SQL Server database
 attaching 625
MEF
 about 224
 used, for connecting view to **ViewModel** 224-228
message-based security
 used, for securing service communication 369-373
MessageBox 241
messenger
 leveraging, to wrap application wide messages 239, 241
metadata class 526
Microsoft Foundation Classes (MFC) 38
Microsoft Push Notifications Service 606
Microsoft SQL Server Database File (MDF) 390
model 207
Model-View-Controller. *See* **MVC**
Model-View-ViewModel. *See* **MVVM**
MouseLeftButtonDown event 112
MovieAuthenticationDomainContext 507
MovieDomainService class 466
MovieMetadata class 534
MovieValidators class 529
MVC 205

MVVM 573
 about 8, 205, 460
 application, building 206-208
 `IPropertyChanged` interface 212
 MVVM light, using to enable MVVM application 214-218
 `NotifyChanged` event 212
 object-based presentation 207
 `PersonViewModel` class 209-213
 `PersonViewModel` instance 213
 using statement 209, 212
 `ViewModel` base class 212
 `ViewModels` folder 208
MVVM Light Toolkit 214

N

navigate() method 575
NavigationService.Navigate() method 571
Net.tcp 362
net.tcp communication 8
New Connection button 412
Ninject 228
NoMoreCustomerSince property 93
NotFound error error message 323
NotFound exception 319
NotificationMessageAction, built-in type 241
NotificationMessageAction<T>, built-in type 241
NotificationMessage, built-in type 241
NotificationMessage<T>, built-in type 241
NotifyChanged event 212, 213, 218
NotifyOnValidationError property 99, 103
NotifyPropertyChangedBase 218
NuGet
 about 627
 packages 627
 URL 192
 URL, for installing 627

O

object-based representation 207
object tracking 421
ObservableCollection 147
ObservableCollection<BluRay> 567
OData 452

OData endpoints	Pictures Lab
domain service, exposing as 550	URL 558
WCF RIA Domain Services, exposing as 550-	P/Invoke 265
552	PInvoke 8
OData (R) endpoints 504	Pivot 577
OnChannelUriChanged() 604	Platform Invocation Services. See P/Invoke
OnError method 547	policy server, socket communication 297
OneTime binding 76	PollingDuplexBinding 346-353
OnNavigatedTo() method 199, 571, 583, 588,	portrait (vertical) 574
592	POST method 445, 600
Open Data Protocol endpoints. See OData	PreferredSince property 95
endpoints	ProcessRequest method 266, 267
OperationContract attribute 13, 306, 354	ProfileService 332
Owner class 80, 93	PropertyChanged event 62
OwnerDetailsEdit control 106	PropertyChangedMessage<T>, built-in type
OwnerService class 128	241
P	ProvideValue() method 125, 127
PagedCollectionView class 153, 155	proxy_GetHotelsCompleted method 19
PagerDisplayMode, values	PublishedDate property 530, 544
FirstLastNumeric 159	push notifications 560
FirstLastPreviousNext 159	about 597
FirstLastPreviousNextNumeric 159	ASP.NET WebForms page 599
Numeric 159	BindToShellTile() method 607
PreviousNext 159	BindToShellToast() method 604, 607
PreviousNextNumeric 159	channel 606
paging	ChannelUri method 602
about 159	ChannelUri property 602
through, data 485, 489	CreateNotificationChannel() method 602
Panorama 561	Find() method 602
Panorama control 566, 569, 577	HttpNotificationChannel 607
Panoramaltem 569	MessageType enumeration 600
Panoramaltem instance 566	Microsoft Push Notifications Service 606
PartCreationPolicy attribute 228	OnChannelUriChanged() 604
performance	POST method 600
optimizing, binary XML used 315-317	process 598
PersistChangeSet() method 502	RegisterForChannelEvents() method 603
Person class 386	RegisterForNotifications() 604
PersonEditViewModel 238	SendMessage() method 600, 601
PersonSelectedMessageReceived method	Send Single Push Notification button 605
238	settings page 605
PersonViewModel 210	Settings.xaml page 602
PersonViewModel class 209, 212-214	UnbindToShellToast() method 602
PersonViewModel constructor 212	unique identifier 606
PersonViewModel instance 213, 223	URL 599, 600

Q

query file, SQL Server database
executing 625

R

Really Simple Syndication. *See RSS feed*
ReceiveAsync method 298
RecentDateValue property 543
RefreshButton 183

RegisterForChannelEvents() method 603
RegisterForNotifications() 604
RegisterTable() method 201
RegisterTables() method 196, 201
RelayCommand implementation 232
remoting 301
RemoveRequest() method 619, 620
Representational State Transfer. *See REST*
request instance 271
RequiresAuthentication attribute 517, 519
RequiresRole attribute 519, 520
resource 394
Resources collection 112
REST
 about 249, 388
 used, for talking to Twitter 434-439
REST API 314
RESTful services 388
REST results
 parsing, with LINQ-To-XML 395-399
REST services
 about 573
 accessing from Windows Phone 7, JSON
 used 591-594
 accessing from Windows Phone 7, XML
 used 587-590
 creating, from WCF 623, 624
 data, reading from 389-395
 JSON, used for communicating with 408-410
 used, for persisting data 399-405
RetrieveEmployeesByUserName method 319
RIA Services 451, 452
RIAServices.DomainServices
 namespace 552
RIAServices.WebHost directory 548

roleManager tag 520
RoleProvider class 519
RoleService 332
RoundTripOriginal attribute 500
RowBackground property 142
RowDetailsTemplate 170
RowHeight property 142
RSS
 about 249
 versions 274
RSS feed
 reading out 271-273
RSS versions 274

S

sample data
 generating, Expression Blend 5 used 84-86
sandbox 175
Save() method 187, 202
SaveToXml() method 180, 181, 190
search function 23
SearchRequest class 315
SearchRequest object 313
Secure Socket Layer communication (SSL)
 363
Security Token Service 374
SelectedPersonChangedExecution method
 234
SelectionChanged event 233, 570
SendAsync method 298
SendMessage() method 600, 601
Send single push notification button 599
Send Single Push Notification button 605
SendUpdate method 348, 354
SendUpdateReceived event 351
server
 data, filtering on 481-484
 data, sorting on 481-484
 Silverlight application, deploying on 32-34
Server Certificates 514
server errors
 handling 545-548
server-side query
 controlling, from client 476-480

service
about 302, 346, 354
accessing, in cloud 279-284
connecting to 311
debugging, in Silverlight 318-323
invoking, that exposes data 302-309

Service Bus 274

service communication
securing, message-based security used 369-373

ServiceContract attribute 13, 306

service-enabled Silverlight 5 application
creating, Visual Studio 2010 used 10-17

service, fault types
declared 324
undeclared 324

set accessor 100

settings page 605

Settings.xaml 602

Settings.xaml page 602

Silverlight
about 7, 139, 243
ASP.NET authentication, using 325-331
data solution, setting up for WCF RIA Services 452-454
for Windows Phone 7 559, 560
service, debugging 318-323
socket communication, using 288-297
WCF Data Services, using with 411-415
WCF RIA Services class library, using 455-458
WCF service, adding 309-311
WCF service, calling from 383-386
WCF service connection, that exposes data 302-309
Windows Identity Foundation (WIF), integrating 374-382

Silverlight 2, 8

Silverlight 3, 302

Silverlight 5 Tools
installing 9

Silverlight application
about 8, 40, 175
Ancestor RelativeSource binding, using 120-123
Bing.com service, invoking 312-315

change-aware collection type, building 127-130
code, cleaning up 66-68
collections, binding to UI elements 56-59
configuration changes, on server 34
connecting 298
connecting with, standardized service 244-250
converters, replacing with BindingBase properties 93-96
converters, used for hooking into data binding process 88-91
cross-domain calls, configuring 254-261
cross-domain calls, making from 262-265
custom markup extensions, creating 124-127
database, accessing in cloud 274-278
data binding 19
data, binding to UI element 53-55
data bound input, validating 97-99
data, caching between 190, 191
data, displaying 40-47
data encryption, ensuring 363-368
data input, validating with attributes 101-103
data, obtaining from UI element 68-72
data, persisting with standardized service 250-254
deploying, on server 32-34
dynamic bindings, creating 49-52
enabling, for automatically updating UI 60-65
environment, setting up for developing 9
IDataErrorInfo, used for validating data 104-108
implicit data templates, using 115-119
INotifyDataErrorInfo, used for validating data 104-108
project structure 18, 19
RSS feed, reading out 271, 273
running, from cloud 284-286
running, with elevated permissions 363
server, disallowing support for XAP 35
service, accessing in cloud 279-284
services 19
setup requisites 363
solution 18, 19
XML, reading with HttpWebRequest 265-270
templates, used for customizing data 109-115

Silverlight client
WCF RIA Services link 458

Silverlight control toolkit
about 427, 626
URL, for downloading 626

Silverlight platform 454

Silverlight solution 18

SilverWitter client 440

size options
about 165
Auto 165
Pixel (Fixed) 165
SizeToCells 165

SizeToHeader 165
Star 165

SketchFlow
about 561
URL 561

smart polling 353

SOAP 314, 388, 452

SOAP (CRUD) 504

SOAP envelope 388

SOAP fault 323

socket communication
about 244, 288
policy server 297
socket server 298
using, in Silverlight 288-297

socket server, socket communication 298

SolidColorBrush 90, 91

sorting 151

SortMemberPath property 156

source control
using, in Expression Blend 5 29-31
using, in Visual Studio 2010 29-31

Source property 92, 162

SourceType enumeration 315

SQL Azure 504
about 548
using, with WCF RIA Services 548-550

SQL Azure Migration Wizard 276

SQL CE (Compact Edition) database. *See SQL CE database*

SQL CE database
database context 615
database, interaction with 616

data, storing in 609-615
local SQL CE database, data storing in 609-615
model classes 615
SubmitChanges() 616

SQL Profiler 480

SQL Server database
MDF file, attaching 625
query file, executing 625

standardized service
connecting with 244-250
reading from 244-250
used, for persisting data 250-254

StaticResource 123

StatusTextBlock 619

Sterling
about 192, 200
adding, to Silverlight project 192
database, creating 201
obtaining 200

Sterling database
data, loading 202
data, saving 202
using 193-200

Sterling engine 201

stock ticker 347

StreamReader instance 593

StringFormat property 94, 96

StringLength attribute 103

SubmitChanges() method 495, 614, 616

SubmitOperation object 495

System.Collections.Generic namespace 125

System.ComponentModel.DataAnnotations namespace 144

System.ComponentModel namespace 65

System.Data.Linq namespace 610

System.IO.IsolatedStorage namespace 571

System.Runtime.Serialization assembly 581

System.Runtime.Serialization.Json namespace 595

System.Threading namespace 348

system.web tag 520

System.Windows.Controls.Data.Input namespace 144

System.Windows.Controls.Data namespace 144

System.Windows.Controls namespace 139,
144

System.Windows.Data namespace 144

System.Windows.Interactivity assembly 233

T

Table attribute 610

Table<BluRay> property 614

Tag property 171

target control 38

TargetNullValue property 93, 96

TCP communication 362

TcpListener class 297

Team Foundation Server. *See* **TFS**

template column

sorting 156

templates

about 109

used, for customizing data 109-115

TextBlock 213

TextBlock control 168

TextBlock controls 10

TextBlock template 539

TextBox 88

Text property 46

TFS 29

TFS terms

about 31

check in 31

check out 31

Get latest 31

TFS workspace 31

TFS workspace 31

Tick event 65, 348

tight coupling 235

TimeoutException 353

Title property 167, 535

TonerService.cs 623

TonerService, WCF service 623

ToObservableCollection<T> 568

ToString() method 114, 109, 145

Total Cost of Ownership (TCO) 283

TrailerAndDirector validator 530

TrailerLocation attribute 527

TrailerLocation properties 530

transactions

about 496

working with 497-502

Transact SQL statements

URL 550

TransferProgressChanged event 619

TransferStatusChanged event 618, 619

TranslatorExtension class 125

transport security 374

trigger 197, 202

trusted application

about 262, 439

credentials, passing to Twitter 439-447

cross-domain access, passing to
Twitter 439-447

Trusted Silverlight application

creating 448

TryValidate-method 539

TryValidateObject 538

TryValidateProperty 538

tweets 434

Twitter

about 261, 388, 434

REST, used for talking to 434-439

TwoWay binding 77, 90, 400

U

UI element

collections, binding to 56-59

data, binding to 53-55

data, obtaining from 68-72

data, obtaining from steps 70

UI virtualization 140

UnbindToShellToast() method 602

undeclared faults 324

unique identifier 606

UpdateUser 512

UriTemplate 624

user access

to service methods, controlling 517-520

to services, controlling 517-520

User class **512**
user identity tracking, custom authentication service **550**
about 507-510
Employee (IUser) class 510
GetUser method 512
IUser interface 512
LinqToEntitiesDomainService<MovieRentalDB Entities> service 512
Logout method 512
MovieAuthenticationDomainService, loading 508
using statement 508
ValidateUser method 512
user identity tracking, default Windows authentication
about 504, 505
Application_Startup method 505

App.xaml, code adding to 506
App.xaml.cs, statements adding to 505
MovieAuthenticationDomainContext 507
new authentication domain service, adding 505
WebContext 507
WebContext.Current 507
XAML code, adding to Home.xaml 506
xmlns imports, adding to App.xaml 506

User Interface (UI) **10**
User Management **388**
userNameAuthentication element **373**
UserNamePasswordValidator class **373**
using statement **209, 212, 508, 540**

V

ValidateButton handler **106**
Validate() method **537, 540**
ValidatesOnExceptions property **99, 103**
ValidatesOnNotifyDataErrors property **524**
ValidateUser method **512**
validationContext **529**
ValidationContext **529, 543**
used, for validating data 541-544
ValidationContextValidation.Items 545

ValidationContext constructor **544**
ValidationContext object **544**
ValidationErrors collection **539**
Validation.Items **545**
ValidationResult **529**
validations **88**
ValidationSummary control
about 100
using 100

validator class **538**

value parameter **90**

Verisign **365**

view
about 206
connecting to ViewModel, MEF used 224-228
connecting to ViewModel, ViewModelLocator used 219-223
connecting to ViewModel, ViewModelLocator used 219-223

View First-approach **223**

ViewModel
about 206
communicating, between 235-238
events passing, commands used 229-234
view connecting to, MEF used 224-228
view connecting to, ViewModelLocator used 219-223

ViewModel base class **212**

ViewModelBase class **218**

ViewModel classes **214**

ViewModelLocator
used, for connecting view to ViewModel 219-223

ViewModel property **225, 227**

ViewModels folder **208**

Visual Studio
data binding expressions, debugging 77-80

Visual Studio 2010
data-driven Silverlight 5 application, creating 10-17
integrating, with Expression Blend 5 20-28
service-enabled Silverlight 5 application, creating 10-17
source control, using 29-31

Visual Web Developer Express 2010
installing 9

W

WCF

about 301, 302, 345
ASMX services 585
credentials, sending 586
data transfer 585, 586
REST services, creating 623, 624
services 585
used, for getting data on Windows Phone 7 578-584

WCF data services 573

WCF Data Services

about 411
used, for persisting data 422-425
used, for reading data 416-420
using, with Silverlight 411-415

WCF net.tcp binding

duplex communication, using with 355-362

WCF RIA Domain Services

exposing, as OData endpoints 550-552
exposing, for other technologies 553-555

WCF RIA services

about 573
WIF, integrating 513

WCF RIA Services. *See also RIA Services*
about 451
better naming, for templates 458, 459
class library, using 455-458
data, acquiring on client 460-468
data, filtering on server 481-484
data solution, setting up for 452-454
data, sorting on server 481-484
enabling 549
entity set, persisting 490-494
LoadBehavior, used for controlling data 472-476
missing properties 469, 471
paging, through data 485-489
properties, excluding 472
server-side query, controlling from client 476-480
SQL Azure, using 548-550
toolkit 455

WCF RIA Services class library

about 455
using 455-458

WCF RIA Services Toolkit

about 455
URL 553
URL, for installing 627

WCF service

about 249, 301
adding 309-311
calling, from Silverlight with ChannelFactory 383-386
connecting to 311
files, uploading to 332-338
images, displaying as stream 338-343
TonerService 623

WebClient class 391, 394, 590

web.config file 309, 321, 374, 553

WebContext 507

WebContext.Current 507

Web Development Helper 319

WebGetAttribute 624

webHttpBinding 588

WebHttpBinding 623

WebRequest instance 270

web role 283

Web Service Description Language (WSDL) 249

WIF

about 346, 374, 513, 627
integrating, in Silverlight 374-382
integrating, with WCF RIA services 513
RequiresAuthentication attribute 517
Windows Identity Foundation Developer Training Kit, installing 514
Windows Identity Foundation Runtime, installing 513
Windows Identity Foundation SDK, installing 513

Windows 7 Phone Mango. *See Windows Phone 7*

Windows Azure 452, 504

Windows Azure Emulator 274

Windows Azure Platform 244

Windows Azure SDK 279

Windows Communication Foundation. *See* **WCF**

Windows Forms **38**

Windows Identity Foundation. *See* **WIF**

Windows Identity Foundation Developer Training Kit **374**
URL **514**

Windows Identity Foundation Runtime **374**

Windows Identity Foundation SDK **374**
URL, for installing **513**

Windows Mobile. *See* **WM**

Windows Phone **504**

Windows Phone 7
about **557**, **558**
environment, preparing **562**
Panorama **561**
push notifications **597-602**
REST services accessing, JSON used **591-594**
REST services accessing, XML used **587-590**
Silverlight for **559**, **560**

Windows Phone 7.5 **562**

Windows Presentation Foundation (WPF) **8**, **139**

Windows Process Activation Service (WAS) **357**

WinForms **504**

WM
about **557**
application **558**

workflow
using, between Visual Studio 2010 and Blend **5** **20-28**

wp\$Count element **608**

wp\$Count field **608**

WP7. *See* **Windows Phone 7**

WP7 SDK
URL, for downloading **562**

WPF **504**

wrap application wide messages
messenger, leveraging to **239-241**

X

XAML **8**, **39**

XAP file **139**

XLinq. *See* **LINQ-To-XML**

XML
about **192**
reading, `HttpWebRequest` used **265-270**
used for accessing REST services, from Windows Phone 7 **587-590**

XmlReader/XmlWriter **395**

XmlSerializer **395**

XmlSerializer class **405**

Y

YouTube **388**



Thank you for buying **Microsoft Silverlight 5 Data and Services Cookbook**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.

For more information, please visit our website: www.PacktPub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Microsoft Silverlight 5: Building Rich Enterprise Dashboards

Create, customize, and design rich enterprise dashboards with Microsoft Silverlight 5

Todd Snyder Joel Eden, Ph.D.
Jeff Smith Matthew Duffield

[PACKT] enterprise[®]
professional expertise distilled

Microsoft Silverlight 5: Building Rich Enterprise Dashboards

ISBN: 978-1-84968-234-3 Paperback: 288 pages

Create, customize, and design rich enterprise dashboards with Microsoft Silverlight 5

1. With this book and e-book, learn how to create, customize and design rich enterprise dashboards with Silverlight
2. Move from scenarios to requirements by applying user-centered design best practices
3. Discover the tips, tricks and hands on experience to create, customize and design rich enterprise dashboards with Silverlight from a distinguished team of User Experience and Development authors



Microsoft Silverlight 4 Data and Services Cookbook

Over 85 practical recipes for creating rich, data-driven business applications in Silverlight

Gill Cleeren Kevin Dockx

[PACKT] enterprise[®]
professional expertise distilled

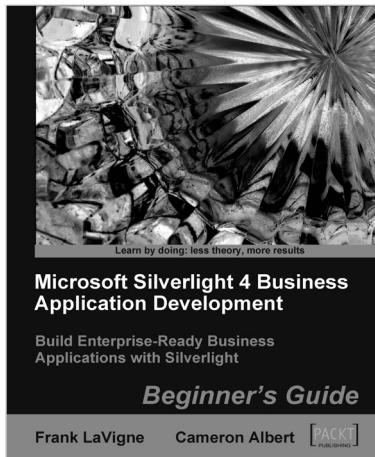
Microsoft Silverlight 4 Data and Services Cookbook

ISBN: 978-1-847199-84-3 Paperback: 476 pages

Over 85 practical recipes for creating rich, data-driven business applications in Silverlight

1. Design and develop rich data-driven business applications in Silverlight
2. Rapidly interact with and handle multiple sources of data and services within Silverlight business applications
3. Understand sophisticated data access techniques in your Silverlight business applications by binding data to Silverlight controls, validating data in Silverlight, getting data from services into Silverlight applications and much more!

Please check www.PacktPub.com for information on our titles

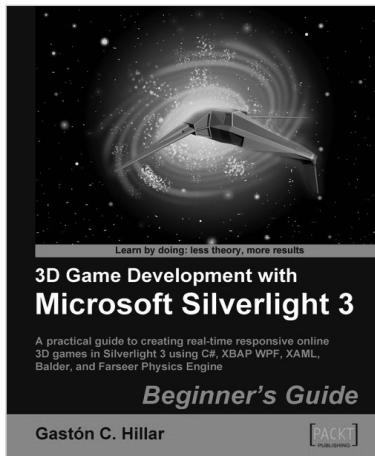


Microsoft Silverlight 4 Business Application Development: Beginner's Guide

ISBN: 978-1-847199-76-8 Paperback: 412 pages

Build enterprise-ready business applications
with Silverlight

1. An introduction to building enterprise-ready business applications with Silverlight quickly.
2. Get hold of the basic tools and skills needed to get started in Silverlight application development
3. Integrate different media types, taking the RIA experience further with Silverlight, and much more!
4. Rapidly manage business focused controls, data, and business logic connectivity.



3D Game Development with Microsoft Silverlight 3: Beginner's Guide

ISBN: 978-1-847198-92-1 Paperback: 452 pages

A practical guide to creating real-time responsive online 3D games in Silverlight 3 using C#, XBAP WPF, XAML, Balder, and Farseer Physics Engine

1. Develop online interactive 3D games and scenes in Microsoft Silverlight 3 and XBAP WPF
2. Enhance development with animated 3D characters, sounds, music, physics, stages, gauges, and backgrounds
3. Packed with inspiring, realistic examples offering impressive graphics, strong performance, and a rich interactive experience

Please check www.PacktPub.com for information on our titles