



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight books

Manage data in Silverlight, build and maintain rich dashboards,
integrate SharePoint with Silverlight, and more

Series Editor
Carl Jones

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.allitebooks.com

Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight books

Manage data in Silverlight, build and maintain rich dashboards, integrate SharePoint with Silverlight, and more

Series Editor

Carl Jones

[PACKT] enterprise 
PUBLISHING professional expertise distilled

BIRMINGHAM - MUMBAI

Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight books

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 published: February 2012

Production Reference: 2200212

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

ISBN 978-1-84968-564-1

www.packtpub.com

Cover Image by Artie Ng (artherng@yahoo.com.au)

Credits

Series Editor

Carl Jones

Technical Editor

Arun Nadar

Contributors

Gastón C. Hillar

Gill Cleeren

Kevin Dockx

Todd Snyder

Joel Eden, PhD

Jeffrey Smith

Matthew Duffield

Cameron Albert

Frank LaVigne

Vibor Cipan

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Contributors

Gastón C. Hillar has been working with computers since he was eight. He began programming with the legendary Texas TI-99/4A and Commodore 64 home computers in the early 80s.

He has a Bachelor's degree in Computer Science – graduated with honors – and an MBA (Masters in Business Administration) – graduated with an outstanding thesis. He worked as a developer, architect, and a project manager for many companies in Buenos Aires, Argentina. Now, he is an independent IT consultant and a freelance author always looking for new adventures around the world. He also works with electronics (he is an electronics technician). He is always researching about new technologies and writing about them. He owns an IT and electronics laboratory with many servers, monitors, and measuring instruments.

Gastón wrote the C# 2008 and 2005 Threaded Programming: Beginner's Guide also published by Packt.

He is also the author of more than 40 books in Spanish about computer science, modern hardware, programming, systems development, software architecture, business applications, balanced scorecard applications, IT project management, the Internet, and electronics.

He contributes to Dr. Dobb's Go Parallel programming portal <http://www.ddj.com/go-parallel/> and he is a guest blogger at Intel Software Network <http://software.intel.com>

He usually writes articles for Spanish magazines Mundo Linux, Solo Programadores, and Resistor.

Gill Cleeren is Microsoft Regional Director (<http://www.theregion.com>), Silverlight MVP (former ASP.NET MVP), and Telerik MVP. He lives in Belgium where he works as a .NET architect at Ordina (<http://www.ordina.be/>). He is passionate about .NET and always plays with the newest bits. In his role as Regional Director, Gill has given many sessions, webcasts, and training on new as well as existing technologies, such as Silverlight, ASP.NET, and WPF at conferences including TechEd Berlin 2010, TechDays Belgium – Switzerland – Sweden, DevDays NL, NDC Oslo Norway, DevReach Bulgaria, NRW Conference Germany, Spring Conference UK, Telerik Silverlight Roadshow in Sweden, and Telerik RoadShow UK. He is the author of Packt's *Microsoft Silverlight 4 Data and Services Cookbook* and is also the author of many articles in various developer magazines and for SilverlightShow.net. He organizes the yearly Community Day event in Belgium and also leads Visug (<http://www.visug.be>), the largest .NET user group in Belgium. You can find his blog at <http://www.snowball.be> and on Twitter by following @gillcleeren.

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, and HTML5). His main focus is on all things related to 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 on various national and international events, such as Microsoft DevDays in the Netherlands, Microsoft Techdays in Portugal, NDC Oslo Norway, and Community Day Belgium. He is the author of Packt's *Microsoft Silverlight 4 Data and Services Cookbook* and also writes articles for various Silverlight-related sites and magazines. His blog, which contains various tidbits on Silverlight, .NET, and the occasional rambling, can be found at <http://blog.kevindockx.com/>, and you can find him on Twitter as well: @KevinDockx.

Todd Snyder has been a software developer/architect for over 16 years. During that time, he has spent several years as a consultant providing technical guidance and leadership for the development of enterprise class systems on the Microsoft Platform. At Infragistics, he is a principal consultant that focuses on the design and construction of RIA and n-tier based applications. Todd is the co-leader for the New Jersey .Net user group (<http://www.njdotnet.net/>) and is a frequent speaker at trade shows, code camps, and Firestarters.

Joel Eden, PhD has been working in the area of user experience and design methods for over 10 years. Currently a Senior Interaction Designer working on UX Tools at Infragistics, he spent three years in the Infragistics Services group, consulting for external clients. Prior to Infragistics, he worked at multiple design agencies in the Philadelphia area, as well as working at Lockheed Martin's Advanced Technology Labs. Joel holds a B.S. in Computer Science, and a Ph.D in Information Science, both from Drexel University.

Jeffrey Smith has been a Visual Designer for six years. During that time he has been an Art Director at various agencies and studied special effects and animation at NYU. A convert from flash and flex, he has been working with .NET technologies for the past two years, specializing in WPF and Silverlight. At Infragistics, he is an UX Visual Designer that focuses on the design, implementation, and user experience. You can view some of his work at <http://www.thinksinkstudio.com>.

Matthew Duffield is a .NET architect in designing and developing enterprise applications. He specializes in .NET with an emphasis on WPF, Silverlight, and WP7 development. He is a Microsoft MVP in Client Application Development and has an MSCD.NET certification. He also works in business intelligence, designing, and developing solutions for data warehouse and data mining initiatives. You can read his blog at mattduffield.wordpress.com and follow him on Twitter at @mattduffield.

Cameron Albert is an independent software development consultant with over 10 years of experience, specializing in Microsoft technologies such as Silverlight, WPF, WCF, SQL Server, and ASP.NET. Having worked in the medical, insurance, and media/entertainment industries, Cameron has been involved in a variety of development solutions featuring a broad range of technical issues. He also dabbles in game development utilizing Silverlight and maintains a blog detailing his exploits into the development world here: <http://www.cameronalbert.com>.

Frank LaVigne has been hooked on with software development since he was 12, when he got his own Commodore 64 computer. Since then, he's worked as developer for financial firms on Wall Street and also in Europe. He has worked on various Tablet PC solutions and on building advanced user experiences in Silverlight and WPF. He lives in the suburbs of Washington, DC. He founded the CapArea.NET User Group Silverlight Special Interest Group and has been recognized by Microsoft as a Tablet PC MVP. He blogs regularly at www.FranksWorld.com.

Vibor Cipan is currently serving as the CEO and Partner of FatDUX Zagreb – a full service interactive UX and service design agency with offices around the world. Before joining FatDUX, Vibor, he worked at Microsoft Development Center in Copenhagen and before that, at Microsoft, Croatia. One thing, however, has stayed constant – his focus on user experience, service design, usability, and information architecture. He has been awarded the prestigious title of *Microsoft Most Valuable Professional* for three years in a row (and is still currently holding that title). He was the youngest awardee and the first one in the CEE, Europe to receive the award while being a full-time student.

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: Layouts and General Content Organization	9
Introduction	10
Fluid layout	10
Creating a navigation pane from scratch	17
Window management and positioning	27
Wizards	37
Progressive disclosure—showing additional controls on demand	49
Control docking with DockPanel	54
Journal navigation	59
Tabs	66
Adding a status bar area	75
Chapter 2: Handling Data	85
Data applications	86
Time for action – creating a business object	86
Windows Communication Foundation (WCF)	89
Time for action – creating a Silverlight-enabled WCF service	90
Collecting data	99
Time for action – creating a form to collect data	99
Validating data	111
Data object	111
Time for action – creating a data object	111
Data binding	116
Time for action – binding our data object to our controls	116
Validation	120
Time for action – validating data input	121
Data submission	125

Time for action – submitting data to the server	125
Summary	130
Chapter 3: An Introduction to Data Binding	131
Introduction	132
Displaying data in Silverlight applications	134
Creating dynamic bindings	145
Binding data to another UI element	148
Binding collections to UI elements	152
Enabling a Silverlight application to automatically update its UI	156
Obtaining data from any UI element it is bound to	163
Using the different modes of data binding to allow persisting data	168
Data binding from Expression Blend 4	173
Using Expression Blend 4 for sample data generation	176
Chapter 4: Advanced Data Binding	179
Introduction	180
Hooking into the data binding process	180
Replacing converters with Silverlight 4 BindingBase properties	184
Validating databound input	188
Validating data input using attributes	193
Validating using IDataErrorInfo and INotifyDataErrorInfo	195
Using templates to customize the way data is shown by controls	200
Building a change-aware collection type	206
Combining converters, data binding, and DataContext into a custom DataTemplate	209
Chapter 5: The Data Grid	217
Introduction	217
Displaying data in a customized DataGrid	218
Inserting, updating, and deleting data in a DataGrid	224
Sorting and grouping data in a DataGrid	228
Filtering and paging data in a DataGrid	233
Using custom columns in the DataGrid	237
Implementing master-detail in the DataGrid	244
Validating the DataGrid	248
Chapter 6: Talking to REST and WCF Data Services	253
Introduction	254
Reading data from a REST service	255
Parsing REST results with LINQ-To-XML	260
Persisting data using a REST service	264
Working with the ClientHttpStack	270

Communicating with a REST service using JSON	273
Using WCF Data Services with Silverlight	276
Reading data using WCF Data Services	281
Persisting data using WCF Data Services	286
Talking to Flickr	291
Talking to Twitter over REST	298
Passing credentials and cross-domain access to Twitter from a trusted Silverlight application	303
Chapter 7: Interacting with Data on the SharePoint Server	315
Managing data in a Silverlight RIA included in a SharePoint solution	316
Working with the SharePoint 2010 Silverlight Client Object Model to insert items	316
Inserting items in a SharePoint list with the Silverlight Web Part	324
Working with successful and failed asynchronous queries	327
Retrieving specific information about fields	331
Creating complex LOB applications composed of multiple Silverlight RIAs	340
Interacting with multiple Silverlight Web Parts in the same page	345
Understanding Line-Of-Business systems as independent Web Parts	347
Expanding LOB systems with delete operations	349
Understanding how to delete an item from a list	352
Expanding LOB systems with update operations	354
Updating an item in a list	357
Summary	359
Chapter 8: Interacting with Rich Media and Animations	361
Bringing life to business applications and complex workflows	362
Creating asset libraries in SharePoint 2010	362
Adding content to an assets library	364
Browsing the structure for SharePoint Asset Libraries	367
Controlling the rich media library by using controls in a Visual Web Part	369
Creating a Silverlight RIA rendered in a SharePoint Visual Web Part	376
Linking a SharePoint Visual Web Part to a Silverlight RIA	390
Adding a SharePoint Visual Web Part in a Web Page	393
Organizing controls in a containing box	399
Reading files from an assets library	400
Working with interactive animations and effects	401
Adding and controlling videos	408
Video formats supported in Silverlight 4	412
Adding and controlling sounds and music	412
Audio formats supported in Silverlight 4	413
Changing themes in Silverlight and SharePoint	414
Summary	417

Chapter 9: Data Access Strategies	419
Data access overview	420
Core networking classes	420
Working with WebClient	420
Using Fiddler	424
Understanding network security	425
Building services with Windows Communication Foundation	427
Working with WCF	429
The data access layer	430
Building a SOAP service	433
Building a REST service	436
Exploring OData data services	439
Building an OData service	441
Consuming an external service	444
Summary	446
Chapter 10: Building Dashboards in SharePoint and Silverlight	447
Overview of SharePoint	448
Setting up SharePoint	448
Building a Silverlight web part	450
Using the Client Object Model	455
Building a SharePoint Silverlight dashboard	459
Setting up our data source	459
Building our dashboard	460
SharePoint Data Access Strategies	466
Summary	466
Chapter 11: Working with 3D Characters	467
The second remake assignment	468
Time for Action – exporting a 3D model without considering textures	468
XAML 3D models	473
Time for action – from DCC tools to WPF	475
XBAP WPF applications with 3D content	476
Time for action – displaying a 3D model in a 2D screen with WPF	477
Understanding the 3D world	481
X, Y, and Z in practice	483
GPU 3D acceleration	484
Understanding meshes	485
Time for action – using other XAML exporter for DCC tools	486
Time for action – adding 3D elements and interacting with them using Expression Blend	488
Interacting with 3D elements using Expression Blend	491

Silverlight and the 3D world	492
Time for action – exporting a 3D model to ASE	492
Time for action – installing Balder 3D engine	493
Time for action – from DCC tools to Silverlight	495
Displaying a 3D model in a 2D screen with Silverlight	498
Using 3D vectors	499
Summary	500
Pop quiz answers	500
Index	501

Preface

A Packt Compendium is a book formed by drawing existing content from several related Packt titles. In other words, it is a mashup of published Packt content – Professional Expertise Distilled in the true sense. Such a compendium of Packt's content allows you to learn from each of the chapters' unique styles and Packt does its best to compile the chapters without breaking the narrative flow for the reader.

Please note that the chapters in this compendium were originally written and intended as a part of various separate Packt titles, so you might find that the information included in this instance is more akin to that of a standalone chapter, rather than creating step-by-step, continuous flowing prose. We are sure that you will find this medley a useful and valuable resource with which you can benefit from a range of Packt books – and their authors' expertise!

Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight focuses on showing .NET developers how to interact with, and handle multiple sources of data in Silverlight business applications, and how to solve particular data problems following a practical hands-on approach, using real-world examples. This book is a collection of media- and data-based chapters from Packt's best selling Silverlight books:

1. *Silverlight 4 User Interface Cookbook*
2. *Microsoft Silverlight 4 Business Application Development: Beginner's Guide*
3. *Microsoft Silverlight 4 Data and Services Cookbook*
4. *Microsoft Silverlight 4 and SharePoint 2010 Integration*
5. *Microsoft Silverlight 4: Building Rich Enterprise Dashboards*
6. *3D Game Development with Microsoft Silverlight 3: Beginner's Guide*

Microsoft Silverlight is a programmable web browser plugin that enables features including animation, vector graphics, and audio-video playback features that characterize Rich Internet Applications. However, Silverlight is a great Line-Of-Business platform and is increasingly being used to build data-driven business applications. This book will enable .NET developers to get their finger on the pulse of data-driven business applications in Silverlight.

In this book, you will find content in various easy-to-follow styles such as a recipe-based cookbook format, tutorial-based beginner's guide, and a reference-styled handbook. The aim of this book is to provide a lot of valuable content to you from various other Packt Silverlight books. It is designed in such a way that you can refer to the topics chapter-by-chapter, and read them in no particular order. It offers clear examples to successfully perform the most important data-related tasks in Silverlight.

What this book covers

The book starts with discussion on layouts and content organization and covers all the options available to access data and communicate with services to get the most out of data in your Silverlight business applications, at the same time providing a rich user experience. 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! 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.

Chapter 1, Layouts and General Content Organization, covers important layout considerations to be made before we start building any application, regardless of being a web application built with Silverlight or a typical desktop application built with Windows Presentation Foundation.

Chapter 2, Handling Data, covers the process of collecting and handling data input from a customer and saving that input on the server. We also looked at how to bind data to control properties and how to provide simple data validation using the built-in visual states provided in the textbox control.

Chapter 3, An Introduction to Data Binding, explains how 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.

Chapter 4, Advanced Data Binding, explores the data binding engine that gives many points where we can extend or change the binding process.

Chapter 5, The Data Grid, shows how to work with the DataGrid. This is an essential control for applications that rely on (collections of) data.

Chapter 6, Talking to REST and WCF Data Services, here, 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.

Chapter 7, Interacting with Data on the SharePoint Server, will cover many topics that help us create simple and complex Line-Of-Business Silverlight RIAs that run as Silverlight Web Parts to interact with data in the SharePoint Server.

Chapter 8, Interacting with Rich Media and Animations, will cover many topics related to retrieving digital assets from SharePoint libraries through the SharePoint Silverlight Client Object Model and consuming them in a Silverlight RIA.

Chapter 9, Data Access Strategies, will introduce you to the features included in SharePoint 2010 for hosting Silverlight dashboard applications. We will explore how to set up a Silverlight web part, and use the SharePoint Silverlight Client Object Model to communicate with data hosted in SharePoint.

Chapter 10, Building Dashboards in SharePoint and Silverlight, will explore the different data access strategies you can use while building a Silverlight application. How to build your own custom data services using SOAP, REST, and OData, a walkthrough of how to consume externally-hosted services, and how the cross-domain security policy system works with Silverlight to call external services.

Chapter 11, Working with 3D Characters, will take 3D elements from popular and professional 3D DCC tools and we will show them rendered in real-time on the screen.

What you need for this book

As this Packt Compendium is a mash-up of published Packt content, the prerequisites may vary between each chapter. Everything you will need for this book is detailed according to the respective source title:

Chapters taken from *Silverlight 4 User Interface Cookbook* uses **Expression Blend 4** for virtually all recipes in it. You might find it useful to use **Visual Studio 2010** (or one of its free "express" editions) for better code editing and development experience, but Expression Blend 4 should be your first choice for this book.

Though the book covers mostly **Silverlight 4** and the user interface patterns and user experience guidelines are referring to Silverlight, first few chapters utilize **WPF 4 (Windows Presentation Foundation)** technology. Almost all ideas, approaches, methods, and guidelines applicable to WPF are also applicable to Silverlight itself.

In order to use and follow all recipes, be sure that, apart from Expression Blend 4 you have installed **Silverlight Toolkit**. Silverlight toolkit adds support for numerous additional controls of your Silverlight 4 controls. You can get it from <http://silverlight.codeplex.com>.

The last part of the book showcases usage of the PathListBox control. This control supported under Silverlight 4 is not (at the moment of writing this book) part of the Silverlight or Silverlight Toolkit. Also, effects, pixel shaders, and numerous other features are available to you when you install **Expression Blend 4 SDK for Silverlight 4** available for free from: <http://www.microsoft.com/downloads/details.aspx?FamilyID=d197f51a-de07-4edf-9cba-1f1b4a22110d&displaylang=en> (short link: <http://bit.ly/9KaiIG>).

Chapters taken from *Microsoft Silverlight 4 Business Application Development: Beginner's Guide* will need the following tools to view the samples and run the code provided. While the Expression tools are discussed and used within the book they are not a requirement to build Silverlight applications, they simply make it easier. Visual Studio 2010 provides a design view of XAML pages so that you can visually design the interface, which saves a lot of hand coding of XAML.

- ▶ Visual Studio 2010
- ▶ Silverlight 4 Tools for Visual Studio
- ▶ WCF RIA Services
- ▶ Expression Blend
- ▶ Expression Encoder
- ▶ SQL Express
- ▶ A SharePoint VPC or development installation (for the SharePoint samples)

Chapters taken from *Microsoft Silverlight 4 Data and Services Cookbook* requires that you have Visual Studio installed. This book targets Silverlight 4, which works only with Visual Studio 2010. Many of the recipes in the book will also work in Silverlight 3, so for these recipes, you have the choice between Visual Studio 2008 and 2010. 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 4 to be installed on your machine (again, if working with Silverlight 3, Blend 3 will suffice here as well).

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.

Chapters taken from *Microsoft Silverlight 4 and SharePoint 2010 Integration* will need the following software products:

- ▶ Visual Studio 2010 Professional, Premium, or Ultimate
- ▶ SharePoint 2010 Server or SharePoint 2010 Foundation, installed on the same computer that runs Visual Studio 2010
- ▶ SharePoint Designer 2010

Chapters taken from *Microsoft Silverlight 4: Building Rich Enterprise Dashboards* will need to have the following available:

- ▶ Visual Studio 2010 Express or Professional Edition
- ▶ SQL Server 2008 R2 Express or Developer Edition
- ▶ Silverlight 5.0 SDK
- ▶ Silverlight 5.0 Toolkit
- ▶ Microsoft Expression Blend 4.0 (with Updates for Silverlight 5.0)
- ▶ Share Point 2010 Foundations (Optional only required for *Chapter 10, Building Dashboards in SharePoint and Silverlight*)

Chapters taken from *3D Game Development with Microsoft Silverlight 3: Beginner's Guide* will need Visual C# 2008 (.NET Framework 3.5) with Service Pack 1, or greater – Visual C# 2010 – installed.

You can use the free Visual Web Developer 2008 Express Edition or greater (<http://www.microsoft.com/express/vwd/>). However, you have to read the documentation to consider its limitations carefully.

Who this book is for

If you are a .NET developer who wants to manage professional data-driven applications with Silverlight, then this book is for you. Basic experience of programming Silverlight and familiarity with accessing data using ADO.NET in normal .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: "The `CakeService.svc.cs` file will contain the implementation of our service interface."


A block of code is set as follows:


```
<Grid x:Name="LayoutRoot">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="69"/>
    <ColumnDefinition Width="0.52*"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="69"/>
  </Grid.ColumnDefinitions>
</Grid>
```

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

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new MainPage();
    // Initialize the ApplicationContext
    ApplicationContext.Init(e.InitParams,
        System.Threading.SynchronizationContext.Current);
}
```

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 the text like this: "Click on **UserControl** and change its height and width to **Auto**".

 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 via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

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 all Packt books you have purchased from your account at <http://www.packtpub.com>. 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.

This book is a compendium title which is extracted from five different best-selling books by Packt. The code bundles for these individual titles can be also downloaded from www.packtpub.com/support.

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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

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

LAYOUTS and General Content Organization



This chapter is taken from *Silverlight 4 User Interface Cookbook* (Chapter 1) by Vibor Cipan.

In this chapter, we will cover:

- ▶ Liquid versus fixed layouts
- ▶ Navigation pane and how to create one from scratch
- ▶ Window management and positioning
- ▶ Wizards
- ▶ Progressive disclosure—showing additional controls on demand
- ▶ Control docking with Dock Panel
- ▶ Journal navigation
- ▶ Tabs
- ▶ Status bar area

Introduction

Before we start building any application, regardless of being a web application built with Silverlight or a typical desktop application built with Windows Presentation Foundation, we will be faced with making some very basic and extremely important decisions. Will our application be able to scale to all of the different screens and resolutions or are we going for a fixed size? Other than that, how are we going to navigate through data presented in our application? How do we deal with windows and their positions, sizes, and states? We might go further and ask ourselves—can we conceive our application as the number of steps that users have to go through to complete one or more tasks? In that case, we might consider using the wizard UI pattern—how to create wizards and use Aero Wizard guidelines instead of older and obsolete Wizard97 guidelines.

What will happen if we present too much data and information to the users, especially the data is irrelevant at the moment? It might lead to user confusion and dissatisfaction. Then, we can consider using the progressive disclosure and interesting UI patterns, which will help us cope with the "control and data overload" challenges.

When we change the screen resolution or resize our windows or pages (please note that throughout this book these terms are being used to designate both, windows as parts of desktop applications and pages as parts of web applications, and in many cases UI patterns are same for both of those), controls might need to change their position, or even size. How do we implement and efficiently use control docking and scaling in those scenarios?

Tabs are really useful: they enable us to put different content and controls on them. but despite them being so well-known, they are often misused and their usage can lead to a user's frustration. How do we use tabs properly and avoid having our users being frustrated and unhappy with our UI?

Often, applications need to communicate information to their users. Some use pop-up windows, message boxes, or status bars. When should you consider using a status bar in your applications and when might it be a better idea to use some other UI pattern?

This introduction has asked a lot of relevant questions. Now, it's time to proceed to some concrete answers and patterns.

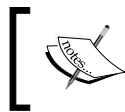
Fluid layout

There are two basic types of layouts that I want to consider here. First is the "fixed" layout, which basically means that all of the elements on your page or window will remain of the same size, irrespective of the screen resolution or other form factor. Then, there is "fluid" layout, which is good for enabling your content to adjust in a size that is appropriate for the current screen resolution, page dimensions, or generally the form factor that your users will be using.

I am going to show you how to create a simple example of fixed and fluid layout in Silverlight. After that, I will give you some guidance on when to use fixed and when to use fluid layouts.

Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears select Silverlight and then Silverlight Application, make sure that Language is set to C# and Version is 4.0. At the end hit OK.



Some recipes in this chapter are dealing with WPF, not with Silverlight. However, ideas, methods, approaches and user experience guidelines are applicable to both technologies.

Now we will see how to create a fluid layout design.

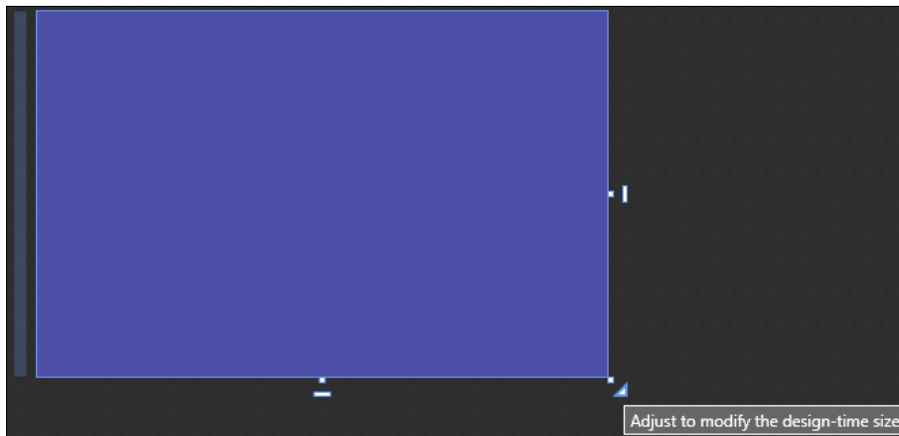
How to do it...

We will create two examples: the first one will demonstrate fluid layouts and the second one, fixed layouts.

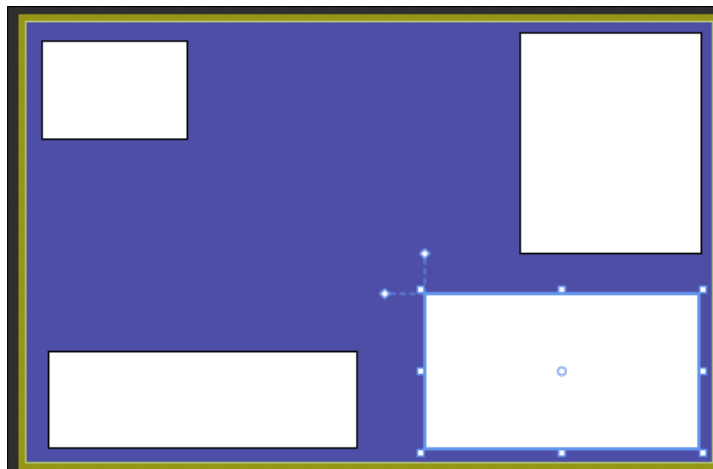
1. After you have created your new project, under the **Objects and Timeline** pane, you will see **UserControl** and **LayoutRoot**. **LayoutRoot** is a grid control hosted in **UserControl**.
2. Click on **UserControl** and change its height and width to **Auto**. To do that, go to **Properties | Layout** and change these properties. You will notice that they are set to **640** and **480** by default. Click on the little cross-arrows to set them to **Auto**.
3. Now, let's change the background color of **LayoutRoot**. Click on it in the **Objects and Timeline** pane, then go to **Properties** pane, navigate to the **Brushes** section, click on **Background**, select **Solid color brush** (second icon in the row), and select any color.



4. Press *F5* or go to **Project | Test Solution**.
5. Your default web browser will start and you will see that the entire surface is covered in the color that you have previously set for your **UserControl**. Try resizing the browser and you will see that the surface is still completely covered with the selected color.
6. Close your browser and return to Blend.
7. Now we will add some objects on top of **LayoutGrid** and explore their behavior.
8. Change the design-time size of your **UserControl** by clicking and dragging handlers. Be sure to select **UserControl** before you start resizing. Note that this is changing only the design-time dimensions; run-time dimensions are still set to **Auto**.



9. On the toolbox (usually a left-aligned, vertical stripe with a number of different controls and tools), locate and select the **Rectangle** tool. You can also select it by pressing the *M* key. Draw several rectangles on top of **LayoutRoot**.



10. Again, press *F5* to start your project.
11. Play with your web browser now. Try changing the height and width and notice the behavior of the rectangles; they will also change their dimensions. You don't need to think about the mechanics behind this behavior right now.
12. Close your browser and return to Blend again.

How it works...

How does it really work? The first step was changing the width and height properties of our **UserControl** to **Auto**. By doing so, we have allowed it to stretch automatically and fill in all available space. Simple as that! As our **LayoutRoot** is Grid control and its height and width properties have been set to **Auto** as well, **LayoutRoot** has also filled in all available space. We have just changed its color to make it more distinguishable. That is why our browser has been filled with **LayoutRoot**.

After that, we added several rectangles to our **LayoutRoot**. And here is where some interesting behaviors start. Depending on the position of rectangles, some of them have been aligned horizontally or vertically to different sides of **LayoutGrid**. You can easily check that by clicking on them and then looking at **HorizontalAlignment** and **VerticalAlignment** under the **Layout** section within the **Properties** pane. Depending on that alignment, rectangles have been resizing when you have resized your browser. This was a very basic illustration of a liquid type of layout. We implement such a layout by using the grid control. Next, we will see how to make a fixed layout design.

How to do it...

The steps are the same as those of the previous example—the creation of liquid fluid layout. I will assume that you have followed those directions and will take it from there.

So, the main difference will be that instead of using **Grid** for **LayoutRoot**, we will use **Canvas**.

1. Right-click on **LayoutRoot** in the **Objects and Timeline** pane and from the menu, select **Change Layout Type | Canvas**.
2. Hit *F5* to test your project.
3. Try resizing your browser again. Notice that the rectangles are keeping their positions as well as their dimensions intact. No matter what you do with your web browser size, they will be the same.

How it works...

What happened after we changed our **Grid** control to **Canvas** type of control in the second example? All our rectangles have retained their size and positions, no matter what we have done with browser. The reason lies in the fact that **Canvas** control used the so-called "absolute positioning" and no layout policy is being applied to its child elements (rectangles in our case). You can literally consider it as a blank canvas.

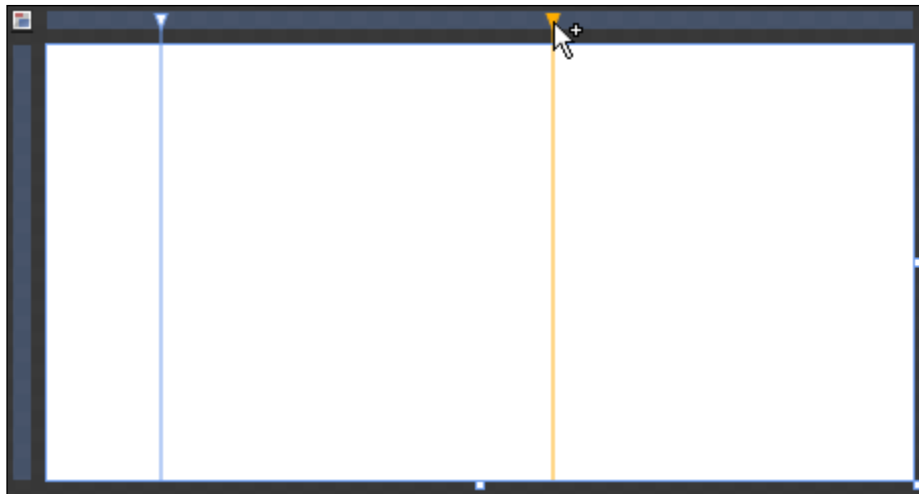
There's more...

To summarize, if you require maximum layout flexibility, you will use **Grid** control. It employs a number of different rules that can be applied to its child elements. For fixed layouts, which are not dependent on the screen size or resolutions, you will use **Canvas** control with absolute positioning.

More info about grid sizing, rows, and columns

The grid is a truly versatile control and it enables you to build very flexible and sometimes complex layouts.

A really important concept includes the possibility of dividing the grid into rows and columns, so that you get even more flexibility for your layouts. With grid control selected, position your mouse pointer over its border. You will notice that the mouse pointer will change its appearance and look like the following screenshot. You can click to add gridlines and define columns and rows in that way.



Now it is important to understand that the grid supports three ways of defining sizes for columns and rows—fixed, star, and auto.

Fixed sizing uses exact pixel values to define row or column dimensions and means that they will not resize. **Star sizing** is using relative dimensions. In fact, it just indicates that width should be relative to the other star-sized columns (or rows, for that matter).

Consider the following examples:

```
<ColumnDefinition Width="0.5*" />
<ColumnDefinition Width="0.5*" />
<ColumnDefinition Width="*" />
<ColumnDefinition Width="*" />
<ColumnDefinition Width="1000*" />
<ColumnDefinition Width="1000*" />
```

These three definitions will have the same column width. They will be identical with each column occupying the same amount of space.

Now, consider this sample:

```
<ColumnDefinition Width="1000*" />
<ColumnDefinition Width="2000*" />
```

Here, the second column will be exactly two times bigger than the first one—a ratio of 2:1.

Auto sizing means that the elements contained in a grid will also resize when the parent element resizes. It also means that the size of that column or row will depend on the size of the object that is located in that specific column or row.

In XAML code, those definitions can look like this:

```
<Grid x:Name="LayoutRoot">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="69" />
    <ColumnDefinition Width="0.52*" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="69" />
  </Grid.ColumnDefinitions>
</Grid>
```

If you omit the width definition, Blend will automatically use star sizing.

More detailed information about Grid and Canvas panel controls can be found in the Microsoft Expression Blend User Guide. Just hit *F1* to access it within your application.

When to use fixed and when to use fluid layouts

As you might have guessed already, there is no definite suggestion as to whether you should use only one of those layout types. Most likely, you will be using both of them. For example, message boxes or some dialogs will not benefit from resizing, but general application workspaces and web pages might want to look into using the most or all the space available to them in a meaningful manner. Also, if you need to preserve some fixed positions, you will use a fixed layout system.

However, there are some proven practices outlining the usage of fixed and fluid layouts.

Fixed layout works well for message boxes displaying information or errors to end users. They will not benefit significantly by adding the ability to resize them. Same goes for dialog boxes presenting users with progress on operations such as file copying or deleting. Keep their sizes fixed. Of course, you will ensure that they look good at different screen resolutions by testing them. You can even try to optimize content positioning, depending on the current screen resolution, but you should not allow users to resize those types of windows. If you consider web environments (you can apply the same principles here), all those modal windows, pop-up dialogs, and message boxes should be of a fixed size.

When it comes to fluid layouts, the situation is more complex and it is more challenging to give design guidelines. But generally, you should use liquid layout type in cases where your application and dialog boxes can benefit from more space available to them. If you are building a file browser, you should implement the "liquid layout" system and use all available space to show as many files and folders as possible, so that your users can select them without scrolling too much. The same goes for web pages—in general, you would want to use as much space as possible for your content. But you should be aware that there are limits. You can put your textual content on the web page in liquid layout and enable it to resize together with your web browser and use the extra space available to them. But from the usability point of view, lines of text that are too long will decrease readability dramatically. Today, screen sizes are often over 20 inches and the trend of increasing sizes will continue into the future for some time—does that mean that you should enable your text or controls to scale accordingly? Absolutely not! The general rule is that your line of text should not be longer than about 66 characters. The reason for that is the fact that the human eye struggles with reading lines longer than that. I recall seeing that suggestion in *The Elements of Typographic Style* written by Canadian typographer, Robert Bringhurst. If you apply some good fonts and think about the overall appearance, you can go up to 75 characters, but don't go over that.

What I like to do is define three sizes for my UI elements: preferred (which is basically default or optimal) size, minimum size, and maximum size. **Preferred size** is the size that the specific UI element will have on a given screen resolution. **Minimum size** is the size that will be the smallest size each element can scale to. For example, you can choose to set the minimum size of your window to 800 by 600 pixels and optimize all controls within it to fit into that. Also, by setting **maximum size** you can ensure that your form and controls will not stretch too much (as I suggested a moment ago when talking about text length). The great thing is that WPF and Silverlight also support these sizes and you can set them for each control you add to your UI.

Other suggestions for dealing with controls in liquid layouts include keeping navigation elements (in web pages) anchored to top and left margins. Controls such as text boxes, tables, grids, and list views should increase their lengths and possibly even widths but again, recall what I have said in the previous paragraph about too long text lines—same applies to all controls. If you are using background colors, patterns, or similar elements, they should fill the new space completely and that is the only case where you should use all space available.

It is also important to think about what will happen if your window or web page gets too small for its content. As I said earlier, set the minimum size for each and every control including container controls such as windows or pages. Optimize the layout for that case. If you don't do that, your controls might become inaccessible during run-time, aesthetic dimension will be also destroyed, and your application will become practically useless or extremely hard to use.

Most WPF and Silverlight controls will adjust nicely by adding scrollbars to help adapt to these changes, so that will spare you from implementing all that logic by yourself.

Don't be afraid to put some constraints on liquid layouts, sometimes too much flexibility will harm your user's experience though your intentions were quite the opposite. I am a firm believer that by limiting certain aspects of UI flexibility we are actually helping our users and building better experiences.

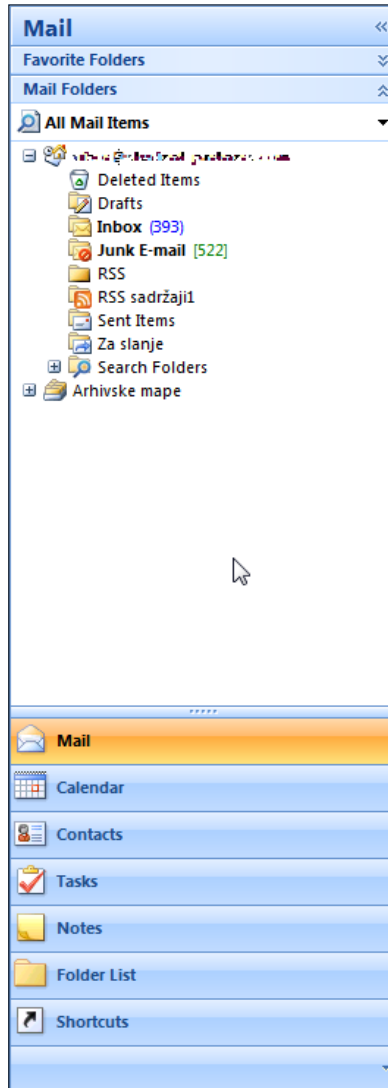
See also

- ▶ *Control docking with DockPanel*

Creating a navigation pane from scratch

If you have ever used Microsoft Outlook 2007 or even earlier versions (and the chances for that are pretty high), then you are already familiar with the concept of the navigation pane.

Navigation pane consists of two major parts—content and buttons. Depending on the selected button, the content part will be changed. The **content part** can host a number of different controls, such as tree view.



In this recipe, I will show you how easy it is to use **tab control** from WPF and create a basic navigation pane control that resembles the look and feel of the Outlook's navigation pane.

However, be sure to understand that this is not a replacement for third-party navigation pane controls, which offer much better and richer functionalities; but of course, all those goodies come at some price.

So, if you need a really simple navigation pane, let's say for your prototyping efforts, then you can use this recipe and create one. If you are creating a commercial, line-of-business application, then you should consider getting one of those fully blown controls available on the market.

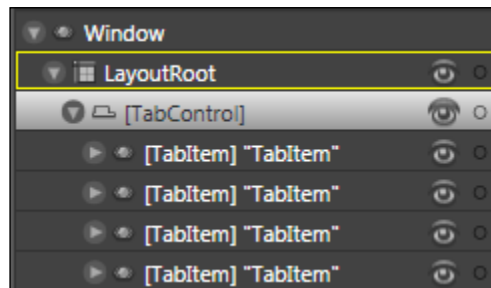
That being said, this recipe is useful even if you go and buy third-party controls because general usage ideas and guidelines are equally applicable. And, it does seem surprising at the moment, but basic guidelines used for tabs may be applicable to a navigation pane pattern too.

Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears, select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK.

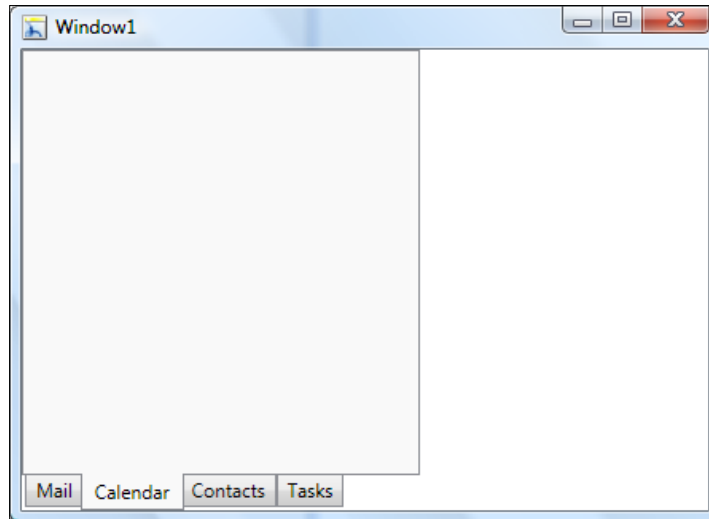
How to do it...

1. After your new WPF project has been created, you should notice that under the **Objects and Timeline** pane, **Window** and **LayoutRoot** grid controls are visible.
2. Go to **Asset Library** and draw a **TabControl** on top of **LayoutRoot**. With **TabControl** selected, go to the **Properties** pane and set all margin values to 0 and **Width** to 250. This will align the tab control to the left-hand side and stretch it from the top to the bottom of your window, giving it a specified width.
3. Right-click on **TabControl** under the **Objects and Timeline** pane and from the drop-down menu, click on **Add TabItem**. Repeat this once again so that you end up with four tab item controls added under the **TabControl** parent. Your visual tree should look somewhat similar to the following screenshot:



4. Now select each **TabItem** and under the **Properties** pane, locate the **Common Properties** section and **Header** property. For each tab item, type the header value. You can go with the values found in Outlook 2007, for example, **Mail**, **Calendar**, **Contacts**, and **Tasks**.

5. Select **TabControl** and under the **Common Properties** section, change **TabStripPlacement** to **Bottom**. If you now go and press *F5* to test your application, you will notice that tabs are located at the bottom. We are still fairly far away from the complete navigation pane found in Outlook but you can start to see where we are heading.



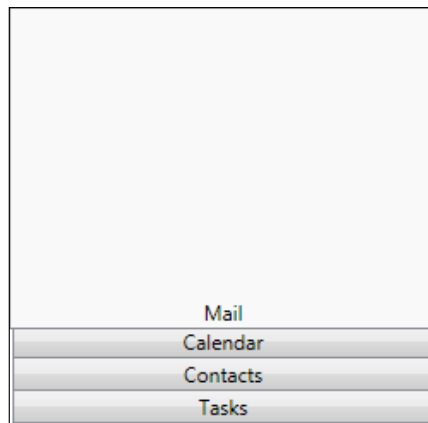
6. The next step is modifying the control template of the tab control. Right-click on **TabControl** and from the drop-down menu, click on **Edit Control Parts (Template) | Edit a Copy...**—a new dialog **Create Style Resource** will appear. Keep the default **Name (Key)**. Notice the possibility to define this template on an application- or document-wide scope. Also, you can create a separate resource dictionary and use that file for all specific styles and templates. I will choose that approach this time and will use the same file for all other styles that I am going to create in this recipe. So, click on **New...** and accept the defaults from the new dialog. Now hit **OK** on the first dialog, and just make sure that under the **Define in** section, the resource dictionary is selected with the name of the RD file that you have just created.
7. Now you are in *template editing* mode. If you now take a look at the **Objects and Timeline** pane, you will notice that the scope has been set to **TabControlStyle1**—our **TabControl** template. This enables us to modify the look of our controls without making an impact and destroying their functionality.
8. Click on **HeaderPanel** and notice that **HeaderPanel** is of the **TabPanel** type. You can see that at the very top under the **Properties** pane. We need to change **TabPanel** to **StackPanel**. To do that, press *F11* to switch to "XAML" or "Split" view, which will enable you to edit the XAML code. Locate the following line:

```
<TabPanel Margin="2,2,2,0" x:Name="HeaderPanel" Grid.Column="0"
    Grid.Row="0" IsItemsHost="true" Panel.ZIndex="1"
    KeyboardNavigation.TabIndex="1"/>
```

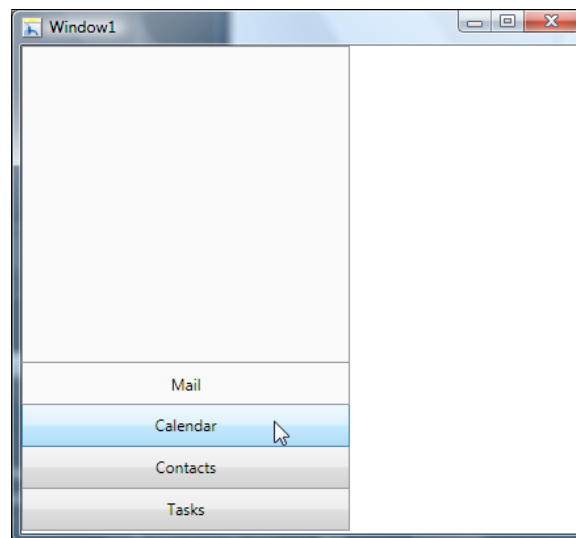
And change it to:

```
<StackPanel Margin="2,2,2,0" x:Name="HeaderPanel" Grid.Column="0"
  Grid.Row="0" IsItemsHost="true" Panel.ZIndex="1"
  KeyboardNavigation.TabIndex="1"/>
```

9. Now save all your files and go to the `Window1.xaml` file (basically, your main file).
10. Press `F5` to test your application now and notice that the tabs are now stacked pretty much in the same way as they are in Outlook 2007: one above the other.



11. Let's go and make more changes. Select all tab item controls (`CTRL + click`) and then change their height to **32**. Also, change the left and right margin values to **-2**. Now, your navigation pane is starting to look like a real one from Outlook.



12. Take a look at the *There's more...* section (of this part) for further possibilities of customization and changes.

How it works...

The key for understanding how this pattern and recipe works is pretty simple. The navigation pane control seen in Outlook is nothing more than **TabControl** with tabs stacked vertically, and that's the whole truth. We have added simple **TabControl** to our artboard and decided to change its control template.

Once we were there, everything became pretty simple and straightforward. First step was to change **TabPanel** to **StackPanel**. Literally, we just changed one thing in the whole of XAML code and our tabs have stacked vertically one above other, which is the basic characteristic of **StackPanel** as a control.

After that everything else was pretty much just cosmetics treatment. We've changed the tab item's height to 32 and we have made some changes to the **Margin** property in order to align and stack each and every tab item nicely.

Again, I find it necessary to repeat that this is not a replacement for third-party navigation pane controls that are offering better and richer functionalities.

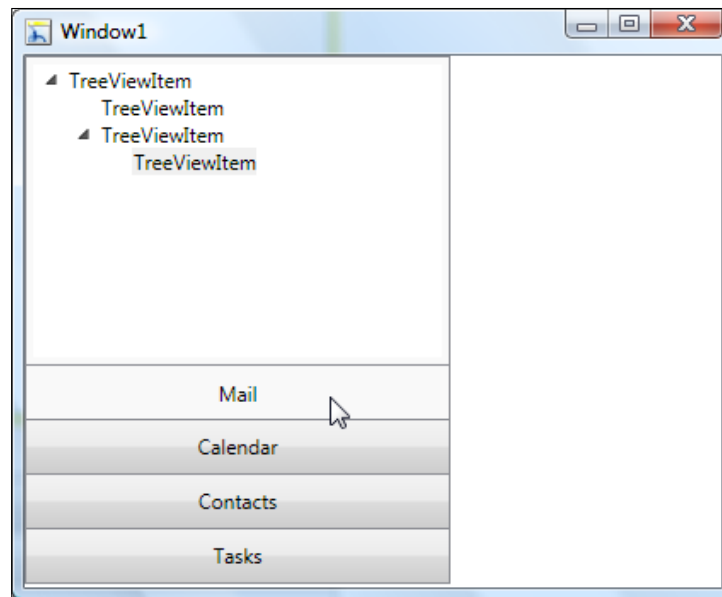
There's more...

In this section, I will show you how to add more functionalities and even further customize your navigation pane.

Hosting content into specific tabs

Remember that we have changed the control template of our **TabControl** but that the functionality has been preserved completely. As a consequence, the process of hosting and adding different content to tabs is the same as it is for regular tabs.

1. Let's continue from the last step in our recipe. If you expand any **TabItem** under the **Objects and Timeline** pane, you will notice that it is comprised of two major parts—**Header** and **Grid**. Click on **Grid** to make it active.
2. Now you can add any control from **Asset Library** or any other object like you would on any other grid-like control. Try experimenting by adding different controls. In the code sample that you can download from this book's website, I've added a simple **TreeView** control under the **Mail** tab.

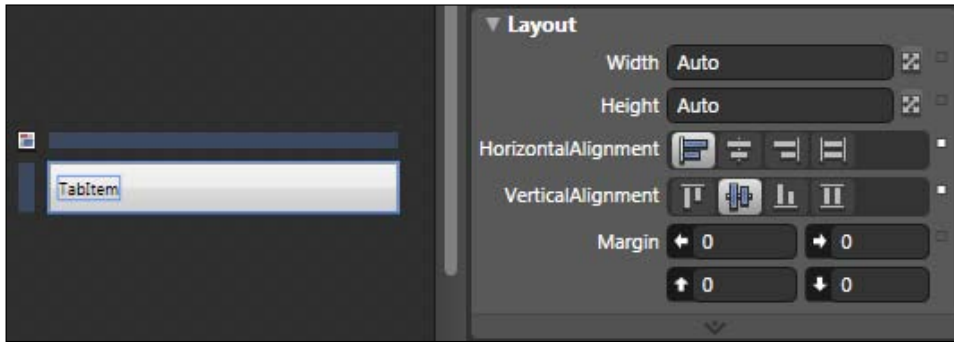


How to align header text to left

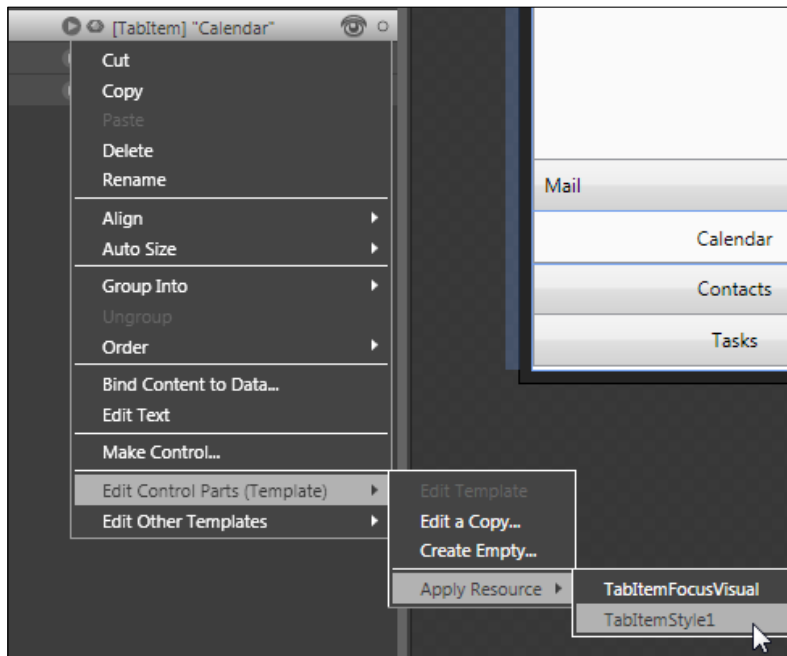
Again, I will take it from the last step. So, our next challenge is to left-align labels such as **Mail**, **Calendar**, and others. To do this, we will need to edit a control template for the **TabItem** control.

1. Right-click on any tab item, let's say the first one with the **Mail** label. From the drop-down menu, click on **Edit Control Parts (Template) | Edit a Copy...**, and the **Create Style Resource** dialog will appear. Accept the suggested **Name (Key)** and under the **Define in** section, select **Resource dictionary**. You'll remember that we defined our resource dictionary earlier as a single location where we will keep all our templates and styles. When ready, click on **OK**.
2. Now we are able to edit the template for our tab item. If you take a look at the **Objects and Timeline** pane, you will see the visual tree for our **TabItemStyle1**. Expand all nodes until you can see an object called **Content** (which is of **ContentPresenter** type). Click on it and go to the **Properties** pane. Locate the **Layout** section and you will see the **HorizontalAlignment** and **VerticalAlignment** properties. They are surrounded with a yellow border, and from them, you can see a little, yellow square, which indicates that these values have been bounded to the control template.
3. We want to set our own bindings. Click on the little square and from the drop-down menu click on **Reset**. Do this for both the **HorizontalAlignment** and **VerticalAlignment** properties.

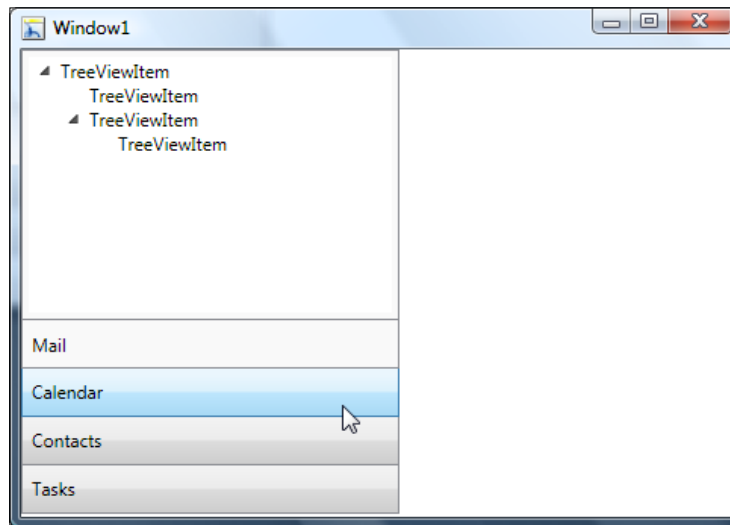
- Now, set **Left** as the value for **HorizontalAlignment** and **Center** for the **VerticalAlignment** property.



- If you now go and hit *F5*, you will notice that the **Mail** label has been left-aligned, but that is not the case for the rest of the labels. The reason for this is the fact that in previous steps we have edited a copy of the control template and applied it to only the first tab item.
- In order to get all labels left-aligned, go to your `Window1.xaml` file and under the **Objects and Timeline** pane, right-click on the next tab item. From the drop-down menu, select **Edit Control Parts (Template) | Apply Resource | TabItemStyle1**.



- Repeat the same procedure for the rest of the tab item controls and then hit *F5* to test your application. Now all the labels are left-aligned.



- Note that you can also add other controls in our **TabItem** style, for example, images.

When to use navigation pane?

As its name implies, the navigation pane is a control or pattern (if you like) that is focused on navigation. But what type of navigation? Is it the same as journal navigation?

First and foremost, a pattern or control (if you prefer calling it that) is never used alone. It is (or should I say, it must be) used as part of a wider navigation concept. While a regular tab control with specific tabs is useful for property pages, dialogs, and different types of content organization, the navigation pane is great for application-specific navigation.

In the real world, it means the following—you will position the navigation pane on the left-hand side of your application. When you click on specific buttons (tabs that is), they will expose content in their upper "container" part. In our recipe, that was the **TreeView** control added on the **Mail** tab.

The most relevant and interesting things happen when users click on different items within the container part of a navigation pane. Then, application-wise navigation occurs. If you have challenges picturing this, go to Outlook and simply try clicking on **Mail**, and after that on the **Inbox**, **Outbox**, or **Sent Items** folders. You will notice that the right-hand part of your application changes. Virtually the same pattern is available in Microsoft Dynamics products such as NAV or AX, and many others as well.

I've mentioned earlier that most guidelines and suggestions that I've outlined for tabs hold true for the navigation pane as well. However, I will reiterate some of them and also point out some differences right here:

- ▶ Tabs (or buttons) such as **Mail**, **Calendar**, and so on must be linear in their structure, which means that there is no hierarchy in their organization—**Mail** is not a parent of **Calendar** and so on. Simply put, each and every tab or button must be mutually independent. But you are free to host hierarchy-based controls such as a "tree view" within your navigation pane.
- ▶ The navigation pane is for navigation. It's not a regular tab control used for organizing your content, controls, and properties. It's not a wizard too: don't ever use it as a control that should guide users through wizard-like processes.
- ▶ The first tab or button should be the one that is most likely to be most used and it should be selected by default. Again, using Outlook as a sample, the first tab is **Mail**—the one that is being used the most.
- ▶ Feel free to use icons. That was not something that I've been encouraging in the case of tabs, but here, you have enough space and you don't have to be afraid that you will make too much of a visual clutter. Sure, that holds true only if you use nice, clear, recognizable, and understandable icons.
- ▶ Don't use any kind of terminating or similar type of button within the container part of a navigation pane; no **OK**, **Cancel**, or **Apply** buttons here. It's called the "navigation pane" and it should do just that—help your users to navigate to or expose different parts of your application and help them get their jobs done quickly.
- ▶ Have I mentioned that in your real-life applications, you should use full-blown, third-party navigation pane controls and not this? We've created a really basic one here? This was a nice learning example and all of the suggestions stated here are applicable to real-world commercial navigation pane controls. And yes, I'm not really able to give you any suggestions on which control you should use, but there are numerous vendors offering them and I'm sure you will find them easily on the Internet.

See also

- ▶ *Tabs*

Window management and positioning

Often overlooked or even completely ignored, window management presents the basic of any window-based desktop application. Though this recipe will be focused on WPF and desktop applications, today, **window management** plays an important role in web environments as well where different types of modal, dialog, and message box windows are appearing. As a consequence, most ideas and general guidelines presented here might be quite useful for web solutions as well.

This recipe will cover several basic ideas and approaches regarding window placement, their positions, and size.

As there are several concepts being described here, I have, when it seemed appropriate to do so, created smaller sub-recipes that are building on previous ones.

WPF provides a fairly rich window management model but it is upon developers and designers to come up with the right usage and interactions involving windows. This recipe aims to help you with that challenge.

Getting ready

Before we proceed any further, it is a good idea to define several concepts that we will be using and relying on during this recipe. You can think of this as a small dictionary and it is compatible with Microsoft UX guidelines for Windows Vista and Windows 7 operating systems.

- ▶ **Top-level or Primary window:** It has no owner window, meaning that this is the one that is displayed on the taskbar and in most cases, can be considered to be the main application window.
- ▶ **Owned or Secondary window:** It has an owner window and as a general rule, it is not displayed on the taskbar.
- ▶ **User-initiated window:** It is always being displayed as a direct result of a user's actions. Action can be clicking on some buttons, commands, menu items, and so on. Also, there are **program-initiated** windows—ones that are initiated by an application itself without user's action—and **system-initiated** windows—ones that are initiated by the underlying operating system itself.
- ▶ **Contextual window:** It is type of a user-initiated window but with a very strong relationship to the UI object, from which it was invoked and launched. Context is extremely important here and positioning often plays a very important role (it will be covered in this recipe).

So, let's start. We are going to explore window management and that's why I will be using WPF. Start your Expression Blend 4 and then select New project... From the dialog that appears select WPF and then WPF Application, make sure that Language is set to C# and Version is 4.0. At the end hit OK.

Title bar controls and window borders - How to do it...

After your new WPF project has been created, you should notice under the **Objects and Timeline** pane that the **Window** and **LayoutRoot** grid controls are visible.

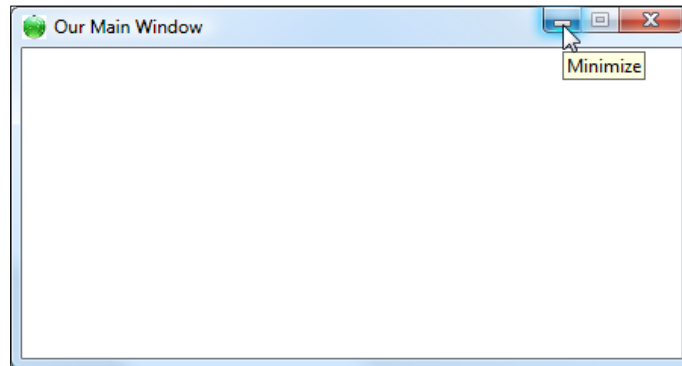
1. Before doing anything, hit *F5* and your application will start. Your window will look like this:



You will see the icon, title, minimize, maximize, and close buttons. If you click on the icon, the system menu will appear. Positioning the mouse cursor on the window borders will allow you to resize the window.

2. Click on **Window | Properties** and locate the **Common Properties** section.
Changing the icon: By changing the **Icon** property, you can change the look of the window icon. You can use a number of picture formats—ICO, PNG, BMP, JPG, TIFF, and GIF. Feel free to choose any picture that is available to you for testing purposes. The selected picture will be automatically added to your project.
Setting the ResizeMode: WPF supports several resize modes and you can select them from the **ResizeMode** drop-down list. You can pick anything between **NoResize**, **CanMinimize**, **CanResize** (which is the default choice), and **CanResizeWithGrip**. Select **CanMinimize**.
3. The **ShowInTaskbar** property enables you to choose whether your window will appear on the Windows taskbar. As in this case our window is the primary window, we will want it to appear on the taskbar so leave the **ShowInTaskbar** property checked.
4. By setting the **Title** property, you can set your window title. Set the **Title** property to **Our Main Window**.

5. Now locate the **Appearance** section and **WindowStyle** property. From the drop-down list, you can select one of the following options: **None**, **SingleBorderWindow**, **ThreeDBorderWindow**, and **ToolWindow**. Select **SingleBorderWindow**, which is the default choice.
6. Hit *F5* now and your window will appear. It should resemble the following screenshot:



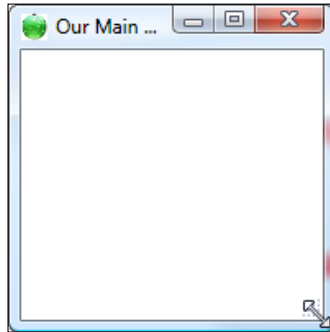
You can see that the icon, title, and control buttons have been affected. We decided to set the **ResizeMode** property to **CanMinimize** and users now can minimize the window, but there is no ability to change its size (positioning the mouse cursor over the borders does not enable us to resize it) or maximize it (the **Maximize** button is disabled). And as you have enabled the ability to display window in the taskbar, you can easily locate it there (like it is the case with most other Windows applications).

Window sizes and states - How to do it...

In this recipe we will deal with window sizes and different states.

1. With your window selected (under **Objects and Timeline** pane), go to **Properties** and locate the **Layout** section and **Height** and **Width** properties.
2. Set the **Width** to **250** and **Height** to **480**. Values are in pixels and they will define the size of your window during the runtime.
3. Now, click on the little arrow pointing downwards (**Show advanced properties**) and some more properties under the **Layout** section will be exposed. Locate **MinWidth**, **MinHeight**, **MaxWidth**, and **MaxHeight**. You can enter your values here and limit the window's maximum and minimum values for both height and width. By default, all minimum values are set to 0, and all maximum values are set to infinity. For the test, set **MinWidth** and **MinHeight** both to **200**.

- Now go to the **Common Properties** section and set the **ResizeMode** to **CanResizeWithGrip**. We want to test the **MinWidth** and **MinHeight** effects but in order to do that we have to set **ResizeMode** to either **CanResize** or **CanResizeWithGrip**.
- Hit *F5* now to test your application. You will notice the resize grip in the lower-right corner. Try sizing your window; you will notice that you cannot resize it to be smaller than 200 by 200 pixels.



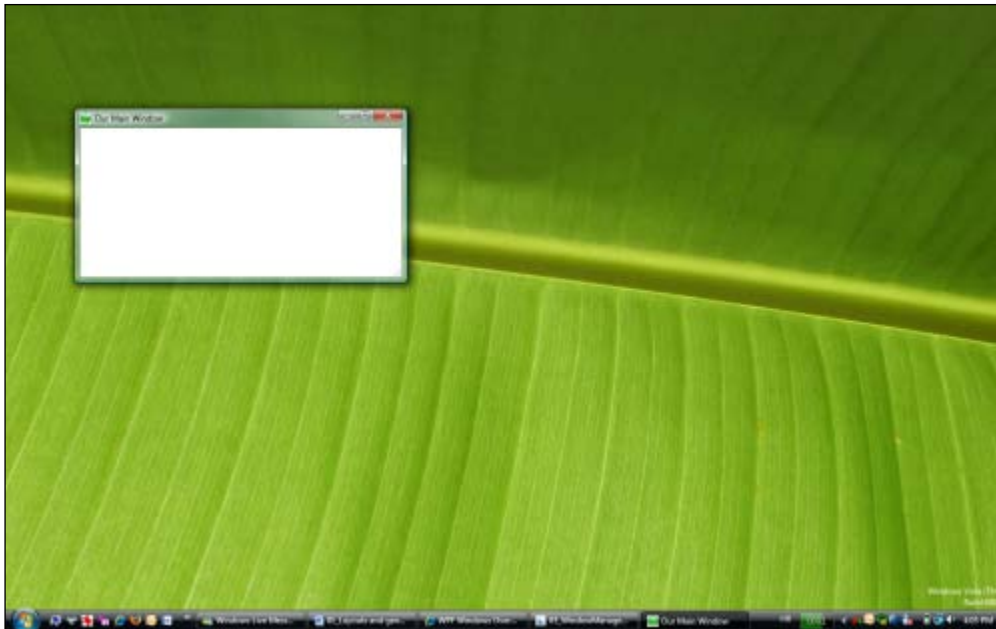
- Close the window and return to Blend.
- Under **Common Properties**, locate the **WindowState** property. Click on the drop-down list and you will see following choices: **Normal**, **Minimized**, and **Maximized**. Select **Maximized** and hit *F5* to start your application.
- Your application will now start **Maximized**. If you click on **Restore Down**, it will be restored to the dimensions you have previously set for its **Height** and **Width** properties.



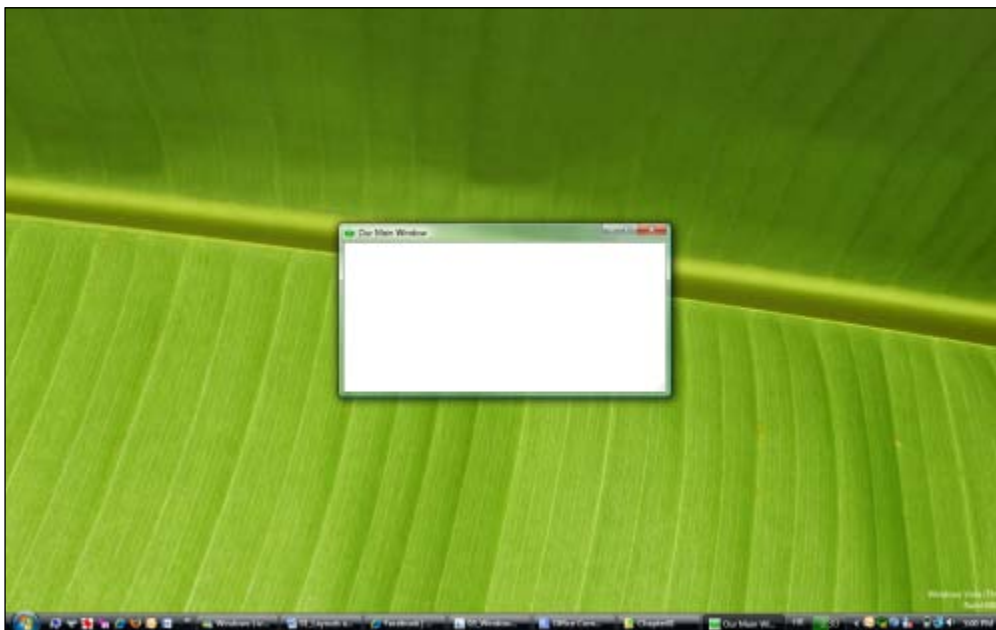
Window positioning - How to do it...

Let's investigate the window positioning options now. As I am going to continue this recipe from the previous one, I will just go to the **WindowState** property and set it back to **Normal** before I do anything else.

- Okay, now we are ready to continue. With your window selected, go to the **Layout** section and locate the **Left** and **Top** properties. Set **Left** to **100** and **Top** to **150**. Now when you press *F5*, your application will start and your window will be positioned 100 pixels from the left and 150 pixels from the top.



2. Close your application and return to Blend. Now under **Common Properties**, locate **WindowStartupLocation**. The drop-down list offers you several choices—**Manual**, **CenterScreen**, and **CenterOwner**. Select **CenterScreen** and hit *F5*.



3. In this case, your application will be automatically centered on your screen and the **Left** and **Top** properties that you have set before will be just ignored giving an advantage to the **CenterScreen** choice set for the **WindowStartupLocation** property.

Title bar controls and window borders- How it works...

Typical title bar controls are icon, title, and minimize, maximize, and close buttons. Expression Blend allows you to set and manipulate all of them by changing the number of properties described in this recipe.

While **Icon** and **Title** are really simple and understandable, let me invest some time and explain the different **ResizeMode** and **WindowStyle** properties.

ResizeMode

ResizeMode is a property that is used to describe window behaviors and abilities when resizing is in question. You can control how and if, at all, a user can resize your window. Your choice is reflected in different combinations of **Minimize**, **Maximize**, and **Close** buttons as well as on the ability to resize window by clicking and dragging its borders.

ResizeMode can be set to one of the following values: **NoResize**, **CanMinimize**, **CanResize**, and **CanResizeWithGrip**.

Let me describe them briefly.

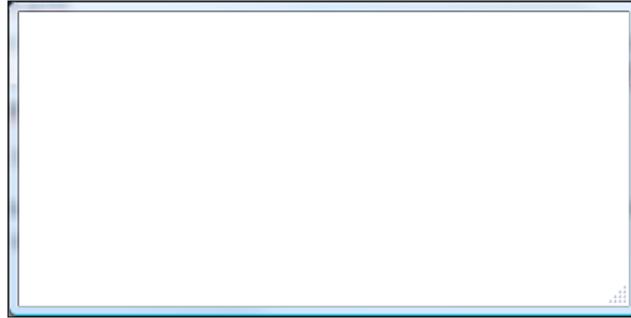
- ▶ **NoResize** will render your window as non-resizable and only a **Close** button will be presented in the title bar, allowing users to only close the current window. Positioning the mouse cursor over borders will not allow for any resizing.
- ▶ **CanMinimize** is the choice that we have taken in our recipe. With this choice, a window can be closed, minimized, and then restored after previous operations. Though both the **Minimize** and **Maximize** buttons are shown, the **Maximize** button is disabled so the user can click only on the **Minimize** or **Close** buttons to terminate the window. There is no option to resize the window.
- ▶ **CanResize** is the default choice. All buttons (**Minimize**, **Maximize**, and **Close**) are present and enabled. Users are able to resize a window by positioning the mouse cursor over its borders—it is the most flexible option available.
- ▶ **CanResizeWithGrip** allows for the same interaction as **CanResize** with the addition of the resize grip that appears in the bottom-right corner of the window.

WindowStyle

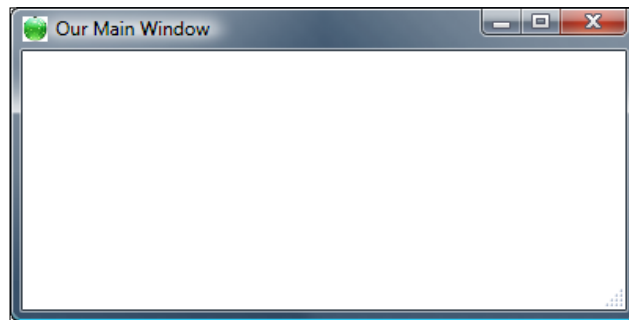
WindowStyle is a property in charge of the window's border appearance. There are four different possible styles at your disposal: **None**, **SingleBorderWindow**, **ThreeDBorderWindow**, and **ToolWindow**.

Let's describe them in more detail:

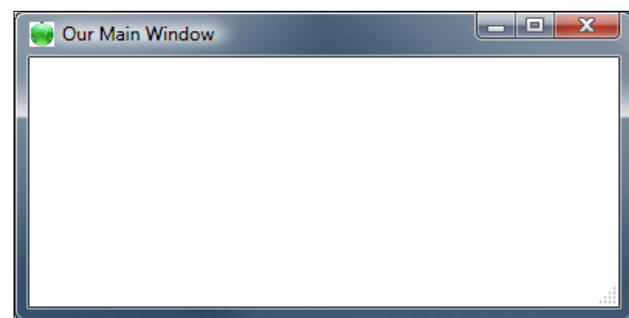
- ▶ **None**, as its name suggests, shows no title bar and border. All you can see is a simple client area.



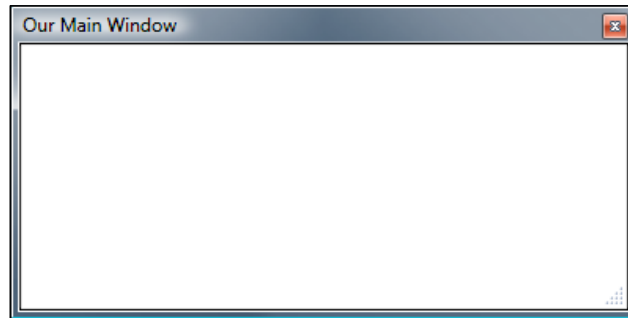
- ▶ **SingleBorderWindow** is the default choice: it shows a window with its standard controls and has a simple, single border.



- ▶ **ThreeDBorderWindow** is same as **SingleBorderWindow** but with a pseudo 3D border and it is much heavier in its appearance, as you can see.



- ▶ **ToolWindow** is a specific window type—a thinner border and a title bar with only a **Close** button available. However, it is resizable. It is often used as a secondary window for applications with a number of tools and options exposed in them (such as palettes and tools in **Paint.NET**).



Window sizes and states - How it works...

Blend allows for a really great number of options related to window sizes and their management.

The easiest and simplest way to set up your window's size is to set its **Height** and **Width** properties. However, you can use even more advanced properties such as **MinWidth**, **MinHeight**, **MaxWidth**, and **MaxHeight**. You can change those properties and limit the window's maximum and minimum values for both height and width. By default, all minimum values are set to 0, and all max values are set to infinity.

WindowState

By setting the **WindowState** property, you can control the appearance of the window in one of the three possible states: normal, minimized, and maximized.

- ▶ **Normal state** is the default state. In this state a window can be moved and resized, if it is resizable.
- ▶ **Minimized state** means that the window is collapsed to its taskbar button (in the default case, when **ShowInTaskbar** is set to **True**). If it is set to **False**, the window will collapse to its minimum possible size and place itself in the bottom-left corner of the desktop. It cannot be resized by using a resize grip or by dragging the border although it can be dragged around your desktop.
- ▶ **Maximized state** means that the window will be expanded to the maximum size it can take. It cannot be resized by using a resize grip or by dragging the border.

Window positioning - How it works...

There are two main ways to go about window positioning. The first one includes setting the **Left** and **Top** property, which enables you to precisely position your window on the screen, while the other approach requires setting the **WindowStartupLocation** property.

WindowStartupLocation

The **WindowStartupLocation** property handles the initial location of your window. Every window's position can be described with the **Left** and **Top** property relative to desktop. You can also use one of the available options for this property: **Manual** (default), **CenterScreen**, and **CenterOwner**.

- ▶ **Manual** startup location means that the window will be displayed based on **Left** and **Top** property values. In case they have been omitted, the underlying operating system will position the window on a specific location.
- ▶ **CenterScreen** will position the window exactly at the center of the screen on which it was opened. It will ignore **Left** and **Top** properties in case they have been set previously.
- ▶ **CenterOwner** works in a manner similar to **CenterScreen** but instead of positioning the window on the center of the screen, it will consider the center of the window that opened it as its startup location.

There's more...

This section will show you some more ideas and approaches that you should consider and take into account while designing and using windows as objects in your solutions.

What is the minimum screen resolution you should be targeting?

This is one of the most challenging questions presented in this book, though it seems to be very simple. Today, on the market, you can find a huge variety of different monitors and supported resolutions—from small netbooks to huge, widescreen monitors. Obviously, there is not a single "works for all" resolution. However, when you are designing your application, you must answer this question.

At the time of writing, the minimum supported resolution for Windows OS is 800×600 pixels, though I can't really remember when was the last time I saw some desktop or notebook computer running on this resolution (netbooks might be an exception, for sure).

Taking all this into account, I would strongly suggest that all your fixed size windows do not exceed the 800×600 dimension and the rest of the windows, ones that can be resized, should be optimized for 1024×768 resolution. It is your responsibility to invest time and explore how resizable windows and their content (UI controls) will behave in 800×600 resolution.

Good designers will try to accommodate their application's design and layout for different resolutions so that in case of a higher resolution, your users can benefit from a bigger workspace. As always, it is a question of the right balance and compromise between possibilities, wishes, and limitations.

If you are one of the rare guys around who knows that your application will be used on exclusively higher resolutions, then you have a nice challenge of creating an application that can take the full advantage of additional screen space.

General window usage guidelines

As I've mentioned in previous paragraphs, all your fixed size windows should not exceed the 800×600 dimension and the rest of the windows, ones than can be resized, should be optimized for 1024×768 resolution. If your application has some critical parts and it is supposed to be used in a *safe mode* environment, you might need to lower the bar and design for 640×480 resolution, but honestly, such cases are really rare, and practically non-existent in typical consumer applications.

When you are testing your windows, use the following table:

DPI / Percentage	Resolution	Performs well?
96dpi – 100%	800 × 600	
120dpi – 125%	1024 × 768	
144dpi – 150%	1200 × 900	

Under the *Performs well?* column, insert Yes or No based on the following criteria:

- ▶ Are there any layout problems?
- ▶ Do you notice control clipping, text readability problems, or anything related?
- ▶ How do icons and bitmaps perform? Are they stretched? What about alignments?
- ▶ Can users access each and every command in all tested cases?

It is better to use larger initial window sizes and use the space effectively (your users will appreciate that). It's a much better solution than trying to fit everything into a small space.

Don't go over 66 characters for text elements.

Windows User Experience Interaction Guidelines suggest that "centering" the window means biasing vertical placement slightly towards the top of the monitor, and not placing the window exactly in the middle of the screen. The reason for this is that our eyes are naturally more biased towards the top of the screen. However, that difference is quite small—you can go for 45% from the top of the monitor or owner window and 55% from the bottom.

If the window that you are launching is contextual (remember, I've explained this term at the very beginning of this recipe), then you can go and display it near the object (button, let's say) that it was launched from. Take into account that you should place it out of the way so that the source object is not obscured; if possible, position it offset down and to the right.

If your window is being launched from the notification area or system tray, you should display it close to that area too.

If your window can be described as a process dialog (one that contains a progress bar—for example, file copy dialog), then you should place it in the lower-right corner of the active monitor, but not as close to the notification area as was the case with the windows launched from that area.

If your window is an owned (secondary) window, then you should initially display it centered on the top of the owner window (you can use the **WindowStartupLocation** property and set it to **CenterOwner**, or you can use the 55% : 45% rule for centering the window (as described in an earlier guideline).

See also

- ▶ *Journal navigation*
- ▶ *Fluid versus fixed layouts*

Wizards

If implemented correctly, wizards can really do some magic for your users and make them happier and satisfied. This recipe goes after desktop-based wizards. I will show you how to implement them using the WPF and provide you with some Microsoft Aero Wizard guidelines.

As was the case with the navigation pane pattern, a large number of different vendors are able to provide you with pre-built, fully capable desktop-based wizard frameworks implementing a number of rich functionalities.

I will show you how to build a simple wizard but you are encouraged to explore third-party options. However, the guidelines and suggestions I will be giving here are applicable to those third-party products, so do yourself and your users a favor and try to take the most out of these suggestions.

Getting ready

As I've already mentioned, we are going to build a desktop-based wizard in this recipe, and that's why I will be using WPF. Start your Expression Blend 4 and then select New project. From the dialog that appears select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK.

How to do it...

There are several page types that can be used in wizards. They are:

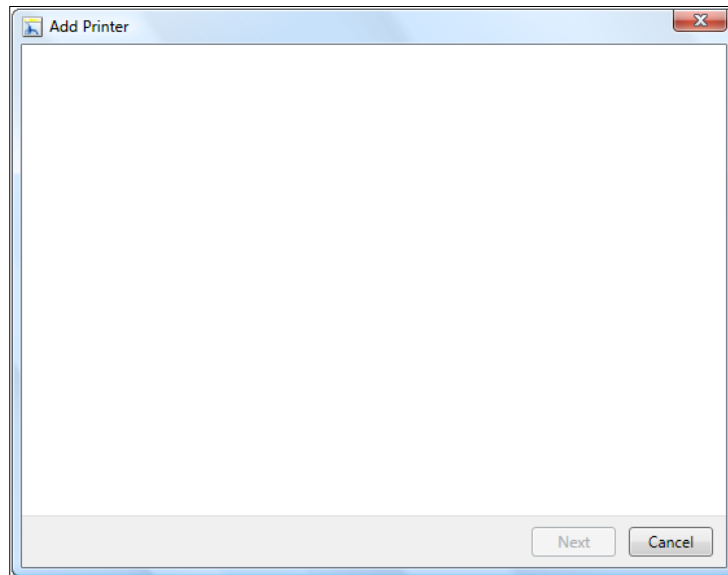
- ▶ Getting started page (optional)
- ▶ Choice page
- ▶ Commit page
- ▶ Progress page (optional)
- ▶ Follow-up page (optional)

Before you start building and implementing your wizard system, you should have a clear understanding of the task flow and user's actions. In this recipe, I will just mimic some typical wizard behavior but be sure to read the *How to use and implement wizards* section in this chapter.

OK, let's start with our building process—in this recipe, I will build a typical choice page. After your new WPF project has been created you should notice that under the **Objects and Timeline** pane, **Window** and **LayoutRoot** grid controls are visible.

1. Click on the window and then under the **Properties** pane, set its **Height** to **429** and **Width** to **549** pixels. Set its **Title** to **Add Printer**. Also under the **Common Properties** section, locate the **ResizeMode** property and set it to **NoResize**.
2. From the **Asset Library**, draw a **Border** control on top of **LayoutRoot**. Click on **Border** control and set the following under the **Properties** pane:
 - ❑ Set the **Width** to **Auto**, **Height** to **40**
 - ❑ Set all margin values to **0**
 - ❑ Set **HorizontalAlignment** to **Stretch**
 - ❑ Set **VerticalAlignment** to **Bottom**
3. Under the **Appearance** section, set the **BorderThickness** value for **Top** to be **1**.
4. Under the **Brushes** section set the following colors:
 - ❑ For **Background**: Solid color brush—**RGB (240,240,240)** or **HEX (#FFF0F0F0)**
 - ❑ For **BorderBrush**: Solid color brush—**RGB (223,223,223)** or **HEX (#FFDFDFDF)**
5. If you now hit *F5* and start your application, you will notice that **Border** control is docked to bottom, looks gray with a bit darker top border. This is the area that we will refer to as to the **command area**.

6. In the command area, we put at least one **Commit** button to commit to the task or proceed to the next step. We will add two buttons now—**Next** and **Cancel**. But as **Border** control can have only one child control, we need to add a **Grid** control and then draw our buttons on top of that **Grid** control.
7. With the **Border** control selected, from toolbox or **Asset Library** draw a grid control on top of **Border** controls itself. Under the **Properties** pane, set the grid's height and width to **Auto**, and all margin values to 0. That will stretch the grid and make it completely fill in the available space within **Border** control.
8. As grid is an extremely versatile and flexible control, we can now add our buttons on it. Select the grid and draw two buttons on it.
9. Call the first button **btnNext** and second one **btnCancel**. You can change their names by selecting them and then under the **Properties** pane (at the very top), you will find the **Name** property.
10. Select **btnCancel**, find the **Content** property, and set it to **Cancel**. Set **Width** to **65** and **Height** to **23**. Choose **Right** as the **HorizontalAlignment** and **Top** for **VerticalAlignment**. For margin values, use **0** for **Left** and **Bottom**, **10** for **Right**, and **8** for **Top**; this should position our **btnCancel** nicely.
11. Now, let's change some properties for **btnNext**. Set the **Content** property to **Next**. Also, set the **Width** to **65** and **Height** to **23**. Choose **Right** as **HorizontalAlignment**, and **Top** for **VerticalAlignment**. For margin values, use **0** for **Left** and **Bottom**, **85** for **Right**, and **8** for **Top**. And under the **Common Properties** section, locate the **IsEnabled** property and uncheck it.
12. Press **F5** now and your wizard should look close to the one in the following screenshot:



13. So far so good. It's good to point out at this stage that you will have to use code to manipulate when the **Next** and other buttons are enabled, which will depend on the current progress and the context of your wizard.
14. Now we need to add the main instructions. It's basically a text label that summarizes what to do in the current wizard page. We can use **Label** as a control for this. So, draw a label (you can get it from **Asset Library** or toolbox) on top of the **LayoutRoot** and set the following properties:
 - ❑ Set **Name** to **IblMainInstruction**
 - ❑ Set **Foreground color** to **RGB (0, 51,153)** or **HEX (#00003399)**
 - ❑ Set both **Width** and **Height** to **Auto**
 - ❑ Set **HorizontalAlignment** to **Left**, and **VerticalAlignment** to **Top**
 - ❑ For margin values, set **Left** to **32** and **Top** to **14**
 - ❑ Finally, set **Choose a local or network printer** for the **Content** property
15. These settings will always be the same for the main instructions for each and every wizard step. Only exception is the **Content** property, which must change appropriately.
16. As you remember, the page that we are currently working on is called "choice page". We have a main instruction, a command area, and now we will design the content area that hosts other controls and objects.
17. We will use regular buttons for this part although for the wizard pattern you should use command links.
18. We will use two buttons that will provide users with choices.
19. Draw two buttons; name the first button **btnLocalPrinter** and the second one **btnNetworkPrinter**.
20. Let's set the following properties for **btnLocalPrinter**:
 - ❑ Under **Brushes** section set **Background** to **No brush**
 - ❑ Set **Height** to **58**
 - ❑ Set **HorizontalAlignment** to **Stretch** and **VerticalAlignment** to **Top**
 - ❑ For margin values, set **50** for **Left**, **10** for **Right**, and **55** for **Top**
 - ❑ Set **HorizontalContentAlignment** to **Left** and **VerticalContentAlignment** to **Top** (You might need to click on **Show advanced properties** to display these properties.)
 - ❑ Under **Padding**, set **Left** to **20**, **Right** and **Bottom** to **1**, and **Top** to **6**.
 - ❑ Set **Content** to **Add a local printer**.

21. Now set the following properties for `btnNetworkPrinter`:

- ❑ Under the **Brushes** section set **Background** to **No brush**
- ❑ Set **Height** to **58**
- ❑ Set **HorizontalAlignment** to **Stretch** and **VerticalAlignment** to **Top**
- ❑ For margin values, set **50** for **Left**, **10** for **Right**, and **145** for **Top**
- ❑ Set **HorizontalContentAlignment** to **Left** and **VerticalContentAlignment** to **Top** (You might need to click on **Show advanced properties** to display these properties.)
- ❑ Under **Padding**, set **Left** to **20**, **Right** and **Bottom** to **1**, and **Top** to **6**.
- ❑ Set **Content** to **Add a network, wireless or Bluetooth printer**

This basically sets the stage for your further improvements and customizations. The next steps are defining other page types and adding interaction logic between them. I will describe those page types and typical considerations that you need to take into account and address when designing and implementing your wizards.

How it works...

It is of utmost importance for you to have a clear understanding of your wizards' purpose and flow. Don't go and start designing and implementing wizards before you have that. You need to come up with a logical flow, specific page designs, and then start designing. However, the page that you will be designing (in almost 100% of cases) will be a simple choice page. In the last recipe, I've outlined the basic idea of how to design a page and now we will dig a bit into some technical details.

First of all, we have set **ResizeMode** to **NoResize** in order to keep our windows' dimension fixed (429 by 549 pixels, in our example). As a consequence, we get only a **Close** button on our window. Guidelines do allow implementation of resizable windows for wizards but I am generally opposed to that approach and I will explain the reason for this under the *There's more...* section.

Okay, in the next step, we have used **Border** control for designing the command area of our choice page. As border, as a control, can have only a single child element, we have added grid as a child element of **Border** and then hosted **Next** and **Cancel** buttons within grid control (which can accept as many child elements as you need).

You must pay special attention when it comes to handling **Next**, **Cancel**, **Apply**, and other buttons or commands that will appear in this area; this includes taking care if they are enabled, disabled, visible at all times, and so on. I am outlining those guidelines later on, so be sure to check them out.

After we have designed the command area, we are ready for setting up the main instruction—the text label that is used for summarization of what to do in the current wizard window. It is really important to make it as understandable as possible so that users can understand what is being done in this specific window just by reading the main instruction.

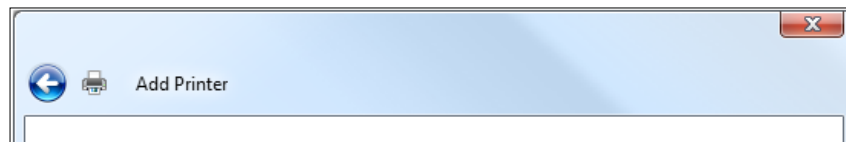
Next step was designing the content area—the main area of your wizard page where you usually place commands and where the most attention of your users will be focused on. We have added two "command links". (I am putting them under quotes for a reason: they are not real action or command links, what I have done basically is just changed some of the properties of regular buttons.

There's more...

As you've learned so far, wizards can be fairly complex and incorporate numerous page types given wizard's purpose. In this section, you will gain much more insights about those types and when and how to use and implement them. Again, this is WPF recipe but the general ideas, user experience considerations and design itself can be easily applied to Silverlight itself.

Brief overview of different wizard page types

In this recipe, I've guided you through the process of designing the very simple, even incomplete, choice page as a part of the wizard. According to the general UI guidelines, all wizard pages have a title bar, main instruction label(s), a content area, and a command area. In the previous recipe, we have designed all parts of our choice page except for the title bar. The fact is that the title bar for the so-called "Aero Wizards" looks pretty different from the one we have defined—it comes with an extended glass surface with the name of the wizard, the **Back** button in the upper-left corner, and a **Close** button with optional **Minimize** or **Maximize** buttons.



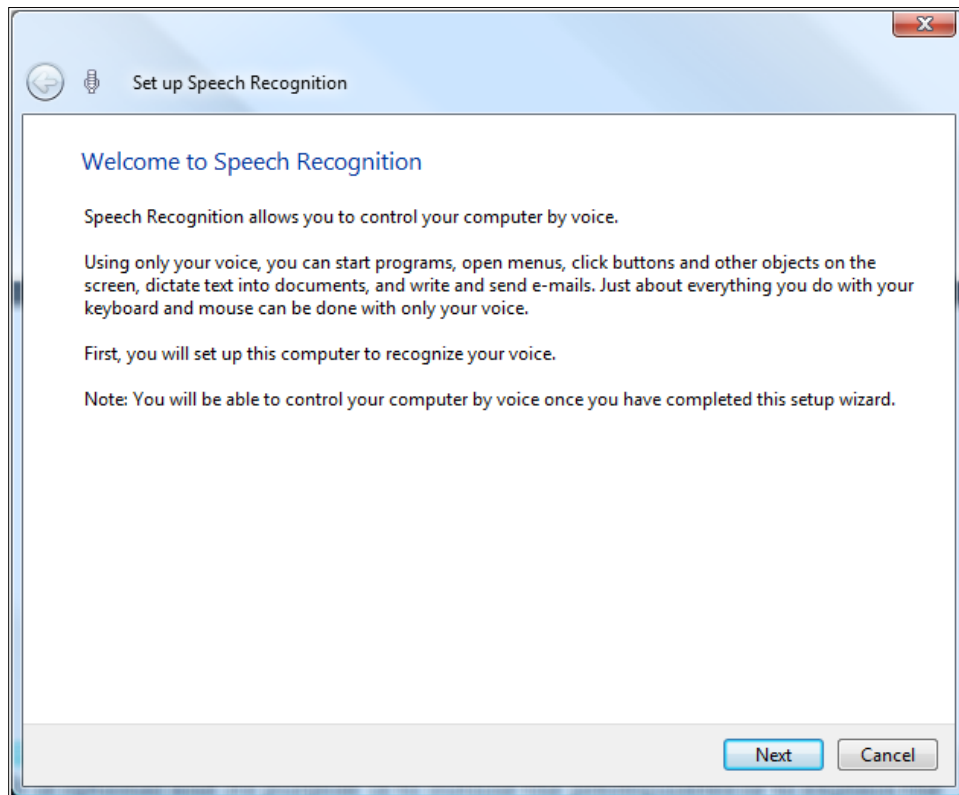
Anyway, let's go through the typical wizard page types.

There are several typical wizard page types:

- ▶ Getting started page (optional)
- ▶ Choice page
- ▶ Commit page
- ▶ Progress page and (optional)
- ▶ Follow-up page (optional)

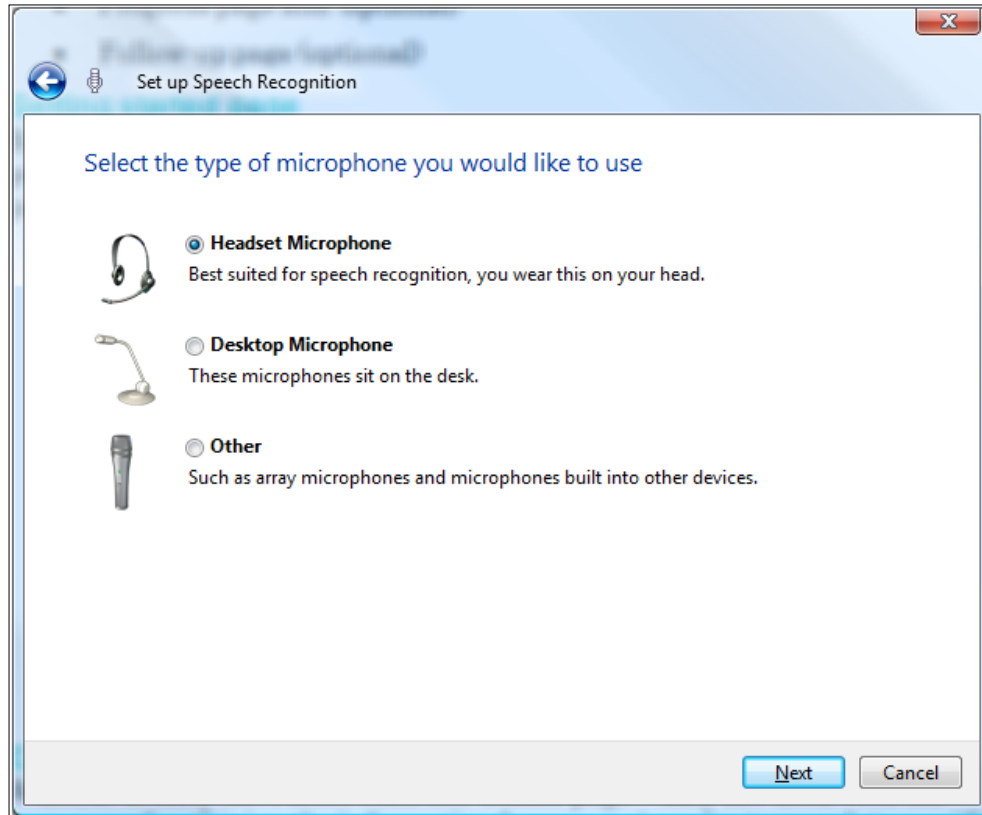
Getting started page

It is optional and its purpose is to outline the prerequisites or to explain the purpose of the wizard. But the general suggestion is not to use this page if all of the necessary information can be shown on the first choice page.



Choice page(s)

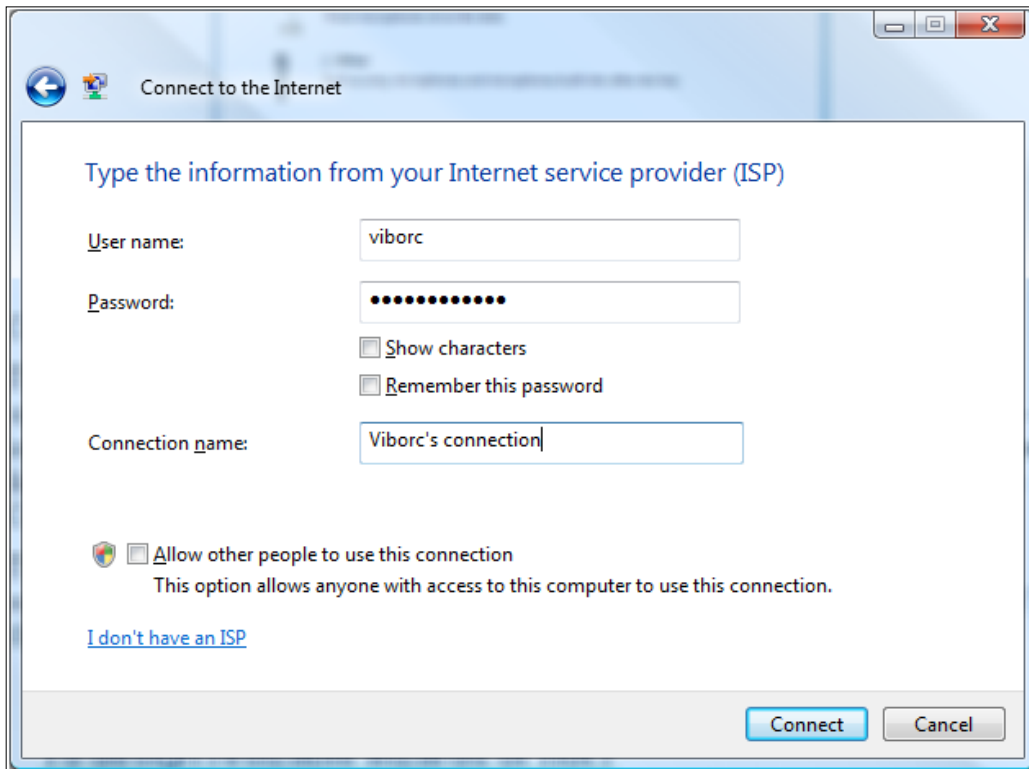
Wizards usually have more than one choice page; they are used with the purpose of gathering information from users in order to complete a specific task. A general suggestion is to use wizards if you know that users will be presented with more than two choice pages. If, however, you are dealing with one or two choice pages, you should consider using the regular dialogs and not the wizard because it is a pretty heavy UI pattern.



Commit page

This type of page looks quite similar to the regular choice page but there is one significant difference: after a user commits an action, there is no going back; in other words, action cannot be undone. That also means that the commit page does not have a **Next** button; it has buttons that clearly state commands such as **Connect** or so.

There is no common agreement as to whether there should be only one or more commit pages in a single wizard. However, personally I am a strong supporter of the single commit page idea.

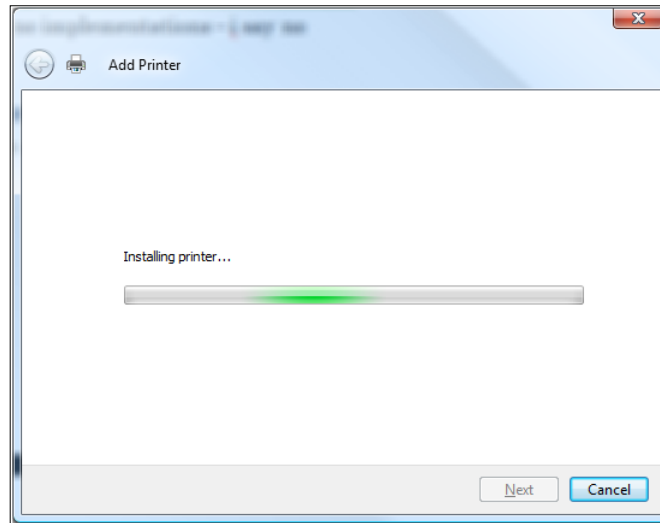


Progress page

If during the wizard there is going to be an operation that will take four or more seconds, you should use the "progress page" type. It is optional, in the sense that if your operations will be shorter than four seconds, you are fine to drop the progress page.

They are, in most cases, called after the **Commit** page and with a progress bar animation (or some other operation's progress indicator such as custom animation); they are good indicators of the operation's progress for the end users.

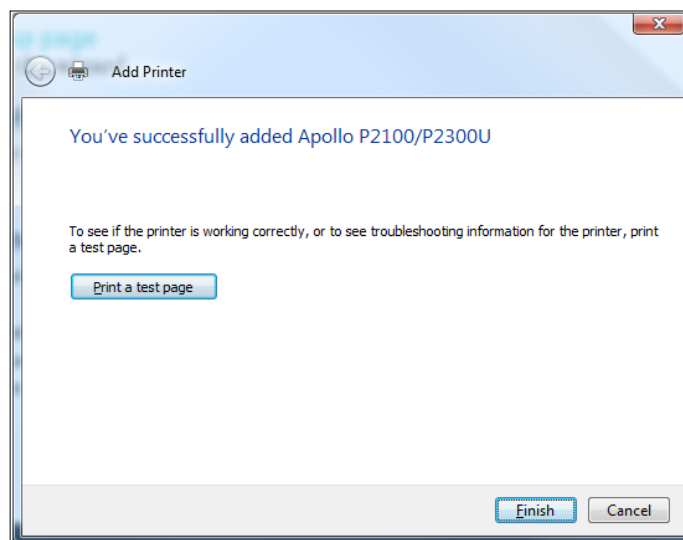
When an operation is done, the wizard should advance automatically to the next wizard page.



Follow-up page

This type of page is optional and is being used to provide users with final results or outcomes.

I am not a big fan of "Thank you" pages. Stick to the task and get to it; your users will appreciate that much more than **Thank you for installing this device**, especially if it has taken them longer and you are just prolonging that with a useless "Thank you" page.



To design resizable wizards or not

Although Microsoft says that usage of resizable wizards is fine under Aero Wizard specifications and guidelines, I would advise you to be careful when you decide to go for resizable wizards.

Let me make it clear that I am not completely against it but I would suggest and play safe and use **ResizeMode = NoResize**.

I guess that my strongest argument for this would be having buttons in the command area always positioned in the same location. Imagine the scenario where you can go through your wizard just by clicking on **Next** several times. It is much faster if the **Next** button is in that case positioned at the same location. Of course, in this case, I am pointing out the problem where you are changing your wizard's dimension on a page-by-page basis. However, having wizards resizable so that they can leverage extra space available might be good idea.

Again, I'd say play it safe; optimize them for minimum resolution supported under Windows Vista or Windows 7 (800 x 600), and don't go for some wild resizing behaviors.

When to use wizards

I bet you all remember older wizards; you can find them if you are still using Windows XP or with some third-party applications that are just breaking all known UI or UX guidelines for wizard applications under Windows Vista or Windows 7. Older wizards were based on a standard called Wizard97 and that "97" is not there for no reason, so we should think about the year we are currently in.

Anyways, new wizard standards are in place and some of the changes include more flexible page layout and text formatting, and removal of the really unnecessary Welcome or Congratulations or Finish pages. (I bet those are not being missed by anyone.)

Some other pillars are the prominent main instructions with the great idea of unifying the earlier heading and subheading. Also, implementation of command links is a nice way of enabling users to have immediate and generally more expressive choices, thus eliminating the usage of several UI controls such as radio buttons followed with a **Next** button.

Navigation within wizards is more aligned with the one that is usually found on the Web and within Windows Explorer. In our recipe, we have not implemented that kind of navigation but you can read about it in the *Journal navigation* recipe of this chapter.

You might have noticed that the **Back** button is not present in the command area, rather it is now located in its new standard location— upper-left corner. I've had several opportunities listening to people saying that in the beginning, this was a bit distracting to them but now they actually see the point; more focus is being given to commit choices.

Guidelines

- ▶ Don't go for wizards like there is no tomorrow. Wizards are considered as heavy user interface elements and should be used sparingly. They are used for multi-step tasks, especially if they are not frequently performed. You might consider some other alternatives to wizards—dialogs, task panes, and others.
- ▶ The **Next** button is used only when advancing to the next page but without commitment, which means that the **Back** button is always available and presented after the **Next** button. According to Microsoft's guidelines, advancing to the next wizard page is considered a commitment when its effect cannot be undone by clicking on **Back** or **Cancel** buttons. Sounds quite logical, doesn't it?
- ▶ Commit buttons are as specific as possible; for example, you should use captions such as **Print** or **Connect** instead of **Finish** or **Done**. Generic labels such as **Next** should not be used because they are suggesting the next step and not a `commit` command. However, there are two exceptions: **Finish** can be used when there is a collection of settings to be applied and if specific responses (**Get**, **Save**, and so on) are generic. The **Commit** button should always start with a verb, never a noun.
- ▶ Command links are here for choices, not commitments. They are unifying the collection of radio buttons and the **Next** button. So when you are using command links, hide or disable the **Next** button but leave the **Cancel** button. This was exactly the case in our recipe where we have used two command links.
- ▶ Wizard is a tricky term; never use "wizard" in wizard names. But it is fine to use "wizard" when referring to it as a specific UI element.
- ▶ User choices must be preserved. This means that when users make a specific selection, then clicks **Next**, and after that **Back**, previous selections should be preserved.
- ▶ Forget about "welcome", "get started", and similar types of pages. Make the first page fully functional whenever possible. There are some exceptions but resort to them sparingly. You can use "getting started" pages only in situations where there are some prerequisites that are necessary for successful completion of the wizard, when the purpose of the wizard may not be clearly understandable from the first choice page, and you don't have enough space on first choice page for additional explanation. In any of these exceptions, the main instruction text should be **Before you begin:** and never some version of welcome, let's get started, or anything like that.
- ▶ Forget about "Thank you" or "Congratulations" pages, too. Wizard's final results should be clear and apparent enough for the users that you can just close the wizard after the final **Commit** button. You can resort to "follow-up" pages if you think that there are some usual follow-up tasks that users are likely to do as a follow-up. However, in that case, avoid using familiar, simple, everyday tasks. Follow-up pages are necessary after progress pages to indicate task completion. But again, if the task is long running (I'd say longer than five minutes) and can be performed in the background, then just close the wizard on the "commit" page and resort to notifications (such as balloons) to give a final feedback to the end users.

- ▶ Commit pages are used to make it clear when users are committing to the task. As a general rule, the commit page is the last page of choices and it does not contain the **Next** button. Rather, the **Next** button is relabeled in a way that is described in the guideline about commit buttons (mentioned above). Sometimes, if the wizard was used for a really risky task or there is a significant doubt that the users have understood their selections, you might want to use a summary page and outline all of the selections and choices of the users, so that they can review them and act upon them.

See also

- ▶ *Action or command links*
- ▶ *Journal navigation*

Progressive disclosure—showing additional controls on demand

In some cases, your user interface needs to host a large number of different controls and present them to the end user. Instead of showing all the available controls at the same moment, you can take an approach where you will progressively disclose more controls on user's demand. This can save you some valuable space and, at the same, increase a user's productivity.

Now I will show you how to implement this UI pattern in a simple WPF application. As always, the same methods, ideas, and principles can be applied to your Silverlight application as well.

Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK.

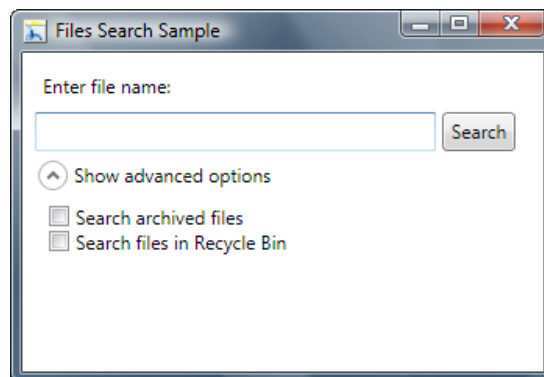
How to do it...

After your new WPF project has been created you should notice that under **Objects and Timeline** pane, **Window** and **LayoutRoot** grid controls are visible.

1. Click on **Window** and under the **Properties** pane change the following things:
 - Under the **Layout** section, set **Width** to **350** and **Height** to **240**
 - Under **Common properties**, set **Title** to **Files Search Sample**

We will create a simple file search application but without real functionality.

2. On the toolbox, click on **Asset Library | Label**. Draw a label on your application. With the **Label** control selected, under the **Properties** pane locate the **Content** property (it is under the **Common properties** section) and set its content to **Enter file name:**. You can set its height and width to **Auto** (under the **Layout** section).
3. Now add the **TextBox** control in the same way and position it under the previously added **Enter file name:** label. Go to the **Text** property under **Common properties** and set it to none.
4. Add the **Button** control, position it right next to the **TextBox** control, and set its **Content** property to **Search**.
5. Now you will go and add an **Expander** control. Select it from **Asset Library**, draw it, and position it below the **TextBox** control on your form. Set its height and width to **Auto**.
6. In the **Objects and Timeline** pane, click on the **Expander** control so that you can access the **Grid** control contained within it. Double-click on the **Grid** control to make it active. We will add our additional controls into this **Grid** control. Under the **Layout** section, set **Height** to **80**.
7. Once again, go to **Asset Library** and select the **CheckBox** control. Draw it onto the grid surface (be sure to make it this way, **CheckBox** must be the child element of the **Grid** control contained within **Expander** control). Repeat the procedure and add one more **CheckBox** control and position it below the previous **CheckBox** control.
8. Select the first **CheckBox** control and change its **Content** property to **Search archived files**, and set the **Content** property of the second **CheckBox** control to **Search files in Recycle Bin**.
9. Ensure that the **IsExpanded** property of the **Expander** is set to false (unchecked) under the **Common Properties** section. We want to keep **Expander** collapsed. Also, change **Expander Header content** to **Show advanced options**.
10. Hit **F5**. When your application starts, click on **Show advanced options** and your application should look like the following screenshot:



How it works...

Our sample application outlines the basic idea of progressive disclosure. We have created a simple file search interface where a user enters the filename and clicks on **Search** to search for files. That is the basic use case and it is available right after the user starts the application.

By clicking on **Show advanced options**, users are presented with two more options that can refine their search results. I have assumed that majority of users will want to use basic search options and just smaller number of them would be interested in searching for archived files or files in the Recycle Bin.

In this example, a progressive disclosure pattern has been implemented using the **Expander** control. The **Expander** control consists of two major parts—header and grid. The **Header** part contains a chevron and label (we changed its content to **Show advanced options**). The **Grid** part can be considered as a container in which you should put all of the controls that you want to disclose progressively to the end user. In this example, we have added two **CheckBox** controls with options to refine our search—they are not visible right away and therefore, they are not adding any visual clutter and distraction to users. If users do not want to use them, they can easily collapse the **Expander** control and hide those options.

There's more...

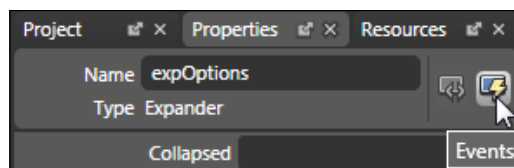
The first part of this recipe has introduced to the basic idea of progressive disclosure. What follows are more details and design and user experience considerations you should take into account when designing and implementing this pattern in your application.

Changing the expander control's header label

For the users to have a better understanding of the actions being performed and the changes being made when they click on chevron, it is highly recommended to change label from **Show advanced options** before clicking on the chevron to **Hide advanced options** after the user has clicked. Same goes for the other way around.

To achieve that, we need some code. As we have created our project as a WPF C# project, we will obviously write our code in C#.

1. Under the **Objects and Timeline** pane, click on the **Expander** control. We will now change its name so that we can access its properties through the code easily.
2. Under the **Properties** pane, type **expOptions** in the **Name** field.



3. We will want to execute different code when users collapse it and when it expands our **Expander** control. To do that, we will define event handlers for the **Collapsed** and **Expanded** events.
4. Click on the **Events** icon under the **Properties** pane and you will be presented with a number of events. Let's add our event handlers for **Collapsed** and **Expanded** events. To do that, you will just have to type in the name for those event handlers and once you press the *Enter* key, Visual Studio will start and allow you to add code logic. You can use **expOptions_Collapsed** and **expOptions_Expanded** as names for your events.
5. Add the following code:

```
private void expOptions_Collapsed(object sender, RoutedEventArgs e)
{
    this.expOptions.Header = "Show advanced options";
}

private void expOptions_Expanded(object sender, RoutedEventArgs e)
{
    this.expOptions.Header = "Hide advanced options";
}
```

Everything what we are doing here is changing the content within the header part of our **expOptions** control.

6. Press *F5* now and try clicking on the chevron and note how the label will change its content, making it easy for users to understand what has happened and what will their next action cause.

When to use progressive disclosure

Many developers tend to expose each and every command and feature that they have built into their solution. Some even go as far as to argue that by doing that they will actually show to the end user how powerful and feature-rich their application is. I'd be happy to say that this story was just my exaggeration but I have witnessed such situations too many times. So, we are facing a challenge on how to enable users to use our application and make it feature-rich, but at the same time not to make it too cluttered.

Guess what—users will not judge your application's power by the number of buttons, icons, and other UI elements that you have exposed to them. They will judge it, among other things, by how easy it was for them to get to the most needed options really quickly. That will make them feel that they are in control and will result in significant productivity gains.

So, what to do?

The first step is to identify the commands and controls that your users will use in most cases. Make a list of all use cases and assign a value to each of those use cases describing how likely users are to use that very option. You must ensure that users should be able to perform about 70-80% of the use cases easily, without having to look for hidden options within your UI. Of course, that percentage can scale depending on certain specifics, but practice has shown that the aforementioned range works very well. Okay, now you have a list with controls and use cases that your users will use in 70-80% of cases, and the rest will be used much less often. This is a great input and you can use it from this point on to define the look and feel of your UI.

Add those most commonly used controls on your windows or pages.

Remember, these are the ones that will be used frequently by your users, so ensure that they are visible and easily accessible. Now, create a separate section and add the rest of the controls to that section. Hide it and make it hidden by default to your end users. But, be sure to allow users to get to those options in a single click. You might consider using buttons with captions such as **More details...** or even chevrons (>>) as a part of the button or other control that is being used to show those hidden controls.



It is extremely important that users are able to hide and show sections with additional controls with a single click. If you show additional controls by clicking on **Show advanced options**, then be sure to enable users to get out from those advanced options by clicking on a button saying something like **Hide advanced options**.

As I said earlier, controls that are being placed in this "hidden section" will be used rather infrequently, but you must ensure that your users will be able to access them and leave them easily.

If you are using chevrons instead of buttons (and that is what I do personally because I feel that they are visually lighter than buttons, and besides that, buttons are usually associated with launching other windows or executing commands), you must take care of rearranging their "pointing direction" after a user clicks on them.

They should always point in the direction of the action being performed, which means the following: if the chevron is pointing down, when the user clicks on it, additional commands should appear below that chevron and now the chevron should point up.

That will always give a clear understanding to the user what will happen when they click on the chevron. I strongly suggest that you use labels that will reinforce users' understanding of the actions being performed.

Look	Behavior when user clicks
 Show advanced options	Section will expand, hidden options will be shown, and chevron will change its direction to up, and text will change to something like Hide advanced options
 Hide advanced options	Section will collapse hiding the previously exposed options and chevron will change its direction to down, and text will change to something like Show advanced options

See also

- ▶ *Responsive enabling*
- ▶ *Contextual fade-in/-out*
- ▶ *Progressive disclosure-showing additional controls on demand*

Control docking with DockPanel

Windows Presentation Foundation comes with **DockPanel**—a really versatile control allowing you to architect and create layout areas within which you can position and organize child elements.

When building your user interface, often you will want to ensure that specific controls and parts of your UI are always docked on the top, left, bottom, or right side. For example, the command bar area is usually docked at the top and the status bar is usually docked at the very bottom of your application.

A good practice is to use the **DockPanel** control as a root control and organize other panel and layout types of controls within it. Of course, you can always think about employing some different approaches and using controls such as **Grid**, but the **DockPanel** control brings you docking capability right out of the box.

In this recipe, I will show you some basic principles that you should be aware of when thinking about using the **DockPanel** control.

Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK.



Apart from WPF, DockPanel is available as a Silverlight control as well. In order to be able to use it, you need to have Silverlight Toolkit installed. The introductory part of this book contains all the information you need to know about how to obtain Silverlight Toolkits and use controls contained in it.

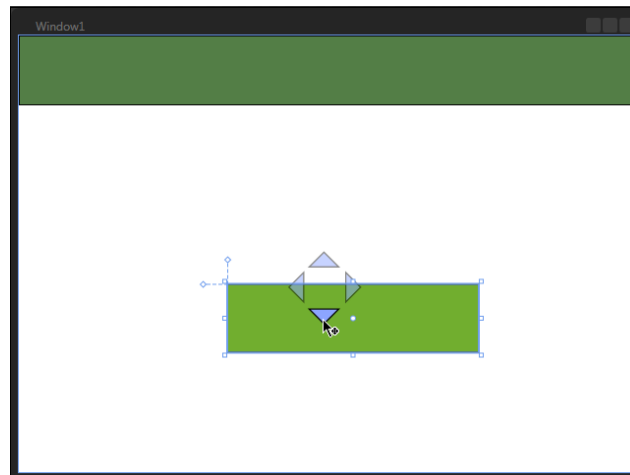
How to do it...

Once your new WPF project has been created you should notice that under the **Objects and Timeline** pane, **Window** and **LayoutRoot** grid controls are visible.

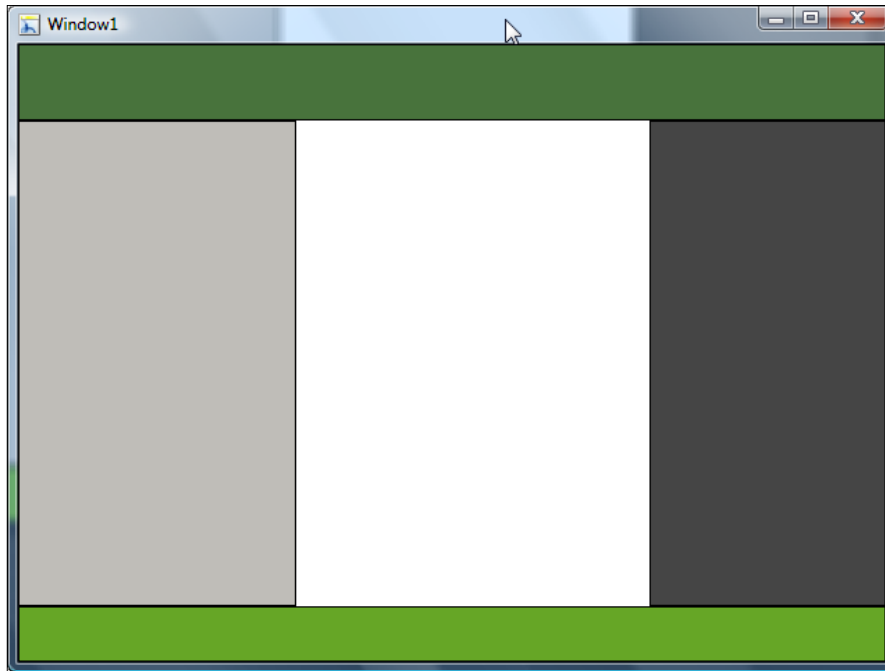
1. Right-click on **LayoutRoot** and in the pop-up menu click on **Change Layout Type | DockPanel**. This is the fastest way for us to use **DockPanel** as our root control. Of course, another possibility is to click on **Asset Library** located on the toolbox and then select and draw **DockPanel** as control.
2. Click on **LayoutRoot** to select it and under the **Properties** tab, within the **Layout** section make sure that the **LastChildFill** property is set to false (unchecked).

Now, let's go and add several rectangles on our **LayoutRoot** so that we can understand basic layout principles.

3. Select **Rectangle** from the toolbox and draw it onto your **LayoutRoot**. With the first rectangle selected, under the **Properties** panel, locate the **Layout** section and from the **Dock** drop-down list select **Top**. Set **Width** to **Auto** and **Height** to **55**.
4. Now draw another rectangle. Instead of selecting and typing in values, we can set the dock position in a different way. Select **Rectangle** and drag it towards the top. You will notice a large four-way cursor showing you directions for possible docking locations of your object. Dock this second **Rectangle** at the bottom.



5. Set the **Width** property to **Auto** and **Height** to **40**.
6. Repeat this procedure for two more rectangles, but align them to the left and right side respectively. Set **Height** to **Auto** for both of them. For the left docked rectangle set the width to **200**, and for the right docked rectangle, set the width to **170**. Set different fill colors for each rectangle so that they are easier to notice and recognize.
7. Press **F5** and start your application. It should look similar to the one in the following screenshot:

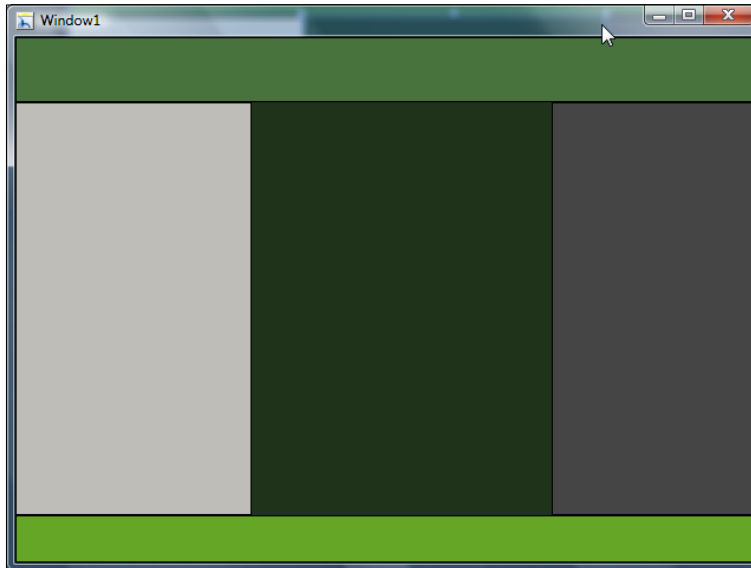


Now let's go one step further and add the **Grid** control to the central part of our application. But now, we want our **Grid** control to completely fill in the available space on the form, so that we can utilize it later; for example, when we decide to extend our application.

8. From the toolbox, select and draw the **Grid** control on the **LayoutRoot** control. Under **Properties** and under **Brushes**, select any color for its **Background** property. The purpose of this color selection is only to make our **Grid** control a bit more distinctive.
9. Now select the **LayoutRoot** element (the easiest way to do this is to select it under the **Objects and Timeline** panel just by clicking on it) and under the **Properties** pane, locate a property called **LastChildFill**, and make sure it is checked.



10. Select the newly added **Grid** control and set its **Height** and **Width** properties to **Auto**.
11. Press *F5* and you should get a layout that looks close to the one in the following screenshot:



Try resizing the form and you will notice that all our rectangles are keeping their positions docked to defined sides, and the central part (**Grid**) is filling in all the available space. I will explain this kind of behavior in more detail in the following paragraph.

How it works...

The first thing we did after starting a new project was change the layout type. The default layout type is grid, but we wanted to use **DockPanel** so that we could explore its behavior.

DockPanel enables us to dock specific child elements. In our example, we have added several rectangles and set their **Dock** property. It was obvious that we could dock our objects in two different ways—by manually setting the **Dock** property within the **Properties** pane or by simply dragging child objects over a large four-way cursor.

We have set specific heights and widths for child objects, but please take into account that in order to achieve that your docked control will scale appropriately, you need to set **Auto** for its **Height** or **Width** properties, depending on their docking direction.

At the very beginning of our recipe, we have set the **LastChildFill** property to false (unchecked it). The idea behind the **LastChildFill** property is to enable the last child element added to **DockPanel** to fill the remaining space. We achieved exactly that when we added **Grid** control and set the **LastChildFill** to true (checked it).

So just to summarize: **DockPanel** arranges its child elements so that they fill a particular edge of the panel. What happens if you set up multiple child elements to have the same docking direction? They will simply stack up against that edge in order.

There's more...

Change the docking order of child elements

Once you have added child elements to your **DockPanel** you might want to change their order. That procedure is fairly simple.

1. Make your **DockPanel** active; the easiest way to achieve that is by double-clicking on it or selecting it under **Objects and Timeline**.
2. Click the child element and simply drag-and-drop it on the desired position (you are *not* dragging-and-dropping child elements on the artboard). Now you are dealing with them within the **Objects and Timeline** pane. Note that by doing this you are not changing the docking orientation; you are changing only the **z-order**. It is also sometimes called **stack order**.

Try experimenting with this and you will notice that the actual docking order is pretty important when it comes to the overall UI layout. The reason for that is that when elements fill up the panel (and that usually happens when we set their heights and widths to **Auto**), some parts might be cut off depending on the screen and child element size.

Change the orientation of a dock panel

Sometimes you will face situations where you want to change the docking orientation for specific child elements after you have already set them. You can do that by following these instructions:

1. Make your **DockPanel** active: the easiest way to achieve that is by double-clicking it or selecting it under **Objects and Timeline**.
2. Click the child element and simply drag-and-drop it on the desired docking position on your artboard. Once you start dragging elements you will notice a large four-way cursor showing you docking directions. Now you only need to drag the element over the direction arrow you want. You will notice that the direction arrow that you select is highlighted indicating the docking direction.
 - Another way of changing the docking orientation is to select child element and under the **Layout** section in the **Properties** pane, select **Top**, **Right**, **Bottom**, or **Left** from the **Dock** drop-down list.

Personal view

I will say it bluntly—I am not a big fan of the **DockPanel** control. Although it is a great and simple-to-use control that enables you to create a basic layout for your UI, I will always prefer the **Grid** control as my control of choice when it comes to layouts. The **Grid** control is slightly more complicated, but it is more flexible as well. Anything you can do with **DockPanel** can be done with the **Grid** control, and then some.

Of course, **DockPanel** gets its credit when we talk about simplicity, and really, if your basic UI structure is really simple and you need to get things done in a fast and simple manner, consider using the **DockPanel** instead of **Grid**. Use **Grid** and other controls within **DockPanel**, work on their docking positions, and you will probably be able to create the UI you were looking for. If that doesn't work, you can always rely on **Grid**—the most powerful of all WPF or Silverlight layout controls.

See also

- ▶ *Fluid versus fixed layouts*

Journal navigation

Journal navigation enables you to utilize "back and forward" metaphors usually seen in web browsers even for your desktop (WPF) applications. As humans, we are often found in positions where we are thinking in a linear way and we tend to associate web page navigation and other UI related concepts with that.

As web browsers do support this kind of navigation by default, more challenging is to achieve this kind of behavior for desktop (WPF) applications. In this recipe, I will show you how to utilize this model and enable users to go back and forth in their navigation history within your WPF application.

Getting ready

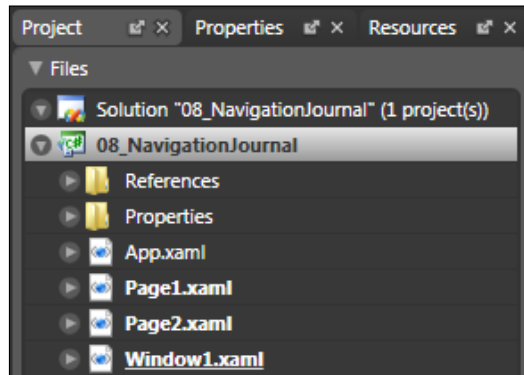
Start your Expression Blend 4 and then select New project... From the dialog that appears select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK..

How to do it...

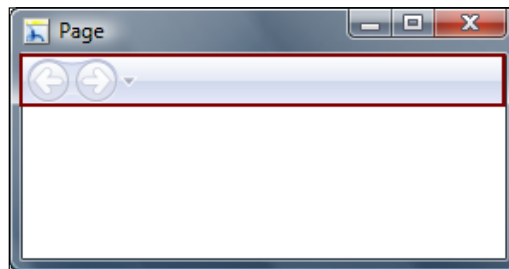
When you create your new WPF project in Expression Blend, you are presented with the **Window** object. In order to implement journal navigation pattern, you need to use **Page**.

For this example, we will add two pages and show how to navigate between them and retain the navigation history.

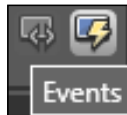
1. Click on the **Project** pane and then right-click on the name of your project. From the drop-down menu choose **Add New Item....**
2. A dialog box will appear showing you your possible choices. Select **Page**. Leave the name as it is (**Page1.xaml**) and be sure to have the **Include code file** option checked.
3. Repeat the procedure and in the same way, add **Page2.xaml** to your project. Now your project tree should look like the one in the following screenshot:



4. Now right-click on **Page1.xaml** and from the drop-down menu, select **Startup**. This will make **Page1.xaml** a **startup object**—the first one to appear when you start your application.
5. If you now hit *F5*, you should be able to see your **Page1.xaml** completely blank, but notice the navigation chrome with back and forward navigation buttons (sometimes also called "travel buttons") added *automagically*. It is not functional at this moment, but very soon you will be able to use it.



6. Let's change some properties and make our pages look a bit more distinctive so that we can recognize them easily later.
7. We will start with **Page1**. Under the **Objects and Timeline** pane, select the **Page** object. On the **Properties** pane, locate **Common Properties**, find the **Title** and **WindowTitle** properties, and set them both to **First page**.
8. Repeat the procedure for **Page2**, but now set the **Title** and **WindowTitle** properties to **Second page**. (I will explain the difference between **Title** and **WindowTitle** properties later on.)
9. Now, let's add a button on **Page1**. We will use that button for navigating from **Page1** to **Page2**. To add a button, locate it on the toolbox or add it from **Asset Library** and draw it on your **Page1**.
10. On the top of the **Properties** pane set its **Name** property to **btnNavigate**. Also, find the **Common Properties** section and change the **Content** property to **Navigate to other page**.
11. Now, we need to add some code that will enable us to really navigate to the second page when a user clicks on this button. Basically, we will define an event handler for the **Click** event.
12. Under the **Properties** pane, locate the **Events** icon and click on it.

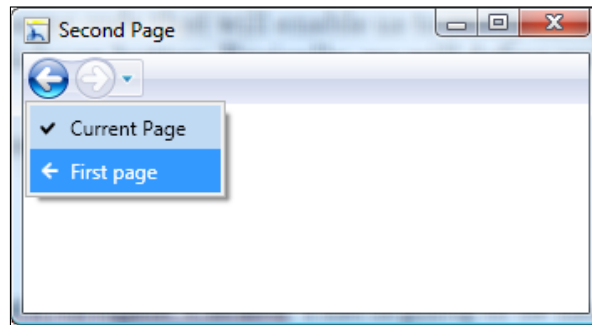


13. Under **Click**, type in **btnNavigate_Clicked**—the name of your event handler. Hit *Enter* and Visual Studio will enable you to type in the code needed.
14. Add the following code in the event handler:


```
private void btnNavigate_Clicked(object sender, RoutedEventArgs e)
{
    ((NavigationWindow) (Application.Current.MainWindow)).Navigate(
        new System.Uri("Page2.xaml", UriKind.RelativeOrAbsolute));
}
```
15. And add the following line at the very beginning of the code:


```
using System.Windows.Navigation;
```
16. Note that **Page2.xaml** is the file name we are navigating to. Now hit *F5* and when the application starts, click on **Navigate to other page**.

17. Note the changes in the navigation chrome. Explore it by clicking on the back and forward buttons or drop-down menu.



How it works...

We have added two **Page** items to our project: **Page1.xaml** and **Page2.xaml**. The basic idea of this recipe was to show you how to navigate from **Page1.xaml** to **Page2.xaml**.

By setting **Page1.xaml** as the startup page, we have ensured that it will be the first page to be shown when the application starts.

We have added a button and called it **btnNavigate** and associated the **Click** event handler.

There is only a single line of code that enables navigation between pages:

```
((NavigationWindow) (Application.Current.MainWindow)).Navigate(  
    new System.Uri("Page2.xaml", UriKind.RelativeOrAbsolute));
```

However, first we need to add a simple `using` directive:

```
using System.Windows.Navigation;
```

The great thing about implementation of the journal navigation pattern is that the navigation chrome is automatically being updated. When I say that I am thinking about the fact that the back and forward button and the drop-down menu are being updated based on the current navigation context. This pattern is often seen in applications such as web browsers, Windows Explorer in Windows Vista and Windows 7, or in Microsoft Dynamics NAV and AX line of products.

There's more...

Journal Navigation pattern is very useful. Sometimes, however, you will want to remove the navigation chrome, or you might be wondering what's the difference between **Title** and **WindowTitle** properties. The following section will explain you that and provide you with even more information.

Removing the navigation chrome

At some point, you might consider removing the navigation chrome and replacing it with your own implementation of the same. Although I won't be going into details and explaining how to replace it with your own, I will show you how easy it is to remove the navigation chrome with just a single line of code.

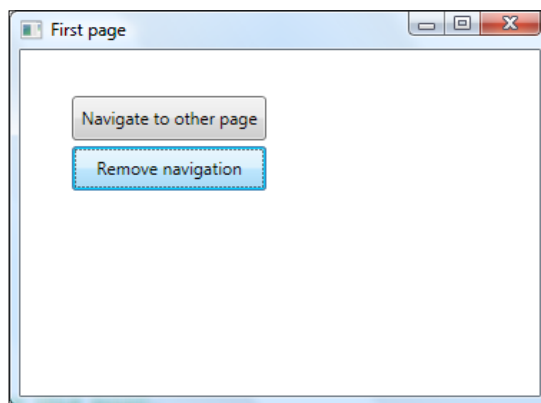
All you need to do is add this line of code:

```
this.ShowsNavigationUI = false;
```

As an example, and for demonstration purposes only, I've added a new button and called it **btnRemoveNavigation** and attached a new **Click** event handler called **btnRemoveNavigation_Clicked**. So my complete code looks like this:

```
private void btnRemoveNavigation_Clicked(object sender,
    RoutedEventArgs e)
{
    this.ShowsNavigationUI = false;
}
```

If you now go and hit *F5* to start your project and then click on **Remove navigation**, the navigation chrome will disappear.



Though you can remove the navigation chrome and navigation functionality will stay intact (one that you have implemented by adding event handlers), your users will suffer immensely if you don't implement some sort of navigation UI (that is, if your application is based on this pattern).

You *must* ensure that simple, easy-to-use, and noticeable navigation chromes exist at all times while your users are using an application based on this pattern.

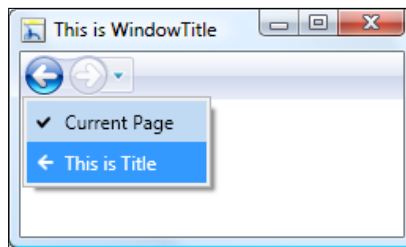
Difference between Title and WindowTitle

In Step 8 of this recipe, I've mentioned that there is a difference between **Title** and **WindowTitle** properties. So, what's the deal?

Both **Title** and **WindowTitle** are **Page** properties located under the **Properties** pane, in the **Common properties** section.

The **Title** property sets a title for a specific page. What you enter here will be displayed in the navigation chrome (including a drop-down menu).

WindowTitle sets the title for a specific **Window** object. Basically, this means that whatever you enter for the **WindowTitle** property will be displayed at the top of the window—in the title bar.

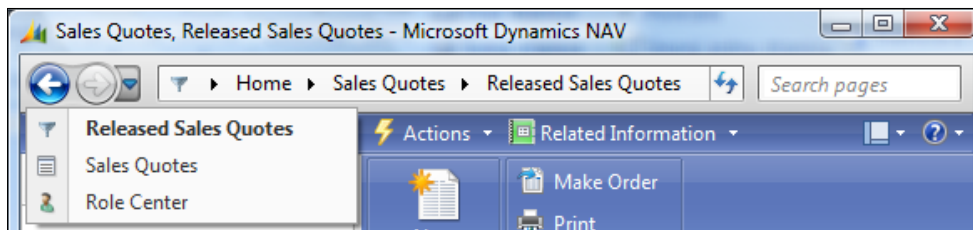


When to use journal navigation

As people tend to think linearly (probably based on their everyday experiences with time flow and similar concepts), we also appreciate the ability to navigate through our applications in the same way. **Journal navigation** is a good pattern that provides us with a mechanism that can be used to go back and forth in the navigation history of our application.

We experience software in the same, more or less single thread, single timeline manner. For example, recall using the web browser of your choice—when you are navigating from one page to another, you have the impression that the main view is being changed. Implementation of journal navigation enables us to track and revert to those views or pages in a pretty simple and straightforward manner.

Journal navigation is not only good (or should I say mandatory) for web browsers but it is also a good choice for a number of different needs. Wizards, desktop, or RIAs (Rich Interactive Applications) are just some of the typical, potentially good candidates for this pattern. Windows Explorer in Windows Vista and Windows 7 uses this same pattern. Microsoft Dynamics NAV and Microsoft Dynamics AX are also great examples; they are combining journal navigation with breadcrumb bars.



So, the basic principle about when to use journal navigation as a pattern is a situation where you want to enable the no forward-only navigational experience, and instead, you are to utilize abilities to go back and forth, thus enhancing the experience of moving from one page to another.

Good practice suggests that you concentrate on three main areas when dealing with the journal navigation implementation. Luckily, when you are using WPF **Page** and the recipe described here, they are (mostly) all being taken care of, but just for your reference, I will describe them.

The first field is surfacing the navigation chrome (user interface). It has to be obvious to the end user that they are dealing with the navigation UI. I've explained how easy it is to remove the default navigation chrome, but you *must* use some sort of navigation UI. Good practice is also to enable users to get access to back and forward buttons via the keyboard and not just by clicking with the mouse.

As you can say by your own intuition or from your personal experience, when interacting with different applications, having a consistent, clear, and easy-to-use navigation system is crucial. It certainly enables users to feel more in control and empowered when using your application.

Second important thing is to have some sort of history so that users can easily see where they have been previously and navigate through their history. In our recipe, that list is easily available by clicking on an arrow pointing downwards, which exposes a drop-down menu with the list of visited pages.

The third and probably most important thing to take care of is the context of your application. Do you want to enable users to navigate back and forth or do you require a one-way task flow? What happens if your users navigate back while some action is still in progress: will you cancel the action, continue the action, or do something else? How will you notify your users?

Never, and I mean never, use the **Back** button as an undo operation! Use a specific, single, understandable command for an undo operation. I've seen too many applications trying to use the **Back** button as a replacement for the undo command, and bluntly said, they "suck". You don't want your application to be characterized as the one that "sucks", do you?

See also

- ▶ *Wizards*

Tabs

Tabs provide a simple way of presenting sets of controls or documents on separate, labeled pages. They are quite popular and most commonly associated with typical property windows. On the Web, they are also very popular as a means for content organization.

As a very general guideline, tabs are used when you are dealing with too much information on one page (it doesn't matter if it is a web or desktop application), which basically creates confusion making the specific content difficult to find and focus on.

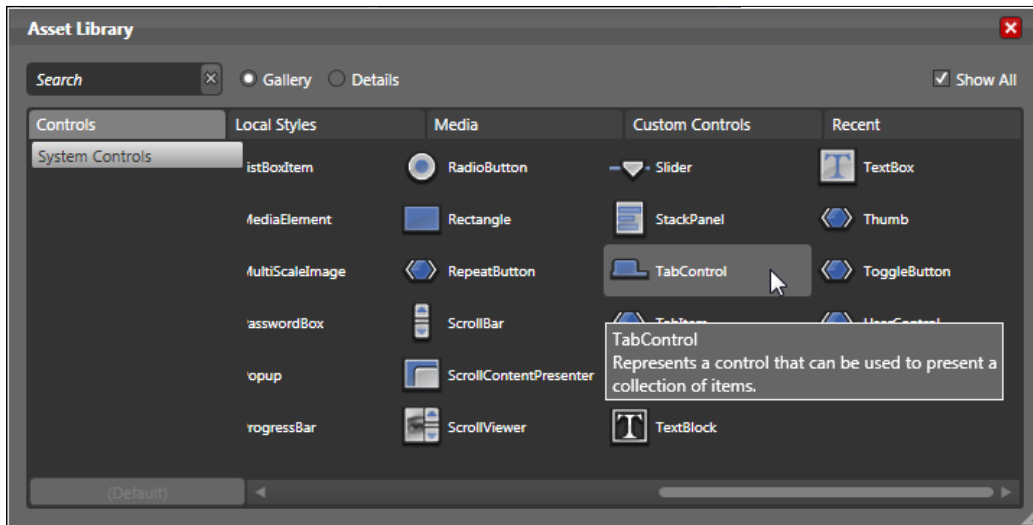
Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears select Silverlight and then Silverlight Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK.

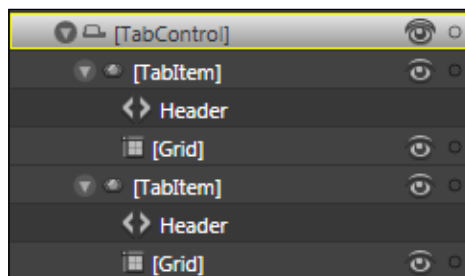
How to do it...

In this example, we will add the **Tab** control as a part of our Silverlight 4 project:

1. After you have created your new project, under the **Objects and Timeline** pane, you will see **UserControl** and **LayoutRoot**. **LayoutRoot** is a **Grid** control hosted in **UserControl**.
2. The **Tab** control is not a part of the "default" Silverlight controls, so in order to use it, we have to add a reference to it. After you installed the Silverlight 4 SDK, you have obtained a library that contains a number of controls, among them **TabControl**.
3. Under the **Project** tab, go to **References | Add Reference....**
4. The **Add Reference** dialog box will appear. Navigate to a file called `System.Windows.Controls.dll`, which is usually located in a path similar to: `C:\Program Files\Microsoft SDKs\Silverlight\v4.0\Libraries\Client`.
5. Select the file and click **Open**.
6. Now you will see `System.Windows.Controls.dll` under your **References** folder.
7. Now, go to **Asset Library** available on the toolbox. Be sure to check **Show All**. Now you should be able to locate **TabControl**.

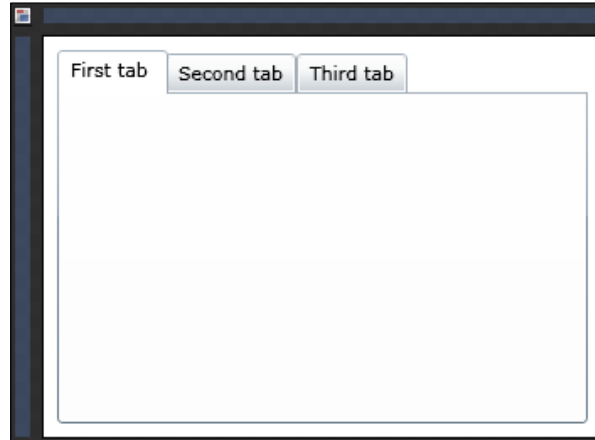


8. Click on **TabControl** and draw it on the artboard.
9. Take a look at the **Objects and Timeline** pane and expand the visual tree so that you can see all parts of the **TabControl**.

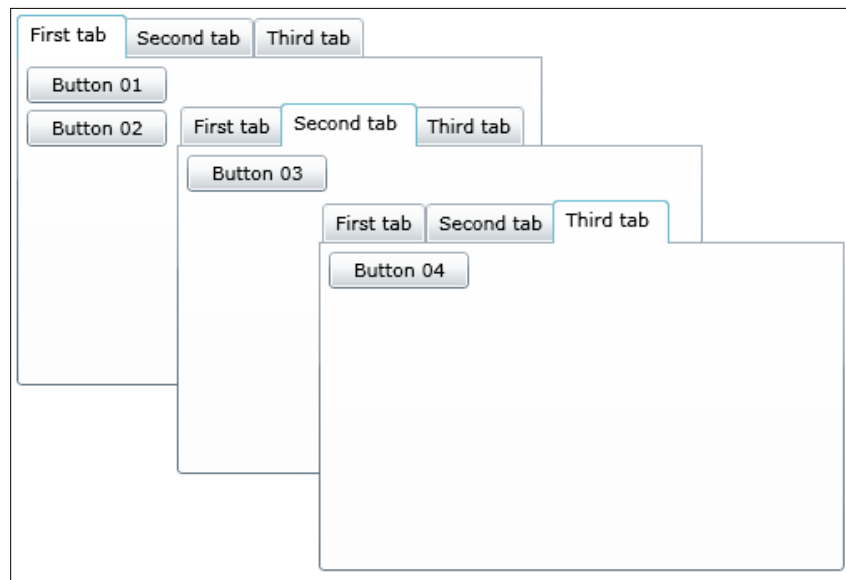


10. Let's add the third **TabItem** to our **TabControl**. Right-click on **TabControl** under the **Objects and Timeline** pane and from the drop-down menu, click on **Add TabItem**. Now you should see the third **TabItem** added to your **TabControl**.

11. Click on the first **TabItem** and under the **Properties** pane, locate the **Header** property (located under **Miscellaneous** section), and type in **First tab**. Do the same for the second and third **TabItem** typing **Second Tab** and **Third Tab**, respectively. Your tab control should look like the one in the following screenshot:



12. As you will have noticed under the visual tree, each **TabItem** is comprised of the **Header** part and **Grid**. The **Grid** part is basically the **Grid** control in which we can put any content we want.
13. Select **First tab** and add two buttons (add them from the toolbox or **Asset Library**). Change their content (**Properties** pane, **Common Properties** section, **Content** property) to **Button 01** and **Button 02**. Be sure that you have selected the **Grid** element of the specific **TabItem** to ensure that the buttons will be added to exactly that **TabItem**.
14. Repeat the same procedure: adding one button to the **Second tab** and one to **Third tab**. Set their content to **Button 03** and **Button 04** respectively.
15. With this, you must have got a pretty clear understanding of how you can add controls to different **TabItems** and how to add **TabItems** as well.
16. Now hit **F5** or go to **Project | Test Solution**.
17. Your default web browser will start and you will see your tab control. Try clicking on different tabs and notice that you can see only the content that you have added to a specific **Grid** under specific **TabItem**.



How it works...

As **TabControl** is not a "default" Silverlight 4 control; we had to add a reference to the `System.Windows.Controls.dll` file (that file is part of the Silverlight 4 SDK pack). After adding the reference, **TabControl** has been added to our **Asset Library** and is available for use from there like all other controls.

Right after you have drawn **TabControl** on the artboard, you will notice two tab items (colloquially called just "tabs"). By right-clicking on **TabControl** and using the option **Add TabItem** you get the opportunity to add more tab items.

It is a general suggestion that you don't add more than seven tabs. Take a look at the *There's more...* section where I will go deeper into general guidelines for using tab as a control.

The next step was adding titles for specific tabs. As shown earlier, specific **TabItem** is comprised of **Header** and **Grid** parts. By clicking on **TabItem** and locating the **Header** property, you have got the possibility to change the title for a specific **TabItem**. We have used just provisional titles, **First tab**, **Second Tab**, and so on.

The next step was adding the specific content to specific tabs. Technically speaking, that content is hosted within the **Grid** control. That, of course, means that all layout, formatting, and other Grid-related mechanisms and properties are applied to all controls hosted. Grid is, without doubt, the most powerful layout control available in WPF and Silverlight.

By hitting *F5* and starting your web browser you got the opportunity to explore tabs and associated buttons (controls).

There's more...

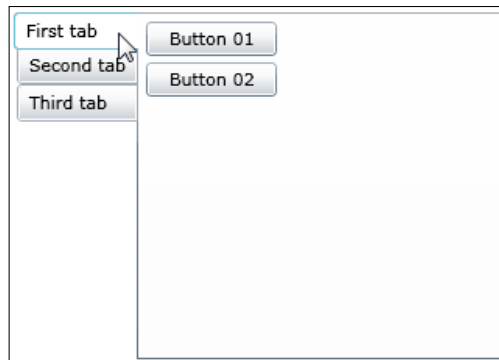
In the first part, you've been introduced to the tab control and its main characteristics. Now, you'll learn how to further customize the control itself and also get the professional and insightful guidelines for using the tabs in your real life applications - no matter what are you using, WPF or Silverlight.

Changing tab orientation

Tabs are most commonly oriented horizontally, but **TabControl**, which is used in our example, supports different orientations.

I will assume that you are continuing from the previous **TabControl** example.

1. Select your **TabControl** under the **Objects and Timeline** pane. Under the **Properties** pane, locate the **TabStripPlacement** property under **Miscellaneous**. The drop-down list offers you the following options: **Left**, **Top**, **Right**, and **Bottom**.
2. Select **Left**.
3. You will notice that the tab orientation has been changed. Hit *F5* and investigate a new look within your browser.



As a small digression, *Last.fm*, a popular music and radio community website, uses this kind of layout for its profile page's design and I find it to be highly usable and pleasing.

Adding icons in tabs

In the *When to use tabs?* paragraph, I will give guidelines regarding the usage of icons in tabs, so be sure to understand when it is correct and acceptable to use icons in tabs.

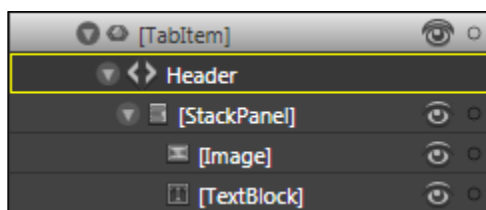
Again, I will take it from our initial example where we have added **TabControl** and three **TabItem** elements (tabs).

Currently, there is no property like **Icon** for each **TabItem**, but we can add them manually by modifying the **Header** part of **TabItem**.

1. Let's just modify the **Header** for the **First tab** and you can apply the same method for the rest of them.
2. The idea is to add **Image** control and **TextBlock** into **Header** and then to place an icon into **Image** control and set the content of **TextProperty** to **First tab**. We want **Image** and **TextBlock** controls to be stacked horizontally, so we will put them in the **StackPanel** container.
3. Double-click on **Header** under the first **TabItem**; this will make it active. Now from the toolbox or **Asset Library** add **StackPanel**. Set its **Height** and **Width** to **Auto** and **Orientation** to **Horizontal**. You can do this by going to the **Properties** pane and locating them under the **Layout** section.
4. Now select the newly added **StackPanel** and add **Image** control and **TextBlock**. The easiest way do to that is just by locating them under the toolbox or **Asset Library** and double-clicking on them. They will be automatically added to the selected container, **StackPanel** in our case.
5. Select **Image** control and under **Properties** locate the **Common Properties** section. Find the **Source** property and click on **Choose an image button** (ellipses). From the dialog box, select an image—the one that you want to add as an image. I suggest using the PNG image with dimensions of about 16x16 pixels.
6. Now select **TextBlock** and change its **Text** property, which is under the **Common Properties** section within the **Properties** pane. Set it to **First tab**.
7. If you now take a look at the XAML code of the selected **TabItem**, it should look similar to this:

```
<basics:TabItem.Header>
  <StackPanel Height="Auto" Width="Auto" Orientation="Horizontal">
    <Image Width="16" Height="16" Source="Help and Support.png"/>
    <TextBlock Text="First Tab" TextWrapping="Wrap"/>
  </StackPanel>
</basics:TabItem.Header>
```

8. Structure, as can be seen under the **Objects and Timeline** panel, looks like this:



9. Press *F5* now and your **TabControl** should look like this:



When to use tabs

I've already mentioned that tabs are usually good in situations when we are dealing with large amounts of information on a single page or window and, as a consequence, users are having difficulties finding, using, and focusing on specific content and possibly, tasks. Tabs are really handy in such situations, as they allow us to break up related pieces of information and organize them on individual tabs, making them available one at a time.

According to Microsoft's own Windows user experience guidelines, there are several questions that are to be asked when deciding whether you should use the tabs as control. They can be summarized as follows:

- ▶ Can all your controls comfortably fit on a single page? If so, there's no need for tabs, and the same. The same holds true if you are using just one tab—do not use tabs just to use them. You might be tempted to do so thinking that this would make your application or web application more professional, but the reality is quite the opposite.
- ▶ Are you using tabs to organize settings and numerous options? Be sure to check if the changing of the options on one tab affects options and settings on other tabs. Each and every tab must be mutually independent. If that is not the case, use wizard pattern.
- ▶ From the hierarchical point of view, ask yourself: Are the tabs mostly peers (on the same levels) or do you have a clear hierarchy organization among them? Tabs must have a peer relationship, never hierarchical. Just think about it: they are *linearly represented*—an orientation that is implying the same hierarchy levels.
- ▶ Are you using tabs for wizard-like step-by-step processes where the first tab represents starting point and tabs that follow are next steps in the process? If that is the case, again, wizard is the appropriate pattern.

This list, of course, is far from comprehensive but offers you some initial guidelines.

Real-world metaphor

Tabs are taken from the real world and they do aim to leverage familiarity that most users have from tabbed folders. As we do use tabs in the real world to group related documents, the same idea must be used in the world of user interfaces.

We benefit from organizing related elements and content onto individual tabs. Furthermore, you should think about the order of your tabs and set them in a way that will make sense to your users. Among other things, that is the reason why tabs are pretty popular events on the Web as navigational controls, although that is not and should not be their primary role.

Implementation guidelines

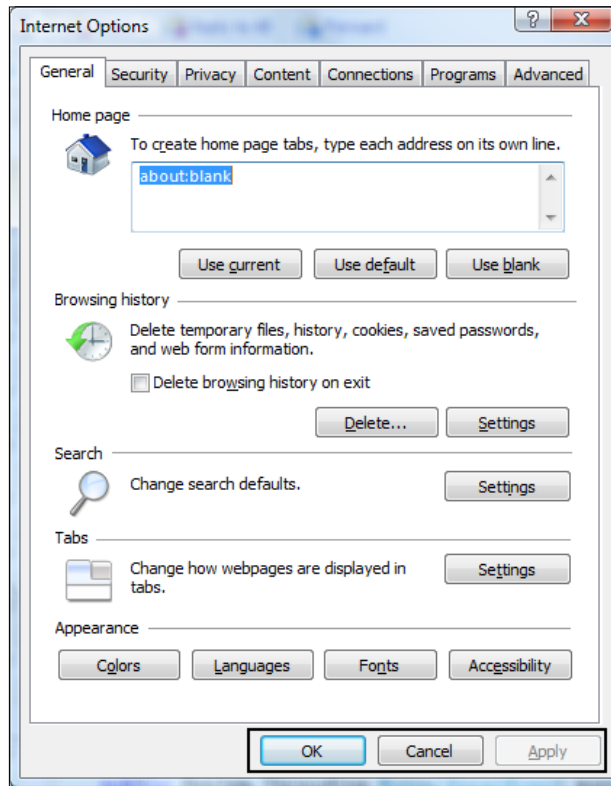
In most cases, tabs will be presented horizontally or vertically. How do we decide? Again, Microsoft's and other UI guidelines suggest, among other things, the following:

- ▶ Horizontal tabs should be used if you plan to use seven or fewer tabs and all tabs fit in one row.
- ▶ Vertical tabs are good for cases when you are dealing with eight or more tabs or if using the horizontal tabs would require more than one row (the case might be that you have five tabs but with very long labels in their header).
- ▶ Forget about using scroll bars for horizontal tabs. This pattern has poor discoverability while usage of scrolling for vertical tabs is acceptable. General idea is to have all tabs always visible. If you have too much content which must fit on a single tab, then put a scrollbar on that specific tab, not on the window that is hosting your tab control.
- ▶ By default, make your first tab selected in all cases. Exclusion from this can be only a specific case when users are likely to start from the last tab that they have selected before dismissing a window or page with tabs. In that case, make the last tab selected to persist on a per-window, per-user basis. My personal experience is that your exceptions like these are extremely rare and you will probably do a better job just by ensuring that the first tab is always selected for users.
- ▶ Don't use and put icons on tabs' headers. They add visual clutter and consume screen space. You might be tempted to use them hoping that they will improve users' comprehension but that is rarely the case.

However, there are some cases when icons might be fitting:

- Icons you are using are standard symbols, well known, and widely recognizable, and understandable
- Use icons if there is not enough space to display significant labels

- ▶ Place the terminating buttons (**OK**, **Cancel**, **Apply**, and so on) on the area outside the tabbed area, onto the dialog. A good example for this is given in the following screenshot:



By doing that you will avoid confusion in respect to the scope of the actions being carried away when user clicks on those buttons. Actions triggered by clicks on terminating buttons placed outside the tabbed area are applied to the whole page or window.

See also

- ▶ *Wizards*

Adding a status bar area

The area at the bottom of the primary application window can be used for a status bar—control that is suited for displaying the current state of the window and accompanying actions, background tasks, or some other contextual information.

In this recipe, I will demonstrate how to add the **StatusBar** control to our WPF application and share some guidelines regarding the implementation and usage of this pattern.

Getting ready

Start your Expression Blend 4 and then select New project... From the dialog that appears, select WPF and then WPF Application. Make sure that Language is set to C# and Version is 4.0. At the end hit OK..

How to do it...

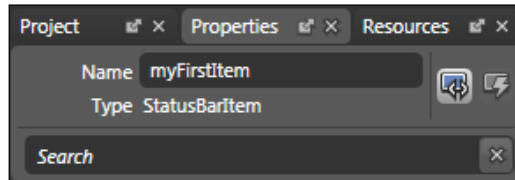
1. Once your new WPF project has been created, you should notice that under the **Objects and Timeline** pane, **Window** and **LayoutRoot** grid controls are visible.
2. Go to **Asset Library** and from there click on the **StatusBar** control. Draw it onto your artboard. Note that the **StatusBar** control will probably be hidden, so you should make sure that the **Show All** option, located at the **Asset Library** dialog, is checked.
3. Select **StatusBar** by clicking on it. Now we will set up some properties. Go to the **Properties** pane and under the **Layout** section change the following:
 - ❑ Set **Width** to **Auto** and **Height** to **24**
 - ❑ Set **HorizontalAlignment** to **Stretch** (last icon in the row)
 - ❑ Set **VerticalAlignment** to **Bottom** (third icon in the row)
 - ❑ Set all values for **Margin** to **0**
4. Your **StatusBar** should now appear docked to the bottom of your window and stretched from side-to-side.
5. As you can notice, **StatusBar** looks completely blank. Let's add a single line at the top of our **StatusBar** just to make it appear more distinct from the rest of the window surface.

- Under the **Appearance** section, set **1** for the **Top Border** value. Now under **Brushes** section, locate the **BorderBrush**, and click on it. Now click on the **Solid color brush** icon and pick some kind of dark gray. You can also enter RGB values: **123,123,123** or HEX: **#FF7B7B7B**.

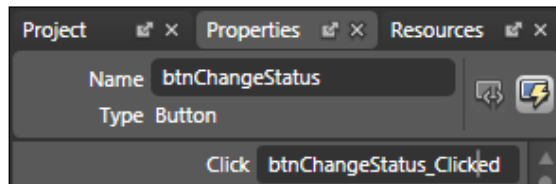


- Your **StatusBar** should now have a simple gray border at the top.
Let's add some controls to our **StatusBar**. However, note that I am just adding random controls now and not following specific guidelines; those can be found under the *There's more...* section and you should refer to them when designing and implementing the status bar pattern.
- Right-click on the **StatusBar** control and from the drop-down menu, select **Add StatusBarItem**; this will add the simplest possible item to your **StatusBar**.
- Now, click on **StatusBarItem** to select it and under the **Properties** pane locate the **Common Properties** section. Enter **Ready** for the **Content** property.

- In order to change the **Content** property of **StatusBarItem** during the run-time, we need to assign it a name. At the top of the **Properties** pane, you will find the **Name** field; type **myFirstItem** in that field.



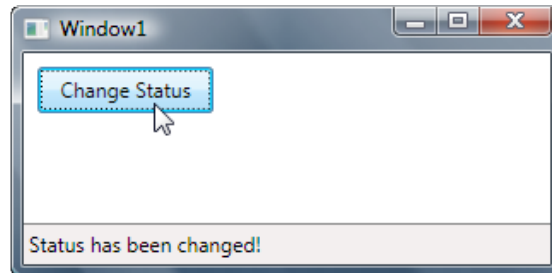
- Now add a **Button** control on top of the **LayoutRoot** control. To do that, select **LayoutRoot** under the **Object and Timeline** pane and then from the toolbox or **Asset Library** draw a simple button on your artboard.
- Give it a name **btnChangeStatus** (follow the same directions given under Step 26).
- Set its **Content** to **Change Status** (This property is located under the **Properties** pane, in the **Common Properties** section).
- The next step is adding an event handler that will change the **Content** property of **StatusBarItem**.
- With **Button** selected, click on the **Events** icon under the **Properties** pane and you will be presented with a number of events. Let's add our event handlers for the **Click** event. To do that, you will just have to type in the name for those event handlers and once you hit *Enter*, Visual Studio will start and allow you to add code logic. You can use **btnChangeStatus_Clicked** as a name for your event.



- Add the following code:

```
private void btnChangeStatus_Clicked(object sender,
    RoutedEventArgs e)
{
    this.myFirstItem.Content = "Status has been changed!";
}
```

17. Press *F5* and your application will start. Click on **Change Status** and note that the text in your status bar area has changed.



How it works...

After the **StatusBar** control was added, I set several of its properties. I've set the height to 24 (it was fairly arbitrarily defined in this particular case), and I've also set the **HorizontalAlignment** to **Stretch** (to ensure that the **StatusBar** will consume all the horizontally available space). Also, **VerticalAlignment** has been set to **Bottom** (to ensure that **StatusBar** will dock to the bottom). By setting values of all margins to zero, I've ensured that there will be no empty space around the status bar when docked and stretched.

In order to make the **StatusBar** a bit more distinct (and to show a really simple way of customizing the control), I've changed the top border width and color.

With all the positioning and the look set, the next step was to add a very simple element to the **StatusBar**—**StatusBarItem**. Later, I will demonstrate how you can add a number of other different elements to **StatusBar**, but **StatusBarItem** was the simplest to do and it served me well to show you how to update its **Content** property via code.

This pattern might be used in real applications to update status messages during runtime, but in case your development knowledge is more advanced, you can take it to a more advanced level. But the idea remains the same: give the name to **StatusBarItem** and change its **Content** property.

There's more...

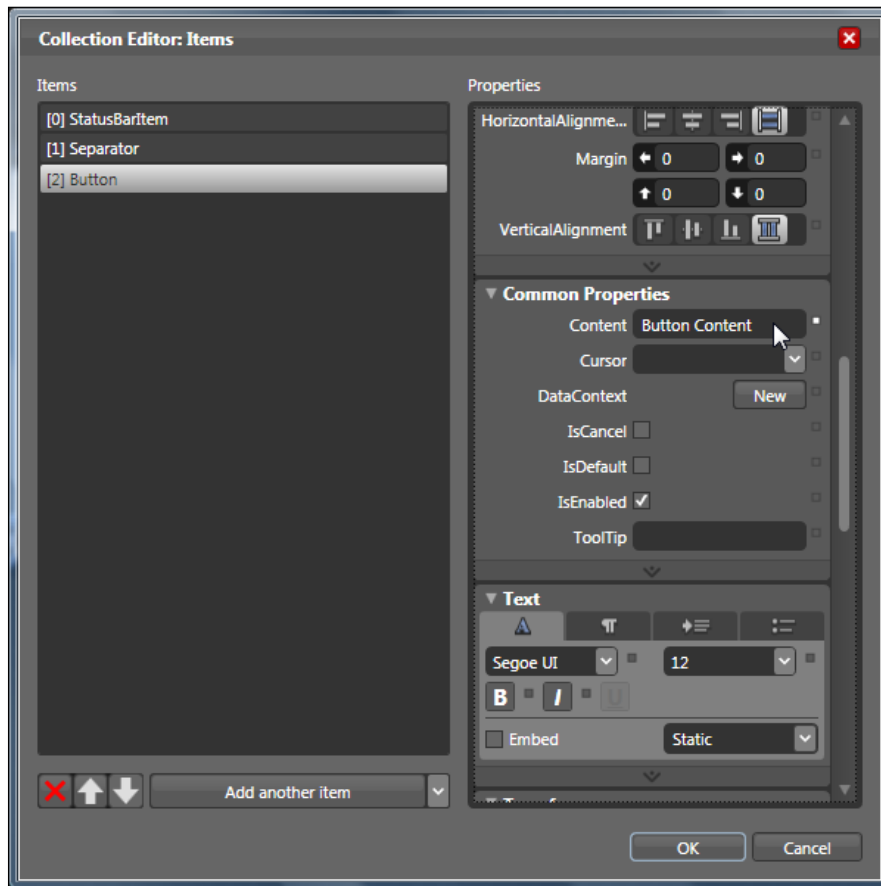
Status bar can be used to host more different controls to help you build better and more solid user experience for your users. In this section, you will learn how to do exactly that but you will also gain knowledge and valuable guidelines for implementing and designing user interfaces which are utilizing **StatusBar** control.

Adding other controls to StatusBar

The previous example showed how to add the simplest of all elements to **StatusBar**. However, **Blend** enables you to add a number of other different controls to your **StatusBar**.

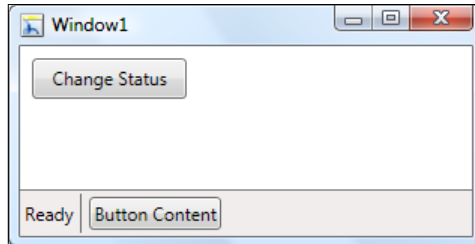
You can take it from the last point in the first example.

1. Select the **StatusBar** control and under the **Properties** tab, locate the **Common Properties** section. Click on the ellipses button near Items(Collection) and the **Collection Editor (Items)** dialog will appear. At the bottom, click on the arrow pointing downwards near **Add another item**. From the list, select **Separator**, and press *Enter*.
2. Repeat the procedure but this time select **Button** control from the list.
3. In the right-hand part, scroll down to **Common Properties** and type **Button Content** in the **Content** property. Press *Enter*.



4. Click on **OK** and now press *F5* to start and test your application. Now you should be able to see a button with the **Button Content** label in your **StatusBar** control.

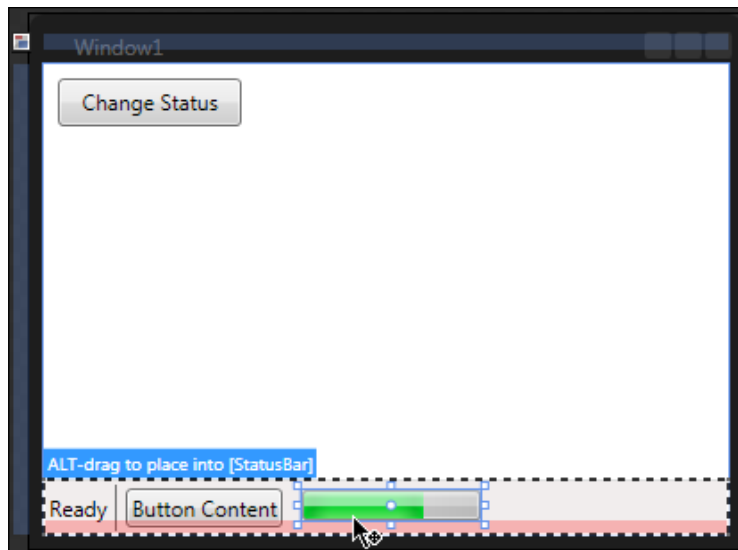
5. If your button is not completely visible, try changing the **StatusBar Height** property to, let's say **32**. Press *F5* now and your application should look like the one in the following screenshot:



What if we want to add controls that are not listed in the drop-down list that we have seen in the **Collection Editor**? For example, how can we add the **ProgressBar** control?

6. Under **Objects and Timeline** pane, select the **StatusBar** control. Now go to **Asset Library** and pick the **ProgressBar** control. The next step is just to draw the **ProgressBar** control on top of your **StatusBar** control.

Alternatively, you can just draw the **ProgressBar** control on your artboard and simply drag-and-drop it on top of the **StatusBar**. If you press the *Alt* key while dragging, **ProgressBar** will be added as an element of the **StatusBar**.



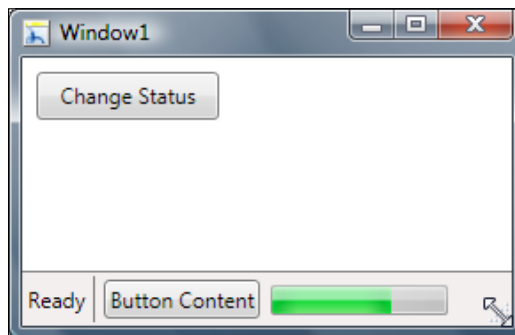
Take into account, if you want to be able to set or get properties from objects added to **StatusBar** control, you need to name them and use that name when referencing them from code.

Adding SizeGrip to StatusBar

Size grip is a commonly seen and used pattern: it enables you to grab a corner of your window and pull it to resize the window. Here is how to add a size grip to your status bar and enable resizing.

I will just continue from the last example. So, return to Expression Blend.

1. Under **Objects and Timeline**, select **Window**.
2. Go to the **Properties** pane and locate the **Common Properties** section. You will find the **ResizeMode** property there.
3. Click on the drop-down list and select **CanResizeWithGrip**.
4. You will now notice the size grip at the bottom-right corner of your window, on top of the **StatusBar** control.
5. Press *F5* to start your application. Position your mouse cursor over the size grip and try resizing the form. As simple as that, zero lines of code!



When to use status bar

A status bar should be used in cases where you want to provide status information to your users but without harsh interruption of their main activities. It is generally located at the very bottom of the main (primary) application window and stretches from left to right with no margins, though specific exceptions might occur.

Information provided in a status bar is usually related to the current window's state, possible tasks in the background, or some contextual information.

The single most important thing is that a status bar should be used to provide information to the end users without interrupting them, which also means that information provided in a status bar should not be extremely critical or something that requires immediate attention. So, displaying status of loading a web page is fine, but letting the user know that a virus has been found on their C drive is something that requires immediate attention and should not be communicated through status bars.

Usually, a status bar uses simple text or images (icons) to communicate information, but a number of other controls can be also used. A progress bar (indicator) is one of them and it is often seen in web browsers showing the loading process of a web page. Also, menus for commands and options can be used in status bars.

However, status bars have one disadvantage: they are not easily discoverable. It is fairly easy to overlook them or even to completely ignore them. What can you do to "fix" this? I hope you are not thinking about using some aggressive animations, blinking, vivid icons, or some other means of grabbing users' attention. Recall what I said a few lines earlier: *information provided in the status bar should not be extremely critical or something that requires immediate attention.*

Ensure that the information you are placing into the status bar is relevant and useful and if that is not the case, well, then don't use the status bar. Also, if a user must see the information, then don't put it in the status bar—the status bar is not for critical information.

So, to summarize, use the status bar for relevant and useful information, but never for critical information!

Implementation guidelines

As always, throughout this book, the implementation and other guidelines provided here are not 100% comprehensive, rather they are suggestions and proven practices extracted from various resources. However, the ones presented in this book are those most commonly used and by following them you will avoid common pitfalls and ensure solid quality.

- ▶ A status bar should be presented only on the primary window of your application. You should never use status bars on all of the windows of your application.
- ▶ Don't use status bars as places to describe usage of controls on your UI. In the past (and I still see that sometimes), there was a trend to display information relevant to specific controls in the status bar. For example, if you positioned your mouse pointer over the printer icon, you would get something like **Click here to print current document** in your status bar. You should use tooltips for this pattern, never the status bar. Same holds true for menu items.
- ▶ Is the information that you want to display in the status bar critical and/or does it require immediate action? If so, then don't use the status bar for it. Consider a dialog or message box, they will break the flow and grab user's attention.
- ▶ Although it's not a rule (because of their relatively low discoverability), status bars might not be suitable for programs intended primarily for users who are beginners.

- ▶ When using icons in the status bar, always choose easily recognizable designs. Also, use icons with more unique shapes; if possible, avoid rectangular- or square-shaped icons. You can use tooltips for icons that are not accompanied by related labels.
- ▶ Don't change the status bar content too often. Status bars should present up-to date information, but they should not appear noisy or distracting.
- ▶ When using textual labels, make them concise. Don't use bold, italic, underline, or colors to put emphasis on status bar text labels. That will only add visual clutter and noise without really helping the users and communicating valuable information.

2

Handling Data



This chapter is taken from *Microsoft Silverlight 4 Business Application Development Beginner's Guide* (Chapter 5) by Frank LaVigne, Cameron Albert.

Business applications are all about data; input received from clients, metrics regarding performance or sales, inventory, assets, and so on. Silverlight provides a robust and easy way to handle, bind, and validate this data.

In addition to data handling capabilities, Silverlight can also communicate via Windows Communication Foundation (WCF) services, providing an extensible means of communication with backend servers and data stores.

In this chapter, we shall:

- ◆ Create a WCF service and business object for receiving data
- ◆ Create a form for allowing users to submit information
- ◆ Bind the data from a data object to Silverlight controls
- ◆ Validate data and display feedback to the user

Data applications

When building applications that utilize data, it is important to start with defining what data you are going to collect and how it will be stored once collected. We know to create a Silverlight application to post a collection of ink strokes to the server. Now, we are going to expand the `inkPresenter` control to allow a user to submit additional information.

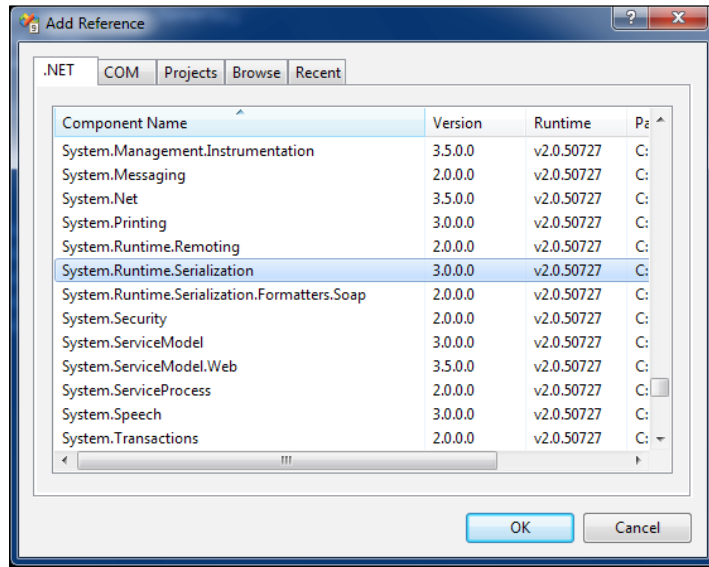
Most developers would have had experience building business object layers, and with Silverlight we can still make use of these objects, either by using referenced class projects/libraries or by consuming WCF services and utilizing the associated data contracts.

Time for action – creating a business object

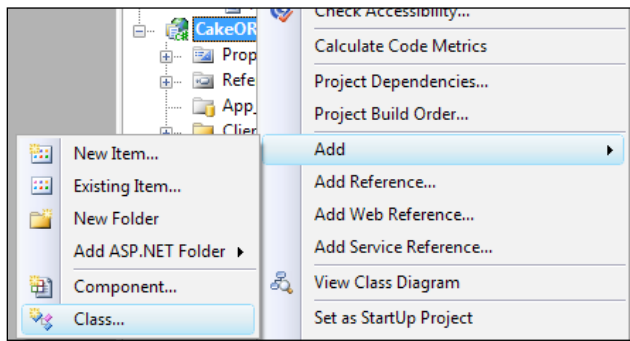
We'll create a business object that can be used by both Silverlight and our ASP.NET application. To accomplish this, we'll create the business object in our ASP.NET application, define it as a data contract, and expose it to Silverlight via our WCF service.

Start Visual Studio and open the **CakeORamaData** solution. When we created the solution, we originally created a Silverlight application and an ASP.NET web project.

1. In the web project, add a reference to the `System.Runtime.Serialization` assembly.



2. Right-click on the web project and choose to add a new class. Name this class `ServiceObjects` and click **OK**.



3. In the `ServiceObjects` class file, replace the existing code with the following code:

```
using System;
using System.Runtime.Serialization;

namespace CakeORamaData.Web
{
    [DataContract]
    public class CustomerCakeIdea
    {
        [DataMember]
        public string CustomerName { get; set; }
        [DataMember]
        public string PhoneNumber { get; set; }
        [DataMember]
        public string Email { get; set; }
        [DataMember]
        public DateTime EventDate { get; set; }
        [DataMember]
        public StrokeInfo[] Strokes { get; set; }
    }

    [DataContract]
    public class StrokeInfo
    {
        [DataMember]
        public double Width { get; set; }
        [DataMember]
        public double Height { get; set; }
        [DataMember]
        public byte[] Color { get; set; }
        [DataMember]
        public byte[] OutlineColor { get; set; }
    }
}
```

```
[DataMember]
public StylusPointInfo[] Points { get; set; }
}

[DataContract]
public class StylusPointInfo
{
    [DataMember]
    public double X { get; set; }
    [DataMember]
    public double Y { get; set; }
}
}
```

4. What we are doing here is defining the data that we'll be collecting from the customer.

What just happened?

We just added a business object that will be used by our WCF service and our Silverlight application. We added serialization attributes to our class, so that it can be serialized with WCF and consumed by Silverlight.

The `[DataContract]` and `[DataMember]` attributes are the serialization attributes that WCF will use when serializing our business object for transmission. WCF provides an opt-in model, meaning that types used with WCF must include these attributes in order to participate in serialization. The `[DataContract]` attribute is required, however if you wish to, you can use the `[DataMember]` attribute on any of the properties of the class.

By default, WCF will use the `System.Runtime.Serialization.DataContractSerializer` to serialize the `DataContract` classes into XML. The .NET Framework also provides a `NetDataContractSerializer` which includes CLR information in the XML or the `JsonDataContractSerializer` that will convert the object into **JavaScript Object Notation (JSON)**. The `WebGet` attribute provides an easy way to define which serializer is used.

For more information on these serializers and the WebGet attribute visit the following MSDN web sites:

<http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractserializer.aspx>.

<http://msdn.microsoft.com/en-us/library/system.runtime.serialization.netdatacontractserializer.aspx>.

<http://msdn.microsoft.com/en-us/library/system.runtime.serialization.json.datacontractjsonserializer.aspx>.

<http://msdn.microsoft.com/en-us/library/system.servicemodel.web.webgetattribute.aspx>.



Windows Communication Foundation (WCF)

Windows Communication Foundation (WCF) provides a simplified development experience for connected applications using the service oriented programming model. WCF builds upon and improves the web service model by providing flexible channels in which to connect and communicate with a web service. By utilizing these channels developers can expose their services to a wide variety of client applications such as Silverlight, Windows Presentation Foundation and Windows Forms.

Service oriented applications provide a scalable and reusable programming model, allowing applications to expose limited and controlled functionality to a variety of consuming clients such as web sites, enterprise applications, smart clients, and Silverlight applications.

When building WCF applications the service contract is typically defined by an interface decorated with attributes that declare the service and the operations. Using an interface allows the contract to be separated from the implementation and is the standard practice with WCF.

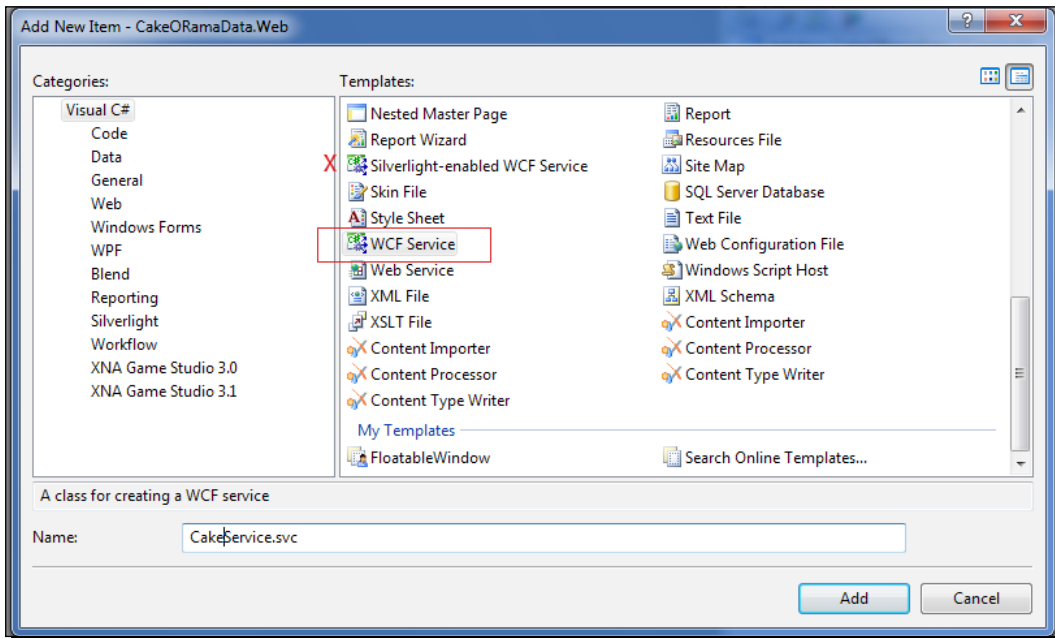
You can read more about Windows Communication Foundation on the MSDN website at: <http://msdn.microsoft.com/en-us/netframework/aa663324.aspx>.




Time for action – creating a Silverlight-enabled WCF service

Now that we have our business object, we need to define a WCF service that can accept the business object and save the data to an XML file.

1. With the **CakeORamaData** solution open, right-click on the web project and choose to add a new folder, rename it to *Services*.
2. Right-click on the web project again and choose to add a new item. Add a new **WCF Service** named *CakeService.svc* to the *Services* folder. This will create an interface and implementation files for our WCF service. Avoid adding the Silverlight-enabled WCF service, as this adds a service that goes against the standard design patterns used with WCF:



 The standard design practice with WCF is to create an interface that defines the `ServiceContract` and `OperationContracts` of the service. The interface is then provided, a default implementation on the server. When the service is exposed through metadata, the interface will be used to define the operations of the service and generate the client classes. The Silverlight-enabled WCF service does not create an interface, just an implementation, it is there as a quick entry point into WCF for developers new to the technology.

3. Replace the code in the `ICakeService.cs` file with the definition below. We are defining a contract with one operation that allows a client application to submit a `CustomerCakeIdea` instance:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;

namespace CakeORamaData.Web.Services
{
    // NOTE: If you change the interface name "ICakeService" here,
    you must also update the reference to "ICakeService" in Web.
    config.
    [ServiceContract]
    public interface ICakeService
    {
        [OperationContract]
        void SubmitCakeIdea(CustomerCakeIdea idea);
    }
}
```

4. The `CakeService.svc.cs` file will contain the implementation of our service interface. Add the following code to the body of the `CakeService.svc.cs` file to save the customer information to an XML file:

```
using System;
using System.ServiceModel.Activation;
using System.Xml;

namespace CakeORamaData.Web.Services
{
    // NOTE: If you change the class name "CakeService" here, you
    must also update the reference to "CakeService" in Web.config.
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Allowed)]
    public class CakeService : ICakeService
    {
        public void SubmitCakeIdea(CustomerCakeIdea idea)
        {
            if (idea == null) return;

            using (var writer = XmlWriter.Create(String.Format(@"C:\
            Projects\CakeORama\Customer\Data\{0}.xml", idea.CustomerName)))
            {
```

```
writer.WriteStartDocument();

//<customer>
writer.WriteStartElement("customer");
writer.WriteAttributeString("name", idea.CustomerName);
writer.WriteAttributeString("phone", idea.PhoneNumber);
writer.WriteAttributeString("email", idea.Email);

// <eventDate></eventDate>
writer.WriteStartElement("eventDate");
writer.WriteValue(idea.EventDate);
writer.WriteEndElement();

// <strokes>
writer.WriteStartElement("strokes");

if (idea.Strokes != null && idea.Strokes.Length > 0)
{
    foreach (var stroke in idea.Strokes)
    {
        // <stroke>
        writer.WriteStartElement("stroke");

        writer.WriteAttributeString("width", stroke.Width.
            ToString());
        writer.WriteAttributeString("height", stroke.Height.
            ToString());

        writer.WriteStartElement("color");
        writer.WriteAttributeString("a", stroke.Color[0].
            ToString());
        writer.WriteAttributeString("r", stroke.Color[1].
            ToString());
        writer.WriteAttributeString("g", stroke.Color[2].
            ToString());
        writer.WriteAttributeString("b", stroke.Color[3].
            ToString());
        writer.WriteEndElement();

        writer.WriteStartElement("outlineColor");
        writer.WriteAttributeString("a", stroke.
            OutlineColor[0].ToString());
        writer.WriteAttributeString("r", stroke.
            OutlineColor[1].ToString());
        writer.WriteAttributeString("g", stroke.
            OutlineColor[2].ToString());
```




One thing to note is that you will need to grant write permission to this directory for the ASP.NET user account when in a production environment.

6. When adding a WCF service through Visual Studio, binding information is added to the `web.config` file. The default binding for WCF is **wsHttpBinding**, which is not a valid binding for Silverlight. The valid bindings for Silverlight are **basicHttpBinding**, **binaryHttpBinding** (implemented with a **customBinding**), and **netTcpBinding**. We need to modify the `web.config`, so that Silverlight can consume the service. Open the `web.config` file and add this `customBinding` section to the `<system.serviceModel>` node:

```
<bindings>
  <customBinding>
    <binding name="customBinding0">
      <binaryMessageEncoding />
      <httpTransport>
        <extendedProtectionPolicy policyEnforcement="Never" />
      </httpTransport>
    </binding>
  </customBinding>
</bindings>
```

7. We'll need to change the `<service>` node in the `web.config` to use our new `customBinding`, (we use the `customBinding` to implement binary HTTP which sends the information as a binary stream to the service), rather than the `wsHttpBinding` from:

```
<service behaviorConfiguration="CakeORamaData.Web.Services.CakeServiceBehavior"
  name="CakeORamaData.Web.Services.CakeService">
  <endpoint address="" binding="wsHttpBinding"
  contract="CakeORamaData.Web.Services.ICakeService">
    <identity>
      <dns value="localhost" />
    </identity>
  </endpoint>
  <endpoint address="mex" binding="mexHttpBinding" contract="IM
  etadataExchange" />
</service>
```

To the following:

```
<service behaviorConfiguration="CakeORamaData.Web.Services.
CakeServiceBehavior"
  name="CakeORamaData.Web.Services.CakeService">
  <endpoint address="" binding="customBinding" bindingConfiguratio
n="customBinding0"
    contract="CakeORamaData.Web.Services.ICakeService" />
  <endpoint address="mex" binding="mexHttpBinding" contract="IMeta
dataExchange" />
</service>
```

8. Set the start page to the `CakeService.svc` file, then build and run the solution. We will be presented with the following screen, which lets us know that the service and bindings are set up correctly:

CakeService Service

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using the s

```
svcutil.exe http://localhost:2268/Services/CakeService.svc?wsdl
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your

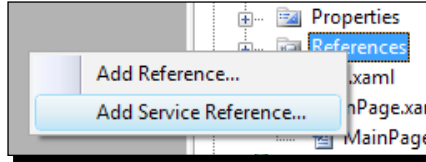
C#

```
class Test
{
    static void Main()
    {
        CakeServiceClient client = new CakeServiceClient();

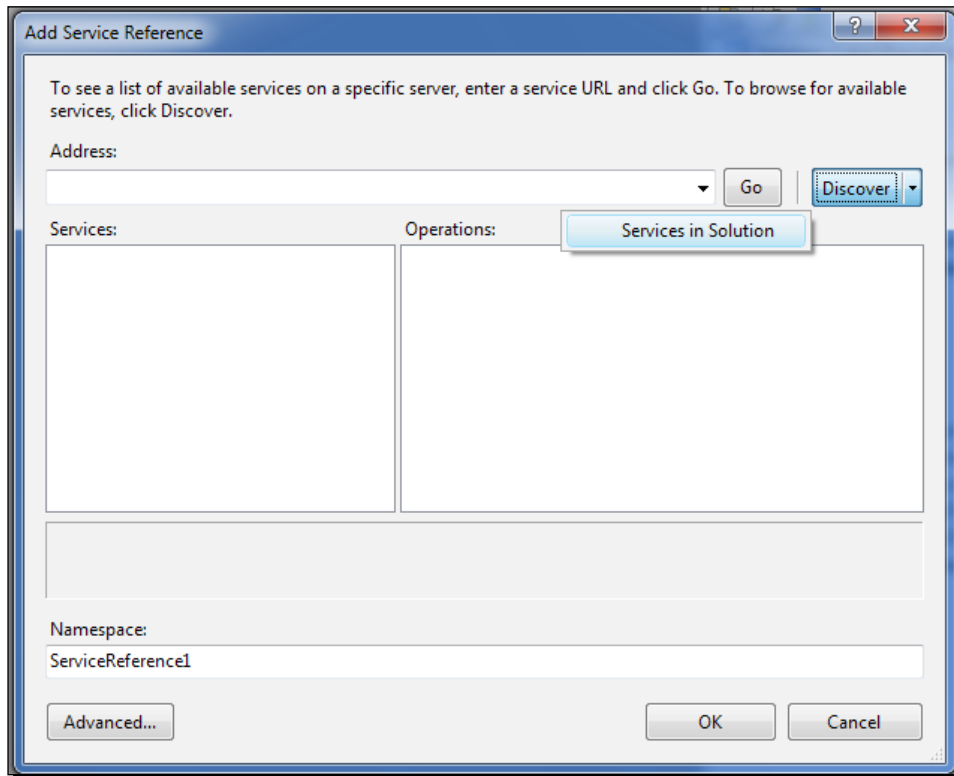
        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

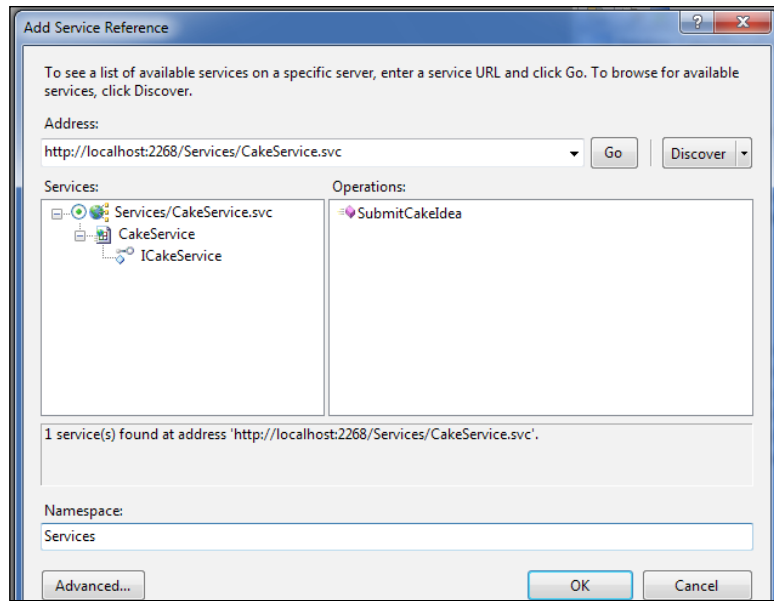
9. Our next step is to add the service reference to Silverlight. On the Silverlight project, right-click on the **References** node and choose to **Add a Service Reference**:



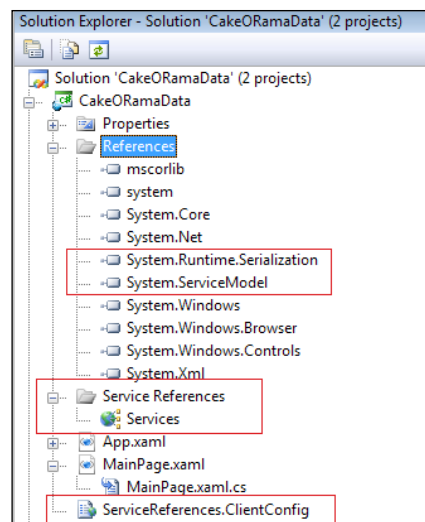
10. On the dialog that opens, click the **Discover** button and choose the **Services in Solution** option. Visual Studio will search the current solution for any services:



- Visual Studio will find our **CakeService** and all we have to do is change the **Namespace** to something that makes sense such as **Services** and click the **OK** button:



- We can see that Visual Studio has added some additional references and files to our project. Developers used to WCF or Web Services will notice the assembly references and the **Service References** folder:



- 13.** Silverlight creates a `ServiceReferences.ClientConfig` file that stores the configuration for the service bindings. If we open this file, we can take a look at the client side bindings to our WCF service. These bindings tell our Silverlight application how to connect to the WCF service and the URL where it is located:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="CustomBinding_ICakeService">
          <binaryMessageEncoding />
          <httpTransport
maxReceivedMessageSize="2147483647" maxBufferSize="2147483647">
            <extendedProtectionPolicy policyEnforcement="Never" />
          </httpTransport>
        </binding>
      </customBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:2268/Services/CakeService.svc"
binding="customBinding" bindingConfiguration="CustomBinding_ICakeService"
contract="Services.ICakeService"
name="CustomBinding_ICakeService" />
    </client>
  </system.serviceModel>
</configuration>
```

What just happened?

We created a Windows Communication Foundation service that is Silverlight ready. In the process, we also followed the best practice guidelines by defining a service interface and a separate implementation. The service accepts a complex data object and writes the data to an XML file.

We included the `AspNetCompatibilityRequirements` attribute to the `CakeService.svc.cs` class which is required in order to host a WCF service from within ASP.NET. We added to the class declaration rather than the interface, because it is implementation-specific and is only valid on class declarations.

We saw how easy it is to create a WCF service and add a service reference to a Silverlight application.

Collecting data

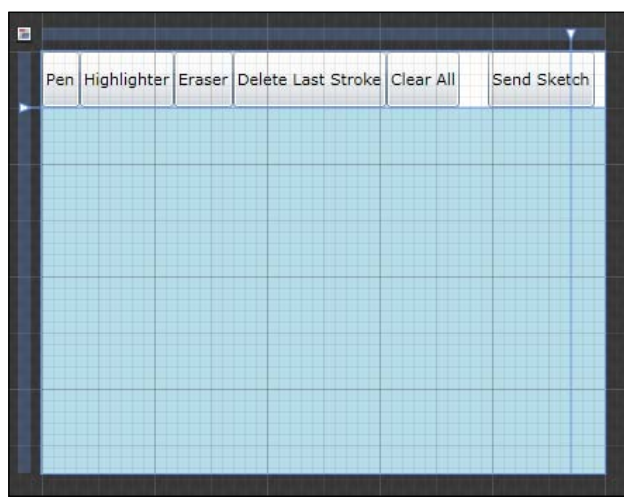
Now that we have created a business object and a WCF service, we are ready to collect data from the customer through our Silverlight application. Silverlight provides all of the standard input controls that .NET developers have come to know with Windows and ASP.NET development, and of course the controls are customizable through styles.

Time for action – creating a form to collect data

We will begin by creating a form in Silverlight for collecting the data from the client. We are going to modify this page to include a submission form to collect the name, phone number, email address, and the date of event for the person submitting the sketch. This will allow the client (Cake O Rama) to contact this individual and follow up on a potential sale.

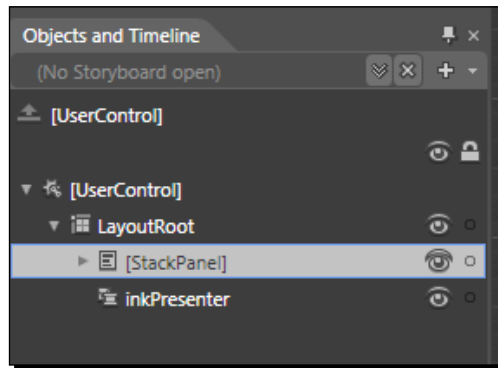
We'll change the layout of `MainPage.xaml` to include a form for user input. We will need to open the **CakeORama** project in Expression Blend and then open `MainPage.xaml` for editing in the Blend art board.

1. Our Ink capture controls are contained within a `Grid`, so we will just add a column to the `Grid` and place our input form right next to the Ink surface. To add columns in Blend, select the `Grid` from the **Objects and Timeline** panel, position your mouse in the highlighted area above the `Grid` and click to add a column:

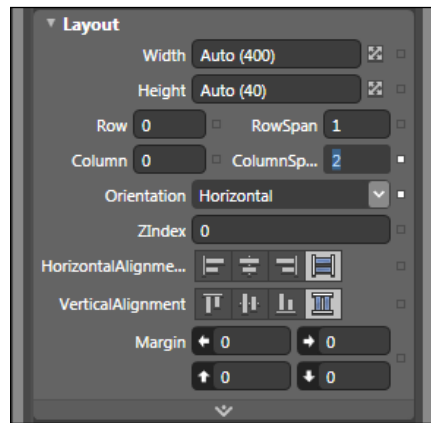


2. Blend will add a `<Grid.ColumnDefinitions>` node to our XAML:

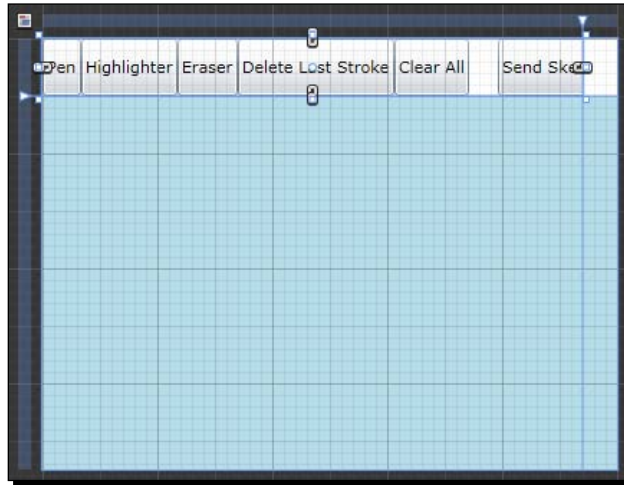
```
<Grid.ColumnDefinitions>
<ColumnDefinition Width="0.94*" />
<ColumnDefinition Width="0.06*" />
</Grid.ColumnDefinitions>
```
3. Blend also added a `Grid.ColumnSpan="2"` attribute to both the `StackPanel` and `InkPresenter` controls that were already on the page.
4. We need to modify the **StackPanel** and **inkPresenter**, so that they do not span both columns and thereby forcing us to increase the size of our second column. In Blend, select the `StackPanel` from the **Objects and Timeline** panel:



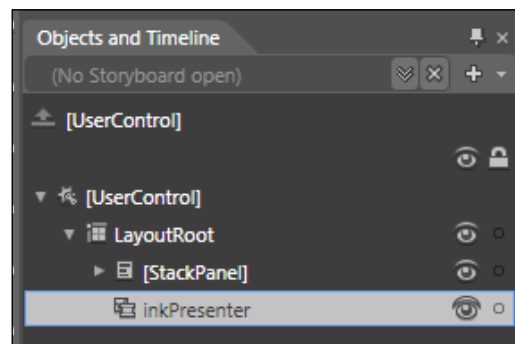
5. In the **Properties** panel, you will see a property called **ColumnSpan** with a value of 2. Change this value to 1 and press the *Enter* key.



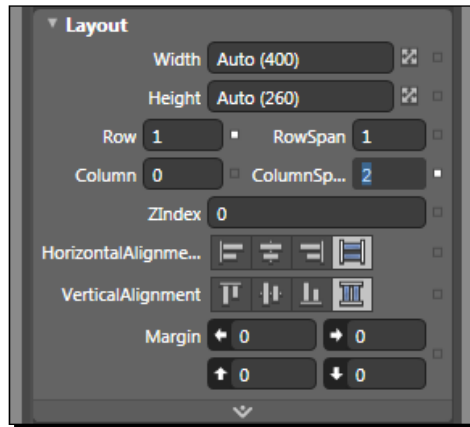
6. We can see that Blend moved the `StackPanel` into the first column, and we now have a little space next to the buttons.



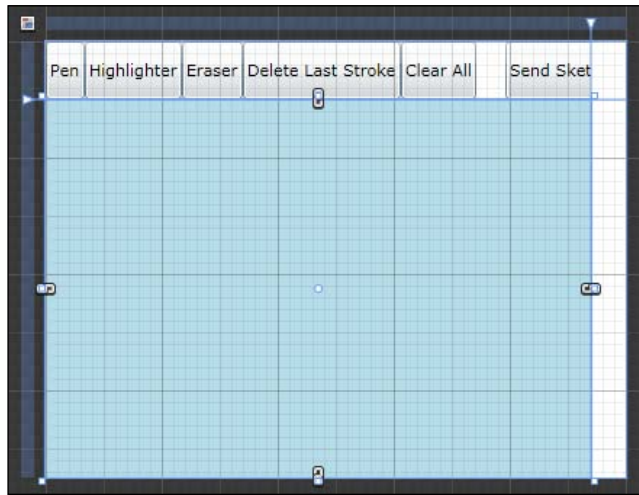
7. We need to do the same thing to the `inkPresenter` control, so that it is also within the first column. Select the `inkPresenter` control from the **Objects and Timeline** panel:



8. Change the **ColumnSpan** from **2** to **1** to reposition the **inkPresenter** into the left column:

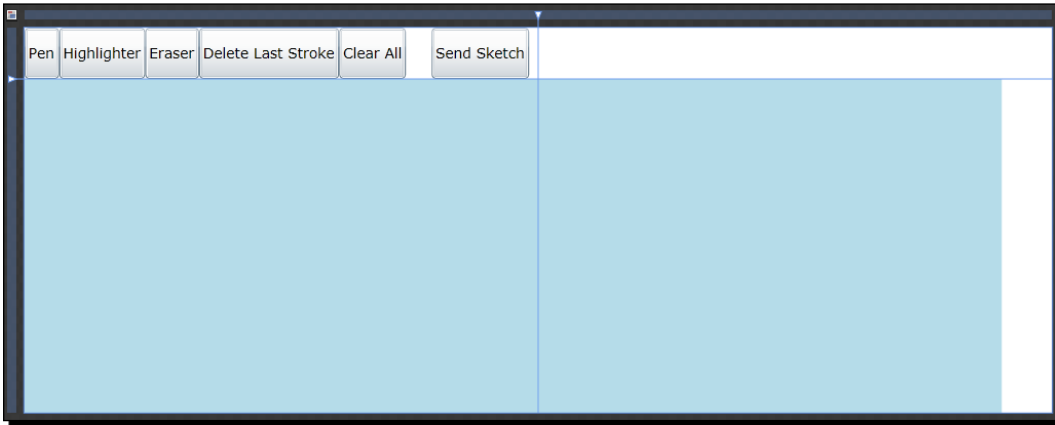


9. The **inkPresenter** control should be positioned in the left column and aligned with the **StackPanel** containing our ink sketch buttons:



10. Now that we have moved the existing controls into the first column, we will change the size of the second column, so that we can start adding our input controls. We also need to change the overall size of the `MainPage.xaml` control to fit more information on the right side of the `ink` control.

11. Click on the [UserController] in the **Objects and Timeline** panel, and then in the **Properties** panel change the **Width** to **800**:



12. Now we need to change the size of our grid columns. We can do this very easily in XAML, so switch to the XAML view in Blend by clicking on the XAML icon:



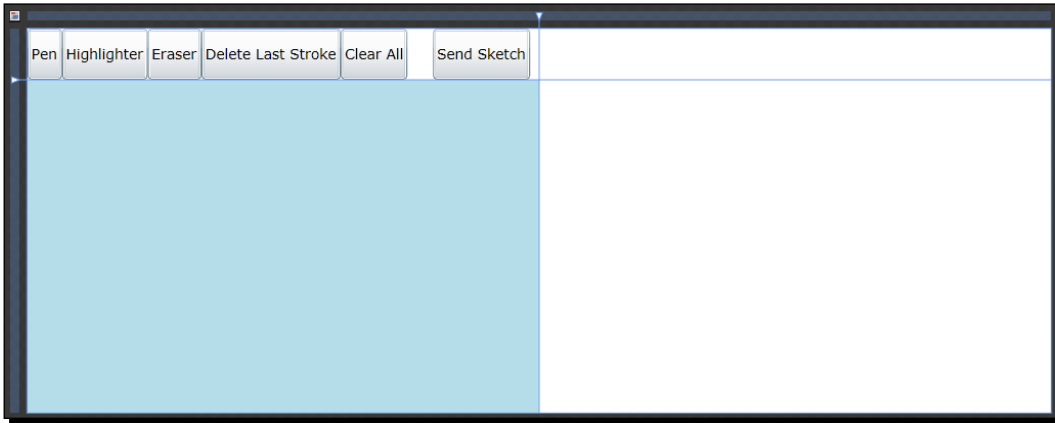
13. In the XAML view, change the grid column settings to give both columns an equal width:

```
<Grid.ColumnDefinitions>  
<ColumnDefinition Width="0.5*" />  
<ColumnDefinition Width="0.5*" />  
</Grid.ColumnDefinitions>
```

14. Switch back to the design view by clicking on the design button:



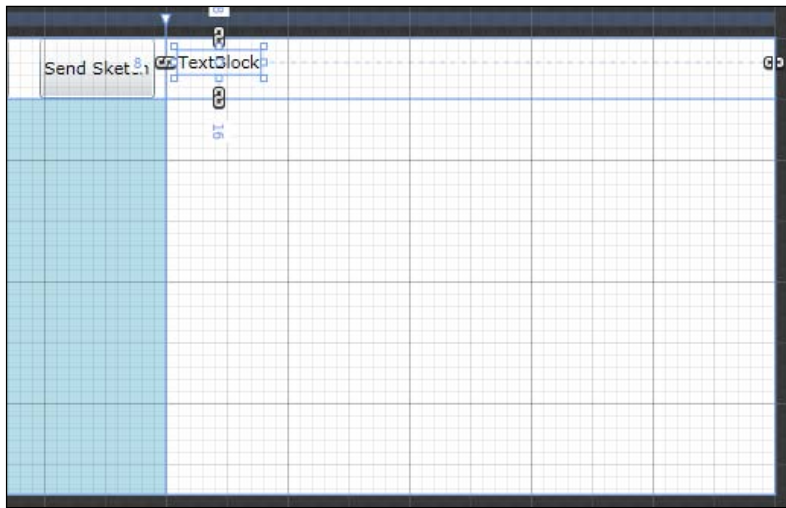
- 15.** Our `StackPanel` and `inkPresenter` controls are now positioned to the left of the page and we have some empty space to the right for our input controls:



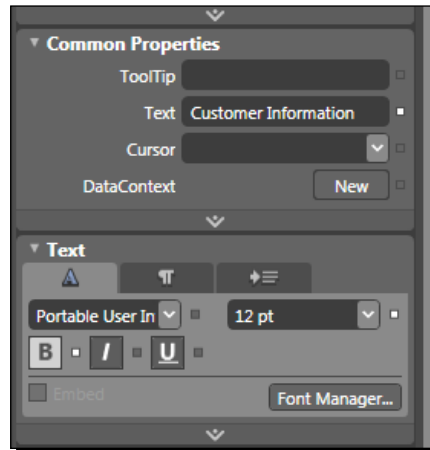
- 16.** Select the `LayoutRoot` control in the **Objects and Timeline** panel and then double-click on the **TextBlock** control in the Blend toolbox to add a new `TextBlock` control:



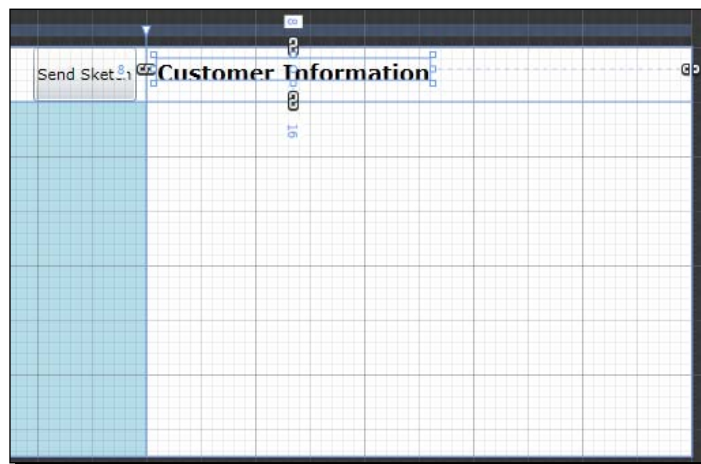
- 17.** Drag the control to the top and right side of the page:



- 18.** On the **Properties** panel, change the **Text** of the `TextBlock` to **Customer Information**, change the `FontSize` to **12pt** and click on the **Bold** indicator:

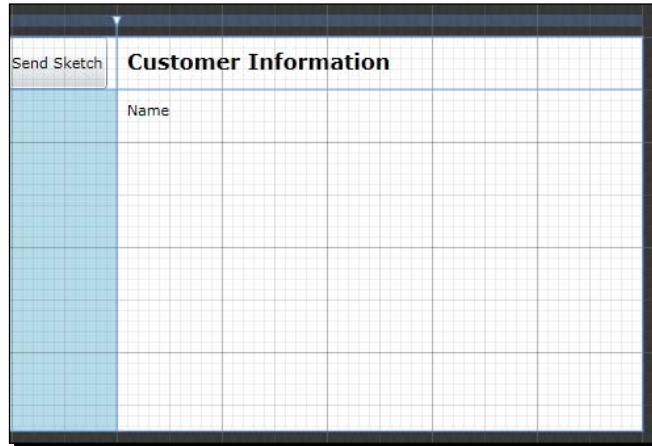


- 19.** The `MainPage.xaml` should look like the following:



- 20.** Double-click the **TextBlock** icon on the toolbox again and drop this into the top-left of column 2, row 2.

- 21.** On the **Properties** panel, change the text of the `TextBlock` to **Name**. This will serve as the label for our `Name` textbox control:

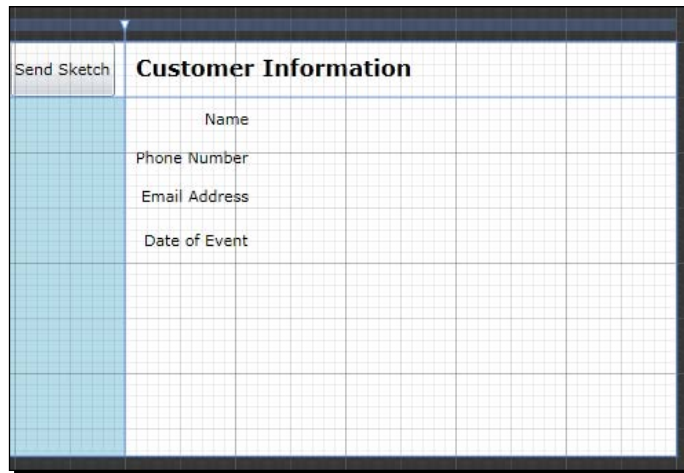


- 22.** Repeat this process, adding **Phone Number**, **Email Address**, and **Date of Event** labels, and rearranging them on the page as illustrated.

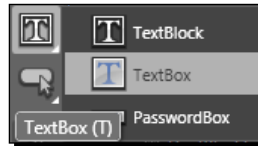


Duplicating Controls

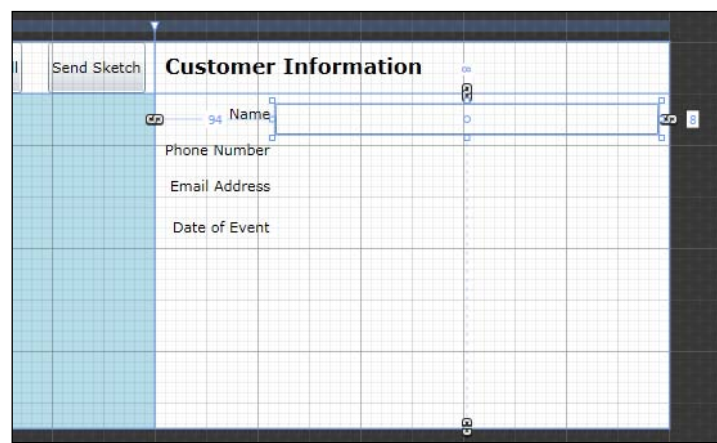
If you click on a control in the **Objects and Timeline** panel, you can make a copy of the control by holding down the *Alt* key, left-click the mouse, and drag the copy into the new position.



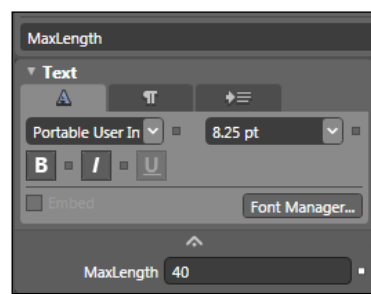
23. Right-click the **TextBlock** icon in the toolbox again and choose the **TextBox** control:



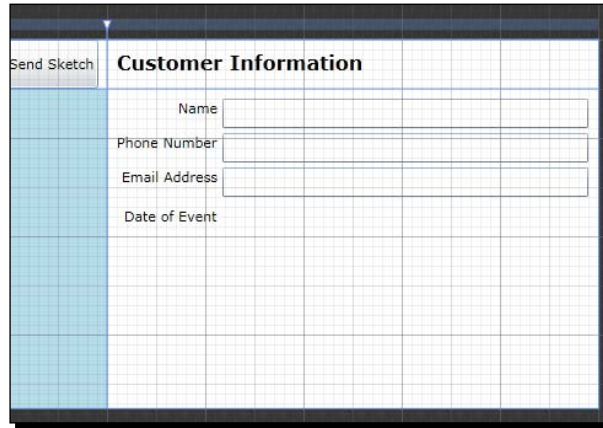
24. Double click the **TextBox** control, which adds a new textbox to the page. Drag this control next to our **Name** label and resize it to maximize the available space:



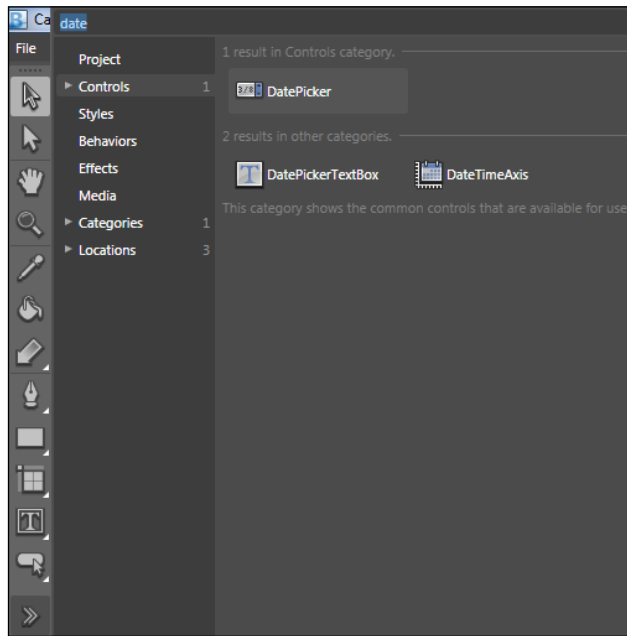
25. Name the textbox **customerName** in the **Properties** panel, and set its **MaxLength** to **40**. The **MaxLength** can be found by typing **MaxLength** in the search field of the **Properties** panel:



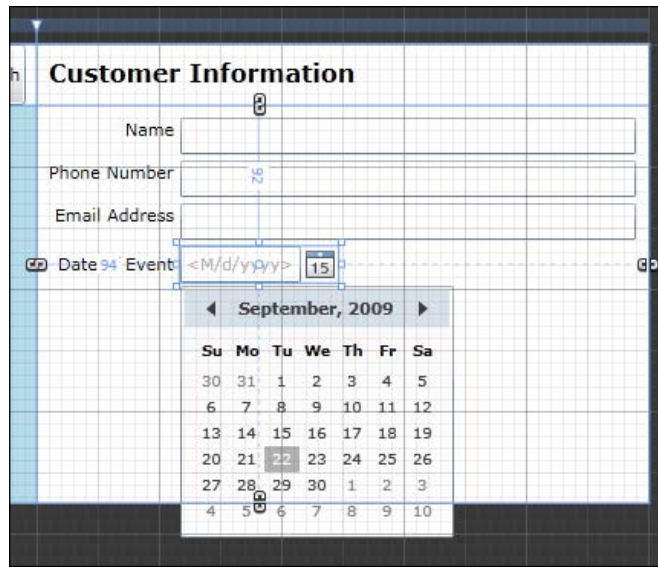
- 26.** Create textbox controls for both the **Phone Number** and **Email Address** fields and name them **phoneNumber** and **emailAddress** respectively; position them on the page next to the appropriate labels. Set the **MaxLength** of the `phoneNumber` field to **15** and the **MaxLength** of the `emailAddress` field to **120**:



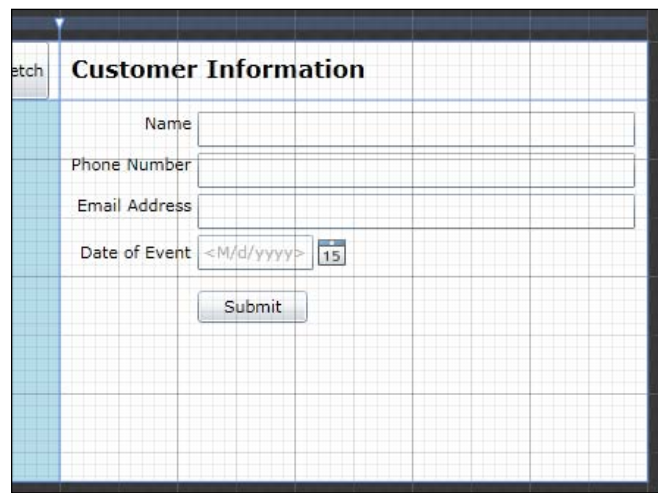
- 27.** To make date entry easier for our users, we will add a **DatePicker** control to our page to allow the user to page through a calendar and select the date of their event. To add a `DatePicker` control, click the **Assets** button, type the word **date** into the search field and select the `DatePicker` control:



- 28.** Double-click on the DatePicker in the toolbox to add it to the page, drag the DatePicker next to the TextBlock label for **Date of Event** and name the control **eventDate**:



- 29.** Add a button control to the page, drag down below the input controls, name the button **submitButton** and change the Content of the control to **Submit**:



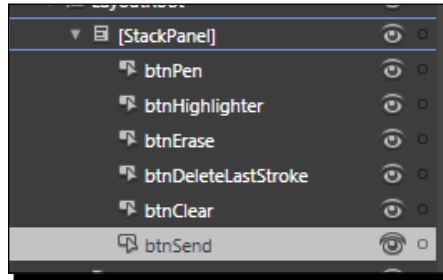
30. Select our **Submit** button and in the **Properties** panel click on the **Events** icon:



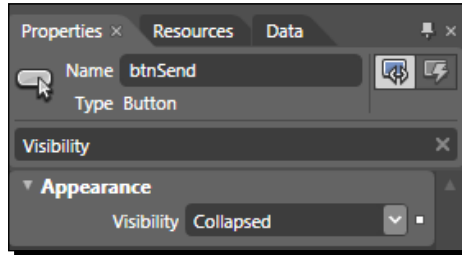
31. Double-click inside of the `Click` event field to have Blend auto create the event handler for the button click event:




32. We added a new **Submit** button, so now we need to hide the **Send Sketch** button. Select the `btnSend` button from the **Objects and Timeline** panel:



33. Set the **Visibility** of the `btnSend` control to **Collapsed**:



[ Be sure to save your work throughout the development process, you would not want to lose all this effort!]

What just happened?

We modified an existing control and added several input controls in order to collect some information from a potential customer. We learned how to add columns to a `Grid` and used `Blend` to create an event handler for our submit button.

By using `Blend`, we are able to set up our input controls very quickly and have visual feedback of our progress the entire time. Hand coding of all this XAML, while possible, is just not what most developers are going to want to spend their time doing, not when there is code to write!

Validating data

With Silverlight, **data validation** has been fully implemented, allowing controls to be bound to data objects and those data objects to handle the validation of data and provide feedback to the controls via the **Visual State Machine**.

The Visual State Machine is a feature of Silverlight used to render to views of a control based on its state. For instance, the mouse over state of a button can actually change the color of the button, show or hide parts of the control, and so on.

Controls that participate in data validation contain a **ValidationStates** group that includes a **Valid**, **InvalidUnfocused**, and **InvalidFocused** states. We can implement custom styles for these states to provide visual feedback to the user.

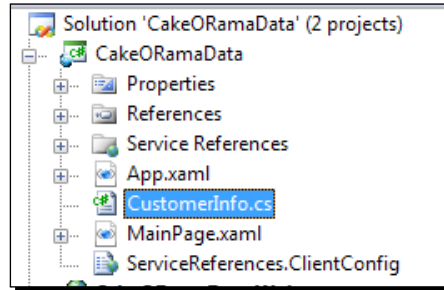
Data object

In order to take advantage of the data validation in Silverlight, we need to create a data object or client side business object that can handle the validation of data.

Time for action – creating a data object

We are going to create a data object that we will bind to our input form to provide validation. Silverlight can bind to any properties of an object, but for validation we need to do two way binding, for which we need both a **get** and a **set** accessor for each of our properties. In order to use two way binding, we will need to implement the `INotifyPropertyChanged` interface that defines a `PropertyChanged` event that Silverlight will use to update the binding when a property changes.

1. Firstly, we will need to switch over to Visual Studio and add a new class named `CustomerInfo` to the Silverlight project:



2. Replace the body of the `CustomerInfo.cs` file with the following code:

```
using System;
using System.ComponentModel;

namespace CakeORamaData
{
    public class CustomerInfo : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged =
        delegate { };

        private string _customerName = null;
        public string CustomerName
        {
            get { return _customerName; }
            set
            {
                if (value == _customerName)
                    return;

                _customerName = value;

                OnPropertyChanged("CustomerName");
            }
        }

        private string _phoneNumber = null;
        public string PhoneNumber
        {
            get { return _phoneNumber; }
            set
```

```
        {
            if (value == _phoneNumber)
                return;

            _phoneNumber = value;

            OnPropertyChanged("PhoneNumber");
        }
    }

    private string _email = null;
    public string Email
    {
        get { return _email; }
        set
        {
            if (value == _email)
                return;

            _email = value;

            OnPropertyChanged("Email");
        }
    }

    private DateTime _eventDate = DateTime.Now.AddDays(7);
    public DateTime EventDate
    {
        get { return _eventDate; }
        set
        {
            if (value == _eventDate)
                return;

            _eventDate = value;

            OnPropertyChanged("EventDate");
        }
    }

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(
            propertyName));
    }
}
```




Code Snippets

Code snippets are a convenient way to stub out repetitive code and increase productivity, by removing the need to type a bunch of the same syntax over and over.

The following is a code snippet used to create properties that execute the `OnPropertyChanged` method and can be very useful when implementing properties on a class that implements the `INotifyPropertyChanged` interface.

To use the snippet, save the file as `propnotify.snippet` to your hard drive.

In Visual Studio go to **Tools | Code Snippets Manager** (*Ctrl + K, Ctrl + B*) and click the **Import** button. Find the `propnotify.snippet` file and click **Open**, this will add the snippet.

To use the snippet in code, simply type **propnotify** and hit the *Tab* key; a property will be stubbed out allowing you to change the name and type of the property.

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippets xmlns="http://schemas.microsoft.com/
VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>propnotify</Title>
      <Shortcut>propnotify</Shortcut>
      <Description>Code snippet for a property that raises
        the PropertyChanged event in a class.</Description>
      <Author>Cameron Albert</Author>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
    </Header>
    <Snippet>
      <Declarations>
        <Literal>
          <ID>type</ID>
          <ToolTip>Property type</ToolTip>
          <Default>int</Default>
        </Literal>
        <Literal>
          <ID>property</ID>
          <ToolTip>Property name</ToolTip>
```

```

        <Default>MyProperty</Default>
    </Literal>
    <Literal>
        <ID>field</ID>
        <ToolTip>Private field</ToolTip>
        <Default>_myProperty</Default>
    </Literal>
    <Literal>
        <ID>defaultValue</ID>
        <ToolTip>Default Value</ToolTip>
        <Default>null</Default>
    </Literal>
</Declarations>
<Code Language="csharp">
    <![CDATA[private $type$ $field$ = $defaultValue$;
public $type$ $property$
    {
        get { return $field$; }
        set
        {
            if (value == $field$)
                return;

            $field$ = value;

            OnPropertyChanged("$property$");
        }
    }
    $end$]]>
</Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>

```

What just happened?

We created a data object or client-side business object that we can use to bind to our input controls.

We implemented the `INotifyPropertyChanged` interface, so that our data object can raise the `PropertyChanged` event whenever the value of one of its properties is changed. We also defined a default delegate value for the `PropertyChanged` event to prevent us from having to do a null check when raising the event. Not to mention we have a nice snippet for stubbing out properties that raise the `PropertyChanged` event.

Now we will be able to bind this object to Silverlight input controls and the controls can cause the object values to be updated so that we can provide data validation from within our data object, rather than having to include validation logic in our user interface code.

Data binding

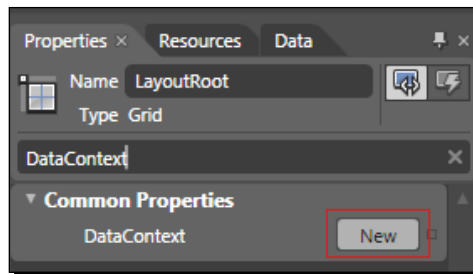
Binding data is one of the most powerful features of .NET Windows and ASP.NET programming, and Silverlight was not left out. Silverlight provides a `Binding` class due to which any property of an object can be bound to any **DependencyProperty** of a control.

Because Silverlight controls are defined in XAML, the `Binding` class can also be defined in XAML using a **Binding Expression**, which is just a XAML way of declaring a `Binding` class.

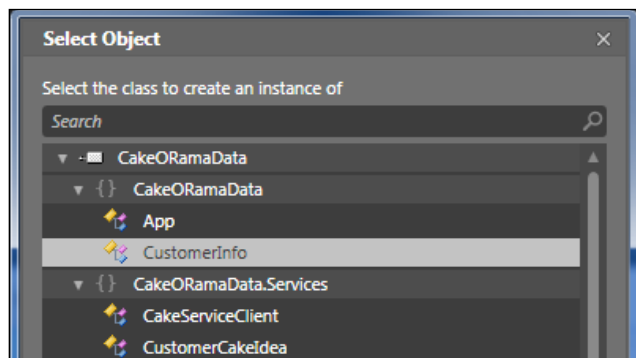
Time for action – binding our data object to our controls

We are going to bind our `CustomerInfo` object to our data entry form, using Blend. Be sure to build the solution before switching back over to Blend.

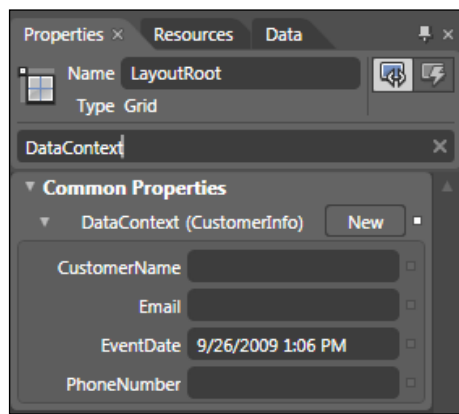
1. With `MainPage.xaml` open in Blend, select the **LayoutRoot** control. In the **Properties** panel enter **DataContext** in the search field and click the **New** button:



2. In the dialog that opens, select the `CustomerInfo` class and click **OK**:



- Blend will set the **DataContext** of the **LayoutRoot** to an instance of a **CustomerInfo** class:

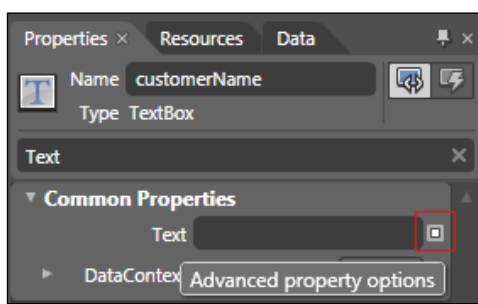


- Blend inserts a namespace to our class; set the `Grid.DataContext` in the XAML of `MainPage.xaml`:

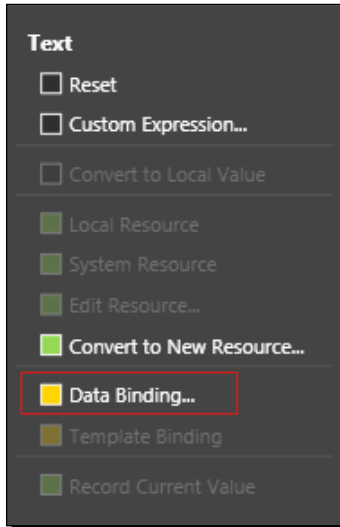
```
xmlns:local="clr-namespace:CakeORamaData"
```

```
<Grid.DataContext>  
  <local:CustomerInfo/>  
</Grid.DataContext>
```

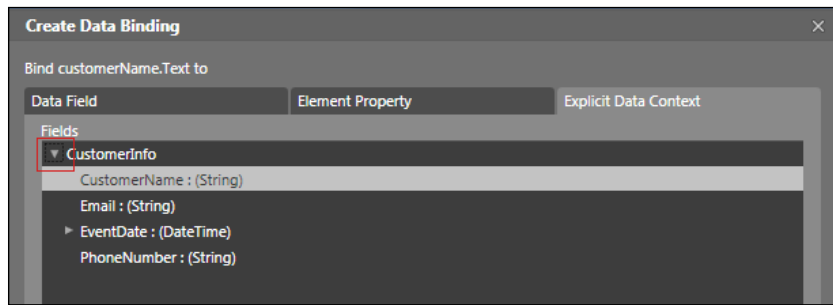
- Now we will bind the value of **CustomerName** to our **customerName** textbox. Select the **customerName** textbox and then on the **Properties** panel enter **Text** in the search field. Click on the **Advanced property options** icon, which will open a context menu for choosing an option:



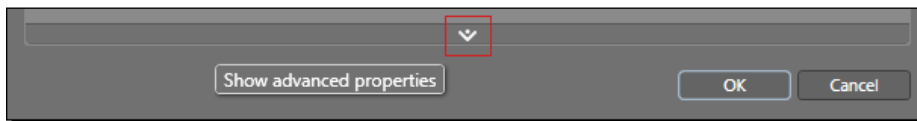
6. Click on the **Data Binding** option to open the **Create Data Binding** dialog:



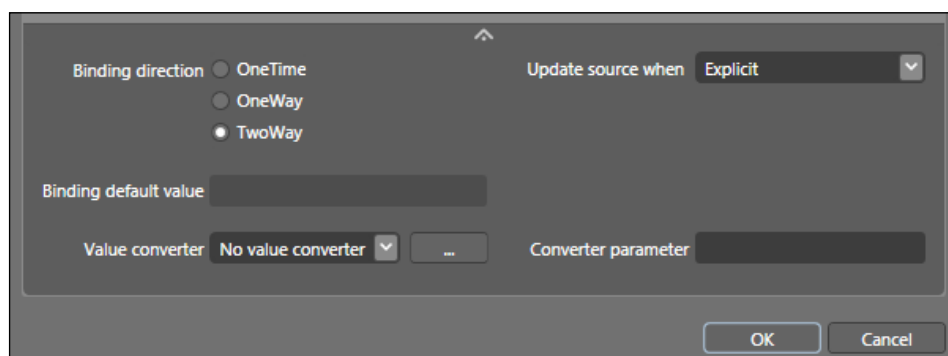
7. In the **Create Data Binding** dialog (on the **Explicit Data Context** tab), click the arrow next to the **CustomerInfo** entry in the **Fields** list and select **CustomerName**:



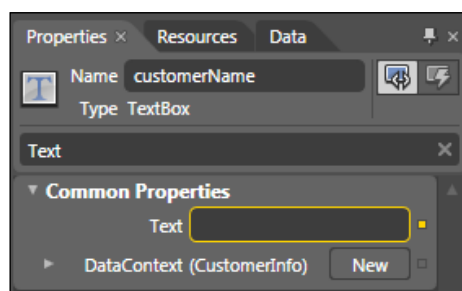
8. At the bottom of the **Create Data Binding** dialog, click on the **Show advanced properties** arrow to expand the dialog and display additional binding options:



9. Ensure that **TwoWay** is selected in the **Binding direction** option and that **Update source when** is set to **Explicit**. This creates a two-way binding, meaning that when the value of the `Text` property of the textbox changes the underlying property, bound to `Text` will also be updated. In our case the **customerName** property of the `CustomerInfo` class:



10. Click **OK** to close the dialog; we can now see that Blend indicates that this property is bound by the yellow border around the property input field:

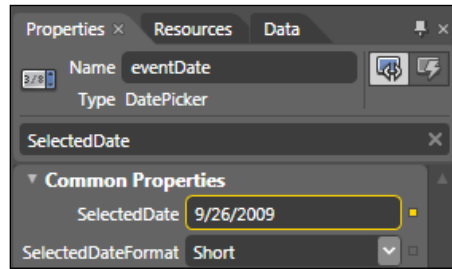


11. Repeat this process for both the **phoneNumber** and **emailAddress** textbox controls, to bind the `Text` property to the **PhoneNumber** and **Email** properties of the `CustomerInfo` class. You will see that Blend has modified our XAML using the Binding Expression:

```
<TextBox x:Name="customerName" Margin="94,8,8,0" Text="{Binding CustomerName, Mode=TwoWay, UpdateSourceTrigger=Explicit}" TextWrapping="Wrap" VerticalAlignment="Top" Grid.Column="1" Grid.Row="1" MaxLength="40"/>
```

12. In the **Binding Expression** code the `Binding` is using the **CustomerName** property as the binding **Path**. The **Path (Path=CustomerName)** attribute can be omitted since the `Binding` class constructor accepts the path as an argument.

- 13.** The **UpdateSourceTrigger** is set to **Explicit**, which causes any changes in the underlying data object to force a re-bind of the control.
- 14.** For the **eventDate** control, enter **SelectedDate** into the **Properties** panel search field and following the same process of data binding, select the **EventDate** property of the `CustomerInfo` class. Remember to ensure that **TwoWay/Explicit** binding is selected in the advanced options:



What just happened?

We utilized Silverlight data binding to bind our input controls to properties of our `CustomerInfo` class. In the process, we setup the binding to be two way, allowing the controls to set the property values of the `CustomerInfo` class, thus removing the need to add a bunch of text changed event handlers to manually do it ourselves, saving us more time in development.

We also had a chance to see how much time using Blend can save and how easy it is to add data bindings to controls. We saw the **Binding Expression** syntax used to define a `Binding` in XAML and also how to setup a `Binding` so that changes to the underlying object cause the control to re-bind the value.

Validation

Before we submit information to the server using our WCF service, we need to validate the data input from the user and provide feedback to the user if invalid information is supplied.

Silverlight can report a validation error in one of three scenarios:

- ◆ Exceptions thrown from the binding type converter
- ◆ Exceptions thrown from the binding object's set accessor
- ◆ Exceptions thrown from one of the validation attributes found in the `DataAnnotations` assembly

We will focus on the `set` accessor method as this provides the simplest way to get our data validated.

Time for action – validating data input

We will make use of some additional properties of `Binding` to allow the controls to display the validation states. Blend does not provide a visual way for us to add these additional properties so we have to do it manually in XAML.

1. Switch to the XAML view of the `MainPage.xaml` in Blend and scroll down to where our textbox controls are located.
2. Within the `Binding Expression` (between the `{` and `}` of the `Binding`), add the following two attributes to each one of the bindings on our input controls:

```
{Binding CustomerName, Mode=TwoWay, UpdateSourceTrigger=Explicit,
NotifyOnValidationError=True, ValidatesOnExceptions=True }
```

3. The `NotifyOnValidationError` and `ValidatesOnException` will both cause the control to display an error message if a validation or exception error occurs when the value of the bound property changes.
4. Now we need to modify our data object to provide validation in the `set` accessor of each property. Change the `CustomerInfo.cs` file to implement our property validation:

```
using System;
using System.ComponentModel;
using System.Text.RegularExpressions;

namespace CakeORamaData
{
    public class CustomerInfo : INotifyPropertyChanged
    {
        private static Regex RegexPhoneNumber = new Regex(@"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}", RegexOptions.Multiline);
        private static Regex RegexEmail = new Regex(@"^([\w\-\.\.]+)@((\[[0-9]{1,3}\.]{3}[0-9]{1,3}\)|(([\w\-\.\.]+) ([a-zA-Z]{2,4}))$)", RegexOptions.Multiline | RegexOptions.IgnoreCase);

        public event PropertyChangedEventHandler PropertyChanged =
            delegate { };

        private string _customerName = null;
        public string CustomerName
        {
            get { return _customerName; }
            set
            {
```



```
        if (value == _customerName)
            return;

        if (String.IsNullOrEmpty(value))
            throw new ArgumentException("Customer Name is
            required.");

        if (value.Length < 3 || value.Length > 40)
            throw new ArgumentException("Customer Name must be at
            least 3 characters and not more than 40 characters
            in length.");

        _customerName = value;

        OnPropertyChanged("CustomerName");
    }
}

private string _phoneNumber = null;
public string PhoneNumber
{
    get { return _phoneNumber; }
    set
    {
        if (value == _phoneNumber)
            return;

        if (String.IsNullOrEmpty(value))
            throw new ArgumentException("Phone Number is
            required.");

        if (!RegexPhoneNumber.IsMatch(value))
            throw new ArgumentException("A valid phone number in the
            format (XXX) XXX-XXXX or XXX-XXX-XXXX is required.");

        _phoneNumber = value;

        OnPropertyChanged("PhoneNumber");
    }
}

private string _email = null;
public string Email
{
    get { return _email; }
}
```

```
        set
        {
            if (value == _email)
                return;

            if (String.IsNullOrEmpty(value))
                throw new ArgumentException("Email Address is
                required.");

            if (!RegexEmail.IsMatch(value))
                throw new ArgumentException("A valid email address is
                required.");

            _email = value;

            OnPropertyChanged("Email");
        }
    }

    private DateTime _eventDate = DateTime.Now.AddDays(7);
    public DateTime EventDate
    {
        get { return _eventDate; }
        set
        {
            if (value == _eventDate)
                return;

            _eventDate = value;

            OnPropertyChanged("EventDate");
        }
    }

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged(this, new PropertyChangedEventArgs
        (propertyName));
    }
}
```

5. Open the `MainPage.xaml.cs` file and in the constructor add the following code to set the `LayoutRoot.DataContext` with a new instance of `CustomerInfo`:

```
public MainPage()
{
    this.Loaded += new RoutedEventHandler(MainPage_Loaded);
    InitializeComponent();
}

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    LayoutRoot.DataContext = new CustomerInfo();
}
```

6. Also within the `MainPage.xaml` file in the `submitButton_Click` event handler, we will add code to force validation of our data object:

```
private void submitButton_Click(object sender, System.Windows.
RoutedEventArgs e)
{
    var bindingExpression = customerName.GetBindingExpression(TextBo
x.TextProperty);
    bindingExpression.UpdateSource();

    bindingExpression = phoneNumber.GetBindingExpression(TextBox.
TextProperty);
    bindingExpression.UpdateSource();

    bindingExpression = emailAddress.GetBindingExpression(TextBox.
TextProperty);
    bindingExpression.UpdateSource();
}
```

7. In Visual Studio, choose **Debug | Start without Debugging** from the file menu. We are not going to debug because our properties throw exceptions and we just want to see the result. Just click the **Submit** button and all the textbox controls will highlight with red borders:



The screenshot shows a window titled "Customer Information" with a white background and a black border. It contains four input fields: "Name", "Phone Number", "Email Address", and "Date of Event". The "Date of Event" field has a calendar icon and shows "10/3/2009" with a "15" in a small box. Below the fields is a "Submit" button. All four input fields have a red border around them, indicating validation errors.

8. If you hover over the small arrow in the top-right corner of the textbox you will see the error message from the data object:

The screenshot shows a form titled "Customer Information" with the following fields: Name, Phone Number, Email Address, and Date of Event. The Name field is empty and has a red border. A red error message "Customer Name is required." is displayed next to it. The Date of Event field is set to 10/3/2009. A "Submit" button is at the bottom.

What just happened?

We implemented simple data validation in our objects and let the built in Silverlight binding process handle the rest by including some additional attributes in the `Binding Expression`. We implemented the `INotifyPropertyChanged` interface in our data object so that the data will be re-bound whenever the values are changed. We also made use of regular expressions to ensure that the phone number and email address are in a valid format.

Data submission

Data collected from users does not provide a benefit unless the user can submit it and we can store the information for later retrieval. The ability to analyze and report on the data is how businesses acquire and maintain clients and customers, which is where the profits are derived from.

Time for action – submitting data to the server

Now that we have setup a form for data input and validated the data, we can now submit the data to the server using our WCF service. We need to submit the information to the server in order for the sales staff of Cake O Rama to be able to review and contact the customer.

1. Switch back over to Visual Studio, open the `MainPage.xaml.cs` file and then add the following to the `using` statements:

```
using CakeORamaData.Services;
```
2. At the bottom of this file add the `ConvertStrokesToStrokeInfoArray` method. This method will convert the Silverlight `Stroke` objects from the `inkPresenter` to `StrokeInfo` objects as defined by our WCF service:

```
private ObservableCollection<StrokeInfo>
ConvertStrokesToStrokeInfoArray()
{
```

```
var strokeCollection = new ObservableCollection<StrokeInfo>();

foreach (Stroke stroke in this.inkPresenter.Strokes)
{
    var strokeInfo = new StrokeInfo
    {
        Width = stroke.DrawingAttributes.Width,
        Height = stroke.DrawingAttributes.Height,
        Color = new byte[]
        {
            stroke.DrawingAttributes.Color.A,
            stroke.DrawingAttributes.Color.R,
            stroke.DrawingAttributes.Color.G,
            stroke.DrawingAttributes.Color.B
        },
        OutlineColor = new byte[]
        {
            stroke.DrawingAttributes.OutlineColor.A,
            stroke.DrawingAttributes.OutlineColor.R,
            stroke.DrawingAttributes.OutlineColor.G,
            stroke.DrawingAttributes.OutlineColor.B
        }
    };
    strokeCollection.Add(strokeInfo);

    var pointCollection = new ObservableCollection
        <StylusPointInfo>();
    strokeInfo.Points = pointCollection;
    foreach (StylusPoint point in stroke.StylusPoints)
    {
        var pointInfo = new StylusPointInfo
        {
            X = point.X,
            Y = point.Y
        };
        pointCollection.Add(pointInfo);
    }
}
return strokeCollection;
}
```



Note here that when we added a reference to the WCF service, our `StrokeInfo[]` array on the `CustomerCakeIdea` object was converted to a `System.Collections.ObjectModel.ObservableCollection<StrokeInfo>` by Silverlight.

3. Go to the `submitButton_Click` method and modify it to resemble the following code:

```
private void submitButton_Click(object sender, System.Windows.
RoutedEventArgs e)
{
    var bindingExpression = customerName.GetBindingExpression(TextBo
x.TextProperty);
    bindingExpression.UpdateSource();

    bindingExpression = phoneNumber.GetBindingExpression(TextBox.
TextProperty);
    bindingExpression.UpdateSource();

    bindingExpression = emailAddress.GetBindingExpression(TextBox.
TextProperty);
    bindingExpression.UpdateSource();

    if (!Validation.GetHasError(customerName)
        && !Validation.GetHasError(phoneNumber)
        && !Validation.GetHasError(emailAddress))
    {
        var info = LayoutRoot.DataContext as CustomerInfo;

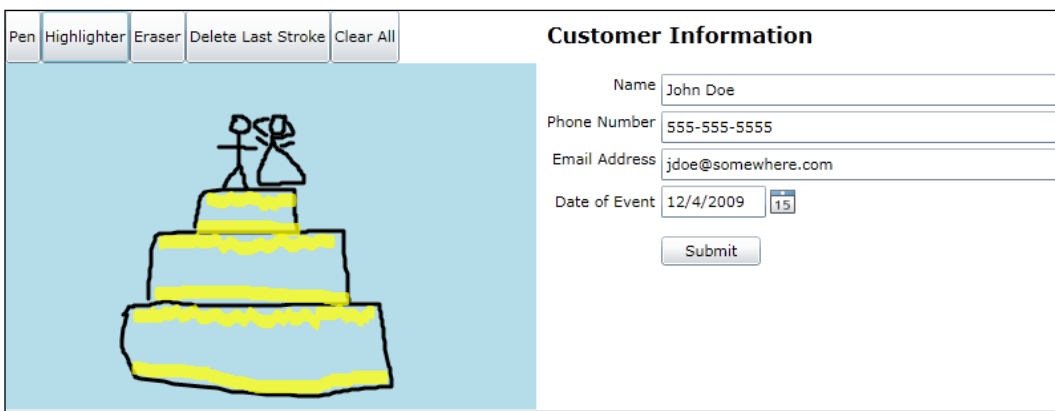
        var idea = new CustomerCakeIdea
        {
            CustomerName = info.CustomerName,
            PhoneNumber = info.PhoneNumber,
            Email = info.Email,
            EventDate = info.EventDate,
            Strokes = ConvertStrokesToStrokeInfoArray()
        };

        var client = new CakeServiceClient();
        client.SubmitCakeIdeaCompleted += new EventHandler
        <AsyncCompletedEventArgs>(OnCakeIdeaSubmissionComplete);
        client.SubmitCakeIdeaAsync(idea);
    }
}
```

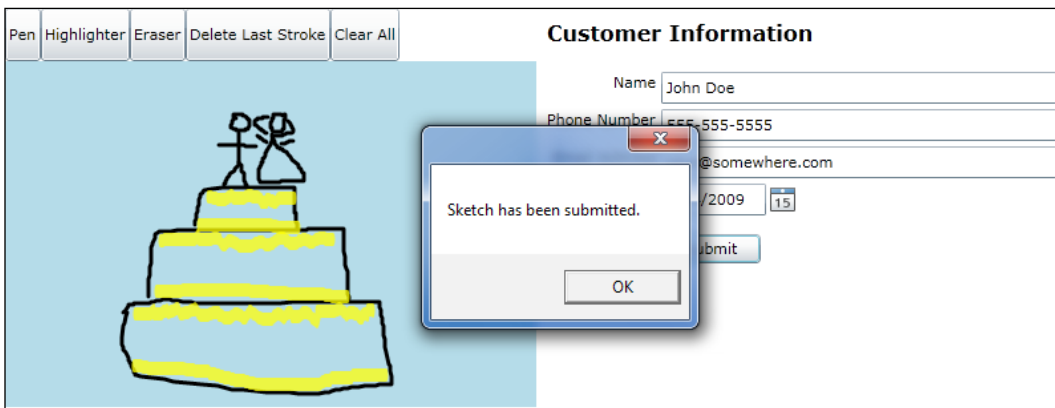
4. Add the following method to handle the `SubmitCakeIdeaCompleted` event to display a `MessageBox` once the submission is complete:

```
private void OnCakeIdeaSubmissionComplete(object sender, AsyncCompletedEventArgs e)
{
    MessageBox.Show("Sketch has been submitted.");
}
```

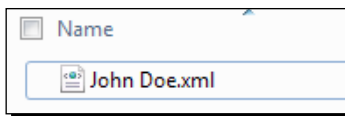
Now we will test out our cake idea submission form and process. Build and run the solution in Visual Studio and when the Silverlight application loads in the browser input some information and draw a cake sketch:



5. When we submit the information to the server, we will get a `MessageBox` telling us that we submitted the information, as shown in the next screenshot:



6. Open **Windows Explorer** and navigate to the path that we setup in the WCF service for storing the customer XML files, and open the newly submitted file. We should now have the data from the cake sketch and customer information in our XML file.



If we open the XML file, we should see the saved customer and ink stroke information:

```
<?xml version="1.0" encoding="utf-8"?>
<customer name="John Doe" phone="555-555-5555" email="jdoe@
somewhere.com">
  <eventDate>2009-10-04T16:03:41.0966771-04:00</eventDate>
  <strokes>
    <stroke width="3" height="3">
      <color a="255" r="0" g="0" b="0" />
      <outlineColor a="0" r="0" g="0" b="0" />
      <points>
        <point x="92" y="189" />
        <point x="91" y="192" />
        <point x="90" y="197" />
        <point x="89" y="199" />
        <point x="88" y="208" />
        <point x="88" y="210" />
        <point x="88" y="212" />
        <point x="88" y="213" />
      </points>
    </stroke>
  </strokes>
</customer>
```

What just happened?

We placed code in the `MainPage.xaml.cs` file to ensure that all of our text input controls did not have any validation errors, by making use of the `Validation` class.

We made use of the `CustomerCakeIdea` business object to store the customer input and ink stroke data and sent that information to the server via the WCF service, where we saved the information to an XML file for later use by the sales staff. We used an anonymous delegate to handle the asynchronous response from the WCF service and utilized a message box to inform the user of the successful submission.

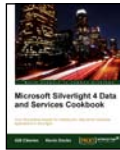
Summary

In this chapter, we covered the process of collecting and handling data input from a customer and saving that input on the server. We also looked at how to bind data to control properties and how to provide simple data validation using the built in visual states provided in the textbox control. We discussed the following:

- ◆ How to create a Windows Communication Foundation service
- ◆ How to mark a business object for serialization in WCF
- ◆ How to create an input form in Silverlight
- ◆ How to create a data object for use with binding
- ◆ How to bind data from a data object to Silverlight controls
- ◆ How to provide input validation using the built-in validation states
- ◆ How to consume a WCF in Silverlight and process an asynchronous request

3

An Introduction to Data Binding



This chapter is taken from *Silverlight 4 Data and Services Cookbook* (Chapter 2) by Gill Cleeren, Kevin Dockx.

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
- ▶ Data binding from Expression Blend 4
- ▶ Using Expression Blend 4 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 takes a look at how data binding works. We'll start by looking at the general concepts of data binding in Silverlight 4 in this chapter.

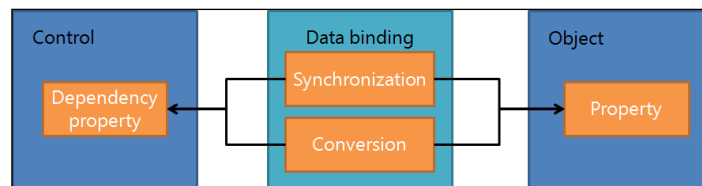
Analyzing the term **data binding** immediately reveals its intentions. It is a technique that allows us to bind properties of controls to objects or collections thereof.

The concept is, in fact, not new. Technologies such as ASP.NET, Windows Forms, and even older technologies such as MFC (Microsoft Foundation Classes) 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 to which property of the source the target should bind. And 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 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**.

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 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). We'll finish this chapter by looking at the support that Blend 4 provides to build applications that use data binding features.

In the recipes of this chapter, 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 screenshot shows the resulting Silverlight banking application:

Silverlight Bank - Account overview

Actions [Calculate loans...](#)

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
[Edit details...](#)

Current balance: 1221.56
Last activity on: 8/7/2009 9:43:34 PM
Amount: -13

Activities on the account:

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

If you want to take a look at the complete application, run the solution found in the `Chapter03/SilverlightBanking` folder in the code bundle that is available on the Packt website.

Displaying data in Silverlight applications

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 to write data-driven applications faster and using less code. In this recipe, we'll build a form that displays the data of the owner of a bank account using data binding.

Getting ready

To follow along with this recipe, you can use the starter solution located in the `Chapter03/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 `Chapter03/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.

```
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);
}
```



```
        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 x:Name="AddressTextBlock"
    Grid.Row="4"
    FontWeight="Bold"
    Margin="2"
    Text="Adress:">
</TextBlock>
<TextBlock x:Name="ZipCodeTextBlock"
    Grid.Row="5"
    FontWeight="Bold"
    Margin="2"
    Text="Zip code:">
</TextBlock>
<TextBlock x:Name="CityTextBlock"
    Grid.Row="6"
    FontWeight="Bold"
    Margin="2"
    Text="City:">
</TextBlock>
<TextBlock x:Name="StateTextBlock"
    Grid.Row="7"
    FontWeight="Bold"
    Margin="2"
    Text="State:">
</TextBlock>
<TextBlock x:Name="CountryTextBlock"
    Grid.Row="8"
    FontWeight="Bold"
    Margin="2"
    Text="Country:">
```



```
</TextBlock>
<TextBlock x:Name="BirthDateTextBlock"
           Grid.Row="9"
           FontWeight="Bold"
           Margin="2"
           Text="Birthdate:">
</TextBlock>
<TextBlock x:Name="CustomerSinceTextBlock"
           Grid.Row="10"
           FontWeight="Bold"
           Margin="2"
           Text="Customer since:">
</TextBlock>
<TextBlock x:Name="OwnerIdValueTextBlock"
           Grid.Row="1"
           Grid.Column="1"
           Margin="2"
           Text="{Binding OwnerId}">
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
           Grid.Row="2"
           Grid.Column="1"
           Margin="2"
           Text="{Binding FirstName}">
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
           Grid.Row="3"
           Grid.Column="1"
           Margin="2"
           Text="{Binding LastName}">
</TextBlock>
<TextBlock x:Name="AddressValueTextBlock"
           Grid.Row="4"
           Grid.Column="1"
           Margin="2"
           Text="{Binding Address}">
</TextBlock>
<TextBlock x:Name="ZipCodeValueTextBlock"
           Grid.Row="5"
           Grid.Column="1"
           Margin="2"
           Text="{Binding ZipCode}">
</TextBlock>
<TextBlock x:Name="CityValueTextBlock"
           Grid.Row="6"
           Grid.Column="1"
```

```
        Margin="2"
        Text="{Binding City}">
</TextBlock>
<TextBlock x:Name="StateValueTextBlock"
    Grid.Row="7"
    Grid.Column="1"
    Margin="2"
    Text="{Binding State}">
</TextBlock>
<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>
<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;
}
```

The result can be seen in the following screenshot:



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 can be seen, 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's 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` is 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`. Silverlight 4 has solved this problem.
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 as 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 this:

```
<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`:

```
<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 code-behind. In this case, binding from XAML isn't possible.

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

In the previous recipe, you've 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 `Chapter03/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 `Chapter03/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"));
```

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 screenshot:



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 which explain all the possibilities in detail. They are, however, already mentioned in this recipe to make it clear you can do everything that is required as far as bindings are concerned in both C# and XAML.

Binding data to another UI element

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 bidirectionally.

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) a lot of 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 `Chapter03/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 `Chapter03/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 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 **Calculate** button 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 creating 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 vice versa. This can be achieved by specifying the `Mode` property on the `Binding` and setting it to `TwoWay`.

The following is the code for this. In this code, 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,
    RoutedEventArgs 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 *Using the different modes of data binding to allow persisting data* recipe in this chapter.

Binding collections to UI elements

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 a bunch of 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 `Chapter03/SilverlightBanking_Binding_Collections_Starter` folder and the completed solution in the `Chapter03/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, that is, `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.

```
<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>
```



```
        <ColumnDefinition Width="330">
        </ColumnDefinition>
        <ColumnDefinitionWidth="100">
        </ColumnDefinition>
    </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 screenshot:

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

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 steps 4 and 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. So, the `DataTemplate` corresponds to one item of your collection: 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 *Enabling a Silverlight application to automatically update its UI* recipe.

The same principle applies for the properties of classes that implement the `INotifyPropertyChanged` interface. More on this can be found in the same recipe, that is, *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 5, The Data Grid*.

See also

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

Enabling a Silverlight application to automatically update its UI

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, that is, **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 direction(s)—using the mode of data binding.

The synchronization works for both single objects bound to the UI as well as entire collections. But 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 `Chapter03/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 `Chapter03/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. The 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 `INotifyPropertyChanged` interface. This interface defines one event, that is, the `PropertyChanged` event. Whenever one of the properties changes, 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 : INotifyPropertyChanged
{
    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 values can be seen in the following screenshot:

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

6. In the *Binding collections to UI elements* recipe, 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 if 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. (For more information on binding modes, refer to the *Using the different modes of data binding to allow persisting data* recipe.) If we had set the binding to `Mode=OneTime`, then only the initial values would have been loaded.

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. Similar to `INotifyPropertyChanged`, an interface 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 on 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"));
    }
}
```

It's a good idea to move the check whether the event is null (which means that there is no one actually subscribed to the event) and the raising of the event 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

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 in 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 `Chapter03/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 `Chapter03/SilverlightBanking_Obtaining_Data_Completed` folder.


```
<TextBlock x:Name="AmountTextBlock"
  Grid.Row="2"
  FontWeight="Bold"
  Margin="2"
  Text="Amount:" >
</TextBlock>
<TextBlock x:Name="ActivityDateTextBlock"
  Grid.Row="3"
  FontWeight="Bold"
  Margin="2"
  Text="Date:" >
</TextBlock>
<TextBlock x:Name="DescriptionTextBlock"
  Grid.Row="4"
  FontWeight="Bold"
  Margin="2"
  Text="Description:" >
</TextBlock>
<TextBlock x:Name="ActivityIdTextBlockValue"
  Grid.Row="0"
  Grid.Column="1"
  Margin="2"
  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;
}
private void btnOK_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
}
```

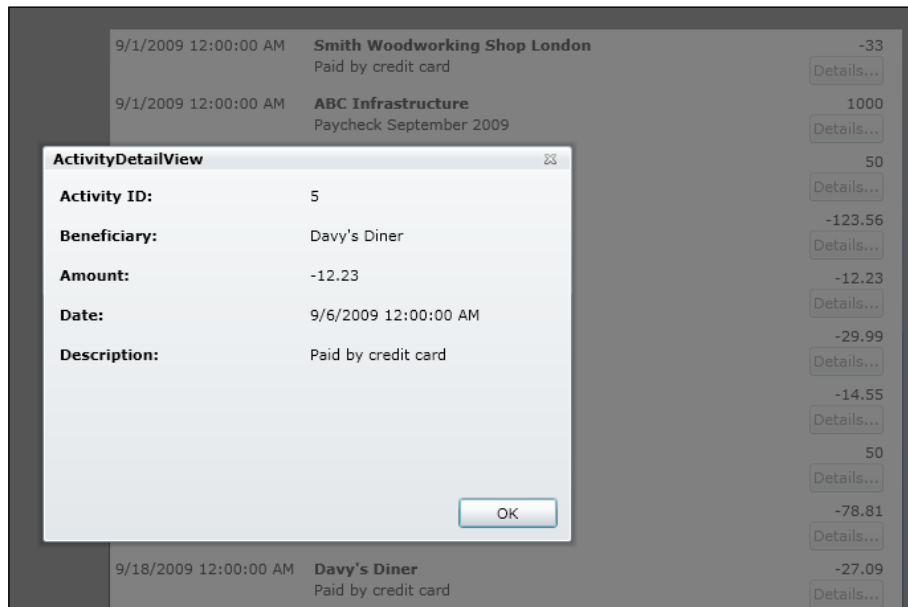
3. Now, we open `MainPage.xaml`, locate the `ListBox` named `AccountActivityListBox`, and add a button named `btnDetails` to the `DataTemplate` of that `ListBox`. This is shown in the following code:

```
<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 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 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`. To be more precise, if the `DataContext` of an item hasn't been set, then 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 trickling 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 `TextBlocks`, 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 `TextBlocks`, 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 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

Until now, the data has flowed from the source to the target (the UI controls). However, it can also flow in the opposite direction, that is, from the target towards the source. This way, not only can data binding help us in displaying data, but also in **persisting data**.

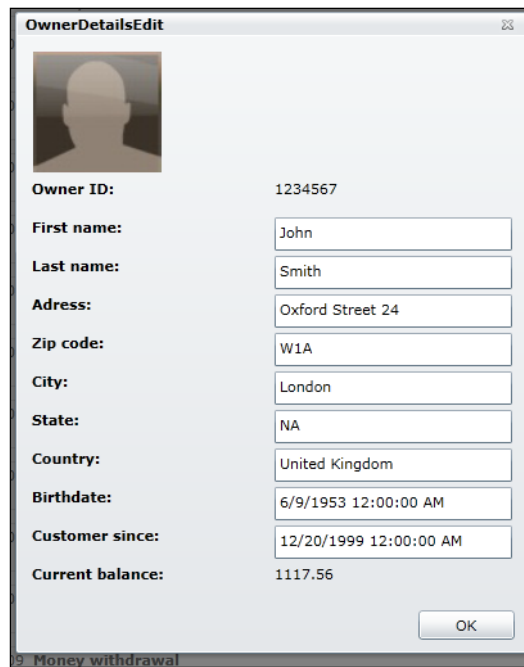
The direction of the flow of data in a data binding scenario is controlled by the `Mode` property of the `Binding`. In this recipe, we'll look at an example that uses all the `Mode` options and in one go, we'll push the data that we enter ourselves to the source.

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 `Chapter03/SilverlightBanking_Binding_Modes_Starter` folder in the code bundle that is available on the Packt website. The `Chapter03/SilverlightBanking_Binding_Modes_Completed` folder contains the finished application of this recipe.

How to do it...

In this recipe, we'll build the "edit details" window of the `Owner` class. On 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 a screenshot 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 `MainPage.xaml`, add a `Click` event on the `OwnerDetailsEditButton`:

```
<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 (Chapter03/ `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>
```

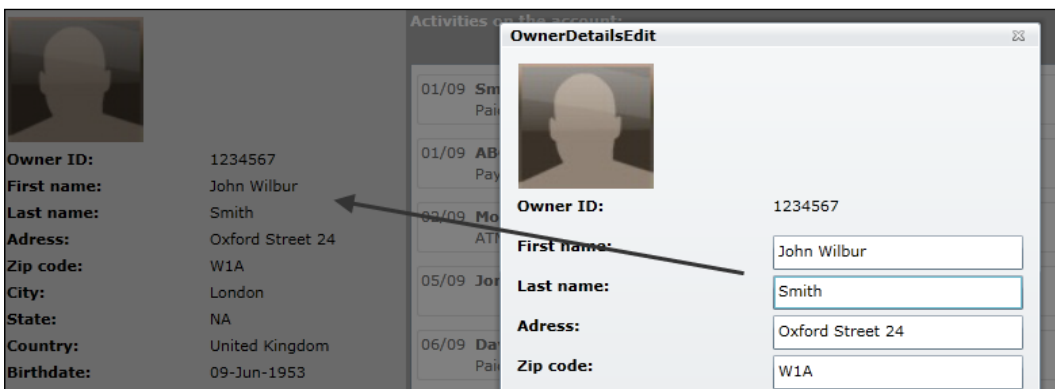
8. The `CurrentBalance` `TextBlock` at the bottom does not need to be editable by the user (allowing a user to change his or her account balance might not be beneficial for the bank), but it does need to change when the source changes. This is the automatic synchronization working for us and it is achieved 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 are in the background bound to the same source object as shown in the following screenshot:



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 the 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; and 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.

```
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.

Data binding from Expression Blend 4

While creating data bindings is probably a task mainly reserved for the developer(s) in the team, Blend 4—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 4 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 4 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 4:

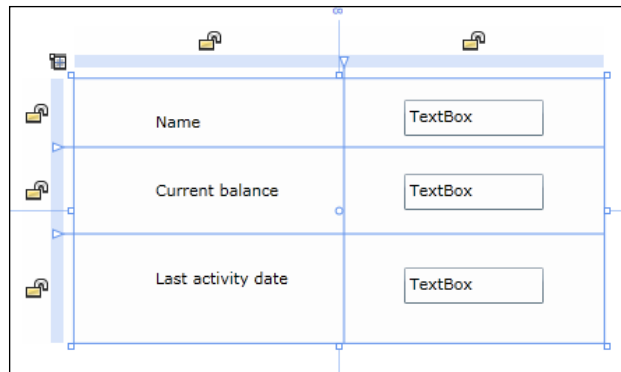
```
public class Owner
{
    public string Name {get; set;}
    public int CurrentBalance {get;set;}
    public DateTime LastActivityDate {get;set;}
}
```

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.

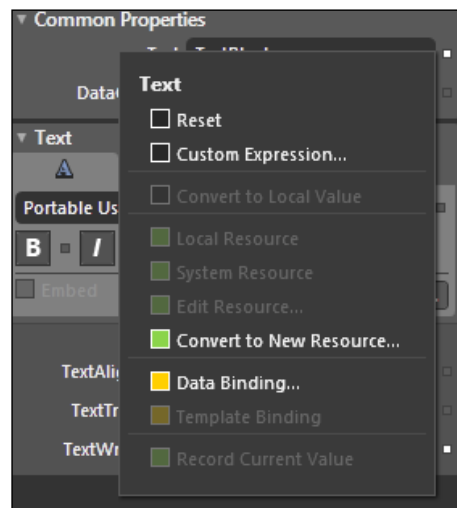
```
public partial class MainPage : UserControl
{
    public Owner owner;
    public MainPage()
    {
        // Required to initialize variables
        InitializeComponent();
        owner = new Owner()
        {
            Name="Gill Cleeren",
            CurrentBalance=300,
```

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

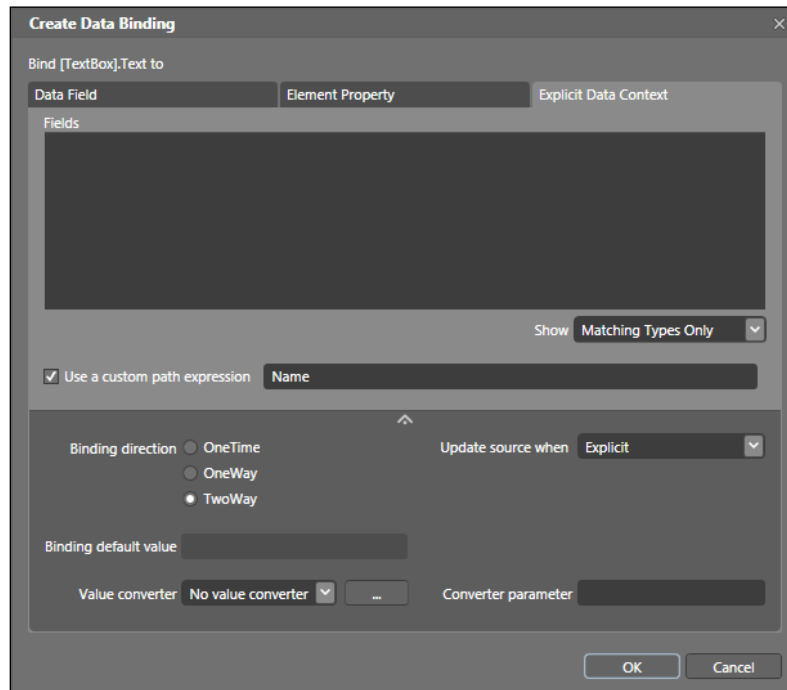
4. Build the solution, so that the `Owner` class is known to Blend and it can use the class in its dialog boxes.
5. Now, in the designer, add a `Grid` containing three `TextBlock` and three `TextBox` controls as shown in the following screenshot:



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.

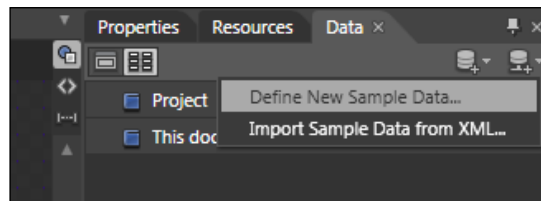
Using Expression Blend 4 for sample data generation

Expression Blend 4 contains a feature that is capable of generating the sample data while developing 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:

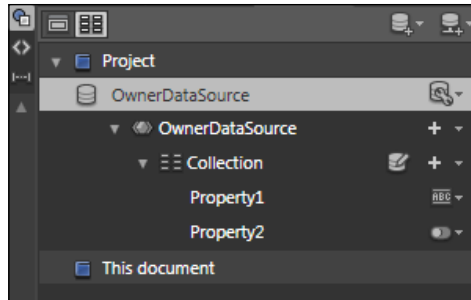
1. Open Blend 4 and go to **File | New Project...** In the dialog box that appears, select **Silverlight 4 Application + Website**. Name the project as **SilverlightBankingManagement** and click on the **OK** button. Blend will now create a Silverlight application and a hosting website.
2. With `MainPage.xaml` 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.

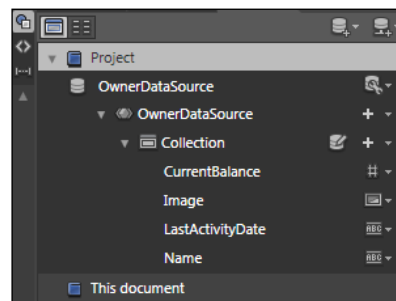
The last option in this window—**Enable sample data when application is running**—allows us to switch off the sample data while running the compiled application. If we leave the checkbox checked, then the sample data will be used for the design time as well as the runtime. We'll keep this option enabled.

Blend will now generate the data source for us. The result is shown in the following screenshot:



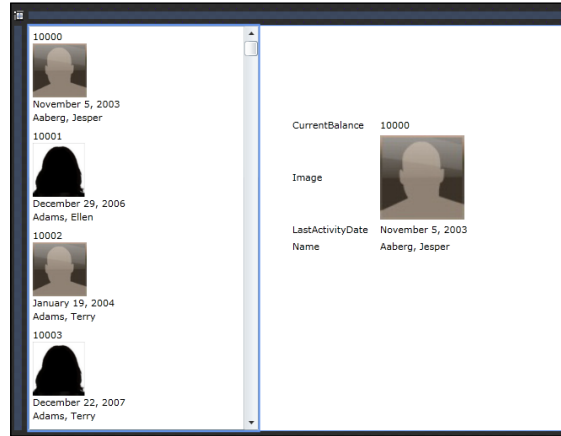
- By default, a **Collection** is created and it contains items with two properties. Each property has a type. Start by adding two more properties by clicking on the **+** sign next to the **Collection** and select the **Add simple property option**.

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.
- 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.

7. Now, to view the details, we have to set the **Data** window to the **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 screenshot:



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

4

Advanced Data Binding



This chapter is taken from *Silverlight 4 Data and Services Cookbook* (Chapter 3) by Gill Cleeren, Kevin Dockx.

In this chapter, we will cover:

- ▶ Hooking into the data binding process
- ▶ Replacing converters with Silverlight 4 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
- ▶ Building a change-aware collection type
- ▶ Combining converters, data binding, and DataContext into a custom DataTemplate

Introduction

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.

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 4 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.

This chapter continues to use the same sample application, Silverlight Banking. If you want to run the completed application, take a look at the code within the `Chapter03/SilverlightBanking` folder in the code bundle that is available on the Packt website.

Hooking into the data binding process

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 `Chapter04/SilverlightBanking_Converters_Starter` folder. The `Chapter04/SilverlightBanking_Converters_Completed` folder contains the completed 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, that is, `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);
}
```

- 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.

```
<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>
```

- 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 Amount,
  Converter={StaticResource localCurrencyConverter}}"
  FontSize="12"
  FontWeight="Bold">
</TextBlock>
```

- While this simple converter converts a double into a string, more advanced conversions can be performed. What if, for example, we want to color negative amounts red and positive amounts 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);
}
```

- 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 Amount,
  Converter={StaticResource localCurrencyConverter}}"
  Foreground="{Binding Amount,
```

```

Converter={StaticResource
    LocalAmountToColorConverter} }">
</TextBlock>

```

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

Current balance:	\$1,234.56
Last activity on:	04-11-2010
Amount:	\$100.00

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 ConvertParameter

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 `ConvertParameter` 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, and 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 4 BindingBase properties

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. Silverlight 4 has added some properties on the `BindingBase` class that can relieve us from writing a converter in some occasions.

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 `Chapter04/SilverlightBanking_BindingBase_Properties_Starter` folder. The completed solution for this recipe can be found in the `Chapter04/SilverlightBanking_BindingBase_Properties_Completed` folder.

How to do it...

The newly added options that are at our disposal in 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");
        }
    }
}
```


3. 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>
```

4. 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>
```

5. `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>
```

6. 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");
    }
}
}
}

```

7. 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.

In 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 databound input

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 is 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.

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 `Chapter04/SilverlightBanking_Validation_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter04/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 `LastNameValueTextBlock` 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");
    }
}
}
```

5. 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:	<input type="text" value="John"/>	
Last name:	<input type="text" value="Smith thisstringwillbetoolong"/>	
Address:	<input type="text" value="Oxford Street 24"/>	
Zip code:	<input type="text" value="W1A"/>	
City:	<input type="text" value="London"/>	
State:	<input type="text" value="NA"/>	
Country:	<input type="text" value="United Kingdom"/>	
Birthdate:	<input type="text" value="invalid date"/>	Input is not in a correct format.
Customer since:	<input type="text" value="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, that is, `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, *Validating data input using attributes*.

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 the `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 data entry form with the following fields and values:

First name:	John
Last name:	Smith thisstringwillbetoolong
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

At the bottom of the form, a red banner displays "2 Errors". Below the banner, the following error messages are listed:

- 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 3, An Introduction to Data Binding*

Validating data input using attributes

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.

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 `Chapter04/SilverlightBanking_Validation_Attributes_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter04/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:	<input type="text" value="John"/>	
Last name:	<input type="text" value="Smith thisvalueistoolong"/>	Length must be <= 20
Address:	<input type="text" value="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, that is, `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, that is, `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 3, An Introduction to Data Binding*.

Validating using `IDataErrorInfo` and `INotifyDataErrorInfo`

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.

In Silverlight 4, a new way of 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 3, An Introduction to Data Binding* or use the provided starter solution in the `Chapter04/SilverlightBanking_Validation_DataError_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter04/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`, rather than 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 3, An Introduction to Data Binding* (or the starter solution) and locate the `OwnerDetailsEdit.xaml` file. In this XAML file, locate and change the `LastNameValueTextBlock` by adding `NotifyOnValidationError=true` to the Binding syntax. This can be seen in the following code:

```
<TextBox x:Name="LastNameValueTextBlock"
         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" />
</StackPanel>
```

```

        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>();
}

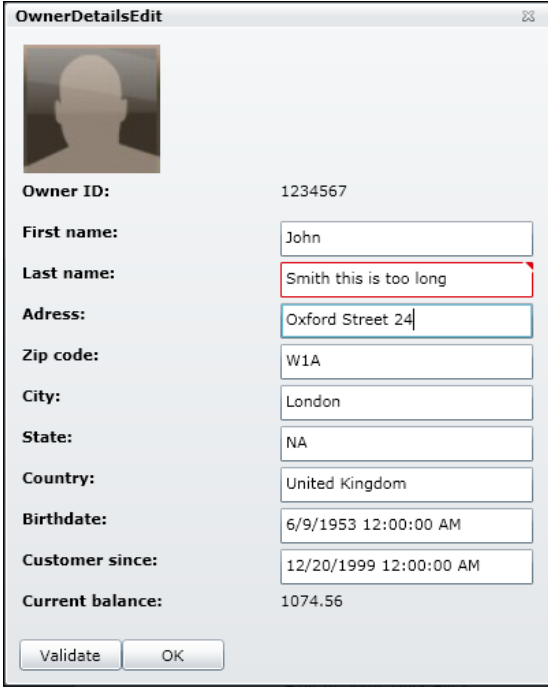
```

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:



The screenshot shows a window titled "OwnerDetailsEdit" with a form containing the following fields and values:

Owner ID:	1234567
First name:	John
Last name:	Smith this is too long
Adress:	Oxford Street 24
Zip code:	W1A
City:	London
State:	NA
Country:	United Kingdom
Birthdate:	6/9/1953 12:00:00 AM
Customer since:	12/20/1999 12:00:00 AM
Current balance:	1074.56

At the bottom of the window, there are two buttons: "Validate" and "OK". The "Last name" field is highlighted with a red border, indicating a validation error.

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 firing validation on 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, that is, the `IDataErrorInfo` interface. The `INotifyDataErrorInfo` interface is typically used for more complex scenarios such as the need for async 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 3, An Introduction to Data Binding*.

Using templates to customize the way data is shown by controls

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 `Chapter04/SilverlightBanking_DataTemplates_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter04/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;
```

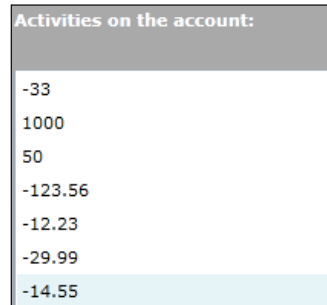
3. When populating this `ListBox` with `AccountActivity` objects and omitting any information that tells the `ListBox` what to display (as we did here), it will simply call the `ToString()` implementation of the object. This mostly results in displaying the string representation of the type of the object,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

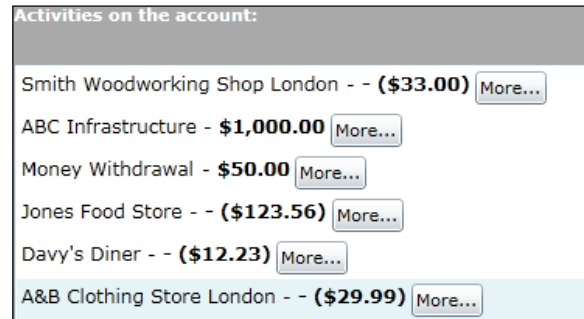

The `ListBox` now displays the value of the `Amount` property, as can be seen in the following screenshot:



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:

```
<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 ComplexTemplate}">
</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. 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>
    <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.

Building a change-aware collection type

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 we are used to.

Getting ready

The finished solution for this recipe can be found in the `Chapter04/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 it. 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 quite 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

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, 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.

You can find the completed solution for this recipe in the `Chapter04/Combining_Converters_Databinding_And_DataContext_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:

```
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. 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}"/>
            <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>

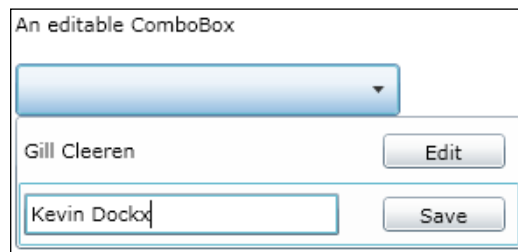
```

```
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
</Grid>
</UserControl>
```

4. 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;
    }
}
```

5. 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 off with the `Person` class. This class represents the people shown in our editable `ComboBox`. It implements the `INotifyPropertyChanged` 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 two-way 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 3, An Introduction to Data Binding*:

- ▶ *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 3, An Introduction to Data Binding*. Additionally, **Converters** are covered in the *Hooking into the data binding process* recipe in this chapter, and for more information on the `ObservableCollection`, have a look at the *Binding collections to UI elements* recipe in *Chapter 3, An Introduction to Data Binding*.

5

The Data Grid



This chapter is taken from *Silverlight 4 Data and Services Cookbook* (Chapter 4) by Gill Cleeren, Kevin Dockx.

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

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

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 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` will still keep running fluently.

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`

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 `Chapter05/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 to be hidden and the horizontal ones to get 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 as `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. Remove the `AutoGenerateColumns="False"` for now so that we'll see all the properties of the `Book` class appear in the `DataGrid`. 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;
    }
}
```

After completing these steps, we have the `DataGrid` 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 screenshot:

Book Library									
Actions									
	Title	Author	PageCount	PurchaseDat	Category	Publisher	Language	ImageName	AlreadyRead
	Book AAA	Author AAA	350	3/10/2009 12:	Computing	Publisher BBB	English	AAA.png	<input checked="" type="checkbox"/>
	Book BBB	Author AAA	667	4/11/2009 12:	Thriller	Publisher AAA	Dutch	BBB.png	<input type="checkbox"/>
	Book CCC	Author AAA	289	12/10/2009 12:	Fiction	Publisher AAA	French	CCC.png	<input checked="" type="checkbox"/>
	Book DDD	Author BBB	200	1/20/2009 12:	Thriller	Publisher DDD	German	DDD.png	<input type="checkbox"/>
	Book EEE	Author BBB	403	3/1/2007 12:0	Biography	Publisher DDD	German	EEE.png	<input checked="" type="checkbox"/>
	Book FFF	Author AAA	296	9/4/2009 12:0	Comics	Publisher AAA	English	FFF.png	<input checked="" type="checkbox"/>
	Book HHH	Author CCC	675	1/31/2007 12:	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>
	Book HHH	Author CCC	675	1/31/2007 12:	Fiction	Publisher CCC	Dutch	HHH.png	<input type="checkbox"/>
	Book III	Author DDD	1300	7/1/2008 12:0	Computing	Publisher DDD	French	III.png	<input checked="" type="checkbox"/>
	Book KKK	Author BBB	200	1/2/2009 12:0	Thriller	Publisher BBB	French	KKK.png	<input checked="" type="checkbox"/>
	Book KKK	Author BBB	200	1/2/2009 12:0	Thriller	Publisher BBB	French	KKK.png	<input checked="" type="checkbox"/>
	Book LLL	Author DDD	400	10/23/2009 12:	Computing	Publisher CCC	English	LLL.png	<input checked="" type="checkbox"/>

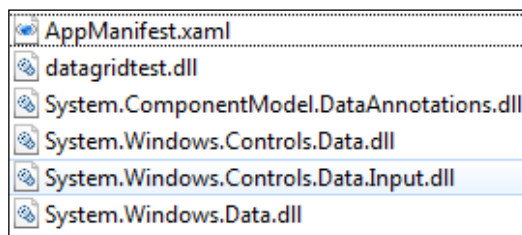
How it works...

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

The `DataGrid` 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 screenshot, 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. 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 `DataGrid`, the performance will still be very good. This is the result of the `DataGrid` implementing UI virtualization, which is enabled by default.

Let's assume that the `DataGrid` 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. Since Silverlight 3, the `ListBox` also features UI virtualization.

See also

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

Inserting, updating, and deleting data in a DataGrid

The `DataGrid` 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 `DataGrid` that supports all of the above 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 `Chapter05/Datagrid_Editing_Data_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter05/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 `DataGrid`, we'll manage an `ObservableCollection<Book>`. We'll make it possible to add, update, and delete the items in the collection through the `DataGrid`. 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 `DataGrid` by clicking on a button. Behind the scene, 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 `DataGrid`.

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 `DataGrid` 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 `DataGrid`. 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 `DataGrid` as shown in the following code:

```
<sdk:DataGrid 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,
    DataGridBeginningEventArgs 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. The application UI contains two buttons. 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 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>
```

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

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 us 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 `DataGrid` comes down to managing the items in the collection. We leverage this here. We aren't adding any items to the `DataGrid` 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 3, An Introduction to Data Binding* and *Chapter 4, Advanced Data Binding*, where we look carefully at all the features offered by Silverlight.

Sorting and grouping data in a DataGrid

Sorting the values within a column in a control such as a `DataGrid` 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 `DataGrid` 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 `DataGrid`.

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 `DataGrid`.

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 `Chapter05/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 `Chapter05/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 `DataGridTextColumns`, 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>
```

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, filtered and so on 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.

```
PagedCollectionView 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"));
```

5. The `DataGrid` will not bind to the collection, but to the view. We specify this by setting the `ItemsSource` property to the instance of the `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 `DataGrid` 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 `DataGrid` is shown in the following screenshot:

Book Library								
Actions Group by Language Expand all Collapse 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	15
	Book HHH	Author CCC	Publisher CCC	Dutch	Fiction	<input type="checkbox"/>	31/01/2007	15
	Book HHH	Author CCC	Publisher CCC	Dutch	Fiction	<input type="checkbox"/>	31/01/2007	15
	Book BBB	Author AAA	Publisher AAA	Dutch	Thriller	<input type="checkbox"/>	11/04/2009	15
English (5 items)								
	Book OOO	Author BBB	Publisher DDD	English	Thriller	<input checked="" type="checkbox"/>	2/02/2009	15
	Book NNN	Author DDD	Publisher CCC	English	Fiction	<input type="checkbox"/>	24/12/2008	15
	Book LLL	Author DDD	Publisher CCC	English	Computing	<input checked="" type="checkbox"/>	23/10/2009	15
	Book FFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	4/09/2009	15
	Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	10/03/2009	15
German (3 items)								
	Book MMM	Author AAA	Publisher DDD	German	Fiction	<input checked="" type="checkbox"/>	5/04/2009	15
	Book EEE	Author BBB	Publisher DDD	German	Biography	<input checked="" type="checkbox"/>	1/03/2007	15
	Book DDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	20/01/2009	15
French (4 items)								
	Book KKK	Author BBB	Publisher BBB	French	Thriller	<input checked="" type="checkbox"/>	2/01/2009	15
	Book KKK	Author BBB	Publisher BBB	French	Thriller	<input checked="" type="checkbox"/>	2/01/2009	15
	Book III	Author DDD	Publisher DDD	French	Computing	<input checked="" type="checkbox"/>	1/07/2008	15
	Book CCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	10/12/2009	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 code, we can control all 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 *Using custom columns in the DataGrid* recipe of this chapter.

See also

In the next recipe, we'll use the `PagedCollectionView` once more.

Filtering and paging data in a DataGrid

Along with offering us support for the sorting and filtering of data, the `PagedCollectionView` has more 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 `Chapter05/Datagrid_Filtering_And_Paging_Starter` folder in the code bundle that is available on the Packt website. The finished code for this recipe can be found in the `Chapter05/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`. The following are the steps to follow:

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 `DataGrid` 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);
BookDataGrid.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 `DataGrid`. Paging is the job of the `DataPager`. This control adds paging support to controls such as the `ListBox` and the `DataGrid`. We simply add a `DataPager` on the XAML page and specify the `PageSize` property as five as 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 `DataGrid`. The following code shows us how to do this:

```

public MainPage()
{
    InitializeComponent();
    LoadBooks();
    view = new PagedCollectionView(bookCollection);
    BookDataGrid.ItemsSource = view;
    BookPager.Source = view;
}

```

We have now implemented a filter on the `DataGrid` using the `PagedCollectionView`. A user can search for a value and filtering of the in-memory data will be done. The resulting `DataGrid` is paged using a `DataPager`. The following screenshot shows the result:

Book Library							
Actions Search book titles AAA Search							
							◀ 1 2 ▶
Title	Author	Publisher	Language	Category	Already read		
▶ Book AAA	Author AAA	Publisher BBB	English	Computing	<input checked="" type="checkbox"/>	10/03/2009	15
Book AAACCC	Author AAA	Publisher AAA	French	Fiction	<input checked="" type="checkbox"/>	10/12/2009	15
Book AAADDD	Author BBB	Publisher DDD	German	Thriller	<input type="checkbox"/>	20/01/2009	15
Book AAEEEE	Author BBB	Publisher DDD	German	Biography	<input checked="" type="checkbox"/>	1/03/2007	15
Book AAFFFF	Author AAA	Publisher AAA	English	Comics	<input checked="" type="checkbox"/>	4/09/2009	15

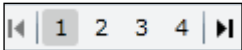
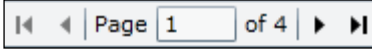
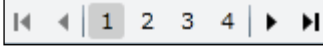
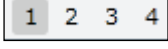
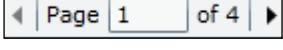
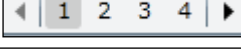
How it works...

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.

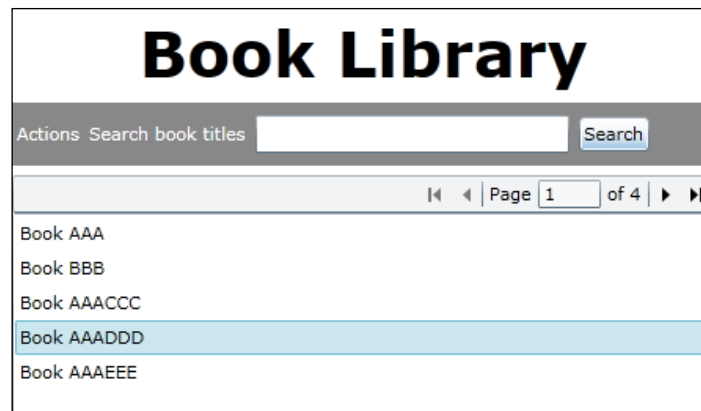
Paging is not directly done by the `DataGrid`. A second control, that is, the `DataPager` comes to the rescue. This control does not have a direct link to the `DataGrid`. Both the `DataGrid.ItemsSource` and the `DataPager.Source` properties point to the same instance of the `PagedCollectionView`. This way, the `DataPager` knows on which control it should add the paging functionality.

The `DataPager` control has a `DisplayMode` property that requires a value of the `PagerDisplayMode` enumeration. The following table shows the different options in action:

PagerDisplayMode 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 screenshot 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

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 the *Displaying the data in a customized DataGrid* recipe. 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. On top of 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 `Chapter05/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 `Chapter05/Datagrid_Custom_Columns_Completed` folder.

How to do it...

There are three types of columns from which we can choose—the `DataGridTextColumn`, the `DataGridCheckBoxColumn` and the `DataGridTemplateColumn`. We can either declare columns from XAML by adding them to the `Columns` collection of the `DataGrid` 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 `DataGrid`. 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 `DataGrid`, 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 `DataGridTextColumn` 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:DataGridTextColumn x:Name="TitleColumn"
                       Binding="{Binding Title}"
                       Header="Title">
</sdk:DataGridTextColumn>
<sdk:DataGridTextColumn x:Name="AuthorColumn"
                       Binding="{Binding Author}"
                       Header="Author">
</sdk:DataGridTextColumn>
<sdk:DataGridTextColumn x:Name="PublisherColumn"
                       Binding="{Binding Publisher,
                               Mode=TwoWay}"
                       Header="Publisher">
</sdk:DataGridTextColumn>
```

3. The `AlreadyRead` property of our `Book` class is of the `bool` type. We can bind such a value to a `DataGridCheckBoxColumn` as shown in the following code:

```
<sdk:DataGridCheckBoxColumn x:Name="AlreadyReadColumn"
                           Binding="{Binding AlreadyRead,
                               Mode=TwoWay}"
                           Header="Already read">
</sdk:DataGridCheckBoxColumn>
```

4. The `DataGridTemplateColumn` 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 lanugageType = typeof(Languages);
            var fields = from c in lanugageType.GetFields()
                where c.IsLiteral
                select c;

            foreach (var f in fields)
            {
                var value = f.GetValue(lanugageType);
                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>
```

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 screenshot:

Book Library									
Actions									
Title	Author	Publisher	Language	Category	Already read				
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	Thriller	<input checked="" type="checkbox"/>	3/1/2007 12:00:00 AM			
Book FFF	Author AAA	Publisher AAA	English	Fiction	<input checked="" type="checkbox"/>	9/4/2009 12:00:00 AM			
Book HHH	Author CCC	Publisher CCC	Dutch	Comics	<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...

Silverlight 4 has added more sizing options for the columns of a `DataGrid`. Silverlight 2 and Silverlight 3 basically 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).

In Silverlight 4, three new options were added, bringing the total to five options to size the columns. 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 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 screenshot 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		
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

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 `Chapter05/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 `Chapter05/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 `DataGrid` 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}"
        HorizontalAlignment="Left"
        Margin="1">
    </TextBlock>
    </StackPanel>
    <StackPanel Grid.Row="1"
      Grid.Column="1"
```

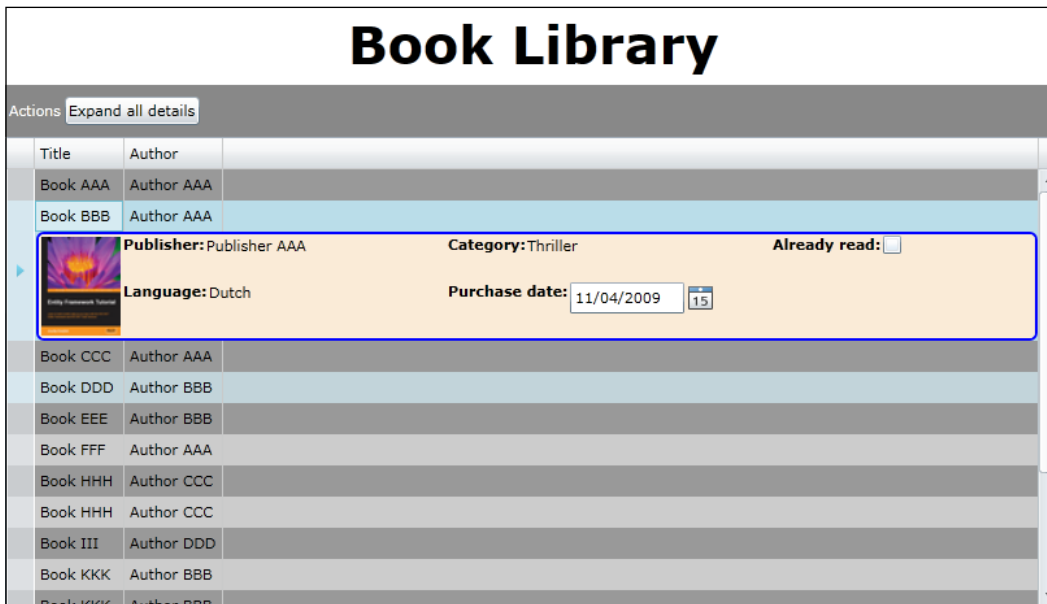
```
        Orientation="Horizontal">
    <TextBlock Text="Language:"
        FontWeight="Bold"
        HorizontalAlignment="Left">
    </TextBlock>
    <TextBlock Text="{Binding Language}"
        HorizontalAlignment="Left"
        Margin="1">
    </TextBlock>
</StackPanel>
<StackPanel Grid.Row="0"
    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>
        <CheckBox IsChecked="{Binding AlreadyRead}">
        </CheckBox>
    </StackPanel>
</Grid>
</Border>
</DataTemplate>

```

We have now created a master-detail scenario. This can be seen in the following screenshot:



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

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 `Chapter05\DataGrid_Validation_Starter` folder in the code bundle that is available on the Packt website. The finished solution for this recipe can be found in the `Chapter05\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 `ID`, `FirstName`, `LastName`, and `DateOfBirth` properties. The `MainPage` should contain an `ObservableCollection` of `Person`.

We're going to add a `DataGrid` to this project and we'll make sure that it react 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 `DataGrid` 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 DataGrid"
    Margin="10"
    FontSize="14" >
  </TextBlock>
  <data:DataGrid x:Name="myDataGrid"
    Grid.Row="1"
    Width="400"
    Height="300"
    Margin="10"
    HorizontalAlignment="Left"
    VerticalAlignment="Top">
  </data:DataGrid>
</Grid>
```

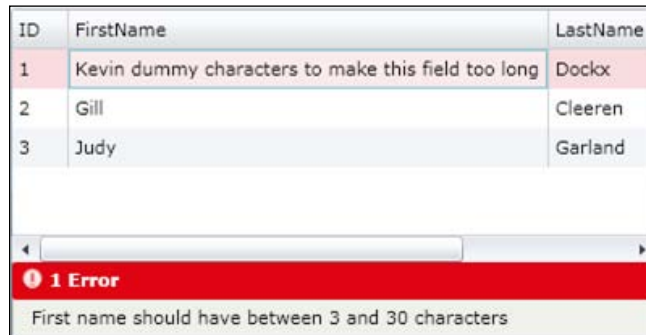
2. Add the following namespace import to `MainPage.xaml` to make sure that the `DataGrid` can be used:

```
xmlns:data="clrnamespace:System.Windows.Controls;
assembly=System.Windows.Controls.Data"
```

3. Add a reference to `System.ComponentModel.DataAnnotations` to your Silverlight project.
4. 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; }
```


5. 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 persist your changes unless they are valid. This can be seen in the following screenshot:



The screenshot shows a DataGrid with three columns: ID, FirstName, and LastName. The first row has ID 1, FirstName 'Kevin dummy characters to make this field too long', and LastName 'Dooxx'. The second row has ID 2, FirstName 'Gill', and LastName 'Cleeren'. The third row has ID 3, FirstName 'Judy', and LastName 'Garland'. Below the grid, a red error bar displays '1 Error' and the message 'First name should have between 3 and 30 characters'.

ID	FirstName	LastName
1	Kevin dummy characters to make this field too long	Dooxx
2	Gill	Cleeren
3	Judy	Garland

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` or a `DataForm`, 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. `DataTypeAttribute`, `RangeAttribute`, `RegularExpressionAttribute`, `RequiredAttribute`, `StringLengthAttribute`, and `CustomValidationAttribute` are the possible data annotations at your disposal.

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.

6

Talking to REST and WCF Data Services



This chapter is taken from *Silverlight 4 Data and Services Cookbook* (Chapter 8) by Gill Cleeren, Kevin Dockx.

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

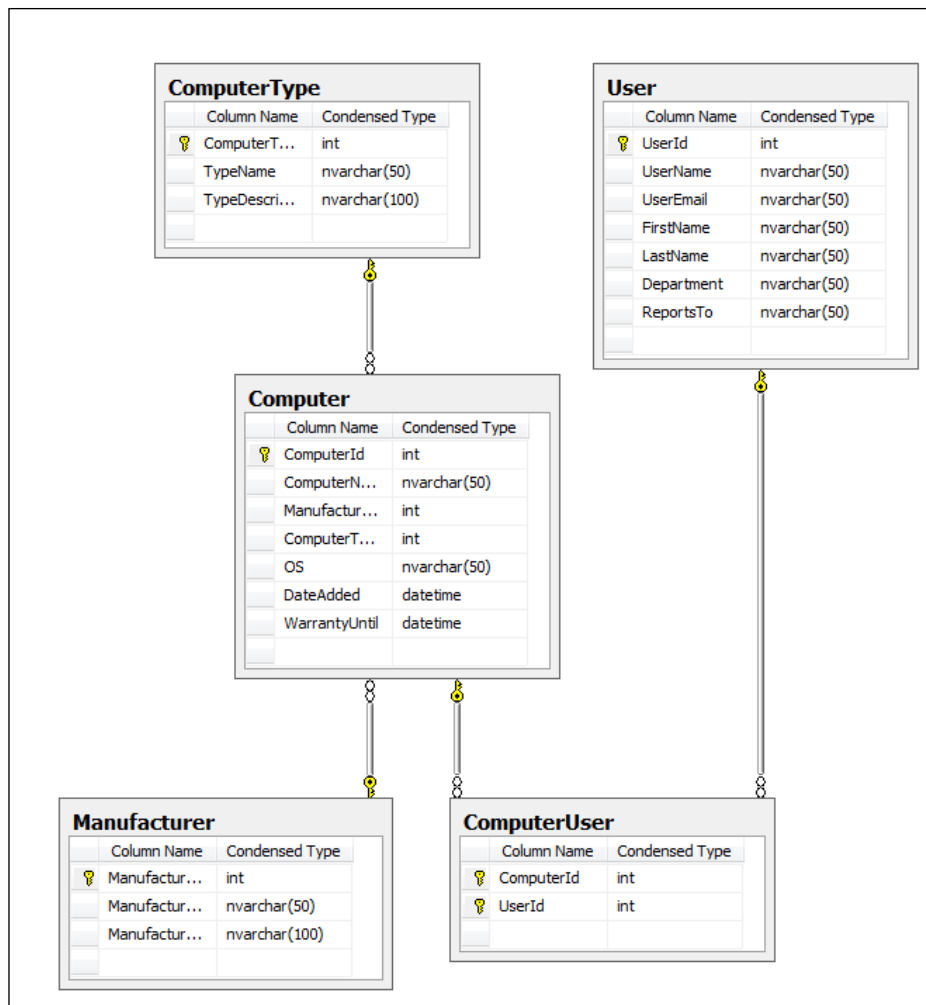
While WCF and ASMX services are very powerful and can address almost every situation, these services might be overkill for some scenarios. Sometimes, a simple exchange of textual information, preferably in XML or JSON (JavaScript Object Notation—an easy-to-read data exchange format), might be enough.

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 the XML format. The XML is clean, meaning there is not a lot of XML markup being added. SOAP messages—the format for web services—are 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? It's safe to say so. 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

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 `Chapter06/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 `Chapter06/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.

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 (`ComputerInventory.mdf`) and as a `*.sql` script file. Both the files are located in the `Chapter06` folder.

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 screenshot 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 Chapter06/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 working with Visual Studio 2010 (either with Silverlight 3 or 4), a reference to this assembly should be added automatically. If you're 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**.
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>
    <RowDefinition Height="40"></RowDefinition>
    <RowDefinition ></RowDefinition>
  </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>
```

- 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 of which 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();
```

- 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.

- 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 `HttpRequest` class. This class should be our choice if we need more control over the call to the service. The `WebClient` class uses the `HttpRequest` class internally.

Calling REST services is possible only in an asynchronous way. Silverlight allows only this type of calls. 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.

Parsing REST results with LINQ-To-XML

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 `XmlReader/XmlWriter`, `XmlSerializer`, 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 `Chapter06/TalkingToSimpleRESTServices_LinqToXml_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/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; }
}
```

- 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").Value.ToString()
    };
}
```

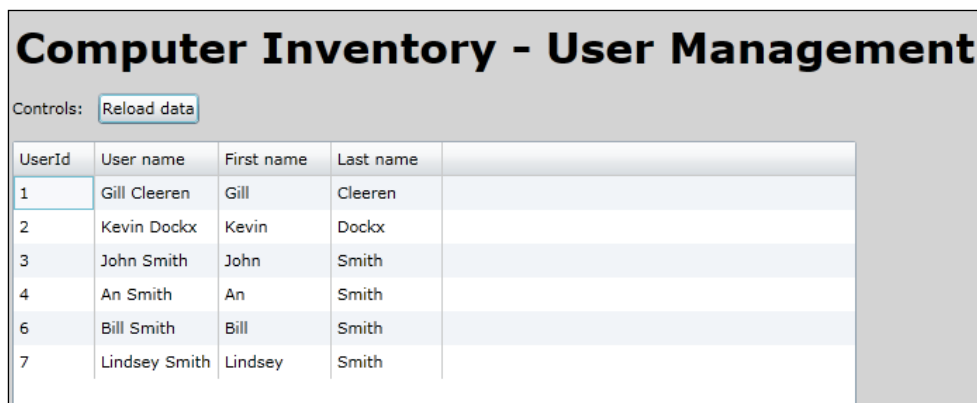
- We will then need to replace the **ResultTextBox** in `MainPage.xaml` with a `DataGrid` called **UsersDataGrid**. The code for this control is as follows:

```
<sdk:DataGrid x:Name="UsersDataGrid"
    Grid.Row="2"
    AutoGenerateColumns="False"
    Width="600"
    Height="500"
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="3">
    <sdk:DataGrid.Columns>
        <sdk:DataGridTextColumn Binding="{Binding UserId}"
            Header="UserId" />
        <sdk:DataGridTextColumn Binding="{Binding UserName}"
            Header="User name" />
        <sdk:DataGridTextColumn Binding="{Binding FirstName}"
            Header="First name" />
        <sdk:DataGridTextColumn Binding="{Binding LastName}"
            Header="Last name" />
    </sdk:DataGrid.Columns>
</sdk: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();
}
```

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 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 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 3, An Introduction to Data Binding* and *Chapter 4, Advanced Data Binding*.

Persisting data using a REST service

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 `Chapter06/TalkingToSimpleRESTServices_PersistingData_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/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`. 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}"
        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 getUserById = "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.

- 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 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.

```
[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; }
}
```

- 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
}
```

- 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();
            UserDetailsGrid.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);
}
```

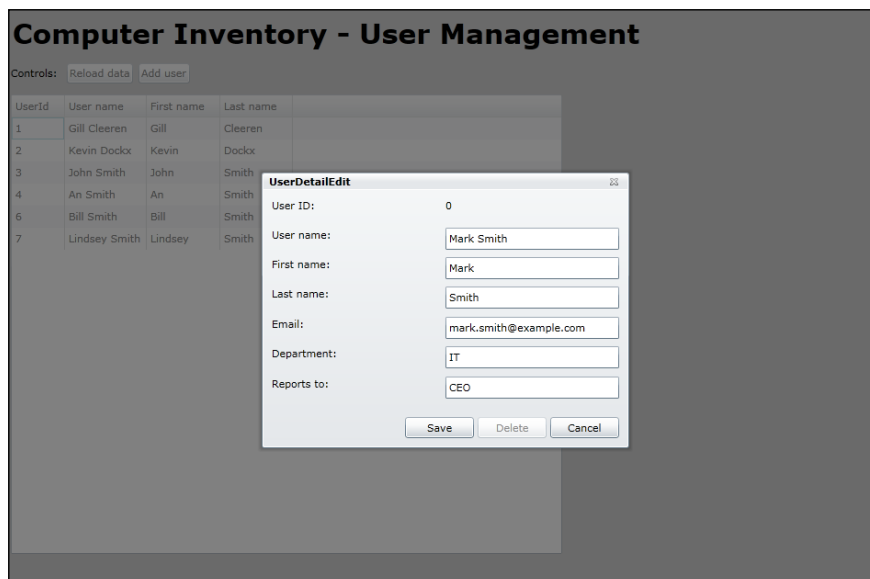
11. At this point, the child window is ready. The only thing left to do is calling it from `MainPage.xaml`. To do so, start by adding a new `Button` called `NewUserButton` in the `StackPanel` within `MainPage.xaml`. This is shown in the following code:

```
<Button x:Name="NewUserButton"
        Content="Add user"
        Click="NewUserButton_Click"
        HorizontalAlignment="Left"
        Margin="3"
        VerticalAlignment="Center">
</Button>
```

12. 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 (de)serialization 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 contains 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 a tenet 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 ClientHttpStack

When communicating with a REST service, Silverlight uses the `BrowserHttpStack` by default. Due to this, Silverlight can't use all HTTP verbs such as `PUT` and `DELETE`. Silverlight 3 added a new option, namely the `ClientHttpStack`. 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 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 `Chapter06/TalkingToSimpleRESTServices_ClientHttp_Starter` folder in the code bundle available on the Packt website. The completed solution can be found in the `Chapter06/TalkingToSimpleRESTServices_ClientHttp_Completed` folder.

How to do it...

To make a Silverlight application that talks with REST services use the `ClientHttpStack` instead of the `BrowserHttpStack`, 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:

To make an application use the `ClientHttpStack`, 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);
}
```

With the previous code executed, all calls will be executed over the `ClientHttpStack`.

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 `ClientHttpStack`.

Working with this new stack requires almost no changes to 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 `ClientHttpStack`. This is shown in the following line of code:

```
HttpRequest.RegisterPrefix("http://",  
    WebRequestCreator.ClientHttp);
```

If we have requests over HTTPS and want these to happen over the client stack as well, we need to register them using the following line of code:

```
HttpRequest.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:

```
HttpRequest.RegisterPrefix("http://www.snowball.be",  
    WebRequestCreator.ClientHttp);
```

Advantages of ClientHttpStack

Using the `ClientHttpStack` has some advantages over using the `BrowserHttpStack`. 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 the `BrowserHttpStack` are limited. With this stack, we have access to only 200 and 404. The `ClientHttpStack` 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 `ClientHttpStack` (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).

When we download an image with the `BrowserHttpStack`, it's automatically cached by the browser as it is its default behavior. However, when working with the `ClientHttpStack`, 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 `ClientHttpStack`. Silverlight 4 adds support for caching though.

The same goes for cookies. When using the `BrowserHttpStack`, 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 `ClientHttpStack`, again this won't work. With the `ClientHttpStack`, we can work with cookies, but this is manual work and can be done 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

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 has also got 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 `Chapter06/TalkingToSimpleRESTServices_ReadingWithJSON_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/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`. To do this, we have to complete 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="NewUserButton"
    Content="Create new"
    Click="NewUserButton_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();
    }
}
```

Build and run your application. If we place a breakpoint in the returning method, we can see that the data is effectively returned in a JSON format.

How it works...

By setting the `RequestFormat` and the `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 this `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

Above our data layer, we may have an entity model that exposes entities (for example, 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.

WCF Data Services is the new name for ADO.NET Data Services. This name change was made in the .NET Framework 4 timeframe.

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.

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.

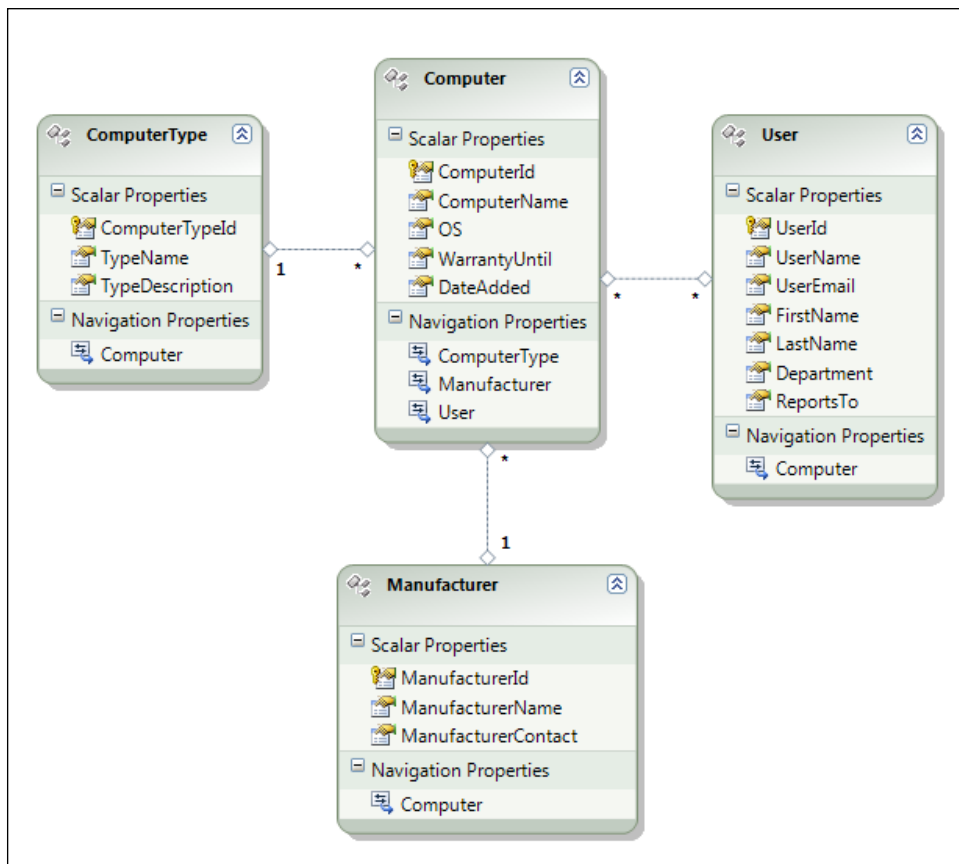
This recipe starts from an empty Silverlight application. Refer to Chapter 1 for more information on how to do so.

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.

- 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 as **ComputerInventory.edmx**.
- 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.
- 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.
- The final step in the wizard allows us to select which items from the database we want to make part of the model. Select **all tables**, excluding the **sysdiagrams**. When we click on **Finish**, Visual Studio generates the model as shown in the following screenshot:



6. Next, we create the actual WCF Data Service. To do so, add an **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. Underlying, it uses REST as its communication platform. So, it's possible to connect to the services using the `WebClient` class or the `HttpRequest` 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 `IQueryable` 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. Sounds 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 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 we want to retrieve `http://localhost:12345/ComputerInventoryService.svc/Computer(1)`.

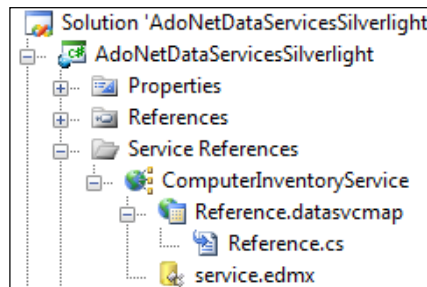
The resulting response can be sent in an XML or JSON format. XML, in the form of AtomPub, is the default format and is actually easiest to read. The **Atom Publishing Protocol (AtomPub)** is a protocol based on HTTP that allows creating and publishing web resources. The response sent when invoking the above 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 URIs 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 an WCF Data Service 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

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 `Chapter06/WorkingWithWcfDataServices_Reading_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/WorkingWithWcfDataServices_Reading_Completed` folder.

How to do it...

In the previous recipe, we introduced the client library that dramatically reduces the amount of code 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:

1. 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 possibility 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 code:

```
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;
}
```

2. 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);
}
```

- 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 on the following screenshot:

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

- 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.
- To show the details of the selected computer in the `DataGrid`, we pass the context as well as the `computerID` of the selected computer via the constructor. This is shown in the following code:

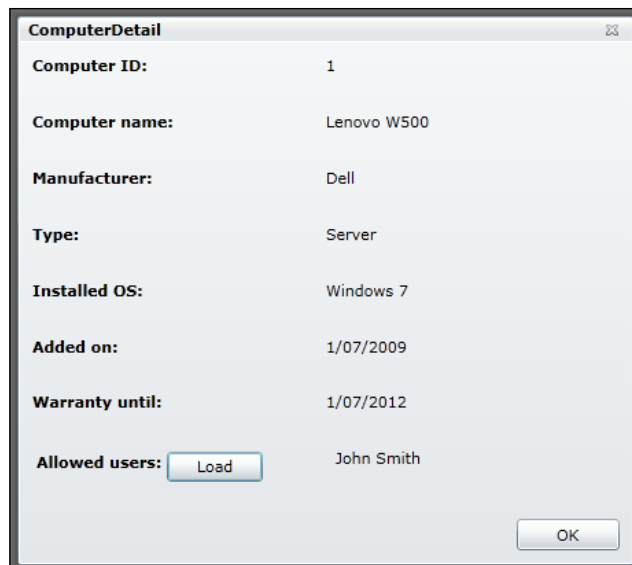
```
private ComputerInventoryEntities context;
private int computerId;
private Computer computer;
```

```
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;
}
```

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 screenshot:



How it works...

The most important part of this recipe is the LINQ query. When executing a LINQ query against an 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).

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

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 screenshot is similar to the View screenshot, 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 `Chapter06/WorkingWithWcfDataServices_Persisting_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/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:

1. 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);
```

- 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 text boxes 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 or 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 URIs. 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.

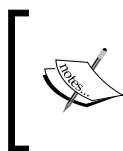
Updating and deleting are very similar. All changes are done initially on the objects tracked by the context. Afterwards, the changes are persisted using exactly the same code as for adding new objects.

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

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 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.

To follow along with this recipe, a starter solution has been provided in the `Chapter06/SilverFlickr_Starter` folder in the code bundle available on the Packt website. The completed solution for this recipe can be found in the `Chapter06/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 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. We start from an empty Silverlight application. Therefore, create a new Silverlight solution called **SilverFlickr**.
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, you need to add a reference to this assembly yourself. Visual Studio 2010 refers this assembly by default for new projects for both Silverlight 3 and 4 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>

```

- 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}";

```

- We now have the URI; we can use it to send a request to. 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 the `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:

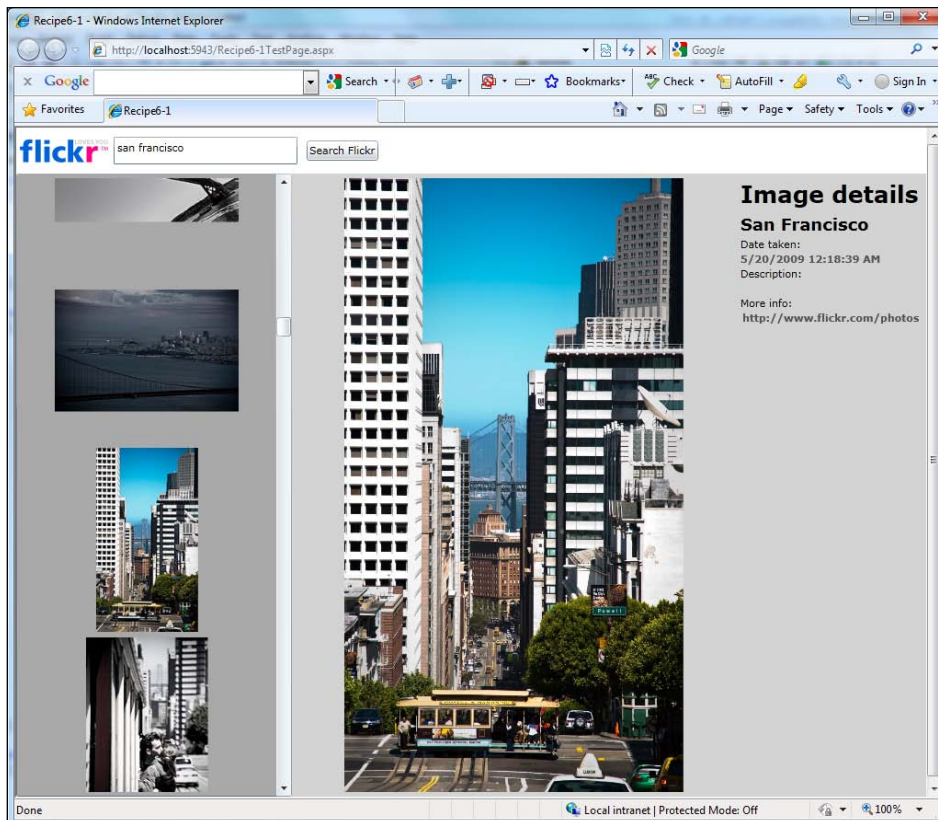
```
public class ImageInfo
{
    public string ImageId { get; set; }
    public string FarmId { get; set; }
    public string ServerId { get; set; }
    public string Secret { get; set; }
    public string ImageUrl
    {
        get
        {
            return string.Format
                ("http://farm{0}.static.flickr.com/{1}/{2}_{3}_m.jpg",
                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);
}
}
```

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 on a picture 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.

```
<?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, that is, 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 3, An Introduction to Data Binding* and *Chapter 4, Advanced Data Binding*.

Talking to Twitter over REST

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.

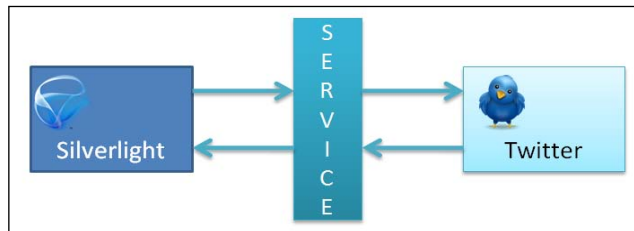
Getting ready

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.

A starter solution for this chapter is provided in the `Chapter06/SilverWitter_Starter` folder in the code bundle available on the Packt website. The finished solution for this recipe can be found in the `Chapter06/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 screenshot 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 time line, 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 userName, string userPassword);
}
  
```

```

    [OperationContract]
    string AddMessage(string message, string userName,
        string userPassword);
    [OperationContract]
    bool CheckCredentials(string userName, string userPassword);
}

```

5. We use the `TwitterUpdate` class in some of the above 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 the `DataContractAttribute`. Its members have the `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 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));
```

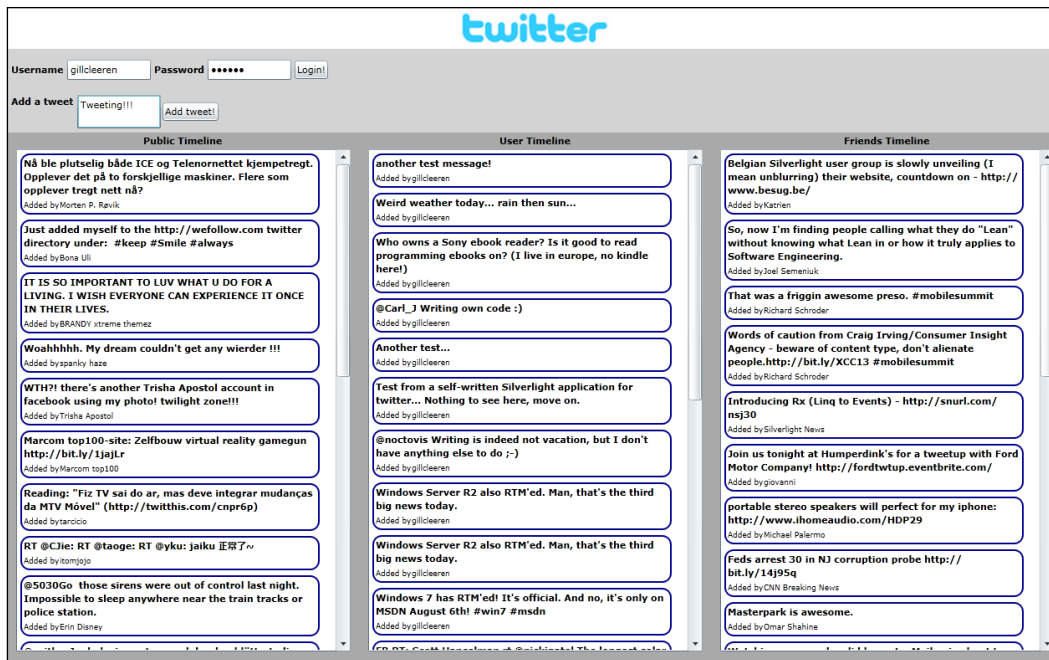
8. Once the result is available, we'll parse it using LINQ-To-XML as shown in the following code. On parsing the XML, we are creating a `List<TwitterUpdate>`.

```
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 screenshot. 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 screenshot 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

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 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.

Another added feature with Silverlight 4 is the ability to send credentials to a service when using a `WebClient` instance.

The combination of these two new 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 on 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 `Chapter06/TrustedSilverWitter_Starter` folder. The finished solution for this recipe can be found in the `Chapter06/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 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."
    </TextBlock>
  </StackPanel>
</Grid>
```

```

        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 stand-alone, 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.";
    }
}
```

5. 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.

6. 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 code below 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**. 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 code below 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 `HttpRequest` for this and set its method to POST. To this `HttpRequest`, we also add the user-entered credentials because this service method requires authorization as well. Silverlight will then send the message to Twitter.

```

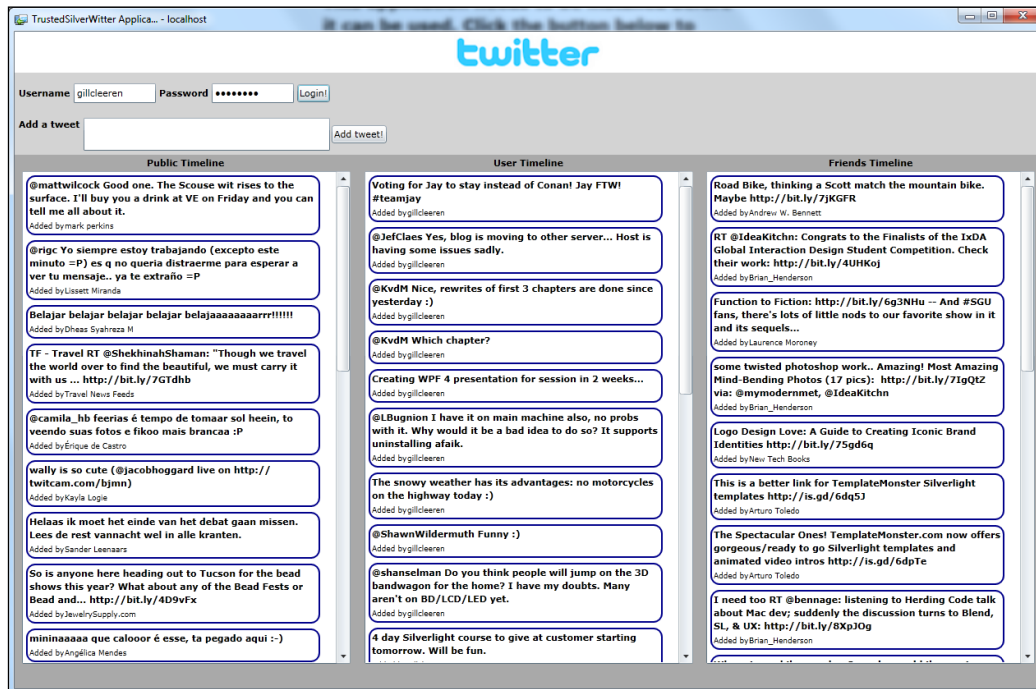
private void AddTweetButton_Click(object sender,
    RoutedEventArgs e)
{
    string uri = @"http://twitter.com/statuses/update.xml";
    try
    {
        string message = AddTweetTextBox.Text.Trim();
        string parameters = string.Format("status={0}&source={1}",
            HttpUtility.HtmlEncode(message), "Trusted SilverWitter");
        WebRequest.RegisterPrefix("http://",
            System.Net.Browser.WebRequestCreator.ClientHttp);
        HttpRequest request = (HttpRequest)
            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 screenshot:



How it works...

To build this application, two major new 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 the `ClientHttpStack`. Let's take a look at these in some more detail.

Let's go cross-domain!

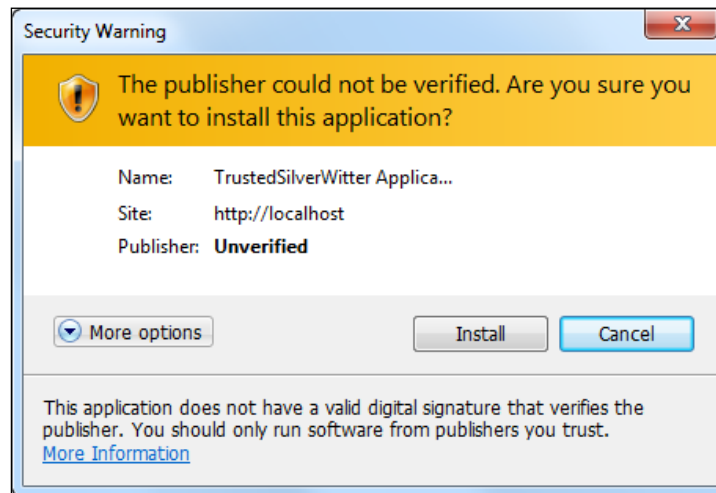
In many recipes in this book, we talked about the cross-domain restrictions that Silverlight has in place. Basically, these come down to Silverlight not allowing us to make requests to services that are not in the same domain as the Silverlight application. Silverlight will make the request only if there's a cross-domain policy file in place that allows the request. Cross-domain restrictions are required for security reasons.

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. Silverlight 4 extends this model.

With version 4, it becomes possible to create **Trusted Silverlight applications**, which run 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.

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 is that Twitter does not expose a policy file, so Silverlight applications can't directly communicate with Twitter's API. With trusted Silverlight 4 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 the **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 that asks the user if he or she fully trusts the application:



Silverlight 4 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. The process of signing the XAP file is outside the scope of this book.

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. Silverlight 4 changed this and now allows us to pass credentials to a service. One thing that is required is that we use the `ClientHttpStack`.

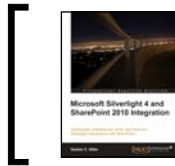
Passing credentials is very simple and can be done using the following code:

```
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 `ClientHttpStack` using the `WebRequest.Register` method: basically we're saying for all traffic that goes over `http://`, use the `ClientHttpStack`.

7

Interacting with Data on the SharePoint Server



This chapter is taken from *Microsoft Silverlight 4 and SharePoint 2010 Integration* (Chapter 3) by Gastón C. Hillar.

In this chapter, we will cover many topics that help us create simple and complex Line-Of-Business Silverlight RIAs that run as Silverlight Web Parts to interact with data in the SharePoint Server.

In this chapter, we will:

- Use a Silverlight RIA to insert items into a SharePoint list
- Retrieve and process metadata information about a SharePoint list
- Prepare code to handle errors when remote operations fail
- Work with messages to allow multiple Silverlight RIAs to communicate with each other
- Work with Visual Studio 2010 code editor features to track the execution flow
- Enhance a Silverlight RIA to delete specific items from a SharePoint list
- Enhance a Silverlight RIA to update specific fields for an item in a SharePoint list

Managing data in a Silverlight RIA included in a SharePoint solution

So far, we have been able to create, deploy, and debug a Silverlight RIA that read data from a list in the SharePoint server. It is also possible to insert, update, and remove items from these lists. In fact, the typical **LOB (Line-Of-Business)** RIA performs **CRUD (Create, Read, Update, and Delete)** operations. Therefore, we can create a Silverlight RIA to perform some of the CRUD operations with the existing list of tasks, by using more features provided by the SharePoint 2010 Silverlight Client OM.

We could improve our existing Silverlight RIA that displays data from the existing list in a grid. However, we are going to create a new Silverlight RIA and then, we will improve both applications to work together to offer a complex LOB solution.

We will analyze diverse alternatives to simplify the deployment process and show how to debug a Silverlight RIA that queries data from a SharePoint server.

Working with the SharePoint 2010 Silverlight Client Object Model to insert items

Now, we are going to create a new solution in Visual Studio. It will include two new projects:

- A Silverlight application project, `SLTasksCRUD`
- An empty SharePoint 2010 project with a module, `SPTasksCRUD`

Follow these steps to create the new Silverlight RIA that allows a user to insert a new item into the list in the SharePoint server:

1. Start Visual Studio as a system administrator user.
2. Select **File | New | Project...** or press *Ctrl+Shift+N*. Select **Other Project Types | Visual Studio Solutions** under **Installed Templates** in the **New Project** dialog box. Then, select **Blank Solution** and enter `TasksCRUD` as the project's name and click **OK**. Visual Studio will create a blank solution with no projects.
3. Right-click on the solution's name in **Solution Explorer** and select **Add | New Project...** from the context menu that appears.
4. Select **Visual C# | Silverlight** under **Installed Templates** in the **New Project** dialog box. Then, select **Silverlight Application**, enter `SLTasksCRUD` as the project's name and click **OK**.

5. Deactivate the **Host the Silverlight application in a new Web site** checkbox in the **New Silverlight Application** dialog box and select **Silverlight 4** in **Silverlight Version**. Then, click **OK**. Visual Studio will add the new Silverlight application project to the existing solution.
6. Follow the necessary steps to add the following two references to access the new SharePoint 2010 Silverlight Client OM:

- `Microsoft.SharePoint.Client.Silverlight.dll`
- `Microsoft.SharePoint.Client.Silverlight.Runtime.dll`

7. Open `App.xaml.cs` and add the following using statement:
`using Microsoft.SharePoint.Client;`

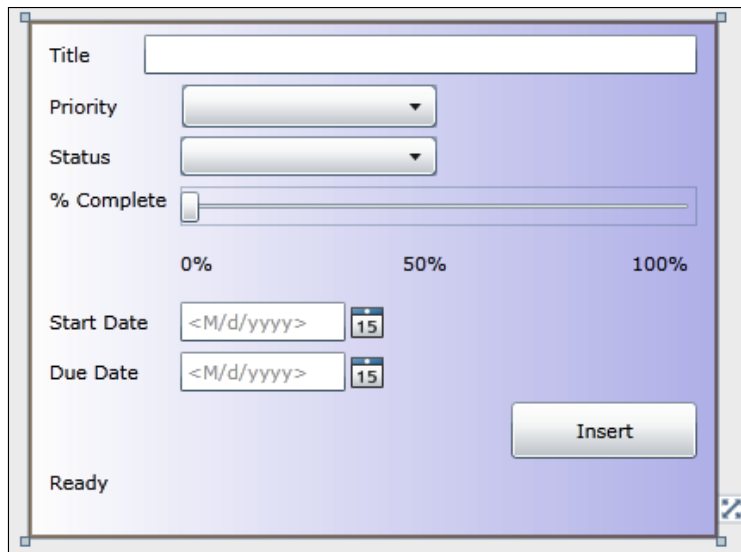
8. Add the following code in the `Startup` event handler to initialize the `Microsoft.SharePoint.Client.ApplicationContext` with the same initialization parameters and the synchronization context for the current thread (the UI thread).

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new MainPage();
    // Initialize the ApplicationContext
    ApplicationContext.Init(e.InitParams,
        System.Threading.SynchronizationContext.Current);
}
```

9. Open `MainPage.xaml`, define a new width and height for the Grid, 800 and 600, add the following controls, and align them as shown in the following screenshot:

- Six Label controls aligned at the left with the following values for their `Content` properties. They are `Title`, `Priority`, `Status`, `% Complete`, `Start Date` and `Due Date`.
- One Label control, located at the bottom, `lblStatus`.
- One TextBox control, `txtTitle`.
- One ComboBox control, `cboPriority`.
- One ComboBox control, `cboStatus`.
- One Slider control, `sldPercentComplete`. Set `LargeChange` to 10, `Maximum` to 100, and `Minimum` to 0. This slider will allow the user to set the percentage of the total work that has been completed.

- One DatePicker control, dtStartDate.
- One DatePicker control, dtDueDate.
- One Button control, butInsert. Set its Title property to Insert.



10. Select the Grid, LayoutRoot. Click on the **Categorized** button to arrange the properties by category. Then, click on **Brushes | Background** and a color palette with many buttons located at the top and the bottom will appear. Click on the **Gradient Brush** button, located at the top and then on the **Vertical Gradient** one, located at the bottom. Define both the `start` and the `stop` colors. The rectangle that defines the background Grid will display a nice linear gradient, as shown in the previous screenshot.
11. Open `MainPage.xaml.cs` and add the following using statements to include the `Microsoft.SharePoint.Client` namespace:

```
using Microsoft.SharePoint.Client;  
using SP = Microsoft.SharePoint.Client;  
Add the following two private variables  
private SP.ClientContext _context;  
private SP.List _projects;
```

Add the following method to fill the drop-down lists that will display the different options for the priority and the status:

```
private void FillComboBoxes()  
{
```

```

cboPriority.Items.Add(" (1) High");
cboPriority.Items.Add(" (2) Normal");
cboPriority.Items.Add(" (3) Low");
cboStatus.Items.Add("Not Started");
cboStatus.Items.Add("In Progress");
cboStatus.Items.Add("Completed");
cboStatus.Items.Add("Deferred");
cboStatus.Items.Add("Waiting on someone else");
}

```



It is possible to retrieve the possible choices for both the **Priority** and **Status** fields. However, we will improve this application later. In this case, we add the possible values in this method and then we will learn how to retrieve the choices through queries to the SharePoint server.

12. Add the following line to the page `MainPage` constructor:

```

public MainPage()
{
    InitializeComponent();
    FillComboBoxes();
}

```

13. Now, it is necessary to add code to execute the following tasks:

- i. Connect to the SharePoint server and load the current user that logged on the server, `ConnectAndAddItemToList` method.
- ii. Add a new item to the `ProjectsList2010` list, considering the values entered by the user in the controls, `AddItemToList` method.

```

private void ConnectAndAddItemToList()
{
    // Runs in the UI Thread
    lblStatus.Content = "Started";
    _context = new
        SP.ClientContext(SP.ApplicationContext.Current.Url);
    _context.Load(_context.Web);
    // Load the current user
    _context.Load(_context.Web.CurrentUser);
    _context.ExecuteQueryAsync(OnConnectSucceeded, null);
}

private void AddItemToList()

```

```
{
    // Runs in the UI Thread
    lblStatus.Content = "Web Connected. Adding new item to List...";
    _projects = _context.Web.Lists.GetByTitle("ProjectsList2010");
    ListItem listItem = _projects.AddItem(new
        ListItemCreationInformation());
    listItem["Title"] = txtTitle.Text;
    listItem["StartDate"] =
        Convert.ToString(dtStartDate.SelectedDate);
    listItem["DueDate"] = Convert.ToString(dtDueDate.SelectedDate);
    listItem["Status"] = "Not Started";
    var fieldUserValue = new FieldUserValue();
    // Assign the current user to the Id
    fieldUserValue.LookupId = _context.Web.CurrentUser.Id;
    listItem["AssignedTo"] = fieldUserValue;
    listItem["Priority"] = "(2) Normal";
    listItem["PercentComplete"] =
        Convert.ToString(Math.Round(sldPercentComplete.Value, 0)/100);
    listItem.Update();
    // Just load the list Title property
    _context.Load(_projects, list => list.Title);
    _context.ExecuteQueryAsync(OnAddItemToListSucceeded,
        OnAddItemToListFailed);
}
```

14. All the previously added methods are going to run in the UI thread. The following methods, which are going to be fired as asynchronous callbacks, schedule the execution of other methods to continue with the necessary program flow in the UI thread:
- When the connection to the SharePoint server, requested by the `ConnectAndAddItemToList` method, is successful, the `OnConnectSucceeded` method schedules the execution of the `AddItemToList` method in the UI thread. If the `ConnectAndAddItemToList` method fails, the `OnConnectFailed` method schedules the execution of the `ShowErrorInformation` method in the UI thread, sending the `ClientRequestFailedEventArgs` args instance as a parameter to the delegate.

- When the insert operation performed on the list available in the SharePoint server, requested by the `AddItemToList` method, is successful, the `OnAddItemToListSucceeded` method schedules the execution of the `ShowInsertResult` method in the UI thread. If the `AddItemToList` method fails, the `OnAddItemToList` method schedules the execution of the `ShowErrorInformation` method in the UI thread, sending the `ClientRequestFailedEventArgs` `args` instance as a parameter to the delegate.

```
private void ShowErrorInformation(ClientRequestFailedEventArgs
args)
{
    System.Windows.Browser.HtmlPage.Window.Alert(
        "Request failed. " + args.Message + "\n" +
        args.StackTrace + "\n" +
        args.ErrorDetails + "\n" + args.ErrorValue);
}

private void ShowInsertResult()
{
    lblStatus.Content = "New item added to " + _projects.Title;
}

private void OnConnectSucceeded(Object sender, SP.ClientRequestSuc
ceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(AddItemToList);
}

private void OnConnectFailed(object sender,
ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Invoke a delegate and send the args instance as a parameter
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));
}

private void OnAddItemToListSucceeded(Object sender, SP.ClientRequ
estSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    //Dispatcher.BeginInvoke(GetListData);
    Dispatcher.BeginInvoke(ShowInsertResult);
}
```



```
}  
  
private void OnAddItemToListFailed(object sender,  
ClientRequestFailedEventArgs args)  
{  
    // This callback isn't called on the UI thread  
    // Invoke a delegate and send the args instance as a parameter  
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));  
}
```

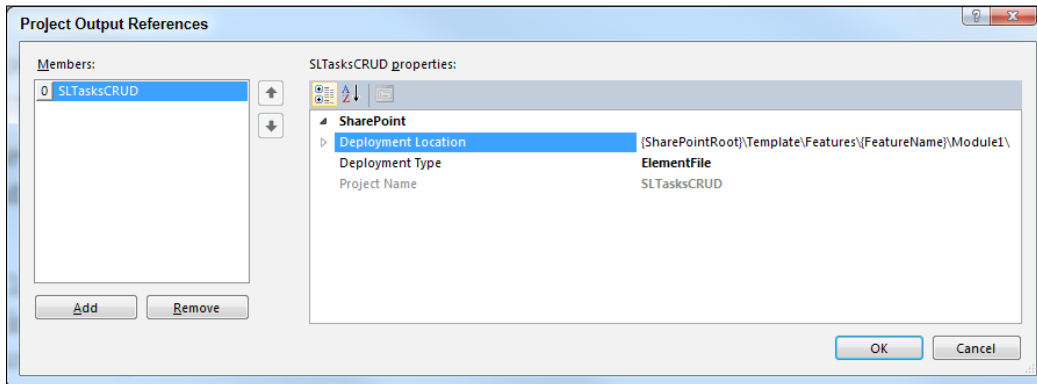
Add the following line to the `Click` event for the `butInsert` button. This way, when the user clicks on this button, the application will connect to the SharePoint server and will insert the new item.

```
private void butInsert_Click(object sender, RoutedEventArgs e)  
{  
    ConnectAndAddItemToList();  
}
```

Now, follow these steps to create a new SharePoint module and link it to the previously created Silverlight RIA, `SLTasksCRUD`.

1. Stay in Visual Studio as a system administrator user.
2. Right-click on the solution's name in **Solution Explorer** and select **Add | New Project...** from the context menu that appears.
3. Select **Visual C# | SharePoint | 2010** under **Installed Templates** in the **New Project** dialog box. Then, select **Empty SharePoint Project**, enter `SPTasksCRUD` as the project's name, and click **OK**. The **SharePoint Customization Wizard** dialog box will appear.
4. Enter the URL for the SharePoint server and site in **What local site do you want to use for debugging?**
5. Click on **Deploy as a sandboxed solution**. Then, click on **Finish** and the new `SPTasksCRUD` empty SharePoint 2010 project will be added to the solution.
6. Add a new item to the project, that is a SharePoint 2010 module, `Module1`.
7. Expand the new SharePoint 2010 module, `Module1`, in the **Solution Explorer** and delete the `Sample.txt` file.
8. Now, right-click on `Module1` and select **Properties** in the context menu that appears. In the **Properties** palette, click the ellipsis (...) button for the **Project Output References** property. The **Project Output References** dialog box will appear.
9. Click on **Add**, below the **Members** list. The empty SharePoint 2010 project's name, `SPTasksCRUD`, will appear as a new member.

10. Go to its properties, shown in the list, located at the right. Select the Silverlight application project's name, `SLTasksCRUD`, in the **Project Name** drop-down list.
11. Select `ElementFile` in the **Deployment Type** drop-down list. The following value will appear in **Deployment Location**: `{SharePointRoot}\Template\Features\{FeatureName}\Module1\`, as shown in the next screenshot:



Click **OK** and the SharePoint project now includes the Silverlight application project, `SLTasksCRUD`.

12. Now, right-click on the SharePoint 2010 project, `SPTasksCRUD`, and select **Properties** in the context menu that appears. Click on the **SharePoint** tab in the properties panel and different options for the SharePoint deployment configuration will be shown.
13. Activate the **Enable Silverlight debugging (instead of Script debugging)** checkbox. Remember that this option will allow us to debug code in the Silverlight application that adds items to the list in the SharePoint server.
14. Right-click on the solution's name in **Solution Explorer** and select **Properties** from the context menu that appears. Select **Startup Project** in the list on the left, activate **Single startup project**, and choose the SharePoint project's name in the drop-down list below it, `SPTasksCRUD`. Then, click **OK**.
15. Build and deploy the solution.
16. Now that the WSP package has been deployed to the SharePoint site, follow the necessary steps to create a new web page, add the Silverlight Web Part, and include the Silverlight RIA in it. Remember that in this case, it is not necessary to upload the `.xap` file because it was already deployed with the WSP package.

Inserting items in a SharePoint list with the Silverlight Web Part

Now, follow these steps to insert an item with the recently deployed Silverlight RIA running in a Silverlight Web Part.

1. Enter the URL for the previously added page that contains the Silverlight Web Part in the web browser. This way, the Silverlight RIA will appear.
2. Enter a value for the **Title**. Select a value for both the **Priority** and the **Status** drop-down lists and use the slider to specify the percentage of the work completed so far and select both the **Start Date** and the **Due Date** by clicking on the datetime pickers. The following screenshot shows some values and the elegant drop-down list that offers the five alternatives for **Status**:

The screenshot displays a web browser window with the address bar showing 'Home > SilverlightProjectsCRUD2010'. The page content includes a search bar and a sidebar with navigation links like 'Recently Modified', 'Pictures', 'Sites', and 'Documents'. The main area is titled 'Silverlight SharePoint Tasks CRUD' and contains a form with the following fields:

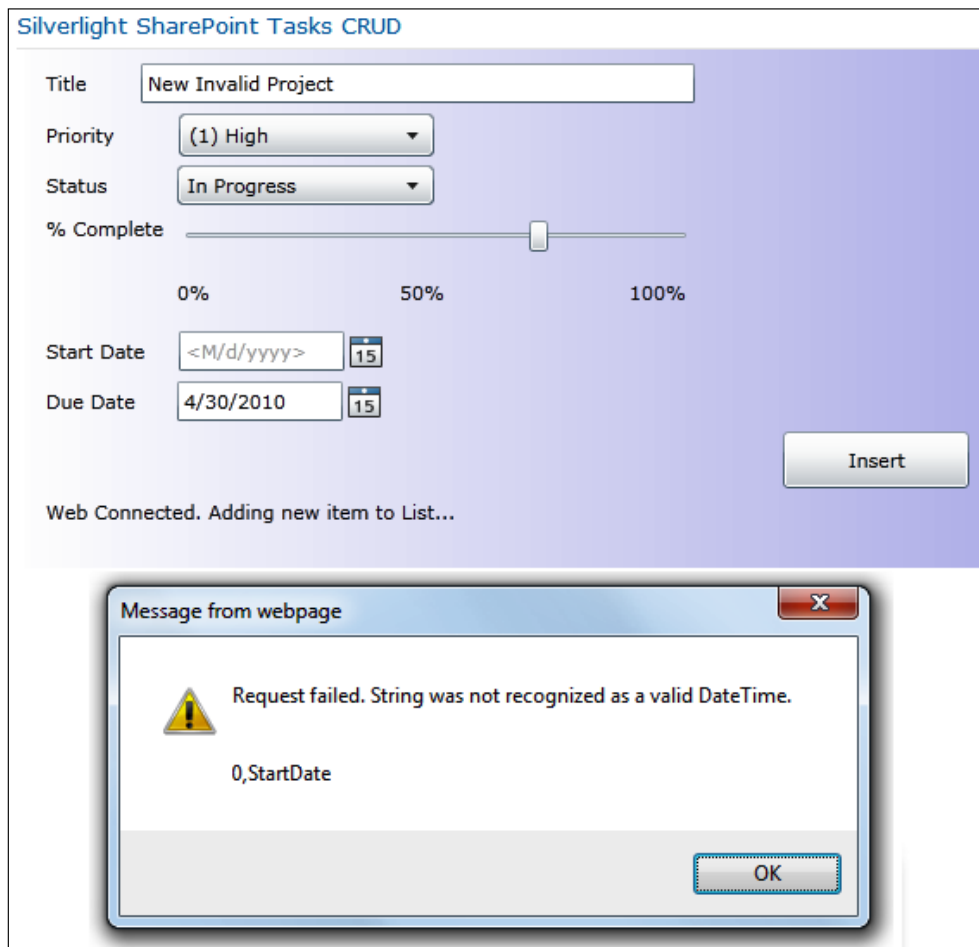
- Title: Testing Silverlight 4 Client OM with Lists in SharePoint 2
- Priority: (1) High
- Status: Not Started (dropdown menu is open, showing options: Not Started, In Progress, Completed, Deferred, Waiting on someone else)
- % Complete: 0% (slider to 100%)
- Start Date: [empty]
- Due Date: [empty]
- Started: [empty]

An 'Insert' button is located at the bottom right of the form.

3. Click on the **Insert** button. The application is going to display its different status values in the label located at the bottom:
 - **Ready**
 - **Started**
 - **Web Connected. Adding new item to List...**
 - **New item added to ProjectsList2010**
4. Open or refresh the items for the list in the corresponding SharePoint 2010 page and you will see the new item added to the list with the values entered in the application and the user logged on to the SharePoint server in the **Assigned To** column. The following screenshot shows the new item in the list:

Type	Title	Assigned To	Status	Priority	Due Date	% Complete
Document	Creating a Silverlight 4 UI	gaston-PC\hillar2010	Not Started	(2) Normal	2/20/2010	30 %
Document	Creating a Complex Silverlight 4 LOB RIA	GASTON-PC\gaston	Not Started	(2) Normal	4/5/2010	50 %
Document	Testing Silverlight 4 Client OM with Lists in SharePoint 2010	gaston-PC\hillar2010	Not Started	(2) Normal	4/29/2010	73 %

- The Silverlight RIA doesn't include code to validate the data that is going to be added to the list in the SharePoint server. Thus, the user can enter inappropriate values for the fields. Enter a new title and delete the value for **Start Date**. Then, click on the **Insert** button and a new dialog box will appear, indicating that the request failed, because the `String` was not recognized as a valid `DateTime`. The problem is that the `StartDate` field has the invalid value `0`. The following screenshot shows the dialog box with the error message:



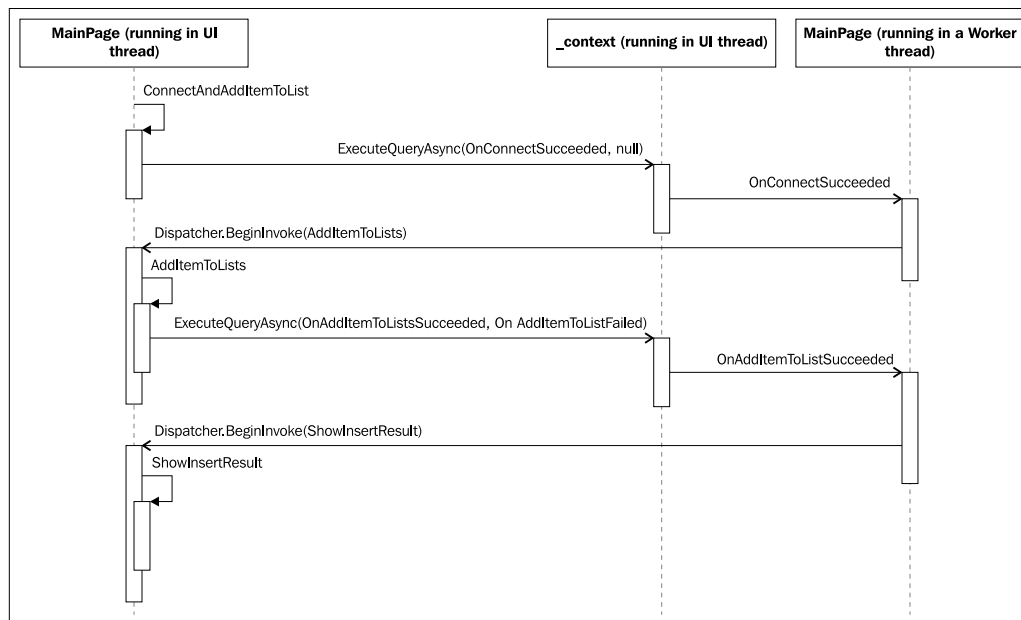
This dialog box is the result of the execution of the `OnAddItemToListFailed` callback, the second parameter of the `_context.ExecuteQueryAsync` method in `AddItemToList`. As something went wrong, this callback invokes a delegate that send the `args` instance as a parameter to the `ShowErrorInformation` method.

```
Dispatcher.BeginInvoke(() => ShowErrorInformation(args));
```

6. Debug the Silverlight RIA.

Working with successful and failed asynchronous queries

The following sequence diagram shows the interaction between the methods defined in `MainPage` that are going to run in the UI thread, the `Microsoft.SharePoint.Client.ClientContext` instance, `_context`, and the methods defined in `MainPage` that are going to run in another thread, that is, a worker thread. This sequence represents the situation in which all the asynchronous operations against the SharePoint server have a successful completion:



When the user clicks the **Insert** button, the `Click` event handler calls the `ConnectAndAddItemToList` method, in the UI thread. This method uses the current URL to generate a `ClientContext` instance, saved in the `private_context` variable. Then, it calls the `Load` method to build a query to load the web and its current user, the user that logged on to the SharePoint site. Then, it calls the `ExecuteQueryAsync` to run it with an asynchronous execution.

```
_context.Load(_context.Web);  
_context.Load(_context.Web.CurrentUser);
```

If the query has a successful execution, the `OnConnectSucceeded` callback schedules the asynchronous execution of the `AddItemToList` method in the UI thread from a worker thread that runs this code after the query has succeeded.

```
private void OnConnectSucceeded(Object sender, SP.ClientRequestSucceededEventArgs args)  
{  
    Dispatcher.BeginInvoke(AddItemToList);  
}
```

In this example, we also specified the `OnConnectFailed` callback and used it as the second parameter for the `ExecuteQueryAsync` method. If something goes wrong, it invokes a delegate that calls the `ShowErrorInformation` method and sends the `ClientRequestFailedEventArgs args` instance as a parameter to it. The code uses a **lambda expression** to define the delegate:

```
private void OnConnectFailed(object sender,  
ClientRequestFailedEventArgs args)  
{  
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));  
}
```



Remember that a lambda expression, introduced in C# 3.0, is an anonymous function that can contain expressions and statements and can be used to create delegates or expression tree types. They are useful to simplify the code when we use delegates. All lambda expressions use the lambda operator `=>` (read as **goes to**). Lambda expressions are described in depth in *WCF Multi-tier Services Development with LINQ* by Mike Liu, Packt Publishing.

The following lines show equivalent code to define a delegate and invoke it to run asynchronously in the UI thread without using a lambda expression. It requires more lines of code, because it is necessary to declare a delegate type, create a new instance with the method to run, use the `Dispatcher.BeginInvoke` method to call the delegate instance, and send the `args` instance as a parameter encapsulated in an array of object.

```
private delegate void ShowErrorInformationCaller
(ClientRequestFailedEventArgs args);

private void OnConnectFailed(object sender,
ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Create the delegate instance
    ShowErrorInformationCaller ShowErrorInformationD =
    new ShowErrorInformationCaller(ShowErrorInformation);
    // Invoke the delegate
    Dispatcher.BeginInvoke(
    ShowErrorInformationD, new object[] { args });
}
```



It is convenient to use lambda expressions, because they require less code to achieve the same goal. However the previously shown lines make it easier to understand the way the method is called in the delegate.

If everything works as expected, then the `AddItemToList` method is going to run in the UI thread. This method calls the `GetByTitle` method to save a reference to the `ProjectsList2010` list, in the `_projects` private variable. Then, it calls its `AddItem` method to add a new `ListCreationInformation` empty instance. This method returns a new `ListCreationInformation` instance that allows us to access the fields for the new item in the list and fill their values.

```
_projects = _context.Web.Lists.GetByTitle("ProjectsList2010");
ListItem listItem = _projects.AddItem(new
ListCreationInformation());
```


Then, the code completes the value for each field by using its `InternalName` and assigning a string value, as shown in the next line.

```
listItem["Title"] = txtTitle.Text;
```

The `AssignedTo` field is a special case, because it is a `FieldUserValue` that references a SharePoint server user through a lookup ID. Remember that this field had a `Microsoft.SharePoint.SPFieldUserValue` value in its `FieldValueType` property. Thus, it is necessary to use the `Id` for the user currently logged on the SharePoint server to assign it to the `LookupId` property of a new `FieldUserValue` instance, `fieldUserValue`. Then, it is possible to assign `fieldUserValue` to `listItem["AssignedTo"]` to store the current user as the value for this field.

```
var fieldUserValue = new FieldUserValue();
fieldUserValue.LookupId = _context.Web.CurrentUser.Id;
listItem["AssignedTo"] = fieldUserValue;
```

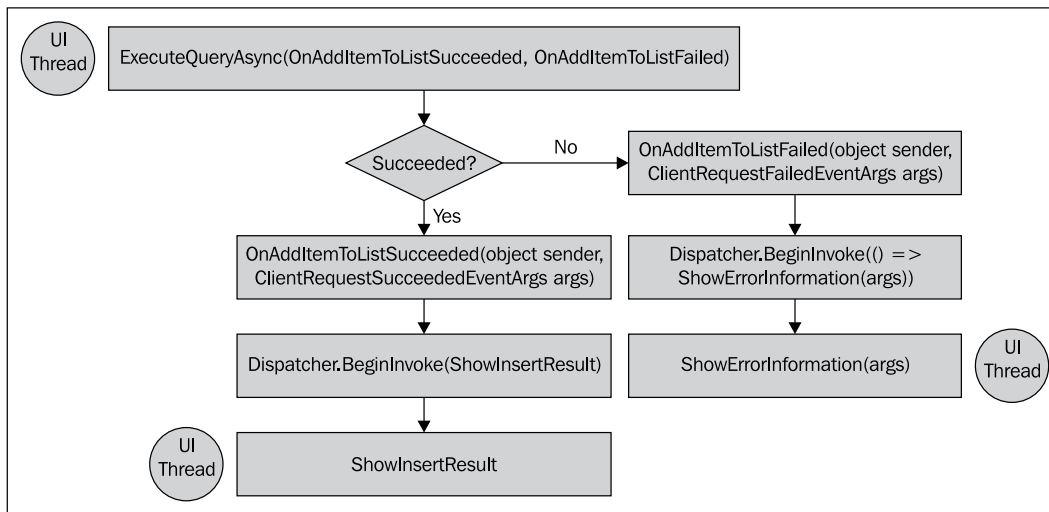


Remember, it is possible to access the `_context.Web.CurrentUser.Id` property because we queried `_context.Web.CurrentUser` in the `ConnectAndAddItemToList` method.

Once all the fields are filled with the corresponding values, the code calls the `Update` method for the new `ListItem` instance that holds the new row, `listItem`. At this point, the new item isn't still inserted in the list, because it is necessary to execute the query. The code requests the `Title` for the list as a response and then calls the `ExecuteQueryAsync` method:

```
listItem.Update();
_context.Load(_projects, list => list.Title);
_context.ExecuteQueryAsync(OnAddItemToListSucceeded,
OnAddItemToListFailed);
```

The following diagram shows the detailed execution flowchart for the asynchronous query that adds the item to the list. Besides, it indicates the code that runs in the UI thread. If the query execution isn't successful, the application will run the `OnAddItemToListFailed` callback and it will display information about the error that made the query fail in a dialog box. If the query execution succeeds, the application will run the `OnAddItemToListSucceeded` callback and it will display status information to let the user know that the item was inserted in the list.



Retrieving specific information about fields

If we examine the default dialog box that allows a user to insert items into the list in the SharePoint server, we will notice that there is a default value for both the **Priority** and **Status** fields, as shown in the following screenshot:

- (2) Normal for Priority
- Not Started for Status

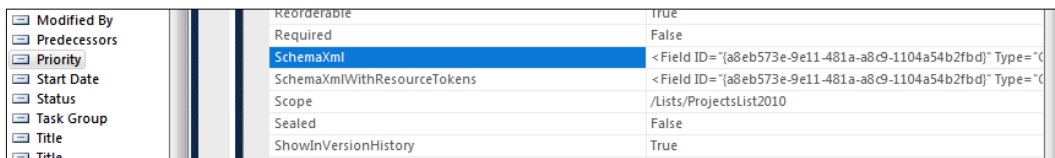
Priority	(2) Normal
Status	Not Started

Besides, when we use this dialog box to insert a new item or edit an existing one, the two drop-down lists offer many choices as their possible values.

Priority	(2) Normal
Status	Not Started
% Complete	Not Started In Progress Completed
Assigned To	Deferred Waiting on someone else
Description	

Follow these steps to access the value for the SchemaXml property in the **Priority** and **Status** fields by using Server Explorer.

1. Stay in Visual Studio as a system administrator user.
2. Activate the **Server Explorer** palette and navigate to the previously created list, **ProjectsList2010**.
3. Now, expand the list of tasks, **ProjectsList2010**, and then expand its **Fields** node. This way, you will see all the fields for this list.
4. Now, click on the **Priority** field, display its properties, and check the value for its SchemaXml property. As the content for this property is XML markup, you won't be able to analyze all the information in the **Properties** palette, because you will see only the first characters



5. You can copy the value for the SchemaXml property and paste it in a new **XML File** in Visual Studio. This way, you will be able to see the three choices and their mappings, and a default value, as shown in the following lines and in the next screenshot with the contents pasted in a Visual Studio XML File that organizes the markup code.

```
XMLFile2.xml* X MainPage.xaml.cs* SequenceDiagram1.sequencediagram M
<Field ID="{a8eb573e-9e11-481a-a8c9-1104a54b2fbd}"
  Type="Choice" Name="Priority" DisplayName="Priority"
  SourceID="http://schemas.microsoft.com/sharepoint/v3"
  StaticName="Priority" ColName="nvarchar3">
  <CHOICES>
    <CHOICE>(1) High</CHOICE>
    <CHOICE>(2) Normal</CHOICE>
    <CHOICE>(3) Low</CHOICE>
  </CHOICES>
  <MAPPINGS>
    <MAPPING Value="1">(1) High</MAPPING>
    <MAPPING Value="2">(2) Normal</MAPPING>
    <MAPPING Value="3">(3) Low</MAPPING>
  </MAPPINGS>
  <Default>(2) Normal</Default>
</Field>
```

```

<Field ID="{a8eb573e-9e11-481a-a8c9-1104a54b2fbd}"
  Type="Choice" Name="Priority" DisplayName="Priority"
  SourceID="http://schemas.microsoft.com/sharepoint/v3"
  StaticName="Priority" ColName="nvarchar3">
  <CHOICES>
    <CHOICE>(1) High</CHOICE>
    <CHOICE>(2) Normal</CHOICE>
    <CHOICE>(3) Low</CHOICE>
  </CHOICES>
  <MAPPINGS>
    <MAPPING Value="1">(1) High</MAPPING>
    <MAPPING Value="2">(2) Normal</MAPPING>
    <MAPPING Value="3">(3) Low</MAPPING>
  </MAPPINGS>
  <Default>(2) Normal</Default>
</Field>

```

6. Now, repeat the aforementioned steps (4 and 5) with the **Status** field. You will be able to see the five choices and their mappings, as shown in the following lines:

```

<Field Type="Choice"
  ID="{c15b34c3-ce7d-490a-b133-3f4de8801b76}"
  Name="Status" DisplayName="Status"
  SourceID="http://schemas.microsoft.com/sharepoint/v3"
  StaticName="Status" ColName="nvarchar4">
  <CHOICES>
    <CHOICE>Not Started</CHOICE>
    <CHOICE>In Progress</CHOICE>
    <CHOICE>Completed</CHOICE>
    <CHOICE>Deferred</CHOICE>
    <CHOICE>Waiting on someone else</CHOICE>
  </CHOICES>
  <MAPPINGS>
    <MAPPING Value="1">Not Started</MAPPING>
    <MAPPING Value="2">In Progress</MAPPING>
    <MAPPING Value="3">Completed</MAPPING>
    <MAPPING Value="4">Deferred</MAPPING>
    <MAPPING Value="5">Waiting on someone else</MAPPING>
  </MAPPINGS>
  <Default>Not Started</Default>
</Field>

```

We now have the information that we need to enhance the Silverlight LOB RIA. Follow these steps to add new code that retrieves the choices defined in the SharePoint list for both the **Priority** and **Status** fields and uses their default values.

1. Stay in Visual Studio as a system administrator user, in the TasksCRUD solution.
2. Open `MainPage.xaml.cs` and add the following private variable:
`private SP.FieldCollection _collField;`
3. Now, it is necessary to add code to execute the following tasks:
 1. Connect to the SharePoint server and load information about the **Priority** and **Status** fields from the `ProjectsList2010`, `ConnectAndFillComboBoxes` method.
 2. Iterate through a collection of fields, find and return a `FieldChoice` instance, according to an `InternalName` value, received as a parameter, `ReturnFieldByInternalName` method.
 3. Add each choice defined for the retrieved **Priority** and **Status** `FieldChoice` fields as a new item in the corresponding combo boxes and set their default value, `AddFieldChoicesToComboBoxes` method.

```
private void ConnectAndFillComboBoxes()
{
    // Runs in the UI Thread
    lblStatus.Content = "Started";
    _context = new SP.ClientContext(
        SP.ApplicationContext.Current.Url);
    // Load the Web
    _context.Load(_context.Web);
    // Get the ProjectsList2010 list
    _projects = _context.Web.Lists.
        GetByTitle("ProjectsList2010");
    _context.Load(_projects);
    // Just load the two necessary fields for the List:
    // Status and Priority
    _collField = _projects.Fields;
    _context.Load(_collField,
        fields => fields.Where(
            field => field.InternalName == "Status"
            || field.InternalName == "Priority")
        .IncludeWithDefaultProperties());
}
```

```
        _context.ExecuteQueryAsync(
            OnConnectAndFillComboBoxesSucceeded,
            OnConnectAndFillComboBoxesFailed);
    }

    private SP.FieldChoice ReturnFieldByInternalName(string
internalName)
    {
        for (int i = 0; i < _collField.Count; i++)
        {
            if (_collField[i].InternalName == internalName)
            {
                return (_collField[i] as FieldChoice);
            }
        }
        return null;
    }

    private void AddFieldChoicesToComboBoxes()
    {
        // Runs in the UI Thread
        SP.FieldChoice statusField =
            ReturnFieldByInternalName("Status");
        SP.FieldChoice priorityField =
            ReturnFieldByInternalName("Priority");
        // Add each choice to the corresponding ComboBox control
        foreach (string item in statusField.Choices)
        {
            cboStatus.Items.Add(item);
        }
        foreach (string item in priorityField.Choices)
        {
            cboPriority.Items.Add(item);
        }
        // Set default values
        cboStatus.SelectedValue = statusField.DefaultValue;
        cboPriority.SelectedValue = priorityField.DefaultValue;
    }
}
```

4. All the previously added methods are going to run in the UI thread. The following methods, which are going to be fired as asynchronous callbacks, schedule the execution of other methods to continue with the necessary program flow in the UI thread:
 - When the connection to the SharePoint server and the query execution, requested by the `ConnectAndFillComboBoxes` method, is successful, the `OnConnectAndFillComboBoxesSucceeded` method schedules the execution of the `AddFieldChoicesToComboBox` method in the UI thread.
 - If the `ConnectAndFillComboBoxes` method fails, the `OnConnectAndFillComboBoxesFailed` method schedules the execution of the `ShowErrorInformation` method in the UI thread, sending the `ClientRequestFailedEventArgs` args instance as a parameter to the delegate.

```
private void OnConnectAndFillComboBoxesSucceeded(Object sender,
    SP.ClientRequestSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(AddFieldChoicesToComboBoxes);
}

private void OnConnectAndFillComboBoxesFailed(object sender,
    ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Invoke a delegate and send the args instance as a parameter
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));
}
```

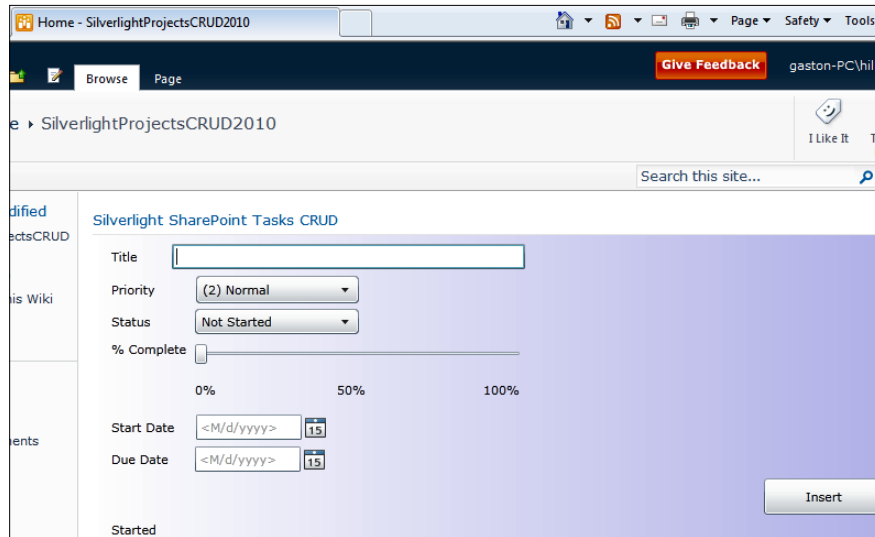
Replace the code in the `MainPage` constructor with the following lines to fill the combo boxes by calling the new `ConnectAndFillComboBoxes` method instead of the previously defined `FillComboBoxes`:

```
public MainPage()
{
    InitializeComponent();

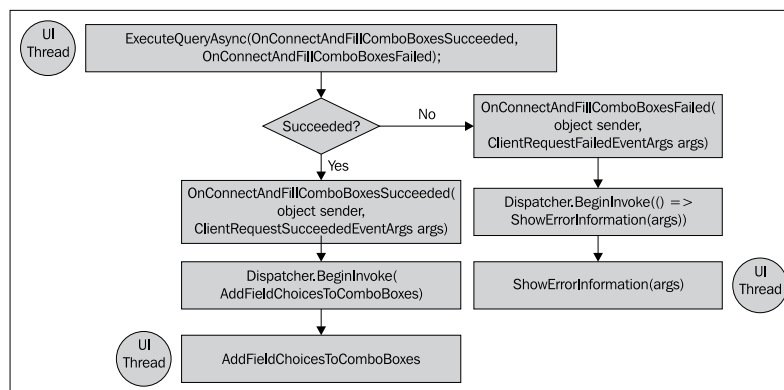
    ConnectAndFillComboBoxes();
}
```

5. Build and deploy the solution.

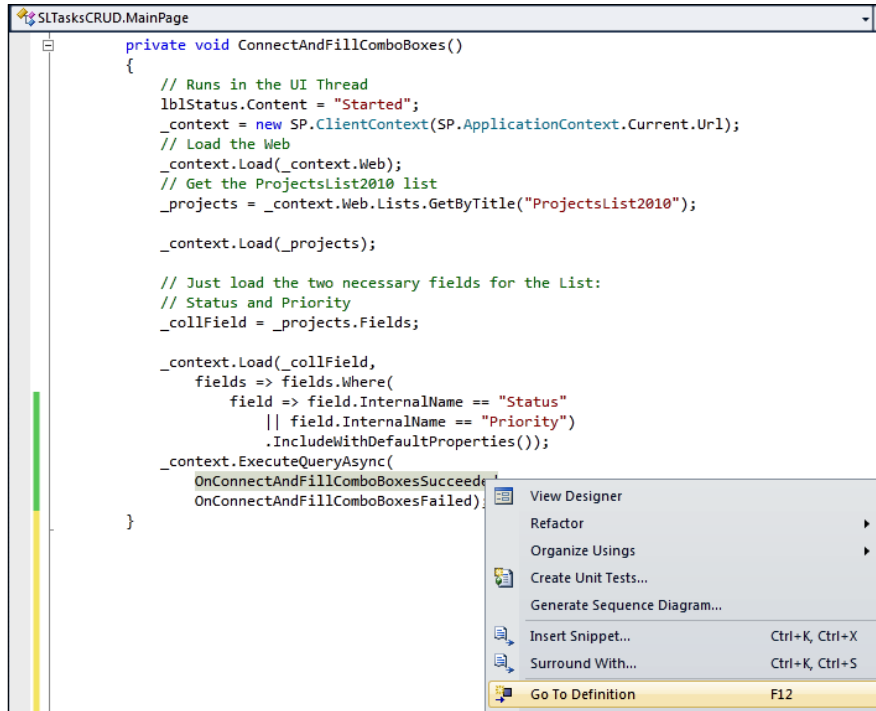
- Enter the URL for the previously added page that contains the Silverlight Web Part in the Web browser. This way, the updated Silverlight RIA will appear and it will load the choices and default values for the `Priority` and `Status` fields from the information retrieved from the list in the SharePoint server:



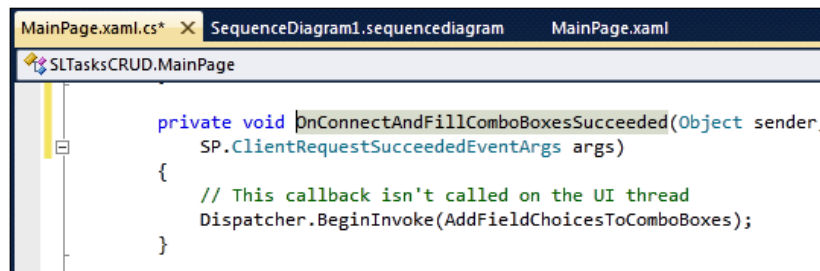
The following diagram shows the detailed execution flowchart for the asynchronous query that retrieves information for both the `Priority` and `Status` fields. The diagram indicates the code that runs in the UI thread. If the query execution isn't successful, the application will run the `OnConnectAndFillComboBoxesFailed` callback and it will display information about the error that made the query fail in a dialog box. If the query execution succeeds, the application will run the `OnConnectAndFillComboBoxesSucceeded` callback and it will fill the two combo boxes with the retrieved choices and will set their default values.



As the asynchronous executions make it a bit difficult to track the execution flow, we can use one of the features offered by the Visual Studio code editor to help us. Locate the cursor over a method's name and press *F12* or right-click on it and select **Go To Definition** in the context menu that appears. For example, you can do it for `OnConnectAndFillComboBoxesSucceeded` in the line that sends this callback as a parameter to the `_context.ExecuteQueryAsync` method.



This way, the code editor will locate the cursor at the line that defines the selected method's name, `OnConnectAndFillComboBoxesSucceeded`. You can take advantage of this feature to track the execution flow by accessing the code for the different methods involved in the sequence.



Now, the `MainPage` constructor calls the new `ConnectAndFillComboBoxes` method that creates a new `ClientContext` instance, requests it to load the web and the list, `ProjectsList2010`. Then, it is necessary to load just two fields, `Priority` and `Status`, and therefore, we specified a LINQ expression as the second parameter for the `Load` method to limit the information that has to be retrieved. We used a lambda expression as a parameter of the `Where` method to filter the sequence of values based on a predicate. We just wanted to retrieve the fields whose `InternalName` property value was `Status` or `Priority`. We added the call to the `IncludeWithDefaultProperties` extension method without parameters, because we just want to retrieve the default properties for the two `Field` instances in the `_collField` `FieldCollection`.

```
_context.Load(_collField,
    fields => fields.Where(
        field => field.InternalName == "Status"
        || field.InternalName == "Priority")
        .IncludeWithDefaultProperties());
```

This way, we query for two fields and we request their default properties. Once this query is successful, the `AddFieldChoicesToComboBoxes` method retrieves the two fields from the `_collField` `FieldCollection` as `FieldChoice` instances, `statusField` and `priorityField`. The `ReturnFieldByInternalName` method receives a string with the required value for the `InternalName` property and returns the field from the `_collField` `FieldCollection` that satisfies this simple condition as a `FieldChoice` instance. The `FieldChoice` class, `Microsoft.SharePoint.Client.FieldChoice`, represents a choice field control. As this is the real class for the two field instances, it is necessary to cast them to `FieldChoice` to access the specific field and properties that allow us to retrieve the choices.

```
SP.FieldChoice statusField =
    ReturnFieldByInternalName("Status");
SP.FieldChoice priorityField =
    ReturnFieldByInternalName("Priority");
```

Then, the code adds a new item to each combo box, `cboStatus` and `cboPriority`, for each string in the `FieldChoice` instance `Choices` string array. The following lines show the `foreach` loop that adds items to the `cboStatus` `ComboBox`:

```
foreach (string item in statusField.Choices)
{
    cboStatus.Items.Add(item);
}
```

Once the code has filled the two combo boxes with the possible choices, it assigns the value for each field `DefaultValue` property to the `ComboBox.SelectedValue` property. This way, the drop-down list displayed by these controls will show the same default value as the dialog box that allows inserting items in the SharePoint list.

```
cboStatus.SelectedValue = statusField.DefaultValue;
```

Creating complex LOB applications composed of multiple Silverlight RIAs

One of the interesting features provided by Silverlight applications is the possibility to establish a simple communication channel between them and use it to send messages between many applications. This feature is very useful when we have many Silverlight applications included in Web Parts. Silverlight applications running on the same computer can communicate over the boundaries of tabs inside a web browser and even over the limits of web browser instances.

Follow these steps to add new code that sends a message to a listener application when the application inserts a new item in the list:

1. Stay in Visual Studio as a system administrator user, in the `TasksCRUD` solution.
2. Open `MainPage.xaml.cs` and add the following using statements to include the `System.Windows.Messaging` namespace:

```
using System.Windows.Messaging;
```
3. Add the following two private constants:

```
private const string MSG_RECEIVER_NAME = "MessageReceiver";  
private const string MSG_TASKSCRUD_NEWITEM = "TASKSCRUD_NEWITEM";
```
4. Add the following method that sends the message received as a parameter to the receiver:

```
private void SendMessage(string message)  
{  
    // Create a new LocalMessageSender instance  
    // with the receiver name as a parameter  
    LocalMessageSender messageSender =  
        new LocalMessageSender(MSG_RECEIVER_NAME);  
  
    // Attach a SendCompletedEventArgs handler  
    messageSender.SendCompleted +=  
        (object s, SendCompletedEventArgs args) =>
```

```

    {
        // Update the status label
        lblStatus.Content = "Message sent successfully. " +
            "Response: " + args.Response;
    };
    // Send the asynchronous message to the receiver
    messageSender.SendAsync(message);
}

```

5. Add the following line in the ShowInsertResult method to send a message to another Silverlight application when the item was successfully added to the list:

```

private void ShowInsertResult()
{
    lblStatus.Content =
        "New item added to " + _projects.Title;

    // Send a message to another Silverlight RIA
    // in order to force a refresh
    SendMessage(MSG_TASKSCRUD_NEWITEM);
}

```

6. Build and deploy the solution.

Now, follow these steps to add new code that listens to the messages sent by the previously modified Silverlight application and runs code as a response to them:

1. Stay in Visual Studio as a system administrator user or run a new instance.
2. Open the TasksViewer solution.
3. Open MainPage.xaml.cs and add the following using statements to include the System.Windows.Messaging namespace:

```
using System.Windows.Messaging;
```

4. Add the following two private constants:

```

private const string MSG_RECEIVER_NAME = "MessageReceiver";
private const string MSG_TASKSCRUD_NEWITEM = "TASKSCRUD_NEWITEM";

```

5. Add the following code to the `MainPage` constructor that starts receiving messages and refreshes the items from the list shown in the `DataGrid`, `dataGridProjects`, each time it receives the `MSG_TASKSCRUD_NEWITEM` message:

```
public MainPage()
{
    InitializeComponent();
    LocalMessageReceiver messageReceiver =
        new LocalMessageReceiver(MSG_RECEIVER_NAME);

    messageReceiver.MessageReceived +=
        ( object sender, MessageReceivedEventArgs e ) =>
        {
            if (e.Message == MSG_TASKSCRUD_NEWITEM)
            {
                Connect();
            }
            e.Response = "OK";
        };

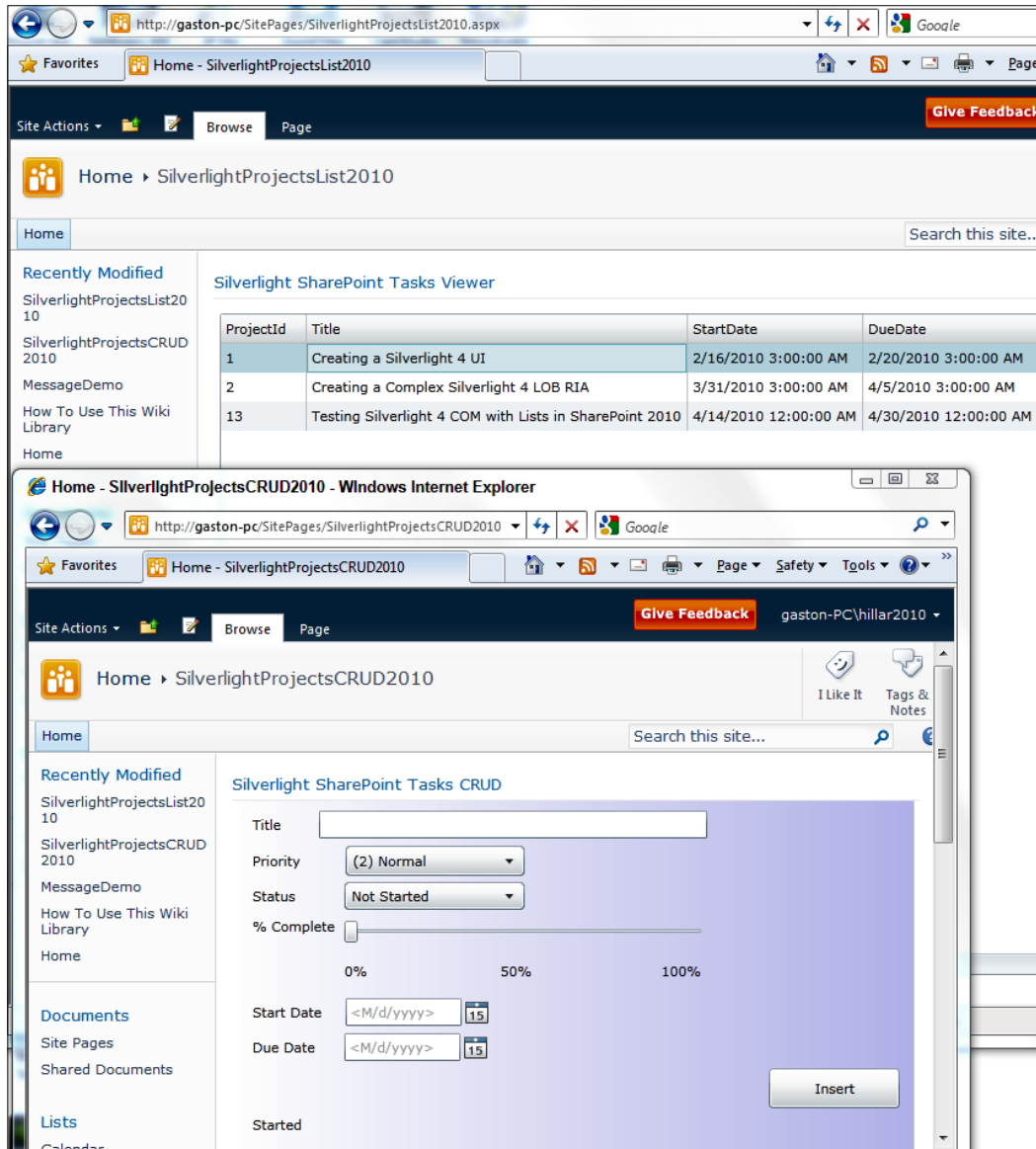
    try
    {
        messageReceiver.Listen();
    }
    catch (ListenFailedException)
    {
        // There is another receiver with the same name
        // and the application cannot receive messages
        lblStatus.Content = "Cannot receive messages.";
    }
}
```

6. Build and deploy the solution.

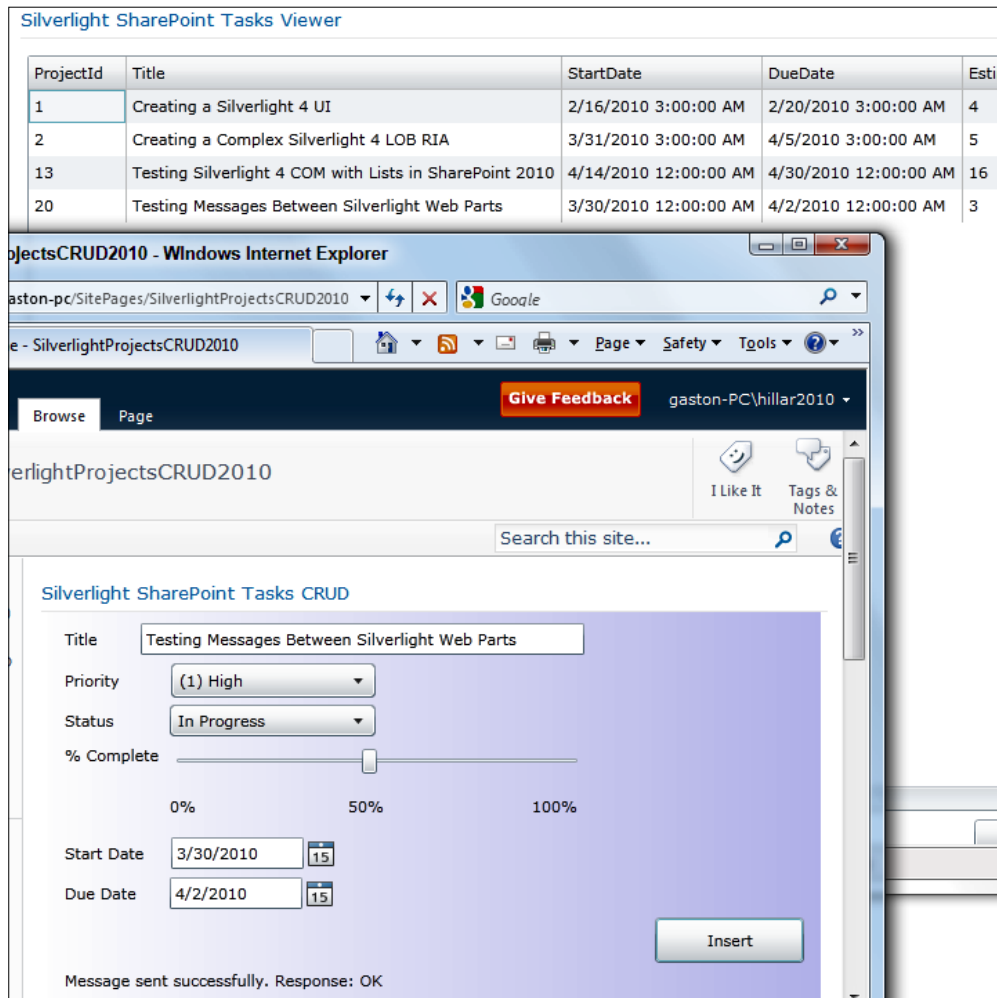
Follow these steps to test the two Silverlight Web Parts in different web browser instances:

1. Start two web browser instances.
2. Load the SharePoint page that shows `SLTasksViewer.xap` as a Silverlight Web Part in one web browser's window, `SilverlightProjectsList2010.aspx`.

3. Load the SharePoint page that shows `SLTasksCRUD.xap` as a Silverlight Web Part in the other web browser's window, `SilverlightProjectsCRUD2010.aspx`. Make both Web Parts visible at the same time.



- Complete the data to insert a new item in the list and click on the **Insert** button. The new item will be added to the list and the Silverlight Web Part that displays the grid will refresh its contents to show the new item, because it receives the message from the other Silverlight Web Part. The status label at the bottom of `SLTasksCRUD.xap` will display **Message sent successfully. Response: OK.**, because it receives OK as a response from `SLTasksViewer.xap`, the listener application:



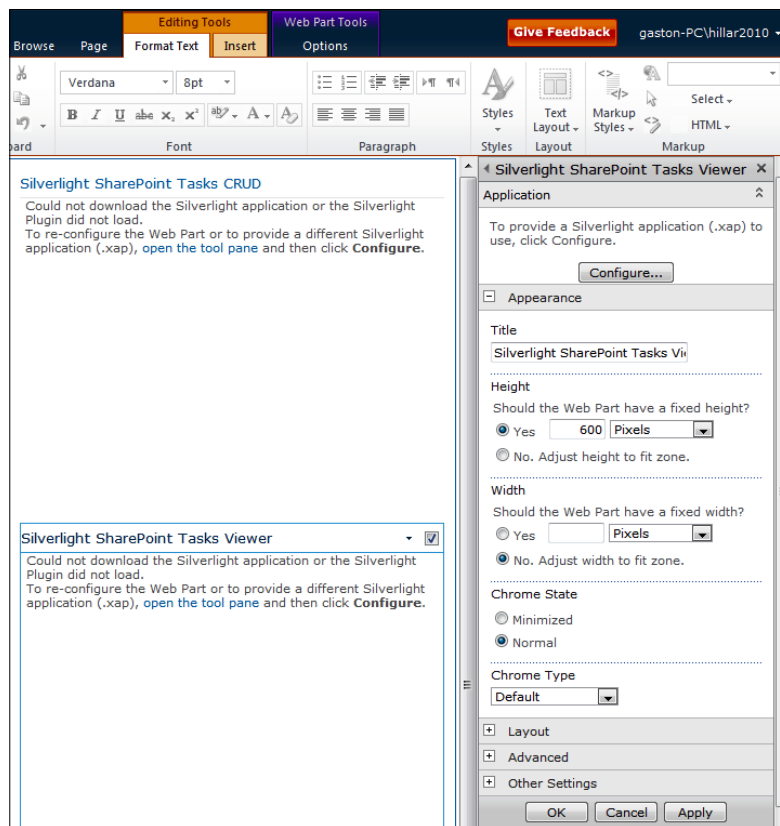
- Test the behavior by showing each Web Part in a different browser tab and you will achieve the same results.

Interacting with multiple Silverlight Web Parts in the same page

We took advantage of Silverlight's messaging capabilities to establish a communication channel between multiple Silverlight Web Parts in SharePoint. This way, it is possible to create complex solutions composed of multiple Silverlight Web Parts that interact with each other. Then, we can decide the best layout for these Silverlight Web Parts in one or many SharePoint pages.

Follow these steps to test the two Silverlight Web Parts in the same SharePoint page:

1. Open the web page that shows `SLTasksCRUD.xap` as a Silverlight Web Part in the other web browser's window, `SilverlightProjectsCRUD2010.aspx`.
2. Click **Site Actions | Edit Page** and SharePoint will display the editing tools for this page.
3. Follow the necessary steps to insert a second Silverlight Web Part to this page, `SLTasksViewer.xap`. This way, the page will show `SLTasksCRUD.xap` at the top and `SLTasksViewer.xap` at the bottom.



- Remember to apply the settings learned in the previous examples to the new Silverlight Web Part and click on the **Save** button in the ribbon. Now, the new page will appear, displaying the two previously created Silverlight RIAs running as two Silverlight Web Parts in the same page.
- Complete the data to insert a new item in the list and click on the **Insert** button. The new item will be added to the list and the Silverlight Web Part that displays the grid at the bottom of the page will refresh its contents to show the new item, because it receives the message from the other Silverlight Web Part.

Silverlight SharePoint Tasks CRUD

Title:

Priority:

Status:

% Complete:


Start Date:

Due Date:

Message sent successfully. Response: OK

Silverlight SharePoint Tasks Viewer

ProjectId	Title	StartDate	DueDate	EstimatedDa
1	Creating a Silverlight 4 UI	2/16/2010 3:00:00 AM	2/20/2010 3:00:00 AM	4
2	Creating a Complex Silverlight 4 LOB RIA	3/31/2010 3:00:00 AM	4/5/2010 3:00:00 AM	5
13	Testing Silverlight 4 COM with Lists in SharePoint 2010	4/14/2010 12:00:00 AM	4/30/2010 12:00:00 AM	16
20	Testing Messages Between Silverlight Web Parts	3/30/2010 12:00:00 AM	4/2/2010 12:00:00 AM	3
26	Testing Two Silverlight Web Parts Interacting in the Same Page	3/29/2010 12:00:00 AM	4/21/2010 12:00:00 AM	23

 The user inserts a new item into the list and doesn't need to refresh the Silverlight Web Part that displays the data for the list, because the application receives the message and updates its content. This simple example demonstrates one of the most interesting possibilities offered by Silverlight RIAs included as Silverlight Web Parts: they make it simpler to refresh content without having to reload a page in the Web browser.

Understanding Line-Of-Business systems as independent Web Parts

The `System.Windows.Messaging` namespace offers the necessary classes and event types that allowed us to send messages between two Silverlight Web Parts running on the same computer.

The `SendMessage` method in the `SLTasksCRUD` Silverlight application receives the message to be sent as a `string` parameter. It creates a new `LocalMessageSender` instance (`System.Windows.Messaging.LocalMessageSender`) with the receiver name, defined in the `MSG_RECEIVER_NAME` constant, as a parameter for the constructor.

```
LocalMessageSender messageSender =  
    new LocalMessageSender(MSG_RECEIVER_NAME);
```

Then, it attaches a `SendCompleted` event handler to the `SendCompleted` event for the `LocalMessageSender` instance, `messageSender`. In this case, a lambda expression defines the code to run in the event handler that receives the `SendCompletedEventArgs` `args` as the second parameter. This event will fire when the message has been sent. If everything worked as expected, `args.Response` will contain the response added by the receiver to acknowledge the message's reception. The code just updates the text shown in the `lblStatus` `Label` and adds the value for `args.Response` to this text. This value should be `OK` if the `SLTasksViewer` Silverlight application received the message.

```
messageSender.SendCompleted +=  
    (object s, SendCompletedEventArgs args) =>  
{  
    // Update the status label  
    lblStatus.Content = "Message sent successfully. " +  
        "Response: " + args.Response;  
};
```

If the message wasn't received, the `args.Error` property would be set to a `SendFailedException` instance. In this case, in order to keep the example simple, the code doesn't consider this situation. However, when you work with more complex Silverlight Web Parts, it is convenient to add more code to handle the potential exceptions that could be thrown.

Then, the code sends the `string` received as a parameter, `message`, as an asynchronous message to the previously specified receiver. Remember that the code defined in the previously explained event handler will run after the receiver receives the asynchronous message.

```
messageSender.SendAsync(message);
```

Each time a user adds a new item to a list, the `SLTasksCRUD` Silverlight application calls the `SendMessage` method to send the `MSG_TASKSCRUD_NEWITEM` string to the registered listener.

```
SendMessage(MSG_TASKSCRUD_NEWITEM);
```

The `MainPage` constructor in the `SLTasksViewer` Silverlight application creates a new `LocalMessageReceiver` instance (`System.Windows.Messaging.LocalMessageReceiver`) with the receiver name, defined in the `MSG_RECEIVER_NAME` constant, and with the same value as the one defined in `SLTasksCRUD`, as a parameter for the constructor.

```
LocalMessageReceiver messageReceiver =  
new LocalMessageReceiver(MSG_RECEIVER_NAME);
```

Then, the code performs the following sequence:

It attaches a `MessageReceived` event handler to the `MessageReceived` event for the `LocalMessageReceiver` instance, `messageReceiver`.

A lambda expression defines the code to run in the event handler that receives the `MessageReceivedEventArgs` `e` as the second parameter.

The event will fire if the `Listen` method was called and a message was received for the registered receiver. `e.Message` contains a string with the received message and it is possible to send a response to the sender by assigning a string to `e.Response`.

If the message received is `MSG_TASKSCRUD_NEWITEM`, it means that the `SLTasksCRUD` Silverlight application added a new item to the list that this application is showing in a `DataGrid`. Thus, it is necessary to refresh the data shown in the grid and the code calls the `Connect` method to query the data from the SharePoint server and assigns `OK` as a response to the sender to acknowledge the message's reception.

```
messageReceiver.MessageReceived += (object sender,  
MessageReceivedEventArgs e) =>  
{  
    if (e.Message == MSG_TASKSCRUD_NEWITEM)  
    {  
        Connect();  
    }  
}
```

```
e.Response = "OK";  
};
```

In order to be able to receive messages, it is necessary to call the `Listen` method for the `LocalMessageReceiver` instance, `messageReceiver`. This method will throw a `ListenFailedException`, if something goes wrong. For example, if there is another receiver registered with the same name, this application is not going to be able to receive messages targeting the indicated receiver. Thus, the code encloses the call to the `Listen` method in a `try-catch` block.

If the call to the `Listen` method is successful, each time a message is sent to the registered receiver, this application will run the code in the previously explained `MessageReceived` attached event handler. In this case, in order to keep things simple, the application doesn't use flags to determine whether a received message that suggests a data refresh is being processed or not. However, as the refresh process can take some time, because it requests data from the SharePoint server and at the same time, other messages can arrive, it is convenient to add some kind of mechanism to run only one refresh process at a time.

Expanding LOB systems with delete operations

So far, we have created Silverlight Web Parts that can retrieve the items from a list in the SharePoint server, add new items to this list, and communicate between themselves to refresh the data shown to the user when necessary.

Now, follow these steps to add a new button in the `SLTasksViewer` Silverlight application to allow the user to delete the item selected in the `DataGrid` from the list in the SharePoint server, `ProjectsList2010`.

1. Stay in Visual Studio as a system administrator user, in the `TasksViewer` solution.
2. Open `MainPage.xaml`, add a new `Button` control, `butDelete`, and set its `Content` property to `Delete`.
3. Add a new `DataPager` control, `dataPager` and locate it at the bottom of the `dataGridProjects DataGrid`. Set its `DisplayMode` property to `Numeric` and `PageSize` to 5. This control will simplify the navigation and will display 5 items per page.

4. Apply data binding to the Source property for the DataPager control, dataPager. Click **Apply Data Binding...**, select ElementName in Source, dataGridProjects and then ItemsSource in Path. This way, the Source property will be set to dataGridProjects.ItemsSource. The XAML markup that defines the data binding will be the following:

```
Source="{Binding ElementName=dataGridProjects, Path=ItemsSource}"
```

5. Open MainPage.xaml.cs and add the following private variable:

```
private System.Windows.Data.PagedCollectionView  
_projectsPagedView;
```

6. Replace the line that assigns the value for ItemSource in the ShowItems method, dataGridProjects.ItemsSource = _projectsList;, with the following lines. Now, the ItemsSource property will have a representation of a view of _projectsList for navigating a paged data collection.

```
_projectsPagedView = new System.Windows.Data.PagedCollectionView(  
_projectsList);  
dataGridProjects.ItemsSource = _projectsPagedView;
```

7. Now, add the following code to the Click event for the butDelete Button. This way, when the user clicks on this button, the application will retrieve the Project instance selected in the dataGridProjects DataGrid, and use its ID to retrieve and delete the item from the list in the SharePoint server.

```
private void butDelete_Click(object sender, RoutedEventArgs e)  
{  
    _context = new  
        SP.ClientContext(SP.ApplicationContext.Current.Url);  
    _projects =  
        _context.Web.Lists.GetByTitle("ProjectsList2010");  
    var selectedProject =  
        (dataGridProjects.SelectedItem as Project);  
    SP.ListItem listItem =  
        _projects.GetItemById(selectedProject.ProjectId);  
    // Remove the item from the list  
    listItem.DeleteObject();  
    _context.ExecuteQueryAsync(  
        OnDeleteSucceeded, OnDeleteFailed);  
}
```

8. The code in the `Click` event for the `butDelete` Button is going to run in the UI thread. The following methods, which are going to be fired as asynchronous callbacks, schedule the execution of other methods to continue with the necessary program flow in the UI thread:
- When the connection to the SharePoint server and the query execution that deletes an item from the list, requested by the `butDelete_Click` method, is successful, the `OnDeleteSucceeded` method schedules the execution of the `Connect` method in the UI thread, to refresh the data shown in the `dataGridProjects` DataGrid.
 - If the `butDelete_Click` method fails, the `OnDeleteFailed` method schedules the execution of the `ShowErrorInformation` method in the UI thread, sending the `ClientRequestFailedEventArgs` `args` instance as a parameter to the delegate.

```
private void ShowErrorInformation(ClientRequestFailedEventArgs
args)
{
    MessageBox.Show("Request failed. " + args.Message + "\n"
+ args.StackTrace + "\n" + args.ErrorDetails + "\n" + args.
ErrorValue);
}

private void OnDeleteSucceeded(Object sender, SP.ClientRequestSucc
eededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(Connect);
}

private void OnDeleteFailed(Object sender,
SP.ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Invoke a delegate and send the args instance as a parameter
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));
}
```

9. Build and deploy the solution.

10. Load the SharePoint page that shows `SLTasksViewer.xap` as a Silverlight Web Part, `SilverlightProjectsList2010.aspx`.
11. Click on the row that you want to delete in the grid and then click on the **Delete** button.

Silverlight SharePoint Tasks Viewer

ProjectId	Title	StartDate	DueDate	EstimatedDaysL
1	Creating a Silverlight 4 UI	2/16/2010 3:00:00 AM	2/20/2010 3:00:00 AM	4
2	Creating a Complex Silverlight 4 LOB RIA	3/31/2010 3:00:00 AM	4/5/2010 3:00:00 AM	5
13	Testing Silverlight 4 Client OM with Lists in SharePoint 2010	4/14/2010 12:00:00 AM	4/30/2010 12:00:00 AM	16
20	Testing Messages Between Silverlight Web Parts	3/30/2010 12:00:00 AM	4/2/2010 12:00:00 AM	3
26	Testing the Delete Operation in a SharePoint List	3/28/2010 3:00:00 AM	4/20/2010 3:00:00 AM	23

Showing items... 1 2 3

12. The application will request the SharePoint server to delete the selected item from the list and then it will refresh the data shown in the grid. If the operation was successful, the row will not appear in the grid.

Silverlight SharePoint Tasks Viewer

ProjectId	Title	StartDate	DueDate	EstimatedDa
1	Creating a Silverlight 4 UI	2/16/2010 3:00:00 AM	2/20/2010 3:00:00 AM	4
2	Creating a Complex Silverlight 4 LOB RIA	3/31/2010 3:00:00 AM	4/5/2010 3:00:00 AM	5
13	Testing Silverlight 4 COM with Lists in SharePoint 2010	4/14/2010 12:00:00 AM	4/30/2010 12:00:00 AM	16
20	Testing Messages Between Silverlight Web Parts	3/30/2010 12:00:00 AM	4/2/2010 12:00:00 AM	3
26	Testing Two Silverlight Web Parts Interacting in the Same Page	3/29/2010 12:00:00 AM	4/21/2010 12:00:00 AM	23

Understanding how to delete an item from a list

When the user clicks the **Delete** button, the `Click` event handler, `butDelete_Click`, runs in the UI thread. It uses the current URL to generate a `ClientContext` instance, saved in the private `_context` variable. Then, it calls the `GetByTitle` method to save a reference to the `ProjectsList2010` list that was previously created in the SharePoint server, in the private `_projects` variable.

```

_context = new SP.ClientContext(SP.ApplicationContext.Current.Url);
_projects = _context.Web.Lists.GetByTitle("ProjectsList2010");

```

Then, it assigns the value for the data item corresponding to the selected row in the `dataGridProjects` `DataGrid` to the `selectedProject` local variable. It does so by accessing the `dataGridProjects.SelectedItem` property and casting it to `Project`, because it represents a `Project` instance.

```

var selectedProject = (dataGridProjects.SelectedItem as Project);

```

The next line calls the `GetItemById` method for the reference to the `ProjectsList2010` list, `_projects`. This method receives the value for the unique ID field, an `int` or `string`, and returns a reference to the `ListItem` instance that is going to be retrieved when the query is executed. In this case, the value for ID was set in `selectedProject.ProjectID` and it is an `int`.

```

SP.ListItem listItem =
    _projects.GetItemById(selectedProject.ProjectId);

```

Once the code has a reference to the desired item in the list, it calls the `DeleteObject` method to remove it. Remember that at this point, we don't have access to the `ListItem` instance properties, because the query hasn't yet been executed. However, we can schedule many queries in a single call to `ExecuteQueryAsync`.

```

listItem.DeleteObject();

```

The call to the `ExecuteQueryAsync` will perform the following actions in the SharePoint server:

Retrieve the element with the specified ID value from the `ProjectsList2010` list. If found, delete it.

If the query has a successful execution, the `OnDeleteSucceeded` callback schedules the asynchronous execution of the `Connect` method in the UI thread from a worker thread that runs this code after the query succeeds. It will refresh the data shown in the `dataGridProjects` `DataGrid` to reflect the new contents of the `ProjectsList2010` list.

```

private void OnDeleteSucceeded(Object sender, SP.ClientRequestSucceededEventArgs args)
{
    Dispatcher.BeginInvoke(Connect);
}

```

In this example, we also specified the `OnDeleteFailed` callback and used it as the second parameter for the `ExecuteQueryAsync` method. If something goes wrong, it invokes a delegate that calls the `ShowErrorInformation` method and sends the `ClientRequestFailedEventArgs args` instance as a parameter to it.

Expanding LOB systems with update operations

Now, follow these steps to add a new feature in the `SLTasksViewer` Silverlight application to allow the user to edit and update the value for the `Title` row in the item selected in the `DataGrid` from the list in the SharePoint server, `ProjectsList2010`.

1. Stay in Visual Studio as a system administrator user, in the `TasksViewer` solution.
2. Open `MainPage.xaml.cs` and add the following code to the `CellEditEnded` event for the `dataGridProjects` `DataGrid`. This way, when the user finishes editing the cell corresponding to the `Title` row, the application will retrieve the `Project` instance selected in the `dataGridProjects` `DataGrid` and use its ID to update the value for the `Title` field in the corresponding item in the list in the SharePoint server.

```
private void dataGridProjects_CellEditEnded(object sender,
DataGridCellEditEndedEventArgs e)
{
    if ((e.EditAction == DataGridEditAction.Commit) &&
        (e.Column.Header.Equals("Title")))
    {
        _context = new SP.ClientContext(
            SP.ApplicationContext.Current.Url);
        _projects = _context.Web.Lists.GetByTitle("ProjectsList2010");
        var selectedProject = (dataGridProjects.SelectedItem as
            Project);
        SP.ListItem listItem = _projects.GetItemById(selectedProject.
            ProjectId);
        // Assign the new value for the Title field
        listItem["Title"] = selectedProject.Title;
        // Update the item in the list
        listItem.Update();
        _context.ExecuteQueryAsync(
            OnUpdateSucceeded,
            OnUpdateFailed);
    }
}
```

3. The code in the `CellEditEnded` event for the `dataGridProjects` `DataGrid` is going to run in the UI thread. The following methods, which are going to be fired as asynchronous callbacks, schedule the execution of other methods to continue with the necessary program flow in the UI thread:
 - When the connection to the SharePoint server and the query execution that updates an item in the list, requested by the `dataGridProjects_CellEditEnded` method, is successful, the `OnUpdateSucceeded` method schedules the execution of a delegate defined in the UI thread with a lambda expression that updates the text shown in the `lblStatus` `Label`. The lambda expression appears highlighted in the next code snippet.
 - If the `dataGridProjects_CellEditEnded` method fails, the `OnUpdateFailed` method schedules the execution of the `ShowErrorInformation` method in the UI thread, sending the `ClientRequestFailedEventArgs` `args` instance as a parameter to the delegate.

```
private void OnUpdateSucceeded(Object sender, SP.ClientRequestSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(
        () =>
        {
            // This code will run on the UI thread
            lblStatus.Content =
                "The title field was updated successfully.";
        }
    );
}

private void OnUpdateFailed(Object sender,
    SP.ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Invoke a delegate and send the args instance as a parameter
    Dispatcher.BeginInvoke(
        () => ShowErrorInformation(args)
    );
}
```

4. Build and deploy the solution.
5. Load the SharePoint page that shows `SLTasksViewer.xap` as a Silverlight Web Part, `SilverlightProjectsList2010.aspx`.
6. Double-click on the cell that contains the title that you want to update and you will enter in the edit mode:

ProjectId	Title	StartDate
1	Creating a Silverlight 4 UI	2/16/2010
2	Creating a Complex Silverlight 4 LOB RIA	3/31/2010
13	Testing Silverlight 4 COM with Lists in SharePoint 2010	4/14/2010
20	Testing Messages Between Silverlight Web Parts	3/30/2010
26	Testing the New Feature that Allows a User to Update	3/29/2010

7. Press *Tab* and the application will request the SharePoint server to update the value for the `Title` field of the selected item from the list. If the operation was successful, the status label will display the following message, **The title field was updated successfully.**

ProjectId	Title	StartDate
1	Creating a Silverlight 4 UI	2/16/2010 3:00:00 AM
2	Creating a Complex Silverlight 4 LOB RIA	3/31/2010 3:00:00 AM
13	Testing Silverlight 4 COM with Lists in SharePoint 2010	4/14/2010 12:00:00 AM
20	Testing Messages Between Silverlight Web Parts	3/30/2010 12:00:00 AM
26	Testing the New Feature that Allows a User to Update a Title	3/29/2010 12:00:00 AM

The title field was updated successfully.

8. You can access the page for the list to check that the value for the `Title` field also appears updated in this view:

Type	Title	Assigned To	Status	Priority
	Creating a Silverlight 4 UI	gaston-PC\hillar2010	Not Started	(2) Normal
	Creating a Complex Silverlight 4 LOB RIA	gaston-PC\hillar2010	Not Started	(2) Normal
	Testing Silverlight 4 COM with Lists in SharePoint 2010	gaston-PC\hillar2010	Not Started	(2) Normal
	Testing Messages Between Silverlight Web Parts	gaston-PC\hillar2010	Not Started	(2) Normal
	<u>Testing the New Feature that Allows a User to Update a Title</u>	gaston-PC\hillar2010	Not Started	(2) Normal

Add new item

Updating an item in a list

When the user finishes editing a cell in the `dataGridProjects` `DataGrid`, the `CellEditEnded` event handler, `dataGridProjects_CellEditEnded`, runs in the UI thread. If the edited column was the one corresponding to the `Title` field and the user committed the edit action, it runs the necessary code to update the corresponding item in the list with the new value for the `Title` field. It uses the values provided in the `EditAction` and `Column.Header` properties from the `DataGridCellEditEndedEventArgs` `e` parameter to determine that the conditions are satisfied.

```
if ((e.EditAction == DataGridEditAction.Commit) &&
    (e.Column.Header.Equals("Title")))
```

It uses the current URL to generate a `ClientContext` instance, saved in the private `_context` variable. Then, it calls the `GetByTitle` method to save a reference to the `ProjectsList2010` list, in the private `_projects` variable.

```
_context = new SP.ClientContext(SP.ApplicationContext.Current.Url);
_projects = _context.Web.Lists.GetByTitle("ProjectsList2010");
```

As it needs to access the new value for the `Title` field, it assigns the value for the data item corresponding to the selected row in the `dataGridProjects` `DataGrid` to the `selectedProject` local variable. It does so by accessing the `dataGridProjects.SelectedItem` property and casting it to `Project`, because it represents a `Project` instance. At this point, the new value is available in the `selectedProject.Title` property.

```
var selectedProject = (dataGridProjects.SelectedItem as Project);
```

The next line calls the `GetItemById` method for the reference to the `ProjectsList2010` list, `_projects`, with `selectedProject.ProjectID` as the ID for `ListItem` instance to retrieve from the list.

```
SP.ListItem listItem =  
    _projects.GetItemById(selectedProject.ProjectId);
```

Once the code has a reference to the desired item in the list, it assigns the new value for the `Title` field to the `ListItem` instance:

```
listItem["Title"] = selectedProject.Title;
```

The next step is to call the `Update` method for the `ListItem` instance to update this row with the new value for the `Title` field. Remember that at this point, we don't have access to the `ListItem` instance properties because the query hasn't yet been executed. However, we can assign new values to its contents and then perform all the operations that require many queries in a single call to `ExecuteQueryAsync`.

```
listItem.Update();
```

The call to `ExecuteQueryAsync` will perform the following actions in the SharePoint server:

- Retrieve the element with the specified ID value from the `ProjectsList2010` list.
- If found, assign the new value to the `Title` field for the row and update it.

If the query has a successful execution, the `OnUpdateSucceeded` callback schedules the asynchronous execution of the delegate defined through a lambda expression in the UI thread from a worker thread that runs this code after the query has succeeded. In this case, the code is written inside the lambda expression that defines the delegate and displays a message in the `lblStatus` Label, indicating that the update operation was successful.

```
private void OnUpdateSucceeded(Object sender, SP.ClientRequestSucceededEventArgs args)  
{  
    // Code that runs in a worker thread
```

```
Dispatcher.BeginInvoke(() =>
{
    // The code inside this delegate
    // runs in the UI thread
    lblStatus.Content =
        "The title field was updated successfully.";
});
// Back in the worker thread
}
```

We also specified the `OnUpdateFailed` callback and used it as the second parameter for the `ExecuteQueryAsync` method. If something goes wrong, it invokes a delegate that calls the `ShowErrorInformation` method and sends the `ClientRequestFailedEventArgs` instance as a parameter to it.

Summary

In this chapter, we learned about developing and deploying Silverlight 4 applications in SharePoint 2010 sites that interact with data in lists by performing insert, update, and delete operations. Specifically, we created a new Silverlight 4 RIA that allowed us to insert new items into a remote SharePoint list. Then, we enhanced this simple application to retrieve metadata information for the fields that compose the list to offer the possible choices and default values for some of the fields.

We worked with messages to allow multiple Silverlight Web Parts to communicate when some events occur. Besides, we performed delete and update operations on the remote SharePoint list through the SharePoint Silverlight Client Object Model. Now, we are able to begin adding Silverlight RIAs that interact with data from the SharePoint server and deploy them by enhancing the examples learned in this chapter.

8

Interacting with Rich Media and Animations



This chapter is taken from *Microsoft Silverlight 4 and SharePoint 2010 Integration* (Chapter 6) by Gastón C. Hillar.

We want to take advantage of Silverlight 4 features to work with rich media and perform animations. In this chapter, we will cover many topics related to retrieving digital assets from SharePoint libraries through the SharePoint Silverlight Client Object Model and consuming them in a Silverlight RIA. We will:

- Create and manage asset libraries in SharePoint 2010
- Access the digital assets in a SharePoint library from a Visual Web Part and a Silverlight RIA
- Create a SharePoint Visual Web Part that sends parameters and renders a Silverlight RIA
- Link a SharePoint Visual Web Part to a Silverlight RIA
- Add a SharePoint Visual Web Part in a web page
- Work with multiple interactive animations and effects
- Display and control videos
- Add background music from the assets library
- Change themes in Silverlight and SharePoint

Bringing life to business applications and complex workflows

So far, we have been able to interact with data from the SharePoint Server through the SharePoint Silverlight Client Object Model and WCF Data Services. Sometimes, we need to share, manage, and consume rich media, related to data.

SharePoint 2010 improves rich media management by introducing asset libraries and we can take advantage of this new feature by consuming it in a Silverlight RIA through the SharePoint Silverlight Client OM. Silverlight 4 offers outstanding features to create amazing User eXperiences (UX) when combining rich media with effects and animations.

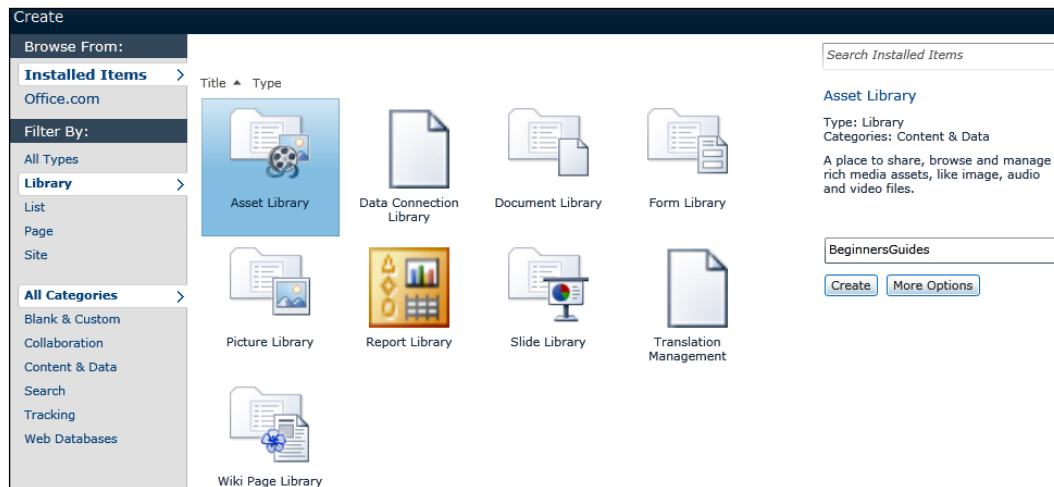
Creating asset libraries in SharePoint 2010

SharePoint Server 2010 introduced a new asset library specially designed for managing and sharing digital assets and rich media files such as images, audio, and video. It is possible to combine workflows, routing, rules, and policies with asset libraries. However, in this case, we will focus on creating simple asset libraries to allow us to store images, videos, and audio files and we will consume them through the SharePoint Silverlight Client Object Model.

We will combine a new SharePoint Visual Web Part with a Silverlight RIA to allow users to select their desired asset library and to browse its images and videos with interactive animations and dazzling effects. The SharePoint Visual Web Part will display a drop-down list with the available asset libraries that store images, videos, or audio files and when the user selects one of them, the Silverlight RIA will use the capabilities offered by the Client OM to retrieve and display the digital assets. This way, with this new composite Web Part, it is going to be possible to create a new asset library and to upload the necessary images and videos to display, and the desired background music as an audio file. The Web Part will allow a user to interact with any asset library.

First, follow these steps to create two asset libraries in a SharePoint site:

1. Open your default web browser, view the SharePoint site, and log in with your username and password.
2. Click **Site Actions | More Options...** in the ribbon and the **Create** dialog box will appear.
3. Select **Library** under **Filter By:** and then **Asset Library** in **Installed Items**, as shown in the following screenshot:

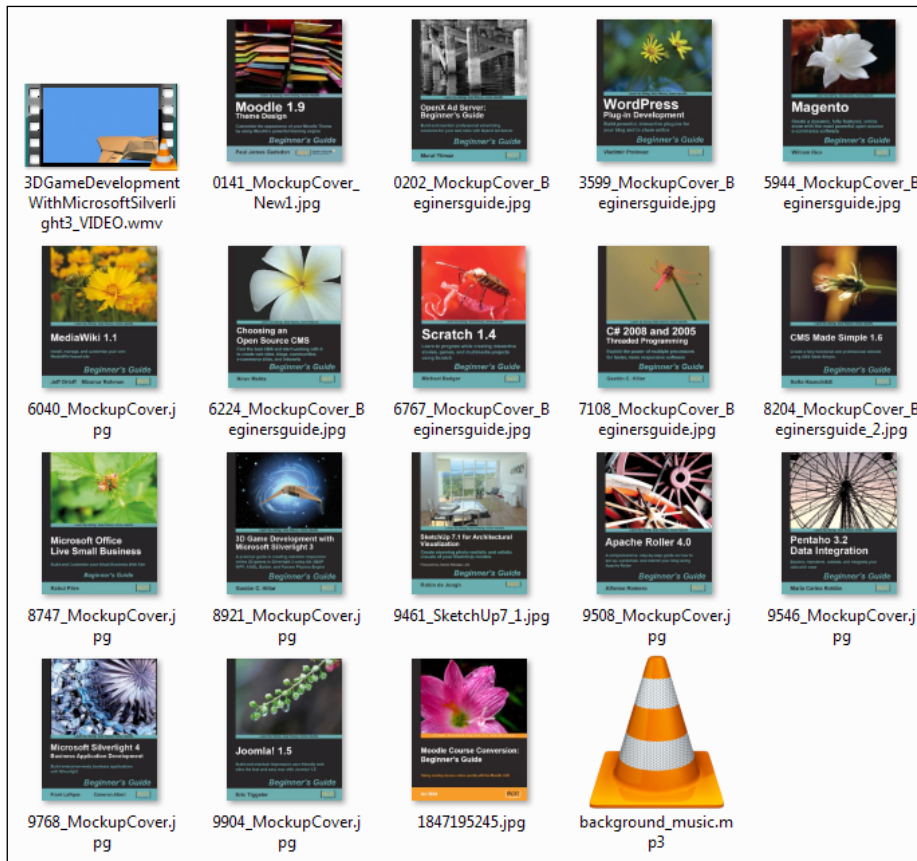


4. Enter `BeginnersGuides` in the **Name** textbox.
5. Click on **More Options**. SharePoint will display a new panel with additional options for the new asset library.
6. Enter `Beginner's Guides` in **Description** and select **Yes** in **Display this list on the Quick Launch?**
7. Click on **Create**; SharePoint will create the new asset library with no digital assets and it will appear in the Quick Launch for the SharePoint site.
8. Now, follow the aforementioned steps (1 to 7) to create another asset library. Use `Cookbooks` as the **Name** and **Description** for this new asset library.

Adding content to an assets library

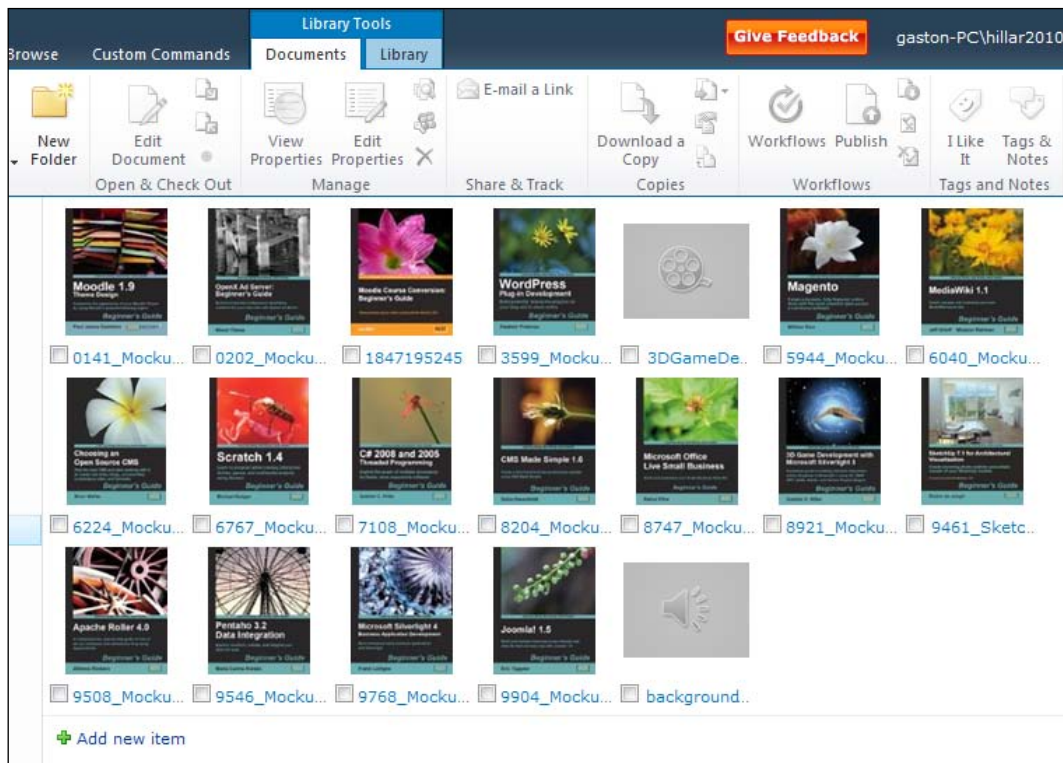
Follow these steps to prepare and add images, videos, and audio files to the previously created asset libraries.

1. Prepare two folders, `BeginnersGuides` and `Cookbooks`. Add many JPG and/or PNG images to these folders. Add a **WMV (Windows Media Video)** video file and an **MP3** audio file to both folders. The following screenshot shows an example of the contents of the `BeginnersGuides` folder with 17 JPG images, a WMV video file, and an MP3 audio file:

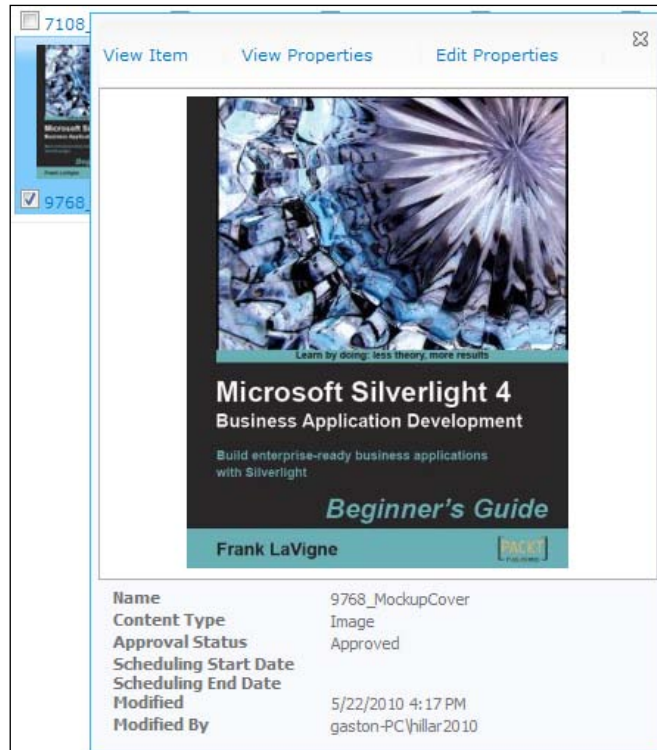


By default, SharePoint 2010 establishes 50 MB as the maximum upload file size setting. This setting specifies the maximum size of a file that a user can upload to the server. If a user tries to upload a file larger than the specified maximum upload size, the upload will fail.

2. Click on the hyperlink for the `BeginnersGuides` asset library in the Quick Launch for the SharePoint site.
3. Click on **Add new item**. The **Upload document** dialog box will appear. Click on **Upload Multiple files...** and a panel to which to drag files and folders will appear.
4. Open an Explorer window and navigate to your `BeginnersGuides` folder. Select all the files within the folder and drag-and-drop them in the **Drag files and folder here** panel within the **Upload Document** dialog box. All the file names will appear in the panel.
5. Click on **OK** and SharePoint will upload all the dropped files to the previously created asset library. Click on **Done** and the new digital assets will appear in the asset library. By default, SharePoint will display a thumbnail preview for the image files:



- Click on one of the thumbnails for the images and a bigger thumbnail will appear with detailed properties for the digital asset:



- Click on **Edit Properties**, located at the top of the bigger thumbnail preview; a new dialog box will appear and you will be able to edit many properties related to the digital asset. The **Content Type** drop-down list will display **Image**, because SharePoint automatically recognized the digital asset as an image. As we uploaded many images dragging and dropping them to the panel, SharePoint assigned the name but it didn't set values for **Title**, **Keywords**, **Comments**, **Author**, and **Copyright**. You can use this dialog box to set the values for these properties in order to organize the contents of the asset library. Then, click on **Save**.

The screenshot shows a web form for uploading an image asset. The form is titled 'Beginners Guides - 9546_MockupCover.jpg'. It features a toolbar with 'Save', 'Cancel', 'Paste', 'Copy', and 'Delete Item' buttons. Below the toolbar, the 'Content Type' is set to 'Image'. The 'Name' field contains '9546_MockupCover.jpg'. The 'Title' field contains 'Pentaho 3.2 Data Integration'. The 'Keywords' field contains 'beginner, guide, pentaho, integrate, validate, data, transform, explore, book'. The 'Comments' field contains 'Explore, transform, validate, and integrate your data with ease'. The 'Author' field contains 'María Carina Roldán'. The 'Date Picture Taken' field contains '12/8/2009 3 PM 26'. The 'Copyright' field contains 'Packt Publishing Ltd.'. At the bottom, there are 'Save' and 'Cancel' buttons. The form also displays creation and modification timestamps: 'Created at 5/24/2010 12:51 AM by gaston-PC\hilar2010' and 'Last modified at 5/24/2010 12:51 AM by gaston-PC\hilar2010'.

- Now, follow the aforementioned steps (1 to 7) to add an audio file, images, and videos to the other asset library, `Cookbooks`. Remember to upload the files stored in the `Cookbooks` folder.

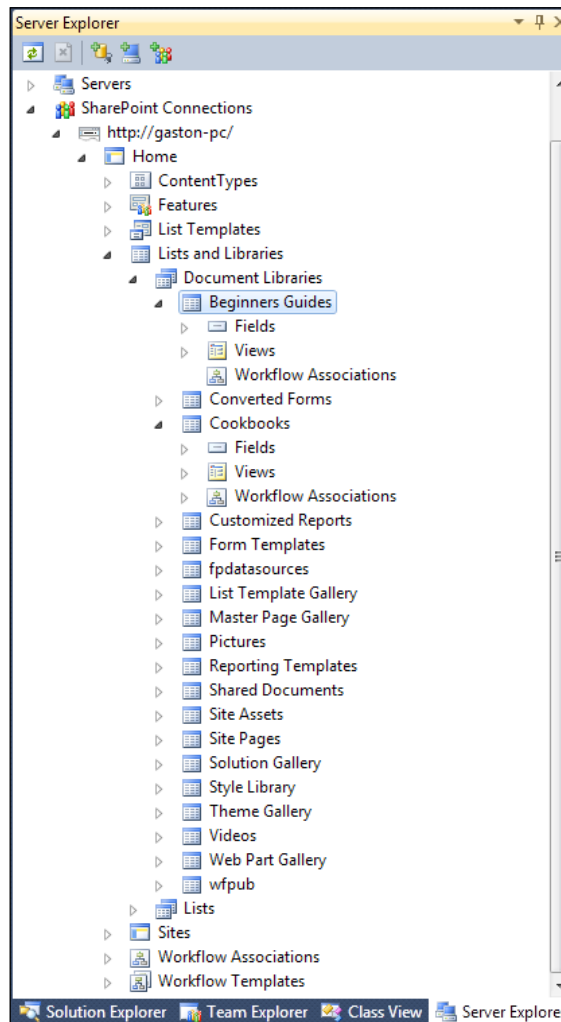
We added images, videos and audio files to the two asset libraries, `BeginnerGuides` and `Cookbooks`, in the SharePoint site. Now, we can browse the asset libraries' structure and then create an interactive Silverlight RIA capable of consuming the uploaded digital assets.

Browsing the structure for SharePoint Asset Libraries

Once we have created the two asset libraries in SharePoint, we can use Server Explorer in Visual Studio to analyze the new asset libraries' structures.

- Start Visual Studio as a system administrator user.

2. Activate the **Server Explorer** palette by clicking on **View | Server Explorer**.
3. Click on the expand button for **SharePoint Connections** and then on the expand button for the SharePoint server. You will be able to browse its different nodes.
4. Expand **Lists and Libraries** and then **Document Libraries** for the Site Collection in which you created the new asset libraries. Remember that the default Site Collection is **Home**. There are asset libraries and document libraries within **Document Libraries** and therefore, it is going to be necessary to use a smart filter to display the right asset library names in the drop-down list that the user will use to select the desired asset library with pictures, videos, and audio files.



Controlling the rich media library by using controls in a Visual Web Part

This time, we are going to create a new solution in Visual Studio that will include two new projects:

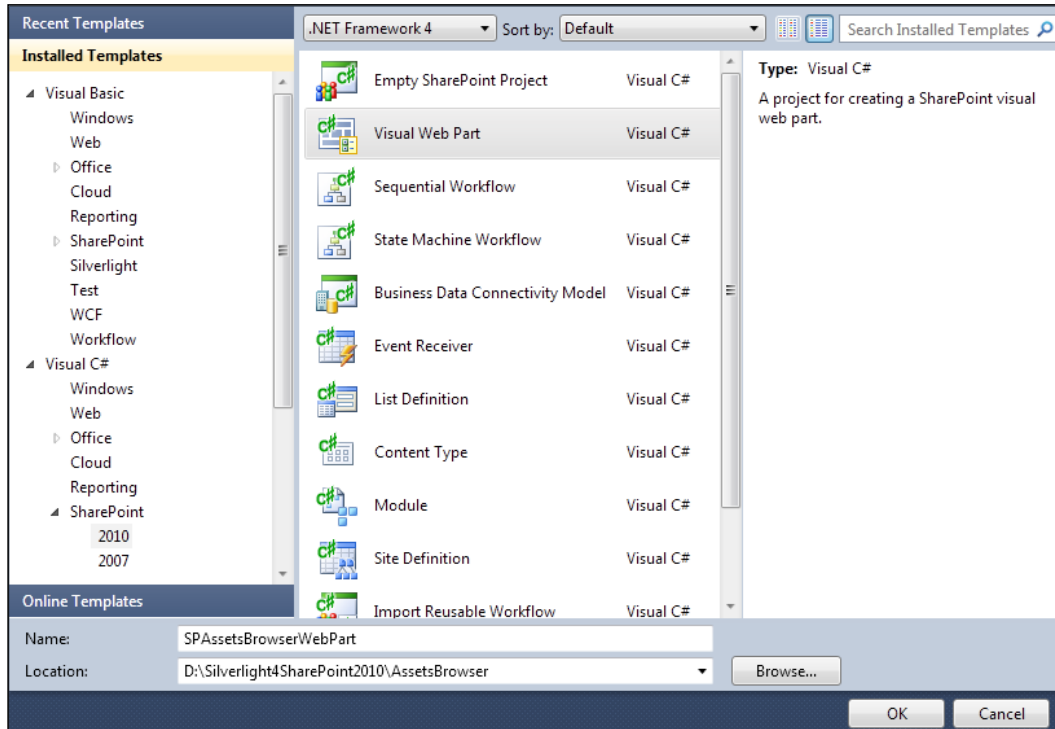
- A SharePoint 2010 Visual Web Part, `SPAssetsBrowserWebPart`
- A Silverlight application project, `SLAssetsBrowser`

SharePoint is built on top of **ASP.NET**, and therefore, a **Visual Web Part** inherits key features from the ASP.NET Web Part architecture. The Visual Web Part will display the available asset libraries with videos, pictures, and/or audio files in a SharePoint site and it will send the selected asset library as a parameter to the Silverlight host control that will render the Silverlight application. We will take advantage of one of the new project templates in Visual Studio 2010, the Visual Web Part project template, which enables us to visually design a Web Part that can be deployed to SharePoint. The necessary steps to display a Silverlight application within the Visual Web Part are a bit complex but the flexibility offered by this combination is worth the effort.

Follow these steps to create the new Visual Web Part that accesses the available asset libraries in a SharePoint site:

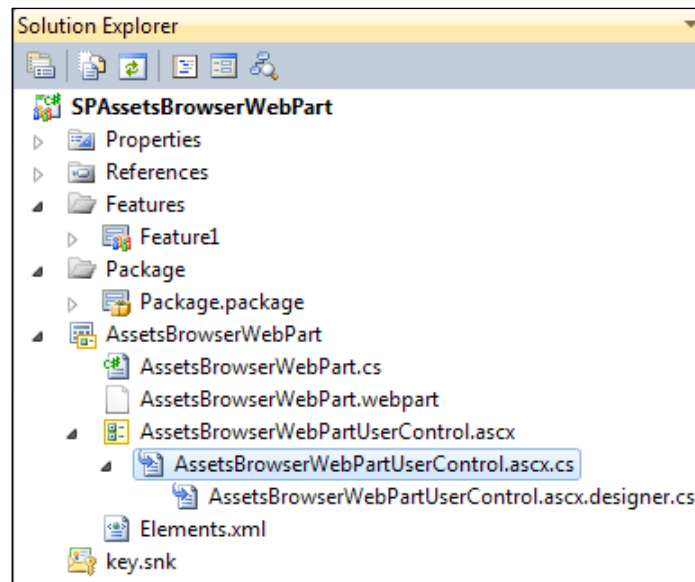
1. Stay in Visual Studio as a system administrator user.
2. Select **File | New | Project...** Expand **Other Project Types** and select **Visual Studio Solutions** under **Installed Templates** in the **New Project** dialog box. Then, select **Blank Solution**, make sure that **.NET Framework 4** version is selected, and enter `AssetsBrowser` as the project's name and click **OK**. Visual Studio will create a blank solution with no projects.

- Right-click on the solution's name in **Solution Explorer** and select **Add | New Project...** from the context menu. Expand **Visual C#** and then expand **SharePoint** and select **2010** under **Installed Templates** in the **Add New Project** dialog box. Then, select **Visual Web Part**, enter `SPAssetsBrowserWebPart` as the project's name, and click **OK**. The **SharePoint Customization Wizard** dialog box will appear.



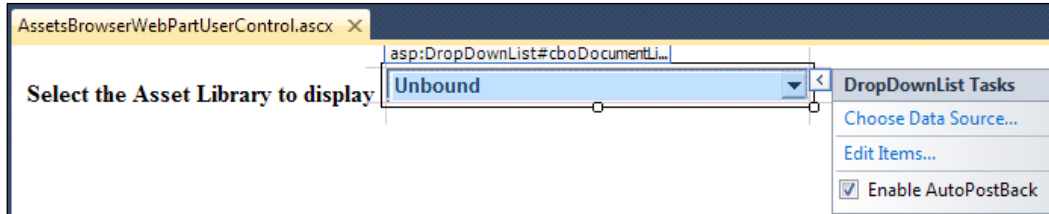
- Enter the URL for the SharePoint server and site in **What local site do you want to use for debugging?**
- Click on **Deploy as a farm solution**. Sandboxed solutions don't support the Visual Web Parts and therefore, it is necessary to deploy projects that include them as a **farm solution**. Then, click on **Finish** and the new `SPAssetsBrowserWebPart` empty SharePoint 2010 Visual Web Part project will be added to the solution. The code editor will open the source code for the `VisualWebPart1UserControl.ascx` `UserControl` (`System.Web.UI.UserControl`). This `UserControl` defines the UI for the Visual Web Part and it has a code-behind file, `VisualWebPart1UserControl.ascx.cs`.
- Right-click on the `VisualWebPart1` folder in **Solution Explorer** and select **Delete** in the context menu. Click **OK** in the confirmation dialog box.

7. Now, right-click on the recently added project's name in **Solution Explorer**, `SPAssetsBrowserWebPart`, and select **Add | New Item...** in the context menu. Expand **Visual C#** and then expand **SharePoint** and then select **2010** under **Installed Templates** in the **New Item** dialog box. Then, select **Visual Web Part**, enter `AssetsBrowserWebPart` in **Name**, and click **OK**. The code editor will open the source code for the `AssetsBrowserWebPart.ascx` `UserControl` (`System.Web.UI.UserControl`). Its new code-behind file is `AssetsBrowserWebPartUserControl.ascx.cs`. Renaming a Visual Web Part can be a very complex process and therefore, it is easier to delete the default `VisualWebPart1` folder and add a new Visual Web Part item with the desired name. This way, Visual Studio will create the container folder and all its related files with the new name.



8. Switch to the **Design** view for the `AssetsBrowserWebPart.ascx` `UserControl` and use the **Toolbox** to drag-and-drop the following server controls. The names for the server controls are assigned in the **ID** property in the **Properties** window.
 - One `Label` control, `Label1`. Set its `Text` property to `Select the Asset Library to display`.

- One DropDownList control, cboDocumentLibraries. Set its AutoPostBack property to true. This way, the page will automatically **post back** to the server after the user changes the selection for this drop-down list.



9. Now, open the code-behind file for the AssetsBrowserWebPart.ascx UserControl, AssetsBrowserWebPartUserControl.ascx.cs and add the following using statements.

```
using Microsoft.SharePoint;  
using Microsoft.SharePoint.WebControls;
```
10. Add the following public property for the AssetsBrowserWebPartUserControl partial class.

```
public string SelectedList { get; private set; }
```
11. Add the following lines to the Page_Load event. This code will run at the server when a user requests the Visual Web Part for the first time and each time a postback occurs. Thus, it is necessary to run different code when it is a postback by checking the Boolean value of the IsPostBack property. When the code runs for the first time (IsPostBack == false), it will add the titles for the lists of SPBaseType.DocumentLibrary type with at least one item (libraryList.RootFolder.ItemCount > 0) and with content types of Picture, Image, Audio, or Video.

```
if (!IsPostBack)  
{  
    var _context = SPContext.Current;  
    var documentLibraries =  
        _context.Web.GetListsOfType(SPBaseType.DocumentLibrary);  
    foreach (SPList libraryList in documentLibraries)  
    {  
        if ((libraryList.RootFolder.ItemCount > 0) &&  
            ((libraryList.ContentTypes[0].Name == "Picture") ||  
             (libraryList.ContentTypes[0].Name == "Image") ||  
             (libraryList.ContentTypes[0].Name == "Audio") ||
```

```

        (libraryList.ContentTypes[0].Name == "Video")))
    {
        // The list has at least 1 element
        cboDocumentLibraries.Items.Add(
            new ListItem(libraryList.Title));
    }
}
// Select the first item in the dropdown list
cboDocumentLibraries.SelectedIndex = 0;
}
SelectedList = cboDocumentLibraries.SelectedValue;

```

12. Go back to the **Design** view for `AssetsBrowserWebPartUserControl.ascx` and define a `SelectedIndexChanged` event handler for the `cboDocumentLibraries DropDownList` and add the following code in it. This way, when the user selects a different item in the drop-down list, the `SelectedList` property will hold the name for the new list that has been selected.

```
SelectedList = cboDocumentLibraries.SelectedValue;
```

13. Now, open the `AssetsBrowserWebPart.cs` code file within the `AssetsBrowserWebPart` folder. This file defines the `AssetsBrowserWebPart` class as a subclass of `WebPart`. Its original code defines a path for the `UserControl`, `AssetsBrowserWebPartUserControl.ascx`, that this `WebPart` subclass will load and add to the `Controls ControlCollection`. This way, the `WebPart` renders the `UserControl`. The following lines show the original code for this file.

```

using System;
using System.ComponentModel;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
namespace SPAssetsBrowserWebPart.AssetsBrowserWebPart
{
    [ToolboxItemAttribute(false)]
    public class AssetsBrowserWebPart : WebPart
    {
        // Visual Studio might automatically update this path when you
        // change the Visual Web Part project item.

```

```
private const string _ascxPath = @"~/_CONTROLTEMPLATES/  
SPAssetsBrowserWebPart/AssetsBrowserWebPart/  
AssetsBrowserWebPartUserControl.ascx";  
protected override void CreateChildControls()  
{  
    Control control = Page.LoadControl(_ascxPath);  
    Controls.Add(control);  
}  
}
```

14. Add the following private variable to the `AssetsBrowserWebPart` class. This variable will hold a reference to the `Control` instance cast as `AssetsBrowserWebPartUserControl`. This way, it will be possible to access the value for the `SelectedList` public property to send it as a parameter to the Silverlight host control in the `OnPreRender` method.

```
private AssetsBrowserWebPartUserControl _control;
```

15. Add the following lines in the `CreateChildControls` method to save the reference to the `AssetsBrowserWebPartUserControl` instance.

```
protected override void CreateChildControls()  
{  
    Control control = Page.LoadControl(_ascxPath);  
    Controls.Add(control);  
    _control = (control as AssetsBrowserWebPartUserControl);  
    base.CreateChildControls();  
}
```

16. Override the `OnPreRender` event to add the Silverlight host control that will load and display the Silverlight RIA and it will send the selected asset library title as a parameter. The highlighted lines define the `.xap` file location and the parameter called `Name`.

```
protected override void OnPreRender(EventArgs e)  
{  
    var name = _control.SelectedList;  
    string webUrl = SPContext.Current.Web.Url;  
    string renderHost = @"<div id='silverlightControlHost'>  
    <object data='data:application/x-silverlight-2,'  
type='application/x-silverlight-2' width='100%' height='100%'>  
    <param name='source' value='/_catalogs/wp/SLAssetsBrowser.xap' />  
    <param name='background' value='white' />  
    <param name='minRuntimeVersion' value='4.0.50303.0' />  
    </object>  
    </div>";  
    Page.RenderControl(new Control { ControlType = ControlType.HtmlElement, InnerText = renderHost });  
}
```

```

    <param name='autoUpgrade' value='true' />
    <param name='initParams' value='Name=" + name.Trim() + @"' />
    <a href='http://go.microsoft.com/fwlink/?LinkId=149156
&v=4.0.50303.0' style='text-decoration:none'>
    <img src='http://go.microsoft.com/fwlink/?LinkId=161376'
alt='Get Microsoft Silverlight' style='border-style:none' />
    </a>
    </object><iframe id='_sl_historyFrame' style='visibility:hidden;
height:0px;width:0px;border:0px'></iframe></div>";
    LiteralControl host = new LiteralControl(renderHost);
    Controls.Add(host);
    base.OnPreRender(e);
}

```

The values for the `renderHost` string define a Silverlight control host. You can check the test page generated by Visual Studio for the Silverlight application to find the most up to date definition.

Once you have built your application, click on the **Show All Files** button in **Solution Explorer**. Then, expand the `Bin\Debug` folder for your Silverlight project. You will find many folders and files; open the HTML file that ends with `TestPage.html`, in our example, `SLAssetsBrowserTestPage.html`. You can copy from `<div id="silverlightControlHost">` to `</div>` and you can assign this value to `renderHost` to create a Silverlight host control. However, you have to change the following line that defines the path for the `.xap` file:

```
<param name="source" value="SLAssetsBrowser.xap"/>
```

It has to be replaced with the path for the `.xap` file inside the SharePoint `_catalogs/wp` folder.

```
<param name='source' value='/_catalogs/wp/SLAssetsBrowser.xap' />
```

In this case, then, it was necessary to add a parameter after the last param name, because we want to send a specific value to the Silverlight RIA.

Creating a Silverlight RIA rendered in a SharePoint Visual Web Part

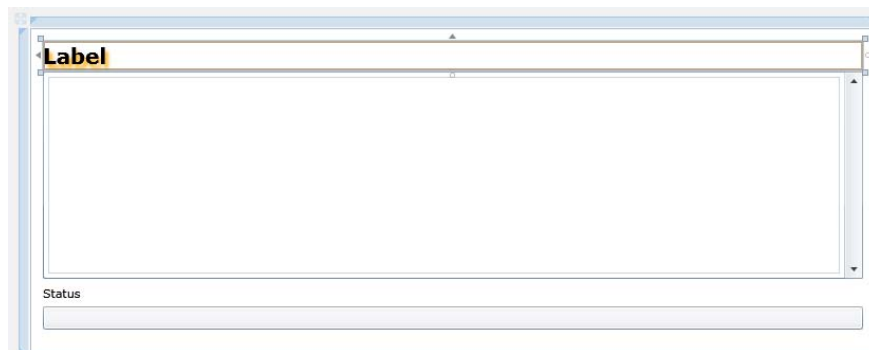
Follow these steps to create the new Silverlight RIA that loads the images, videos, and audio from the asset library selected in the Visual Web Part that renders this application and sends the selected name as a parameter:

1. Stay in Visual Studio as a system administrator user.
2. Select **File | New | Project...** Expand **Visual C#** and select **Silverlight** under **Installed Templates** in the **New Project** dialog box. Then, select **Silverlight Application**, enter `SLAssetsBrowser` as the project's name, choose **Add to Solution** in the **Solution** drop-down list, and click **OK**.
3. Deactivate the **Host the Silverlight application in a new Web site** checkbox in the **New Silverlight Application** dialog box and select **Silverlight 4** in **Silverlight Version**. Then, click **OK**. Visual Studio will add the new Silverlight application project to the existing solution.
4. Follow the necessary steps to add the following two references to access the SharePoint 2010 Silverlight Client OM:
 - `Microsoft.SharePoint.Client.Silverlight.dll`
 - `Microsoft.SharePoint.Client.Silverlight.Runtime.dll`
5. Open `App.xaml.cs` and add the following using statement:
`using Microsoft.SharePoint.Client;`
6. Replace the code in the `Startup` event handler with the following lines. The code stores the value for the `Name` parameter, specified by the Visual Web Part in the string that creates the Silverlight host control, in the `parameterName` local variable. Then, it creates a new instance of `MainPage` sending this value as a parameter to the constructor.

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    string parameterName = e.InitParams["Name"];

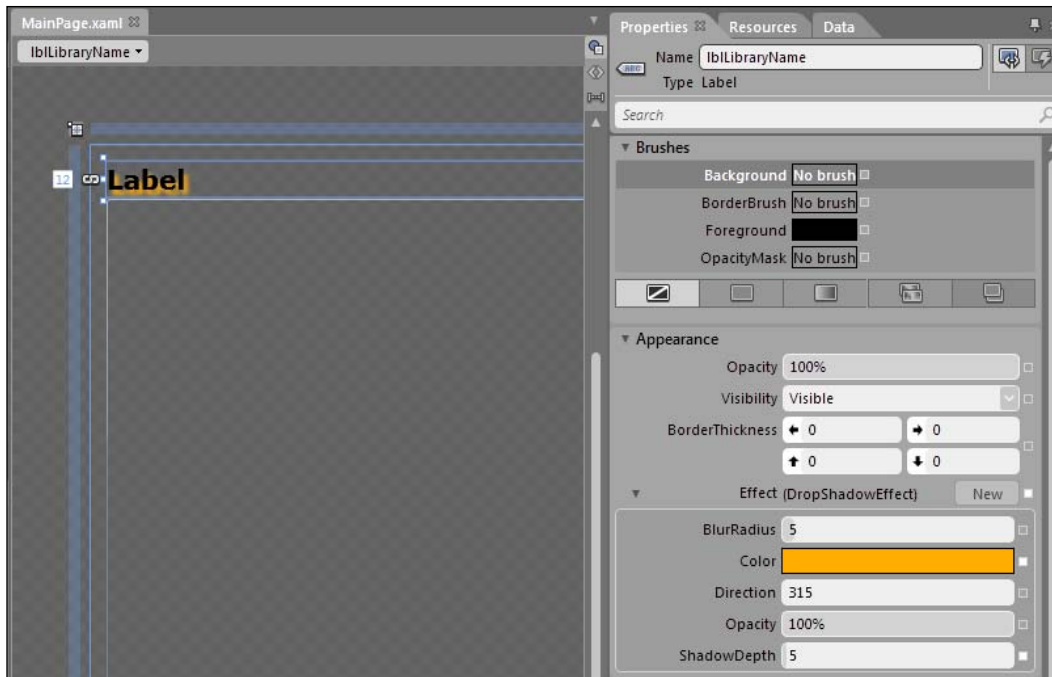
    this.RootVisual = new MainPage(parameterName);
    ApplicationContext.Init(e.InitParams,
        System.Threading.SynchronizationContext.Current);
}
```

7. Select **Start | All Programs | Microsoft Silverlight 4 Toolkit April 2010 | Binaries** and Windows will open the folder that contains the Silverlight Toolkit binaries. By default, they are located at `C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Toolkit\Apr10\Bin` in 64-bit Windows versions, and at `C:\Program Files\Microsoft SDKs\Silverlight\v4.0\Toolkit\Apr10\Bin` in 32-bit Windows versions.
8. Add a reference to `System.Windows.Controls.Toolkit.dll`. Remember that it is located in the aforementioned `Bin` sub-folder. This way, we will have access to the `WrapPanel` control.
9. Open `MainPage.xaml` and activate the **Toolbox**. Right-click on the **All Silverlight Controls** header and select **Choose Items...** in the context menu. The **Choose Toolbox Items** dialog box will appear with the **Silverlight Components** tab activated. Make sure that the checkbox located at the left of the **WrapPanel** item in the **Name** column is checked. This way, the **Toolbox** will display the `WrapPanel` control and you will be able to add it by dragging and dropping it to the desired location within the design view.
10. Define a new width and height for the `Grid`, `800` and `600`, and add the following controls. The following lines show the XAML that defines all the controls and some effects for the `lblLibraryName` Label and the `wrapPanel` `WrapPanel`.
 - One `Label` control, `lblLibraryName`, located at the top
 - One `ScrollViewer` control, `scrollViewer`
 - One `WrapPanel` control, `wrapPanel`, within the `ScrollViewer` control
 - One `Label` control, `lblStatus`, located at the bottom
 - One `ProgressBar` control, `pgbLoadingStatus`, located at the bottom




```
<UserControl x:Class="SLAssetsBrowser.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:toolkit="http://schemas.microsoft.com/winfx/2006/xaml/
  presentation/toolkit"
  xmlns:mc="http://schemas.openxmlformats.org/markup-
  compatibility/2006"
  mc:Ignorable="d"
  d:DesignHeight="600" d:DesignWidth="800" xmlns:sdk="http://
  schemas.microsoft.com/winfx/2006/xaml/presentation/sdk">
  <Grid x:Name="LayoutRoot" Loaded="LayoutRoot_Loaded"
  Width="Auto" Height="Auto">
    <sdk:Label Height="28" HorizontalAlignment="Left"
  Margin="12,12,0,0" Name="lblLibraryName" VerticalAlignment="Top"
  Width="776" FontSize="20" FontWeight="Bold" >
      <sdk:Label.Effect>
        <DropShadowEffect ShadowDepth="5" Color="Orange" />
      </sdk:Label.Effect>
    </sdk:Label>
    <ProgressBar Height="22" HorizontalAlignment="Left"
  Margin="12,554,0,0" Name="pgbLoadingStatus"
  VerticalAlignment="Top" Width="776" />
    <sdk:Label Height="22" HorizontalAlignment="Left"
  Margin="12,534,0,0" Name="lblStatus" VerticalAlignment="Top"
  Width="776" Content="Status" />
    <ScrollViewer Height="487" HorizontalAlignment="Left"
  Margin="12,41,0,0" Name="scrollViewer" VerticalAlignment="Top"
  Width="776">
      <toolkit:WrapPanel Name="wrapPanel" Width="Auto"
  Height="Auto" RenderTransformOrigin="0.497,0.493">
        <toolkit:WrapPanel.Effect>
          <DropShadowEffect ShadowDepth="10"/>
        </toolkit:WrapPanel.Effect>
      </toolkit:WrapPanel>
    </ScrollViewer>
  </Grid>
</UserControl>
```

11. You can also define the effects in Expression Blend without having to edit the XAML code. You can do so by right-clicking on `MainPage.xaml` and selecting **Open in Expression Blend...** in the context menu. This way, you will be able to work with the additional effects offered by this tool.




 There are many open source projects that provide additional effects that you can use in your RIAs, such as Silverlight.FX, <http://projects.nikhilk.net/SilverlightFX>.

- Open `MainPage.xaml.cs`. Now, it is necessary to add a `using` statement to include the `Microsoft.SharePoint.Client` namespace, as we want to work with the SharePoint Silverlight Client OM. We also have to work with the `BitmapImage` class, included in `System.Windows.Media.Imaging`.

Add the following lines of code:

```
using Microsoft.SharePoint.Client;
using SP = Microsoft.SharePoint.Client;
using System.Windows.Media.Imaging;
```

Add the following seven private variables:

```
private ClientContext _context;
private SP.List _documents;
private string _assetLibraryName;
private int _maxImageWidth = 150;
```

```
private int _imageMargin = 5;
// The background music can be added just once
private bool _backgroundMusicAdded = false;
// The current document to load
private int _documentToLoad;
```

13. Replace the `MainPage` constructor with this new constructor that receives the asset library name as a parameter, assigns its value to the `_assetLibraryName` private variable, and displays it in the `lblLibraryName` Label.

```
public MainPage(string assetLibraryName)
{
    InitializeComponent();
    _assetLibraryName = assetLibraryName;
    lblLibraryName.Content = assetLibraryName;
}
```

We are going to work with three media file types, `Audio`, `Video`, and `Picture`. Add the following code to define an enumeration and a method that returns the media file type according to the received file name's extension:

```
private enum MediaFileType
{
    Audio,
    Video,
    Picture
}

private MediaFileType GetMediaFileType(string fileName)
{
    switch (System.IO.Path.GetExtension(fileName).ToUpper())
    {
        // It isn't necessary to add break;
        // after each line because the code
        // exits with the return statement
        case ".JPG":
            return MediaFileType.Picture;
        case ".JPEG":
            return MediaFileType.Picture;
        case ".GIF":
            return MediaFileType.Picture;
    }
}
```

```

        case ".WMA":
            return MediaFileType.Audio;
        case ".MP3":
            return MediaFileType.Audio;
        case ".AAC":
            return MediaFileType.Audio;
        case ".WMV":
            return MediaFileType.Video;
        case ".MP4":
            return MediaFileType.Video;
        default:
            return MediaFileType.Picture;
    }
}

```

14. Add the following event handlers that will define and start animations when the user right-clicks on a button that displays an image or a video:

```

private void imageButton_MouseRightButtonDown(object sender,
MouseEventArgs e)
{
    // This ensures that Silverlight won't show up
    // the default Silverlight context menu
    e.Handled = true;
    var hlButton = (sender as HyperlinkButton);
    var image = hlButton.Content as Image;
    // Add a doubleAnimation for a MaxWidth animation
    var doubleAnimMaxWidth = new DoubleAnimation();
    doubleAnimMaxWidth.Duration =
        new Duration(TimeSpan.FromSeconds(6));
    doubleAnimMaxWidth.From = image.ActualWidth;
    doubleAnimMaxWidth.To = scrollViewer.ActualWidth -
        (_imageMargin * 2);
    doubleAnimMaxWidth.FillBehavior = FillBehavior.HoldEnd;
    // Create a new Storyboard to handle the MaxWidth animation
    var storyboardMaxWidth = new Storyboard();
    storyboardMaxWidth.Children.Add(doubleAnimMaxWidth);
    Storyboard.SetTarget(doubleAnimMaxWidth, image);
    Storyboard.SetTargetProperty(doubleAnimMaxWidth,
        new PropertyPath("MaxWidth"));
    storyboardMaxWidth.AutoReverse = true;
    storyboardMaxWidth.RepeatBehavior = new RepeatBehavior(1);
    // Add a doubleAnimation for a MaxHeight animation
}

```

```
var doubleAnimMaxHeight = new DoubleAnimation();
doubleAnimMaxHeight.Duration = new
    Duration(TimeSpan.FromSeconds(6));
doubleAnimMaxHeight.From = image.ActualHeight;
    doubleAnimMaxHeight.To = scrollViewer.ActualHeight -
        (_imageMargin * 2);
doubleAnimMaxHeight.FillBehavior = FillBehavior.HoldEnd;
// Create a new Storyboard to handle the MaxHeight animation
var storyboardMaxHeight = new Storyboard();
storyboardMaxHeight.Children.Add(doubleAnimMaxHeight);
Storyboard.SetTarget(doubleAnimMaxHeight, image);
Storyboard.SetTargetProperty(doubleAnimMaxHeight,
    new PropertyPath("MaxHeight"));
storyboardMaxHeight.AutoReverse = true;
storyboardMaxHeight.RepeatBehavior = new RepeatBehavior(1);
// Start the previously defined storyboards
storyboardMaxWidth.Begin();
storyboardMaxHeight.Begin();
}

private void videoButton_MouseRightButtonDown(object sender,
    MouseButtonEventArgs e)
{
    // This ensures that Silverlight won't show up
    // the default Silverlight context menu
    e.Handled = true;
    var hlb = (sender as HyperlinkButton);
    var element = hlb.Content as MediaElement;
    // Add a doubleAnimation for a MaxWidth animation
    var doubleAnimMaxWidth = new DoubleAnimation();
    doubleAnimMaxWidth.Duration = new
        Duration(TimeSpan.FromSeconds(9));
    doubleAnimMaxWidth.From = element.ActualWidth;
    doubleAnimMaxWidth.To = scrollViewer.ActualWidth -
        (_imageMargin * 2);
    doubleAnimMaxWidth.FillBehavior = FillBehavior.HoldEnd;
    // Create a new Storyboard to handle the MaxWidth animation
    var storyboardMaxWidth = new Storyboard();
    storyboardMaxWidth.Children.Add(doubleAnimMaxWidth);
    Storyboard.SetTarget(doubleAnimMaxWidth, element);
```

```

Storyboard.SetTargetProperty(doubleAnimMaxWidth,
    new PropertyPath("MaxWidth"));
storybookMaxWidth.AutoReverse = true;
storybookMaxWidth.RepeatBehavior = new RepeatBehavior(1);
// Add a doubleAnimation for a MaxHeight animation
var doubleAnimMaxHeight = new DoubleAnimation();
doubleAnimMaxHeight.Duration = new
    Duration(TimeSpan.FromSeconds(9));
doubleAnimMaxHeight.From = element.ActualHeight;
doubleAnimMaxHeight.To = scrollView.ActualHeight -
    (_imageMargin * 2);
doubleAnimMaxHeight.FillBehavior = FillBehavior.HoldEnd;
// Create a new Storyboard to handle the MaxHeight animation
var storyboardMaxHeight = new Storyboard();
storybookMaxHeight.Children.Add(doubleAnimMaxHeight);
Storyboard.SetTarget(doubleAnimMaxHeight, element);
Storyboard.SetTargetProperty(doubleAnimMaxHeight,
    new PropertyPath("MaxHeight"));
storybookMaxHeight.AutoReverse = true;
storybookMaxHeight.RepeatBehavior = new RepeatBehavior(1);
// Start the previously defined storyboards
storybookMaxWidth.Begin();
storybookMaxHeight.Begin();
}

```

Add the following event handler that will restart the reproduction of a video after it ends:

```

private void media_MediaEnded(object sender, RoutedEventArgs e)
{
    var media = (sender as MediaElement);
    // It is necessary to stop it or to set its Position to
    TimeSpan.Zero
    media.Stop();
    // Play again
    media.Play();
}

```

15. Add the following two methods that add and return a `HyperlinkButton` to the `wrapPanel WrapPanel` with an image and a video:

```

private HyperlinkButton AddImage(string url)
{
    var image = new Image();

```

```
        image.MaxWidth = _maxImageWidth;
        image.Stretch = Stretch.Uniform;
        var bitmapImage = new BitmapImage(new Uri(url,
            UriKind.Absolute));
        image.Source = bitmapImage;
        var imageButton = new HyperlinkButton();
        imageButton.Visibility = System.Windows.Visibility.Collapsed;
        imageButton.Margin = new Thickness(_imageMargin);
        imageButton.Content = image;
        imageButton.NavigateUri = new Uri(url);
        imageButton.MouseRightButtonDown += new
            MouseButtonEventHandler(imageButton_MouseRightButtonDown);
        imageButton.TargetName = "_blank";
        imageButton.Cursor = Cursors.Hand;
        // Add the new Hyperlink button with the image
        // to the WrapPanel wrapPanel
        wrapPanel.Children.Add(imageButton);
        return imageButton;
    }

private HyperlinkButton AddVideo(string url)
{
    MediaElement media = new MediaElement();
    media.MaxWidth = (_maxImageWidth * 3);
    media.Stretch = Stretch.UniformToFill;
    media.Source = new Uri(url, UriKind.Absolute);
    media.AutoPlay = true;
    media.MediaEnded += new RoutedEventHandler(media_MediaEnded);
    var videoButton = new HyperlinkButton();
    videoButton.Visibility = System.Windows.Visibility.Collapsed;
    videoButton.Margin = new Thickness(_imageMargin);
    videoButton.Content = media;
    videoButton.NavigateUri = new Uri(url);
    videoButton.MouseRightButtonDown += new
        MouseButtonEventHandler(videoButton_MouseRightButtonDown);
    videoButton.TargetName = "_blank";
    videoButton.Cursor = Cursors.Hand;
    // Add the new Hyperlink button with the video
    // to the WrapPanel wrapPanel
    wrapPanel.Children.Add(videoButton);
    return videoButton;
}
```

16. Add the following method that defines and starts animations for the `HyperlinkButton` that displays an image or a video received as a parameter:

```
private void AddImageVideoAnimation(HyperlinkButton hlButton)
{
    // Add a projection to the button
    var projection = new PlaneProjection();
    hlButton.Projection = projection;
    // Add a doubleAnimation for a Projection's RotationZ animation
    var doubleAnimProjectionZ = new DoubleAnimation();
    doubleAnimProjectionZ.Duration = new
        Duration(TimeSpan.FromSeconds(5));
    doubleAnimProjectionZ.From = 0.0;
    doubleAnimProjectionZ.To = 360.0;
    doubleAnimProjectionZ.FillBehavior = FillBehavior.HoldEnd;
    // Create a new Storyboard to handle the Projection's RotationZ
    animation
    var storyboardProjectionZ = new Storyboard();
    storyboardProjectionZ.Children.Add(doubleAnimProjectionZ);
    Storyboard.SetTarget(doubleAnimProjectionZ, projection);
    Storyboard.SetTargetProperty(doubleAnimProjectionZ,
        new PropertyPath("RotationZ"));
    // Add a doubleAnimation for a Projection's RotationY animation
    var doubleAnimProjectionY = new DoubleAnimation();
    doubleAnimProjectionY.Duration = new
        Duration(TimeSpan.FromSeconds(3));
    doubleAnimProjectionY.From = -45.0;
    doubleAnimProjectionY.To = 45.0;
    doubleAnimProjectionY.FillBehavior = FillBehavior.HoldEnd;
    doubleAnimProjectionY.RepeatBehavior = RepeatBehavior.Forever;
    doubleAnimProjectionY.AutoReverse = true;
    // Create a new Storyboard to handle the Projection's RotationY
    animation
    var storyboardProjectionY = new Storyboard();
    storyboardProjectionY.Children.Add(doubleAnimProjectionY);
    Storyboard.SetTarget(doubleAnimProjectionY, projection);
    Storyboard.SetTargetProperty(doubleAnimProjectionY,
        new PropertyPath("RotationY"));
    // Add a doubleAnimation for an Opacity animation
    var doubleAnimOpacity = new DoubleAnimation();
    doubleAnimOpacity.Duration = new
        Duration(TimeSpan.FromSeconds(5));
    doubleAnimOpacity.From = 0.0;
```



```
doubleAnimOpacity.To = 1.0;
doubleAnimOpacity.FillBehavior = FillBehavior.HoldEnd;
// Create a new Storyboard to handle the Opacity animation
var storyboardOpacity = new Storyboard();
storyboardOpacity.Children.Add(doubleAnimOpacity);
Storyboard.SetTarget(doubleAnimOpacity, hlButton);
Storyboard.SetTargetProperty(doubleAnimOpacity,
    new PropertyPath("Opacity"));
// Start the previously defined storyboards
storyboardProjectionZ.Begin();
storyboardOpacity.Begin();
storyboardProjectionY.Begin();
hlButton.Visibility = System.Windows.Visibility.Visible;
}
```

17. Add the following method that plays an audio file as background music for the application. It will just play background music once, no matter the number of times it is called.

```
private void AddBackgroundMusic(string url)
{
    if (_backgroundMusicAdded)
    {
        // Background music already loaded
        return;
    }
    _backgroundMusicAdded = true;
    MediaElement backgroundMusic = new MediaElement();
    LayoutRoot.Children.Add(backgroundMusic);
    backgroundMusic.Volume = 0.8;
    backgroundMusic.Source = new Uri(url);
    backgroundMusic.Play();
}
```

18. Now, it is necessary to add code to connect to the SharePoint server, connect to the lists, retrieve data from the assets library name stored in `_assetLibraryName`, request its files, and process each picture, video, and audio file to add it to the `wrapPanel WrapPanel`. These methods will run in the UI thread. Replace "http://gaston-pc" with the SharePoint website's URL.

```
private void Connect()
{
    // Runs in the UI Thread
```

```
        lblStatus.Content = "Started";
        // Replace http://gaston-pc with
        // your SharePoint 2010 Server URL and Site
        _context = new SP.ClientContext(new Uri("http://gaston-pc",
UriKind.Absolute));
        _context.Load(_context.Web);
        _context.ExecuteQueryAsync(OnConnectSucceeded, null);
    }

private void ConnectLists()
{
    // Runs in the UI Thread
    lblStatus.Content = "Web Connected. Connecting to Lists...";
    _context.Load(_context.Web.Lists);
    _context.ExecuteQueryAsync(OnConnectListsSucceeded, null);
}

private void GetListData()
{
    // Runs in the UI Thread
    lblStatus.Content = "Lists Connected. Getting List data...";
    _documents = _context.Web.Lists.GetByTitle(_assetLibraryName);
    _context.Load(_documents);
    _context.Load(_documents.RootFolder);
    // Request the files
    _context.Load(_documents.RootFolder.Files);
    _context.ExecuteQueryAsync(OnGetListDataSucceeded, null);
}

private void LoadItems()
{
    // Runs in the UI Thread
    lblStatus.Content = String.Format("Loading {0} items...",
        _documents.RootFolder.Files.Count);
    pgbLoadingStatus.Maximum = _documents.RootFolder.Files.Count;
    pgbLoadingStatus.Value = 0;
    _documentToLoad = 0;
    // Clear the WrapPanel children
    wrapPanel.Children.Clear();
    foreach (File file in _documents.RootFolder.Files)
    {
```

```
        _context.Load(file);
        _context.ExecuteQueryAsync(
            OnLoadItemsSucceeded,
            OnLoadItemsFailed);
    }
}

private void ShowItem()
{
    // Runs in the UI Thread
    lblStatus.Content = String.Format("Processing item # {0}",
        _documentToLoad);
    string fileName =
        _documents.RootFolder.Files[_documentToLoad].Name;
    string Url = _context.Url + _documents.RootFolder.Files
        [_documentToLoad].ServerRelativeUrl;
    switch (GetMediaFileType(fileName))
    {
        case MediaFileType.Audio:
            AddBackgroundMusic(Url);
            break;
        case MediaFileType.Picture:
            var imageButton = AddImage(Url);
            AddImageVideoAnimation(imageButton);
            break;
        case MediaFileType.Video:
            var videoButton = AddVideo(Url);
            AddImageVideoAnimation(videoButton);
            break;
    }
    // Update the progress bar
    pgbLoadingStatus.Value++;
    _documentToLoad++;
    if (_documentToLoad >= _documents.RootFolder.Files.Count)
    {
        // All documents loaded
        lblStatus.Content = "Displaying animations for all the
documents.";
    }
}
```

19. Most of the methods added in the previous step execute asynchronous queries to the SharePoint server. Both the successful and failed requests fire asynchronous callbacks that are going to run in another thread, different from the UI thread. Hence, if you have to update the UI, it is necessary to invoke the code to run in the UI thread. The following methods, which are going to be fired as asynchronous callbacks, schedule the execution of other methods to continue with the necessary program flow in the UI thread:

```
private void ShowErrorInformation(ClientRequestFailedEventArgs
args)
{
    MessageBox.Show("Request failed. " + args.Message + "\n" +
        args.StackTrace + "\n" +
        args.ErrorDetails + "\n" + args.ErrorValue);
}

private void OnConnectSucceeded(Object sender,
    SP.ClientRequestSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(ConnectLists);
}

private void OnConnectListsSucceeded(Object sender, SP.ClientReque
stSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(GetListData);
}

private void OnGetListDataSucceeded(Object sender, SP.ClientReques
tSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(LoadItems);
}

private void OnLoadItemsFailed(Object sender,
    SP.ClientRequestFailedEventArgs args)
{
    // This callback isn't called on the UI thread
    // Invoke a delegate and send the args instance as a parameter
    Dispatcher.BeginInvoke(() => ShowErrorInformation(args));
}
```

```
private void OnLoadItemsSucceeded(Object sender, SP.ClientRequestSucceededEventArgs args)
{
    // This callback isn't called on the UI thread
    Dispatcher.BeginInvoke(ShowItem);
}
```

20. Add the following line to the `LayoutRoot_Loaded` event:

```
Connect();
```

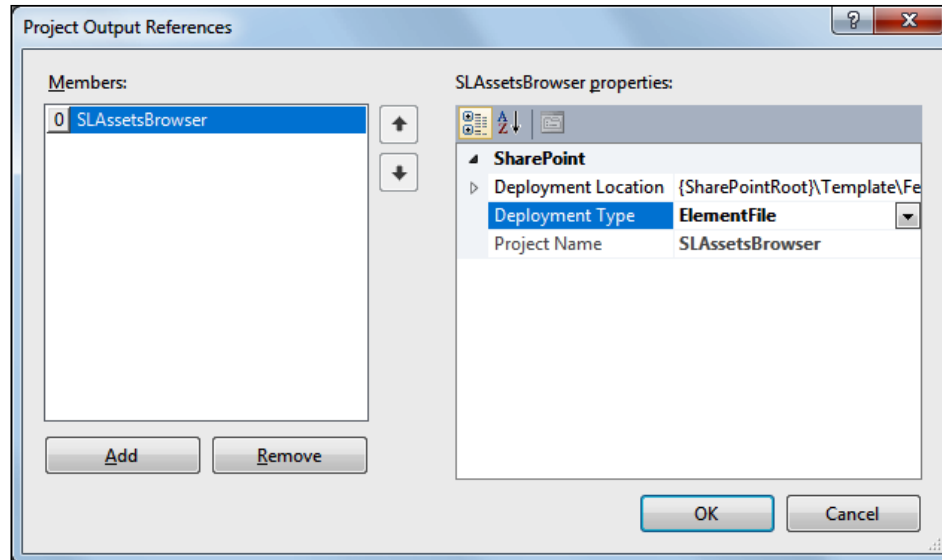
We created a new Silverlight RIA that receives an asset library name as a parameter from the Visual Web Part that renders this application. When the user selects an asset library from a drop-down list in the Visual Web Part, the Silverlight RIA will load the images, videos, and audio from the chosen asset library. We added the necessary code to create an application that displays the images and videos with many animations and effects.

Linking a SharePoint Visual Web Part to a Silverlight RIA

Follow these steps to link the previously created Visual Web Part, `AssetsBrowserWebPart`, with this new Silverlight RIA, `SLAssetsBrowser`. This way, the Silverlight RIA will be part of the package that contains the Visual Web Part.

1. Stay in Visual Studio as a system administrator user.
2. Expand the SharePoint Visual Web Part folder, `AssetsBrowserWebPart`, in the **Solution Explorer**.
3. Now, right-click on `AssetsBrowserWebPart` and select **Properties** in the context menu that appears. You will see the values for its properties in the **Properties** panel.
4. In the **Properties** palette, click the ellipsis (...) button for the **Project Output References** property. The **Project Output References** dialog box will appear.
5. Click on **Add** below the **Members:** list. The SharePoint 2010 Visual Web Part's project name, `SPAssetsBrowserWebPart`, will appear as a new member.
6. Go to its properties, shown on the list located at the right. Select the Silverlight application project's name, `SLAssetsBrowser`, in the **Project Name** drop-down list.

7. Select **Element File** in the **Deployment Type** drop-down list. The following value will appear in **Deployment Location**, `{SharePointRoot}\Template\Features\{FeatureName}\AssetsBrowserWebPart\`. The following screenshot shows the dialog box with the explained values:



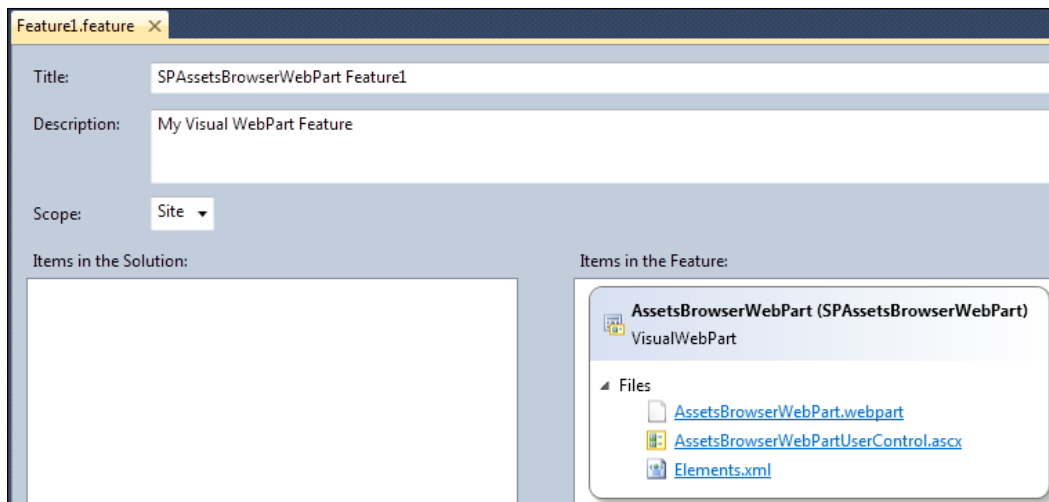
8. Click **OK**. The SharePoint Visual Web Part project now includes a reference to the Silverlight application project, `SLTasksViewer`. However, it is still necessary to add a line to the `Elements.xml` file to make the Silverlight RIA be part of the Visual Web Part.
9. Open the `Elements.xml` file. The following lines are the initial contents of this XML file. They describe the elements that compose this SharePoint 2010 Visual Web Part.

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/" >
  <Module Name="AssetsBrowserWebPart" List="113"
    Url="_catalogs/wp">
    <File Path="AssetsBrowserWebPart\AssetsBrowserWebPart.webpart"
      Url="AssetsBrowserWebPart.webpart" Type="GhostableInLibrary" >
      <Property Name="Group" Value="Custom" />
    </File>
  </Module>
</Elements>
```

10. Add the highlighted line before `</Module>`. The new contents of this XML file will include a reference to the linked Silverlight project `.xap` file, `SLAssetsBrowser.xap`. This is a new element for this SharePoint 2010 Visual Web Part. During the deployment process, the `SLAssetsBrowser.xap` file will be located in the `AssetsBrowserWebPart` folder in the **SharePoint package file**, also known as the **WSP package**, because it has a `.wsp` extension. Thus, the WSP package will also deploy the Silverlight application to the SharePoint server.

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/" >
  <Module Name="AssetsBrowserWebPart" List="113" Url="_catalogs/
wp">
  <File Path="AssetsBrowserWebPart\AssetsBrowserWebPart.webpart"
Url="AssetsBrowserWebPart.webpart" Type="GhostableInLibrary" >
  <Property Name="Group" Value="Custom" />
  </File>
  <!-- Added -->
  <File Path="AssetsBrowserWebPart\SLAssetsBrowser.xap"
  Url="SLAssetsBrowser.xap" />
  <!-- EOF Added -->
  </Module>
</Elements>
```

11. Remember to enable Silverlight debugging instead of the default script debugging capabilities.
12. Right-click on the solution's name in **Solution Explorer** and select **Properties** from the context menu that appears. Select **Startup Project** in the list on the left, activate **Single startup project**, and choose the SharePoint Visual Web Part project's name in the drop-down list below it, `SPAssetsBrowserWebPart`. This way, the solution is going to start with the SharePoint project and not with the Silverlight application. This is very important because it will allow us to debug the Silverlight application when it runs in a SharePoint site. Then, click **OK**.
13. Expand **Features | Feature1** in Solution Explorer and double-click on `Feature1.feature`. Visual Studio will display a new panel with the feature title, description, scope, and its items. The feature will include three files in the **Items in the feature** list, `AssetsBrowserWebPart`, `AssetsBrowsersWebPartUserControl.ascx` and `Elements.xml`.



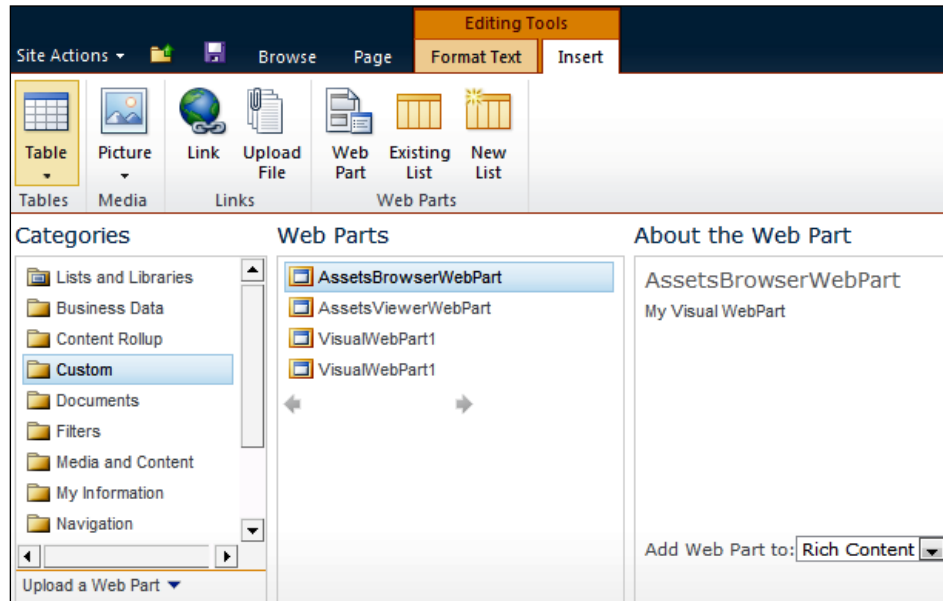
14. Build and deploy the solution.

Adding a SharePoint Visual Web Part in a Web Page

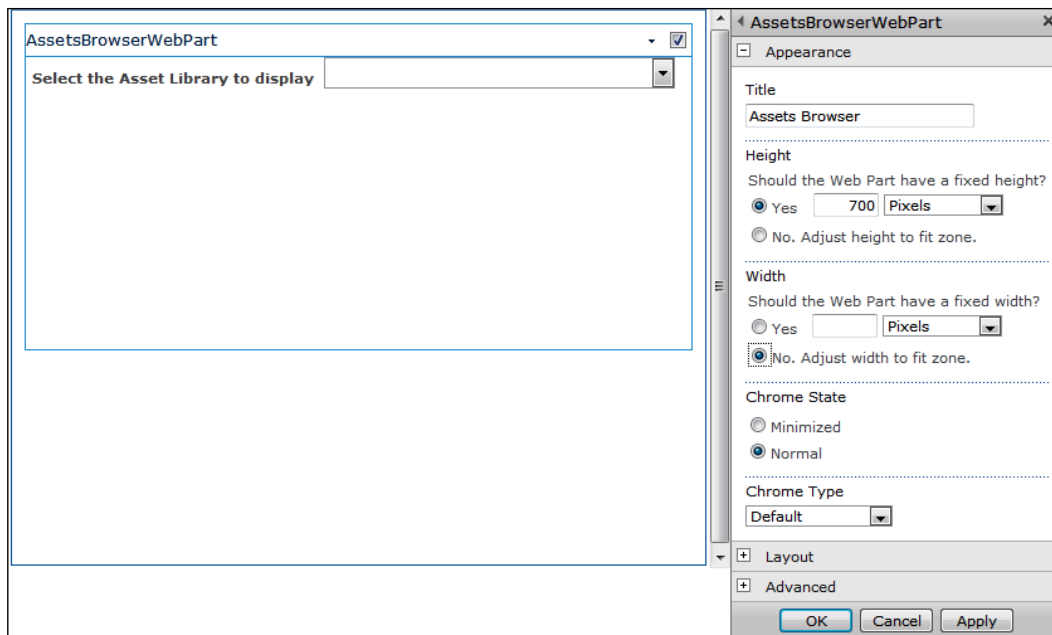
Now that the WSP package has been deployed to the SharePoint site, follow these steps to create a new web page and add the Visual Web Part that includes and renders the Silverlight RIA. In this case, it isn't necessary to upload the .xap file, because it was already deployed with the WSP package.

1. Open your default web browser, view the SharePoint site, and log in with your username and password.
2. Click **Site Actions** | **New Page** and SharePoint will display a new dialog box requesting a name for the new page. Enter `AssetsBrowser` and click on **Create**. SharePoint will display the editing tools for the new page.

3. Click **Insert | Web Part** in the ribbon and a new panel will appear. Select **Custom** in **Categories** and then the previously deployed Visual Web Part name, `AssetsBrowserWebPart`, in **Web Parts**.

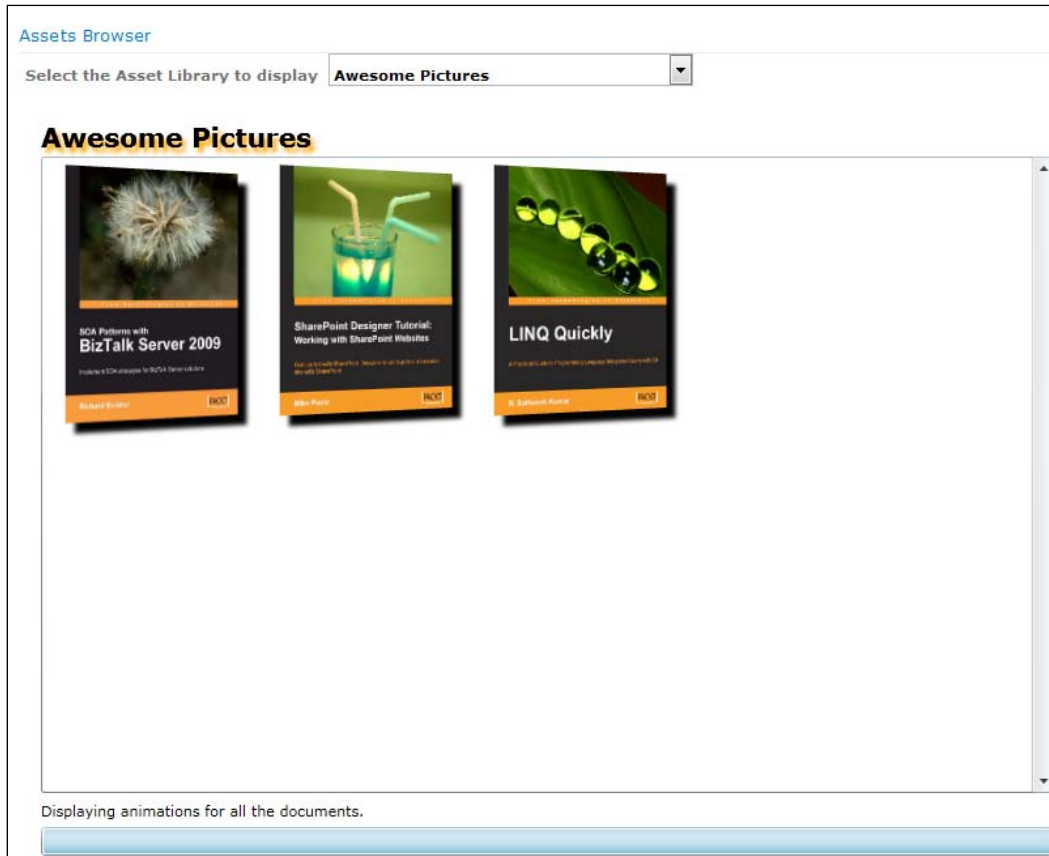


4. Click **Add**. The **Select the Asset Library to display** legend and the drop-down list will appear. Click on the down arrow, located at the top, and then select **Edit Web Part**. The `AssetsBrowserWebPart` pane will appear at the right. It will enable us to define many properties that affect the appearance and behavior for this Visual Web Part that renders a Silverlight RIA.
5. Enter `Assets Browser` in **Title**.
6. Click on **Yes** in **Should the Web Part have a fixed height?** and enter `700` in **Pixels**.



7. Click on **No. Adjust width to fit zone.** in **Should the Web Part have a fixed width?**, and then on **OK**.
8. Click on the **Save** button in the ribbon. Now, the new page will appear displaying the previously created Visual Web Part. This Web Part is going to display the drop-down list of asset libraries with pictures, videos, and audio files. The Silverlight RIA will appear below the drop-down list displaying the images and videos found in the first asset library in the drop-down list with interactive animations and dazzling effects. It is going to load and then it will display its different status values in the label located at the bottom:
 - **Started**
 - **Web Connected. Connecting to Lists...**
 - **Lists Connected. Getting List data...**
 - **Loading n items...**
 - **Processing item #x...**, where x is the number of picture, video or audio file being processed
 - **Displaying animations for all the documents. (this should be Bullet end)**

The following screenshot shows this value for the label and the Silverlight RIA displaying the images and videos for the chosen asset library.



9. Now, go back to Visual Studio and open the code-behind file for `AssetsBrowserWebPartUserControl.ascx`, `AssetsBrowserWebPartUserControl.ascx.cs`. Insert a breakpoint in the first line of the `Page_Load` event handler, `if (!IsPostBack)`. Insert another breakpoint in the line of code of the `cboDocumentLibraries_SelectedIndexChanged` event handler, `SelectedList = cboDocumentLibraries.SelectedValue;`

```

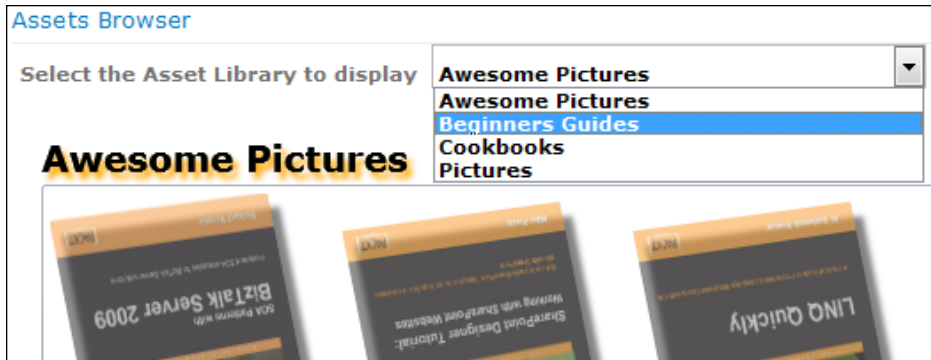
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        var _context = SPContext.Current;
        var documentLibraries = _context.Web.GetListsOfType(SPBaseType.DocumentLibrary);
        foreach (SPList libraryList in documentLibraries)
        {
            if ((libraryList.RootFolder.ItemCount > 0) &&
                ((libraryList.ContentTypes[0].Name == "Picture") ||
                 (libraryList.ContentTypes[0].Name == "Image") ||
                 (libraryList.ContentTypes[0].Name == "Audio") ||
                 (libraryList.ContentTypes[0].Name == "Video")))
            {
                // The list has at least 1 element
                cboDocumentLibraries.Items.Add(new ListItem(libraryList.Title));
            }
        }
        // Select the first item in the dropdown list
        cboDocumentLibraries.SelectedIndex = 0;
    }
    SelectedList = cboDocumentLibraries.SelectedValue;
}

protected void cboDocumentLibraries_SelectedIndexChanged(object sender, EventArgs e)
{
    SelectedList = cboDocumentLibraries.SelectedValue;
}

```

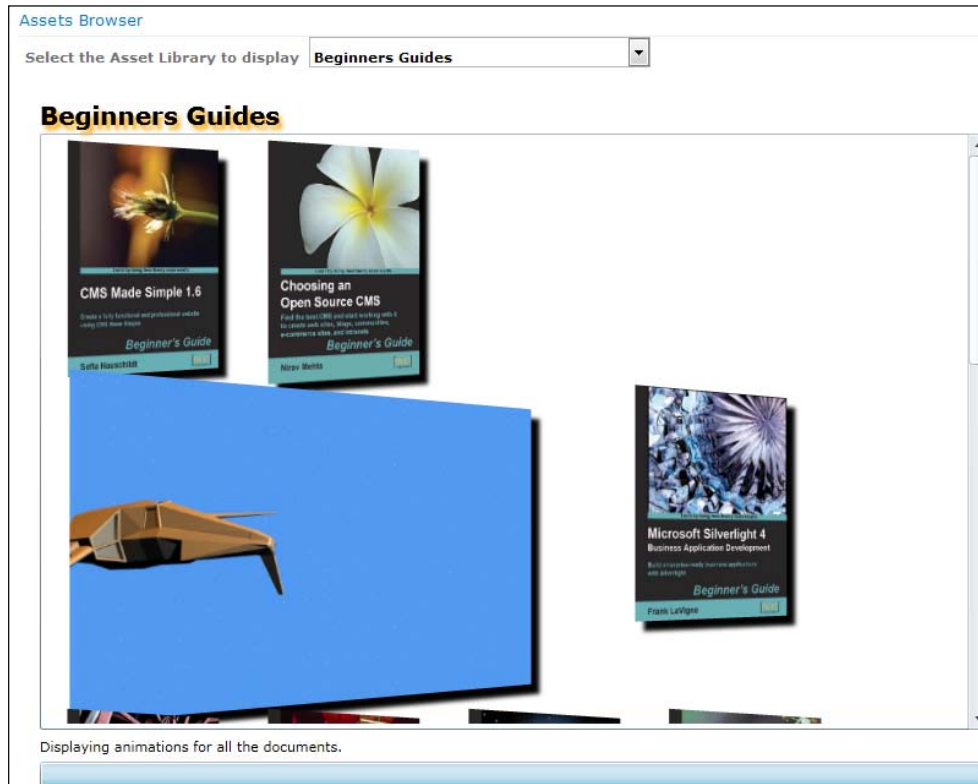
10. Open `AssetsBrowserWebPart.cs` and insert a breakpoint in the first line of the `OnPreRender` event handler.
11. Select **Debug | Start Debugging** from the Visual Studio's main menu or press *F5* to start debugging the solution.
12. Visual Studio will display a new window for your default web browser with the server and Site Collection in which you deployed the WSP package.
13. Enter the URL for the previously added page that contains the Visual Web Part in the web browser. This way, the ASP.NET code for the Visual Web Part will start running and Visual Studio will stop in the breakpoint established in the `Page_Load` event handler in the code-behind file, `AssetsBrowserWebPartUserControl.ascx.cs`.
14. Inspect the value for `IsPostBack` and it will be `false`, because it is the first time that the Visual Web Part is rendered. Thus, the method will run the code to add the titles of the document libraries that have pictures, images, audio, or video files. The first item for the `cboDocumentLibraries` `DropDownList` will be selected as the default library and the `SelectedList` property is going to save the selected title. Run the code step-by-step to understand the execution flow.

15. Then, Visual Studio will stop in the breakpoint established in the `OnPreRender` event handler, in `AssetsBrowserWebPart.cs`. The `renderHost` string will include a line that defines the value for the `Name` parameter. This parameter will specify a string with the value stored in the `SelectedList` public property. In this method, the code defines a new `LiteralControl` instance initialized with the `renderHost` string and adds it to the `Controls` `ControlCollection`.
16. Press `F5` and the web browser will display the Silverlight RIA with the first asset library contents.
17. Now, select a different asset library to display in the drop-down list located at the top of the Visual Web Part. This way, the ASP.NET code for the Visual Web Part will start running again, performing a postback, and Visual Studio will stop in the breakpoint established in the `Page_Load` event handler in the code-behind file, `AssetsBrowserWebPartUserControl.ascx.cs`.



18. Inspect the value for `IsPostBack` and it will be `true` because it is a postback for the `UserControl`. Thus, the method won't run the code to add the titles of the document libraries to the drop-down list. It will just run the line that sets the `SelectedList` property to the selected title. Run the code step-by-step to understand the execution flow.
19. Then, Visual Studio will stop in the breakpoint established in the `OnPreRender` event handler, in `AssetsBrowserWebPart.cs`. The `renderHost` string will include a line that defines the new value for the `Name` parameter, held in the previously explained `SelectedList` public property. This way, the new `LiteralControl` instance will add a Silverlight RIA with a different parameter value.

20. Press *F5* and the web browser will display the Silverlight RIA with the new asset library contents. The images and the videos will appear with animations and effects.



We added the SharePoint Visual Web Part to a new Web page in the SharePoint Site Collection. Then, we used Visual Studio to debug the Visual Web Part and we learned how Visual Web Parts renders a Silverlight RIA with parameters. We inserted many breakpoints to analyze the postback performed by the `UserControl` within the Visual Web Part.

Organizing controls in a containing box

The Silverlight RIA displays a `WrapPanel` control, `wrapPanel`, within a `ScrollViewer`, `scrollViewer`. The `WrapPanel` control works as a container and it locates its child elements in sequential positions from left to right, in columns, when its `Orientation` property is set to `Horizontal`. At the edge of the containing box, it breaks the content to the next row and therefore, it simplifies the organization of `HyperlinkButton` controls.

As we don't know the number of rows that will be necessary to display all the pictures and videos in the `WrapPanel` control, the `ScrollViewer` control defines a scrollable viewport. When the content of the `WrapPanel` is not entirely visible, the `ScrollViewer` will display scrollbars to allow the user to move the content area that is visible. The visible content is known as, **viewport** and all of the content included in the `ScrollViewer` is known as the **extent**.

The XAML markup in `MainPage.xaml` defines a `DropShadowEffect` for the `WrapPanel` control, with its `ShadowDepth` property set to 10. This way, all the `HyperlinkButton` controls added as `wrapPanel`'s children will inherit this effect and will drop a shadow with a depth of 10 pixels.

```
<ScrollViewer Height="487" HorizontalAlignment="Left"
Margin="12,41,0,0" Name="scrollViewer" VerticalAlignment="Top"
Width="776">
  <toolkit:WrapPanel Name="wrapPanel" Width="Auto" Height="Auto"
RenderTransformOrigin="0.497,0.493">
    <toolkit:WrapPanel.Effect>
      <DropShadowEffect ShadowDepth="10"/>
    </toolkit:WrapPanel.Effect>
  </toolkit:WrapPanel>
</ScrollViewer>
```

Reading files from an assets library

The `GetListData` method requests the asset library, a special list, specified in `_assetLibraryName`, and loads it, its `RootFolder` `Folder`, and its `RootFolder.Files` `FileCollection`.

```
_documents = _context.Web.Lists.GetByTitle(_assetLibraryName);
_context.Load(_documents);
_context.Load(_documents.RootFolder);
_context.Load(_documents.RootFolder.Files);
```

After the successful asynchronous execution of the queries, the `LoadItems` method clears the children for the `wrapPanel` `WrapPanel`. Then, it runs an asynchronous query to load each `File` in the asset library, `_documents`, `RootFolder.Files` `FileCollection`.

```
wrapPanel.Children.Clear();
foreach (File file in _documents.RootFolder.Files)
{
  _context.Load(file);
  _context.ExecuteQueryAsync(
    OnLoadItemsSucceeded,
    OnLoadItemsFailed);
}
```

Each successful asynchronous query will schedule the `ShowItem` method to run in the UI thread. The first time this method is called, `_documentToLoad` is set to 0 and the code in this method will increment `_documentToLoad` each time it finishes processing a file. The method retrieves the file name, stored in the `Name` property for the `File` instance to determine the media file type and saves it in the local `fileName` string. Then, it computes an absolute `Url` to access the file, `_context.Url` concatenated with the `ServerRelativeUrl` property for the `File` instance, and saves it in the local `Url` string.

```
string fileName = _documents.RootFolder.Files[_documentToLoad].Name;
string Url = _context.Url + _documents.RootFolder.Files[_
documentToLoad].ServerRelativeUrl;
```

Working with interactive animations and effects

A `switch` statement considers the results of the `GetMediaFileType` method that receives the `fileName` string as a parameter. As previously explained this method determines the media file type according to the extension and returns a `MediaFileType` as a result.

If the file type is `MediaFileType.Picture`, the method calls the `AddImage` method with the `Url` string as a parameter and it saves the `HyperlinkButton` instance returned by this method in `imageButton`. Then, it calls the `AddImageVideoAnimation` with `imageButton` as a parameter.

```
case MediaFileType.Picture:
    var imageButton = AddImage(Url);
    AddImageVideoAnimation(imageButton);
    break;
```

The `AddImage` method creates a new `Image` instance, sets values for its `MaxWidth` and `Stretch` properties, creates a `BitmapImage`, `bitmapImage`, with the absolute `Uri` from the URL received as a parameter, `url`, and assigns `bitmapImage` to the `image.Source` property.

```
var image = new Image();
image.MaxWidth = _maxImageWidth;
image.Stretch = Stretch.Uniform;
var bitmapImage = new BitmapImage(new Uri(url, UriKind.Absolute));
image.Source = bitmapImage;
```


Then, the code creates a new invisible `HyperlinkButton`, `imageButton`, and sets its `Content` property to the previously created `Image` instance, `image`. When `imageButton` becomes visible, it will show the bitmap image. The `NavigateUri` property for `imageButton` is set to a new `Uri` from the URL received as a parameter, `url`. The `TargetName` property is set to `_blank`, and therefore, when the user clicks the `HyperlinkButton`, the web browser will open a new window and will display the image from the URL.

The code attaches an event handler to the `MouseRightButtonDown` event that occurs when the user clicks the right mouse button and the mouse pointer is over the `Hyperlinkbutton`. It assigns a new `MouseButtonEventHandler` that will fire the `imageButton_MouseRightButtonDown` method. This method runs an animation for the `Hyperlinkbutton`.

```
var imageButton = new HyperlinkButton();
imageButton.Visibility = System.Windows.Visibility.Collapsed;
imageButton.Margin = new Thickness(_imageMargin);
imageButton.Content = image;
imageButton.NavigateUri = new Uri(url);
imageButton.MouseRightButtonDown += new MouseButtonEventHandler(imageButton_MouseRightButtonDown);
imageButton.TargetName = "_blank";
imageButton.Cursor = Cursors.Hand;
```

Finally, it is necessary to add the `HyperlinkButton` as a child to the `wrapPanel` `WrapPanel` and return the instance. As previously explained, `wrapPanel` will take care of organizing the layout of all the `HyperlinkButton` instances added as children.

```
wrapPanel.Children.Add(imageButton);
return imageButton;
```

At this point, the `HyperlinkButton` is invisible, because its `Visibility` property was set to `System.Windows.Visibility.Collapsed`. However, when the `AddImage` method returns, the `AddImageVideoAnimation` receives the `HyperlinkButton` control as a parameter, `hlButton`, and brings life to the image that it displays.

Firstly, it adds a `PlaneProjection` instance to the `HyperlinkButton`, `hlButton`, by setting its `Projection` property to a new `PlaneProjection` instance, `projection`. `PlaneProjection` is a subclass of the `Projection` class. The latter allows describing how to project a 2D object in the 3D space by using perspective transforms. Then, the code will run an animation with the values that define the perspective transform for `hlButton`.

```
var projection = new PlaneProjection();
hlButton.Projection = projection;
```



The `RotationX`, `RotationY`, and `RotationZ` properties for a `PlaneProjection` instance specify the number of degrees to rotate the `HyperlinkButton` in the space. The `LocalOffsetX` and `LocalOffsetY` properties specify the distance the `HyperlinkButton` is translated along each axis of the `HyperlinkButton`'s plane.

Then, the code defines three `DoubleAnimation` (`System.Windows.Media.Animation.DoubleAnimation`) instances and adds them as children of their corresponding `Storyboard` (`System.Windows.Media.Animation.Storyboard`) instances. A `DoubleAnimation` instance allows us to animate the value of a `Double` property between two target values specified by their `From` and `To` properties. It uses a linear interpolation over a specified duration, specified by the `Duration` property. Each `Storyboard` instance defines a container timeline that provides object and property targeting information for its child `DoubleAnimation` instance. The code creates the `DoubleAnimation` and `Storyboard` instances summarized in the following table:

DoubleAnimation instance	Storyboard instance	Animates	From	To	Duration (seconds)
doubleAnimProjectionZ	storyboardProjectionZ	projection.RotationZ	0.0	360.0	5
doubleAnimProjectionY	storyboardProjectionY	projection.RotationY	45.0	45.0	3
doubleAnimOpacity	storyboardOpacity	hlButton.Opacity	0.0	1.0	5

The following lines create the `doubleAnimProjectionZ` `DoubleAnimation` and set its properties. The `FillBehavior` property is set to `FillBehavior.HoldEnd` to specify that the animation must hold its value after it reaches the end of its active period. This way, the target property for this animation will remain at its end value after the animation ends and it won't revert to its non-animated value.

```
var doubleAnimProjectionZ = new DoubleAnimation();
doubleAnimProjectionZ.Duration = new Duration(TimeSpan.
FromSeconds(5));
doubleAnimProjectionZ.From = 0.0;
doubleAnimProjectionZ.To = 360.0;
doubleAnimProjectionZ.FillBehavior = FillBehavior.HoldEnd;
```

The next lines create the `Storyboard` instance and add `doubleAnimProjectionZ` as a child. Then, it is necessary to set the target object and the target property by calling the static methods `Storyboard.SetTarget` and `Storyboard.SetTargetProperty` with `doubleAnimProjectionZ` as its first parameter.

```
var storyboardProjectionZ = new Storyboard();
storyboardProjectionZ.Children.Add(doubleAnimProjectionZ);
Storyboard.SetTarget(doubleAnimProjectionZ, projection);
Storyboard.SetTargetProperty(doubleAnimProjectionZ, new
PropertyPath("RotationZ"));
```

The animations defined in `doubleAnimProjectionZ` and `doubleAnimOpacity` will run just one. However, `doubleAnimProjectionY` will run forever and it will auto reverse its execution, because its `RepeatBehavior` is set to `RepeatBehavior.Forever` and `AutoReverse` to `true`. Once it reaches the value specified by `To` for `projection.RotationY`, it will start a new animation from this value to the value specified by `From`, in the reverse direction.

```
doubleAnimProjectionY.RepeatBehavior = RepeatBehavior.Forever;
doubleAnimProjectionY.AutoReverse = true;
```

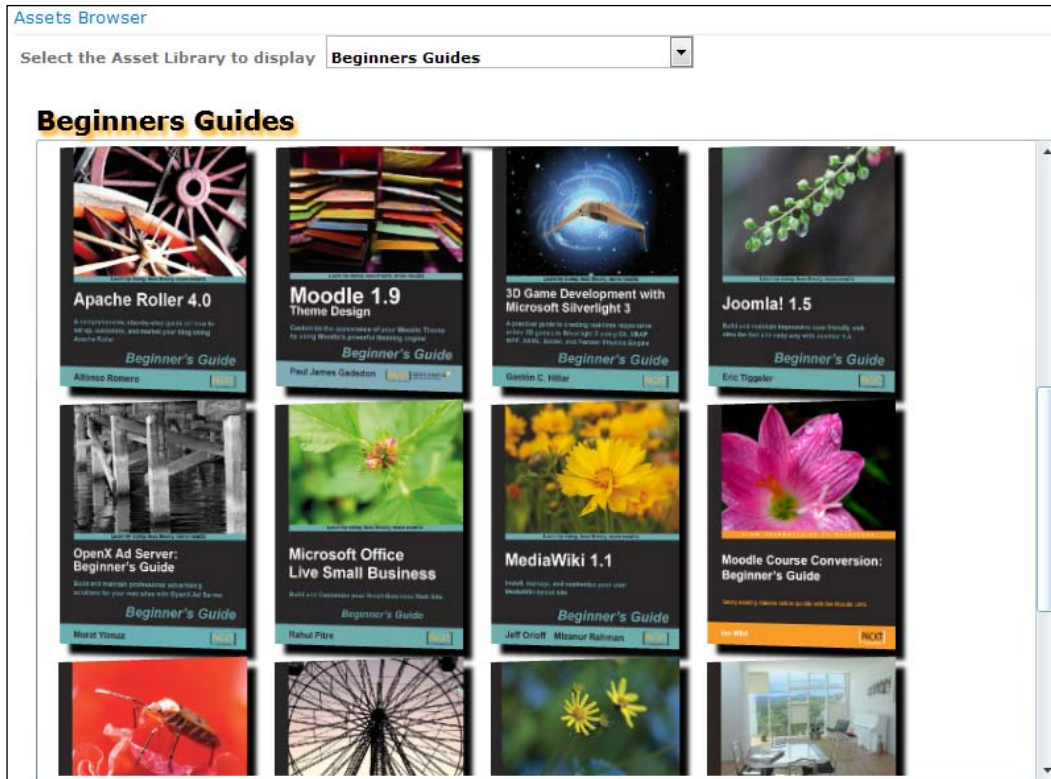
Once the method defines all the properties for the `DoubleAnimation` and `Storyboard` instances, it applies the animations associated with each `Storyboard` to their targets and initiates them by calling the `Begin` method.

```
storyboardProjectionZ.Begin();
storyboardOpacity.Begin();
storyboardProjectionY.Begin();
hlButton.Visibility = System.Windows.Visibility.Visible;
```

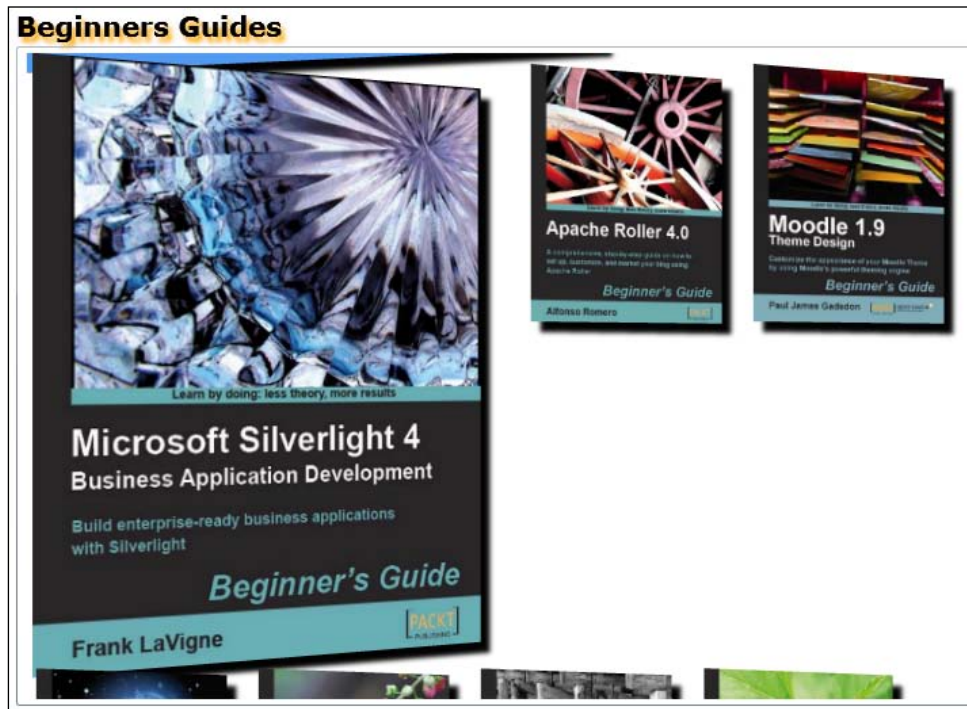


It is also possible for a single `Storyboard` instance to have many `DoubleAnimation` or other `Animation` subclasses as children. In this case, we used an independent `Storyboard` instance for each animation, because we want to have full control over each one to allow us to start and/or stop each animation to experience different alternatives for the UX in the future. However, if we just need to start all the animations at the same time, we can create a single `Storyboard` instance, add all the `DoubleAnimation` instances as their children, set the `Target` and `TargetProperty` for each `DoubleAnimation`, and call the `Begin` method.

When you open the page that contains the Visual Web Part, the Silverlight RIA will display all the hyperlink buttons that display images and videos with dazzling movements. `doubleAnimProjectionY` will run forever. The following screenshot shows one of the frames for the animations:



When you click on a dancing image, an animated `Hyperlink` button, the web browser will open a new window with the image displayed with its full size. When you right-click on a dancing image, the code in the `ImageButton_MouseRightButtonDown` method will run and the image will go on performing the same animation but it will also grow and then stretch. The container `WrapPanel` will make sure that the different elements displayed reorganize as the hyperlink button grows and stretches. The following picture shows one of the frames for the animation.



The `ImageButton_MouseRightButtonDown` method receives two parameters, object sender and `MouseButtonEventArgs e`. The first line sets the `Handled` property for `e` to `true`. This way, it ensures that Silverlight won't show the default Silverlight context menu that appears when the user right-clicks within the Silverlight application area.

```
e.Handled = true;
```

As we attached this method as an event handler for the `MouseRightButtonDown` event for a `HyperlinkButton`, `sender` can be cast to `HyperlinkButton`, `hlb` and we can access its `Content` property to access its associated `Image` and store its reference in `image`.

```
var h1Button = (sender as HyperlinkButton);
var image = h1Button.Content as Image;
```

Then, the code defines two `DoubleAnimation` instances and adds them as children of their corresponding `Storyboard` instances. The code creates the `DoubleAnimation` and `Storyboard` instances summarized in the following table.

DoubleAnimation instance	Storyboard instance	Animates	From	To	Duration (seconds)
doubleAnimMaxWidth	storyboardMaxWidth	image.MaxWidth	image.ActualWidth	scrollView.ActualWidth - (_imageMargin * 2)	6
doubleAnimMaxHeight	storyboardMaxHeight	image.MaxHeight	image.ActualHeight	scrollView.ActualHeight - (_imageMargin * 2)	6

Both `DoubleAnimation` instances have their `AutoReverse` property set to `true` and `RepeatBehavior` set to `RepeatBehavior(1)`. This means that the image will grow and then it will auto-reverse the animation to stretch to its original width and height.

Once the method defines all the properties for the `DoubleAnimation` and `Storyboard` instances, it applies the animations associated with each `Storyboard` to their targets and initiates them by calling the `Begin` method.

```
storyboardMaxWidth.Begin();
storyboardMaxHeight.Begin();
```

You can right-click on many images and the animation will run for all these images. The following screenshot shows one of the frames for the animation.



Adding and controlling videos

When the file type is `MediaFileType.Video`, the `ShowItem` method calls the `AddVideo` method with the `Url` string as a parameter and it saves the `HyperlinkButton` instance returned by this method in `videoButton`. Then, it calls the `AddImageVideoAnimation` with `videoButton` as a parameter.

```
case MediaFileType.Video:
    var videoButton = AddVideo(Url);
    AddImageVideoAnimation(videoButton);
    break;
```

The `AddVideo` method creates a new `MediaElement` instance, `media`, and sets values for its `MaxWidth` and `Stretch` properties. Then, it assigns the absolute `Uri` from the URL received as a parameter, `url`, to the `media.Source` property.

```
MediaElement media = new MediaElement();
media.MaxWidth = (_maxImageWidth * 3);
media.Stretch = Stretch.UniformToFill;
media.Source = new Uri(url, UriKind.Absolute);
```

Then, the code sets the `AutoPlay` property to `true` to automatically start the playback of the video specified in the `Source` property. The code attaches an event handler to the `MediaEnded` event that occurs when the video finishes. It assigns a new `RoutedEventHandler` that will fire the `media_MediaEnded` method. This method plays the video again from the beginning and therefore, the video is going to play forever while the Silverlight RIA performs all the animations.

```
media.AutoPlay = true;
media.MediaEnded += new RoutedEventHandler(media_MediaEnded);
```

Then, the code creates a new invisible `HyperlinkButton`, `videoButton`, and sets its `Content` property to the previously created `MediaElement` instance, `media`. When `videoButton` becomes visible, it will show the video being reproduced. The `NavigateUri` property for `videoButton` is set to a new `Uri` from the URL received as a parameter, `url`. The `TargetName` property is set to `_blank` and therefore, when the user clicks the `HyperlinkButton`, the web browser will open the video from the URL in the default player associated with the file extension.

The code attaches an event handler to the `MouseRightButtonDown` event that occurs when the user clicks the right mouse button and the mouse pointer is over the `Hyperlinkbutton`. It assigns a new `MouseButtonEventHandler` that will fire the `videoButton_MouseRightButtonDown` method. This method runs the previously explained animation for the `Hyperlinkbutton`. This animation is very similar to the one explained for the `imageButton_MouseRightButtonDown` method.

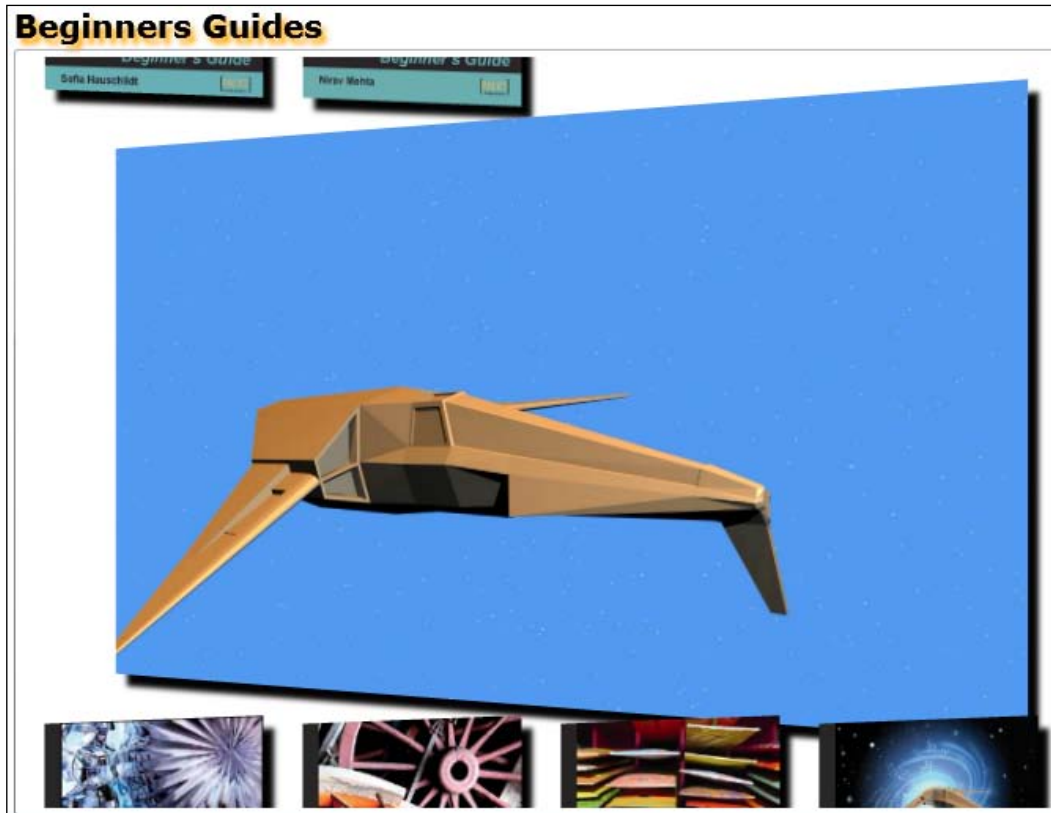
```
var videoButton = new HyperlinkButton();
videoButton.Visibility = System.Windows.Visibility.Collapsed;
videoButton.Margin = new Thickness(_imageMargin);
videoButton.Content = media;
videoButton.NavigateUri = new Uri(url);
videoButton.MouseRightButtonDown += new MouseButtonEventHandler(videoB
utton_MouseRightButtonDown);
videoButton.TargetName = "_blank";
videoButton.Cursor = Cursors.Hand;
```

Finally, it is necessary to add the `HyperlinkButton` as a child to the `wrapPanel` `WrapPanel` and return the instance.

```
wrapPanel.Children.Add(videoButton);
return videoButton;
```

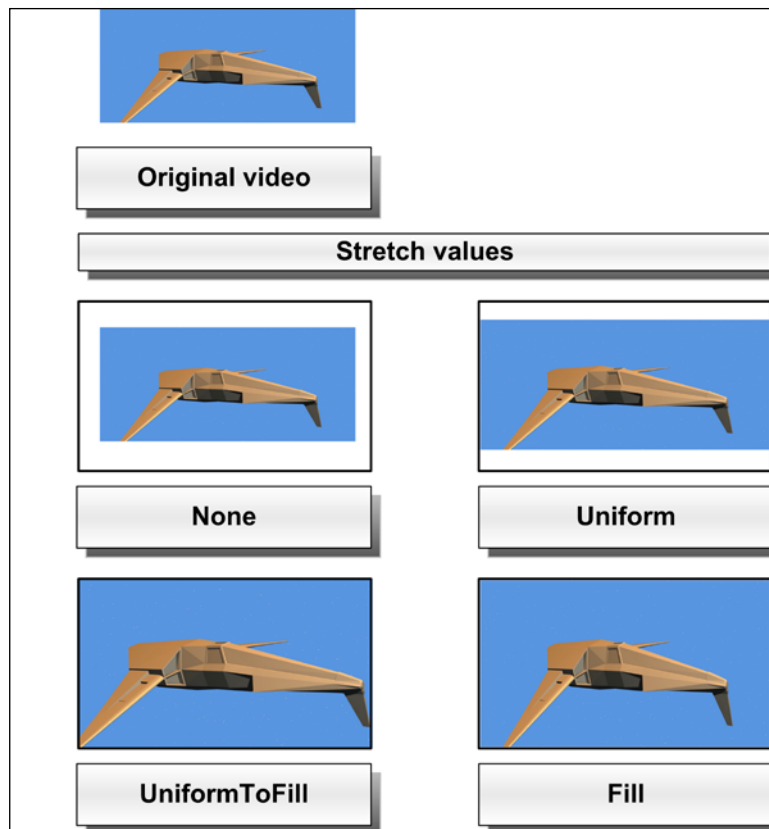
At this point, the `HyperlinkButton` is invisible, because its `Visibility` property was set to `System.Windows.Visibility.Collapsed`. However, when the `AddVideo` method returns, the `AddImageVideoAnimation` receives the `HyperlinkButton` control as a parameter, `hlButton`, and brings life to the video that it displays, as explained for the images.

The following screenshot shows one of the frames for the animated `HyperlinkButton` reproducing the video and growing after the user right-clicked on it:



We defined the horizontal reproduction area for the video to be `_maxImageWidth * 3` pixels and we assigned the `UniformToFill` value to the `Stretch` property. Thus, the `MediaElement` resizes the original to fill the container's dimensions while preserving the video's native aspect ratio.

The following screenshot shows the results of using the four possible values in the `Stretch` property with the same original video:



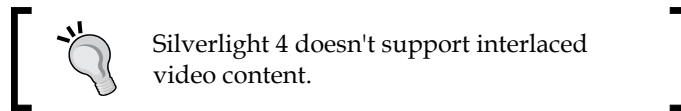
The following table explains the results of using the aforementioned values:

Stretch value	Description	Aspect ratio
None	The video preserves its original size.	Preserved
Uniform	The video is resized to fit in the destination dimensions.	Preserved
UniformToFill	The video is resized to fill the destination dimensions. The video content that does not fit in the destination rectangle is clipped.	Preserved
Fill	The video is resized to fill the destination dimensions.	Not preserved

Video formats supported in Silverlight 4

Silverlight 4 supports the video encodings shown in the following table:

Encoding name	Description and restrictions
None	Raw video
YV12	YCrCb(4:2:0)
RGBA	32-bit Red, Green, Blue, and Alpha
WMV1	Windows Media Video 7
WMV2	Windows Media Video 8
WMV3	Windows Media Video 9
WMVA	Windows Media Video Advanced Profile (non-VC-1)
WMVC1	Windows Media Video Advanced Profile (VC-1)
H.264 (ITU-T H.264 / ISO MPEG-4 AVC)	H.264 and MP43 codecs; base main and high profiles; only progressive (non-interlaced) content and only 4:2:0 chroma sub-sampling profiles



If we want to use a video with an encoding that does not appear in the previously shown table in a Silverlight RIA, we will have to convert it to one of the supported formats before uploading it to a SharePoint assets library.

Adding and controlling sounds and music

When the file type is `MediaFileType.Audio`, the `ShowItem` method calls the `AddBackgroundMusic` method with the `Url` string as a parameter.

```
case MediaFileType.Audio:  
    AddBackgroundMusic(Url);  
    break;
```

The `AddBackgroundMusic` method checks whether it was called before (`_backgroundMusicAdded == true`) before running the rest of the code, because it doesn't want to reproduce two audio files as the background music. If `_backgroundMusicAdded` is true, it assigns true to `_backgroundMusicAdded`.

The code creates a new `MediaElement` instance, `backgroundMusic`, adds it to a parent container, `LayoutRoot`, and sets its `Volume` property to 80% (0.8). The `Volume` ranges from 0 to 1. It uses a linear scale.

```
_backgroundMusicAdded = true;
MediaElement backgroundMusic = new MediaElement();
LayoutRoot.Children.Add(backgroundMusic);
```

Then, it assigns the absolute `Uri` from the URL received as a parameter, `url`, to the `backgroundMusic.Source` property and calls the `Play` method to start reproducing the audio file with the specified volume level. The background music will be reproduced just once.

```
backgroundMusic.Volume = 0.8;
backgroundMusic.Source = new Uri(url);
backgroundMusic.Play();
```

Audio formats supported in Silverlight 4

Silverlight 4 supports the audio encodings shown in the following table:

Encoding name	Description and restrictions
LPCM	Linear 8 or 16-bit Pulse Code Modulation.
WMA Standard	Windows Media Audio 7, 8, and 9 Standard.
WMA Professional	Windows Media Audio 9 and 10 Professional; Multichannel (5.1 and 7.1 surround) is automatically mixed down to stereo. It supports neither 24-bit audio nor sampling rates beyond 48 kHz.
MP3	ISO MPEG-1 Layer III.
AAC	ISO Advanced Audio Coding; AAC-LC (Low Complexity) is supported at full fidelity (up to 48 kHz). HE-AAC (High Efficiency) will decode only at half fidelity (up to 24 kHz); Multichannel (5.1) audio content is not supported.

If we want to use an audio file with an encoding that does not appear in the previously shown table, we will have to convert it to one of the supported formats before uploading it to a SharePoint assets library.

Changing themes in Silverlight and SharePoint

The Visual Web Part is a great candidate for applying the themes included in Silverlight's Toolkit to offer the user a more exciting UI.

1. Stay in Visual Studio as a system administrator user.
2. Add a reference to `System.Windows.Controls.Theming.Toolkit.dll`. Remember that it is located in the `Bin` sub-folder.
3. Add a reference to the DLL for `System.Windows.Controls.Theming.ShinyRed` in the `Themes` sub-folder. This way, we are going to be able to apply the `ShinyRed` theme.
4. Add the following line to include the namespace that defines the theme in the `UserControl` defined in `MainPage.xaml`:

```
xmlns:shinyRed="clr-namespace:System.Windows.Controls.Theming;assembly=System.Windows.Controls.Theming.ShinyRed"
```
5. Add the following line before the definition of the main `Grid, LayoutRoot`:

```
<shinyRed:ShinyRedTheme>
```
6. Add the following line after the definition of the main `Grid, LayoutRoot`:

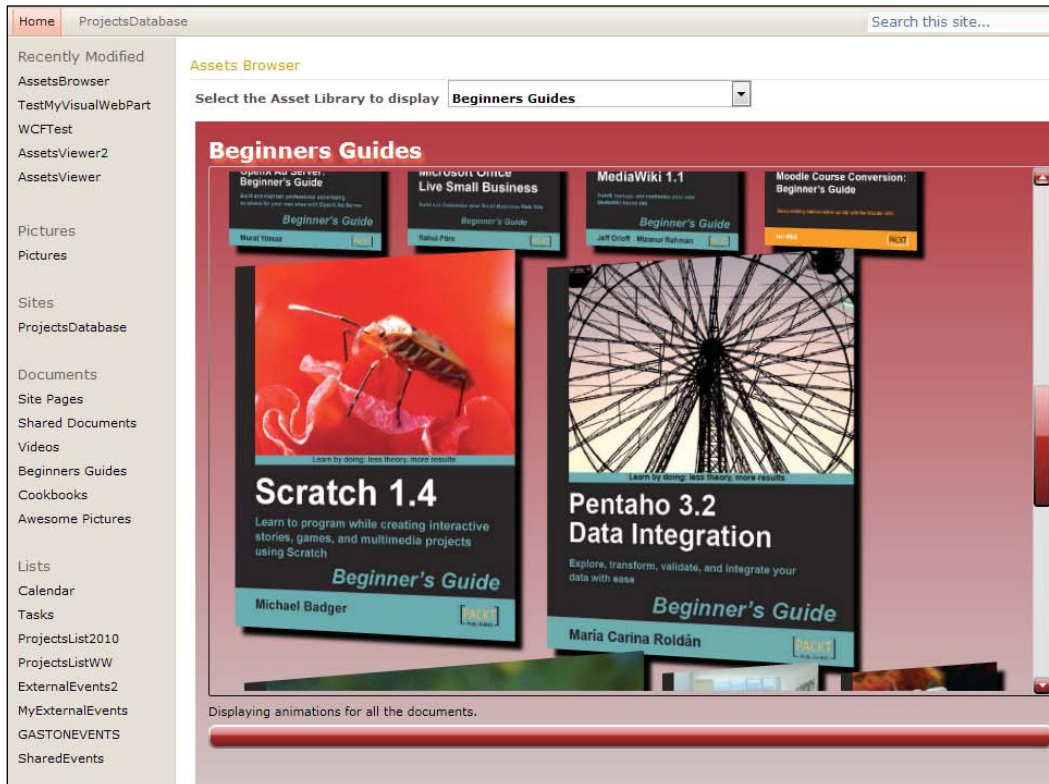
```
</shinyRed:ShinyRedTheme>
```
7. This way, the `ShinyRed` theme will be applied to the main `Grid, LayoutRoot`, and all its child controls. Build and deploy the solution and open the page that displays the Visual Web Part. The Silverlight RIA looks really more attractive. However, the colors displayed by the rest of the SharePoint UI don't match the `ShinyRed` theme colors.

10. Select the **Municipal** theme in the list located at the right. This theme uses a color scheme that is appropriate for Silverlight applications that use the ShinyRed theme.

The screenshot shows the 'Select a Theme' dialog box in SharePoint. It is divided into three main sections: 'Select a Theme', 'Customize Theme', and 'Preview Theme'.
1. **Select a Theme:** On the left, there is a grid of color swatches for 'Hyperlink' and 'Followed Hyperlink'. Below the grid, 'Heading Font' and 'Body Font' are both set to 'Lorem ipsum dolor sit amet...'. On the right, a list of themes is shown, with 'Municipal' highlighted in blue. Other themes include Default (no theme), Azure, Berry, Bittersweet, Cay, Classic, Construct, Convection, Felt, Graham, Grapello, Laminate, Mission, Modern Rose, Pinnate, Ricasso, Summer, Vantage, and Viewpoint.
2. **Customize Theme:** This section allows for further customization. It lists various elements with corresponding color swatches and a 'Select a color...' button: Text/Background - Dark 1, Text/Background - Light 1, Text/Background - Dark 2, Text/Background - Light 2, Accent 1 through 6, Hyperlink, and Followed Hyperlink. At the bottom, 'Heading Font' and 'Body Font' are both set to 'Verdana'.
3. **Preview Theme:** At the bottom right, there is a 'Preview' button. A note at the bottom left says 'Click the button to open a new window and preview the selected theme applied to this'.

11. Click on **Preview** and the web browser will open a new window with your site's home page with the new color schemes and fonts that the selected theme defines.
12. Close this window and click on **Apply**. SharePoint will apply the new theme to the pages that haven't been individually themed. The new theme won't affect the site's layout.

13. Now, refresh the page that displays the Visual Web Part and the combination of a new SharePoint theme with the theme applied to the Silverlight RIA will look really nice.



Summary

We learnt a lot in this chapter about accessing asset libraries in a Silverlight RIA rendered in a SharePoint Visual Web Part. Specifically, we were able to send parameters to the Silverlight control host to define the desired asset library to display. We worked with classic lists but this time we consumed files. We took advantage of Silverlight 4 rich media features to add effects and interactive animations to the images and videos.

We have learned many alternatives to integrate Silverlight 4 with SharePoint 2010 and we have understood the great possibilities offered by Silverlight RIAs in SharePoint sites.

9

Data Access Strategies



This chapter is taken from *Microsoft Silverlight 4: Building Rich Enterprise Dashboards* (Chapter 9) by Todd Snyder, Joel Eden, Ph.D., Jeff Smith, Matthew Duffield.

Silverlight is a Rich Internet Application (RIA) platform that allows you to use a rich multimedia experience to showcase application data. This is one of the main advantages that Silverlight has over other platforms for making your dashboard come alive. No matter how flashy you make the application, the most important part that your users care about is information (data). Whether it's the CIO of a fortune 500 company looking to track sales, or a fantasy football owner tracking his team's trends, information is power.

In this chapter, we will explore the different data access strategies you can use while building a Silverlight application. How to build your own custom data services using SOAP, REST, and OData, a walkthrough of how to consume externally hosted services, and how the cross-domain security policy system works with Silverlight to call external services.

In this chapter, we will cover the following topics:

- Data access overview
- Understanding network security
- Building services with Windows Communication Foundation
- Exploring OData data services

Data access overview

Silverlight offers a rich set of options for integrating data services into your applications:

- You can use basic HTTP and socket-based networking classes to perform low-level network calls.
- Take advantage of the Windows Communication Foundation (WCF) and Language Integrated Query (LINQ) application programming interfaces included in the Silverlight runtime to consume simple and complex data services.
- Integrate externally hosted services using the Silverlight cross-domain security policy system.

Core networking classes

The Silverlight networking stack includes classes for communication using raw sockets and over the HTTP protocol. The HTTP classes allow for simple communication to external resources, such as an image or XML file. For more advanced scenarios, you can use the socket classes for communication.

- HTTP classes (`HttpRequest`, `HttpResponse` or `WebClient`) included in the `System.Net` namespace. These classes allow you to communicate to a server using the HTTP or HTTPS protocol. They are the best options when you need to call an external web service or download files directly.
- Sockets and Multicast classes included in the `System.Net.Sockets` namespace. These classes allow you to perform general networking calls using a socket interface, real-time duplex communication with a remote network service, or set up a one-to-many or many-to-many multicast communication.

Working with WebClient

The simplest way to get data into your Silverlight application is to use the `WebClient` class to access a XML data file stored on the server. By default, `WebClient` can use a relative path to access files in the Silverlight application's XAP file's host directory (`ClientBin`) or one of its sub directories. For more advanced scenarios, where you may need to set the HTTP headers sent to a server, you would want to use the `HttpRequest` class.

All network operations in Silverlight are performed asynchronously; so you will need to create a callback method for when the operation is complete. Depending on the type of networking call you make, you may need to use the `Dispatcher` class to marshal data back to the UI thread. For example, `WebClient` operations are automatically returned to the UI thread, but `HttpRequest` requires you to use a `Dispatcher` to marshal data.

The following code uses the `WebClient` class to retrieve the `Products.xml` from the host domain of the running Silverlight application. Because the operation is asynchronous, you will need to set up a callback method that will be called after the file has finished being downloaded.

```
WebClient client = new WebClient();
client.DownloadStringCompleted +=
    new DownloadStringCompletedEventHandler
        (client_DownloadStringCompleted);
client.DownloadStringAsync(new Uri("products.xml", UriKind.Relative));
```

We will be using XLINQ to load data from the `Products.xml` file into a collection of products. To make it easier to map the data into the product class, let's set up a couple of extensions methods. Extensions methods were added in .NET 3.5, and allow you to add helper methods to externally defined classes. To create an extension method, define a static class with one or more static methods. The first parameter to each method must be the class being extended. To use your own custom extension method, just add the `using` statement to the namespace where you defined your extension methods as follows:

```
using System.Xml.Linq;
namespace Chapter9.Common
{
    public static class XElementExtension
    {
        public static int GetIntValue(this XElement element)
        {
            int value = 0;
            int.TryParse(element.Value, out value);
            return value;
        }
        public static double GetDoubleValue(this XElement element)
        {
            double value = 0d;
            double.TryParse(element.Value, out value);
            return value;
        }
    }
}
```

The data in the `Products.xml` file will be loaded into a collection of the `Product` class. The class contains properties (ID, Name, Unit Price, etc...) that match the columns in the data grid that will be used for displaying the data from the `Products.xml` file. Throughout the chapter, we will be reusing the `Product` class, so refer back to this code snippet when creating the other samples.

```
public class Product
{
    public string ProductId {get; set;}
    public string ProductName {get; set;}
    public string Supplier {get; set;}
    public string Category {get; set;}
    public double UnitPrice {get; set;}
    public int UnitsInStock {get; set;}
}
```

After the `WebClient` asynchronous asynchronously completes, the `client_DownloadStringCompleted` method will be called. If the operation was successful, the contents of the `Products.xml` file will be stored in the `e.Result` property. We will now use `XLINQ`, along with the extension methods and product class we previously defined, to load a collection of products. Once the collection is loaded, we will set the item source of the `ProductList` data grid as follows:

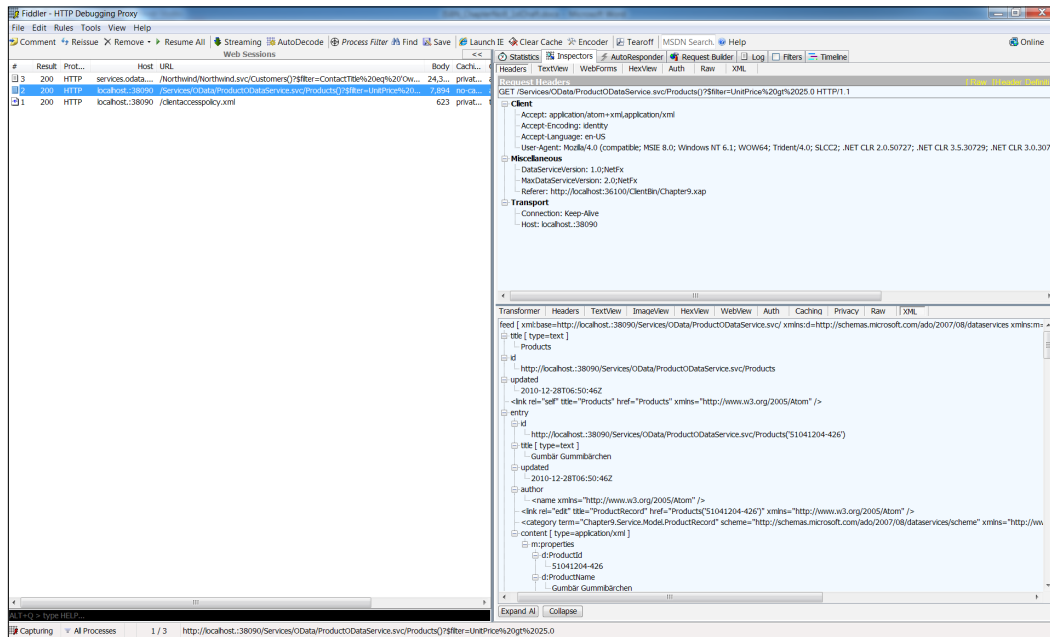
```
void client_DownloadStringCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    XDocument document = XDocument.Load(new StringReader(e.Result));
    var result = from d in document.Root.Descendants("Product")
    select new Product
    {
        ProductId = d.Element("ProductID").Value,
        ProductName = d.Element("ProductName").Value,
        Supplier = d.Element("Supplier").Value,
        Category = d.Element("Category").Value,
        UnitPrice = d.Element("UnitPrice").GetDoubleValue(),
        UnitsInStock = d.Element("UnitsInStock").GetIntValue()
    };
    this.ProductList.ItemsSource = result;
}
```

The following XAML defines the DataGrid we will be using to display the collection of products returned from downloading the requested file. Throughout the chapter, we will be using a similarly defined grid to display the data for our sample applications.

```
<navigation:Page x:Class="Chapter9.Views.WebClientData"
  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"
  xmlns:navigation="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls.Navigation"
  xmlns:Controls="clr-namespace:System.Windows.Controls;
    assembly=System.Windows.Controls.Data"
  d:DesignWidth="640" d:DesignHeight="480"
  Title="WebClientData Page">
  <Grid x:Name="LayoutRoot">
    <Controls:DataGrid x:Name="ProductList"
      AutoGenerateColumns="False">
      <Controls:DataGrid.Columns>
        <Controls:DataGridTextColumn Binding=
          "{Binding ProductId}" Header="Product ID" />
        <Controls:DataGridTextColumn Binding=
          "{Binding ProductName}" Header="Product" />
        <Controls:DataGridTextColumn Binding=
          "{Binding Supplier}" Header="Supplier" />
        <Controls:DataGridTextColumn Binding=
          "{Binding Category}" Header="Category" />
        <Controls:DataGridTextColumn Binding=
          "{Binding UnitPrice}" Header="Unit Price" />
        <Controls:DataGridTextColumn Binding=
          "{Binding UnitsInStock}" Header="Units in Stock" />
      </Controls:DataGrid.Columns>
    </Controls:DataGrid>
  </Grid>
</navigation:Page>
```

Using Fiddler

While building a Silverlight application, it can be problematic to troubleshoot network communication. Fiddler (<http://www.fiddler2.com/fiddler2/>) is a HTTP communication tool that you should download to troubleshoot networking issues.

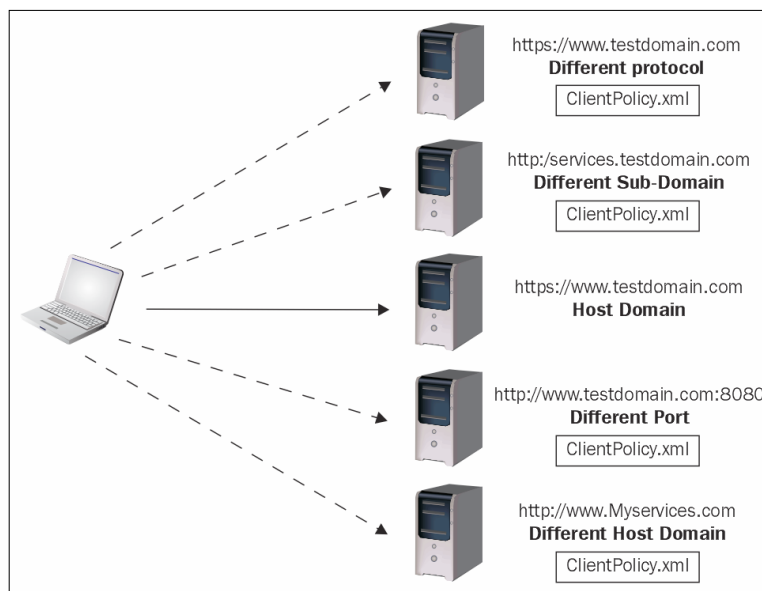


By default, Fiddler cannot trace traffic sent to `http://localhost`. To work around this, you can use your computer name instead of the local host, or add a period in front of the port number (`http://localhost.:38090`) of your services URI. To monitor your Silverlight application, fire up Fiddler. It will monitor all the HTTP traffic on your machine. As each HTTP request is sent, you can see the contents of the request and result returned from the server. Always remember to fire up Fiddler if you start encountering errors communicating from your Silverlight application.

Understanding network security

Silverlight uses a security policy-based approach for allowing cross-domain access. Depending on how you have configured your Silverlight application and the location of the resource (File or Web Service) you are trying to access, Silverlight will download and verify that your application (Host Domain) can access the resource. If you are running an out-browser Silverlight application with evaluated trust or accessing a resource on the same host domain (and port) as your Silverlight XAP file, the policy file is not required.

The following image shows the different rules that Silverlight uses for deterring when to download and verify the cross-domain policy file. For resources on the same domain, the file is not required. Unless you have an out-of-browser application with evaluated trusted enabled, the cross-domain policy file is required for the other scenarios: Different Protocols (HTTP vs HTTPS), Sub-Domains, Ports, or Host Domains.



Silverlight supports two different types of security policy files. The Flash policy file (`Crossdomain.xml`) and the Silverlight policy file (`ClientAccessPolicy.xml`). If neither file can be found when your application makes a cross-domain, calling Silverlight will raise a security exception.

- Flash policy file (`crossdomain.xml`): This is the same cross-domain policy file that Adobe Flash uses. This file can only be used by the HTTP and Web Client classes in the `System.Net` namespace. The Silverlight runtime requires that all domains have full access to work.
- Silverlight policy files (`ClientAccessPolicy.xml`): The policy file supports cross-domain access for the Web Client and HTTP classes; the classes in the `System.Net.Socket` namespace, and the client libraries used by Windows Communication Foundation (WCF).

The structure of the Silverlight cross-domain policy file allows you to define the host URI and the network path that external domains have access to. If you want to allow full access to your server, set up the policy file using "*" to allow all domains, and set the resource path to the root "/".

```
<?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>
```

You can restrict access by defining specific domains and paths. For example if you want to restrict access to only the `http://www.mycompany.com` domain and services located in the services folder, set up your `ClientAccessPolicy.xml` like the policy file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="www.mycompany.com" />
      </allow-from>
```

```
<grant-to>
  <resource path="/services/" include-subpaths="true"/>
</grant-to>
</policy>
</cross-domain-access>
</access-policy>
```

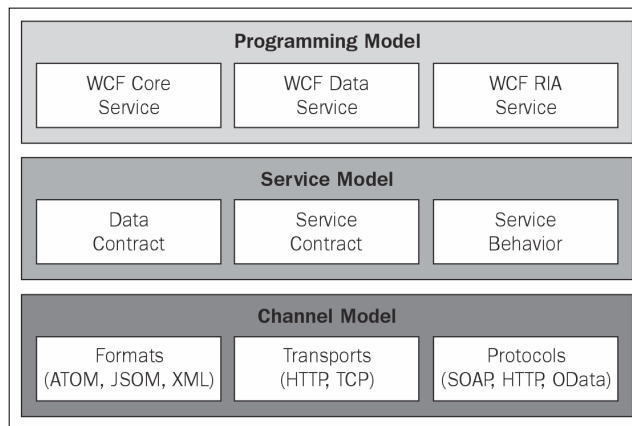
To allow clients to send HTTP headers, you need to include the `http-request-header` attribute. To set up different access rules, define one or more policy elements with the correct corresponding configuration. To allow externally hosted Silverlight applications to call your services, you need to deploy a `ClientAccessPolicy.xml` at the root folder (domain) of the service's URI. For example if your services are deployed at the `http://www.mycompany.com/services/v1/dataservice.svc`, you need to deploy the policy file at the root URI (`http://www.mycompany.com`).

Building services with Windows Communication Foundation

When building your own data services, you want to take advantage of the rich programming model that Windows Communication Foundation (WCF) provides. Depending on the type of application and its requirements, there are different service frameworks you can use as follows:

- WCF Core Services offers the most flexible programming model. It allows you to define a web service class that can be consumed by your Silverlight application. You can use Visual Studio to generate proxy classes for the service, and use the WCF client library to asynchronously interact with the service. These types of services offer the most flexibility by allowing you to configure end points using different data formats and protocols.
- WCF Data Services allows you to expose your data source using a Representational State Transfer (REST) style. Data is exposed as an Open Data Protocol (OData) feed. You can use the HTTP networking classes or the WCF Data Service Client Library to make REST-based queries or actions (Create, Update, or Delete) method calls.
- WCF RIA Services offers a simplified programming model for building n-tier applications. It allows you to define and share data model classes between a Silverlight client and a web service. In addition, RIA services offers built-in methods for validation, concurrency, and security. RIA services are ideally suited for simple forms over data type applications.

Windows Communication Foundation (WCF) is a rich and extensible Application Programming Interface (API) for building and consuming services. As mentioned above, there are several different programming models you can use when working with WCF. When designing a WCF service, you will create service and data contracts. Service contracts define the operations (methods) of your service and data contracts define the message structure you are passing and returning from the services. Service behaviors are used for defining cross-cutting operations that run before or after each request. The channel used by WCF for communicating is based on the protocol and formats you defined for your services.

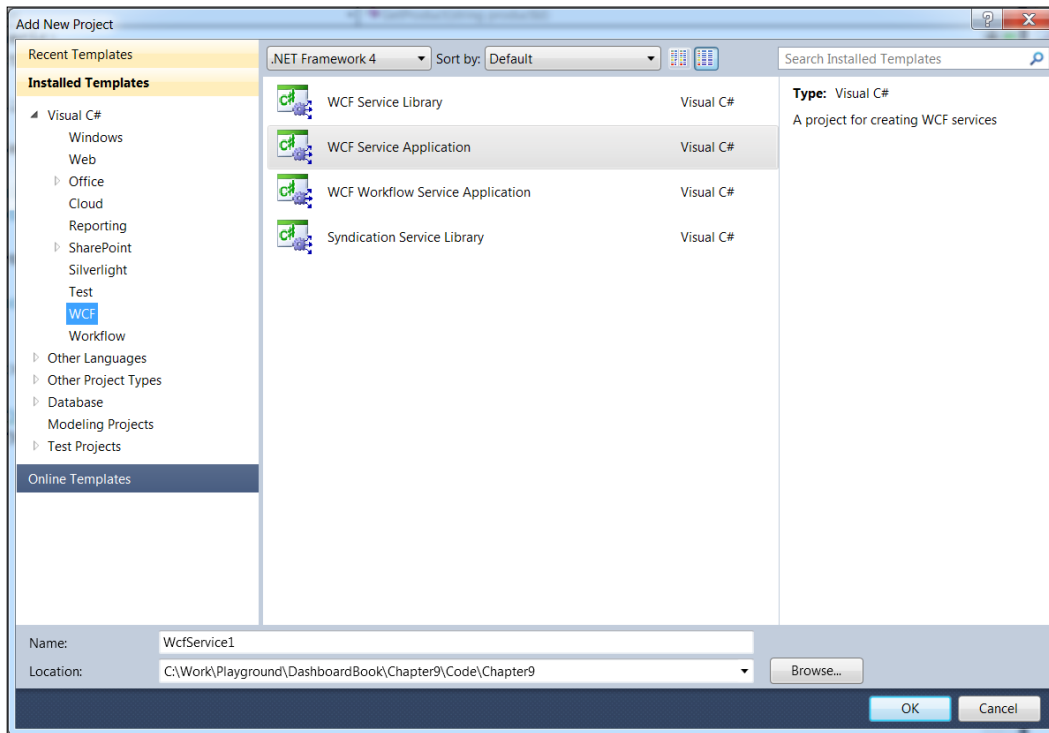


Now that we understand the different data access options and security restrictions for building a Silverlight consumable data service, let's put this knowledge to work. When building a WCF service, one of the most important technical design decisions you need to make is what protocol and data formats you want your service to support. Depending on your application needs, it might make sense to build a SOAP-based service that uses a binary format, or a REST-based service that returns data using the JSON format.

There is a lot of debate in the industry over which is better with regards to SOAP and REST. For now, we will skip the debate; both protocols have advantages and disadvantages, and ultimately it's up to you to decide which options work best for the type of application you need to build. If you're consuming an externally hosted or cloud-based service, more than likely it will be REST or OData-based service.

Working with WCF

Before we dig into the details of how to build a SOAP or REST-based WCF service, let's walk through the common data access classes our services will share. After we walk through building the service layer, we will dig into how to consume each type of service in our Silverlight application.



Instead of adding our WCF services directly to the ASP.NET application that hosts our Silverlight application, we can instead create a separate host application. Using Visual Studio, add a new WCF Service application. After Visual Studio has finished generating the WCF project, add a `ClientAccessPolicy.xml` file, and set it up to allow access to all domains. As we previously discussed, this file is required because our services will be hosted on a different URI than the Silverlight application XAP file.

If you haven't done so already, download and run the sample project for this chapter. The project includes the web client sample we previously talked about, along with the samples and WCF services we are about to build.

The data access layer

To keep things simple, our WCF services will be returning a collection of products stored in an XML data file (`Products.xml`). We will be using XLINQ to load the data stored in the XML file into a collection of products inside of the WCF service we will be creating. Following is the structure of the `Products.xml` file. Each product contains an ID, Name, Category, and so on.

```
<Products>
  <Product>
    <ProductID>401043204-423</ProductID>
    <ProductName>Tunnbröd</ProductName>
    <Supplier>PB Knäckebröd AB</Supplier>
    <Category>Grains/Cereals</Category>
    <QuantityPerUnit>12 - 250 g pkgs.</QuantityPerUnit>
    <UnitPrice>9.00</UnitPrice>
    <UnitsInStock>61</UnitsInStock>
    <UnitsOnOrder>10</UnitsOnOrder>
  </Product>
  <Product>
    <ProductID>534041202-345</ProductID>
    <ProductName>Guaraná Fantástica</ProductName>
    <Supplier>Refrescos Americanas LTDA</Supplier>
    <Category>Beverages</Category>
    <QuantityPerUnit>12 - 355 ml cans</QuantityPerUnit>
    <UnitPrice>4.50</UnitPrice>
    <UnitsInStock>20</UnitsInStock>
    <UnitsOnOrder>15</UnitsOnOrder>
  </Product>
  ...
</Products>
```

The product class looks very similar to the one we used previously. The only change is that the class and its member properties are decorated with the `DataContract` and `DataMember` attributes. These attributes are used by WCF to serialize the product class. If you omit the `DataMember` attribute, the property will not be serialized.

```
using System.Runtime.Serialization;
namespace Chapter9.Model
{
    [DataContract]
    public class Product
    {
        [DataMember]
```

```

public string ProductId { get; set; }
    [DataMember]
public string ProductName { get; set; }
    [DataMember]
public string Supplier { get; set; }
    [DataMember]
public string Category { get; set; }
    [DataMember]
public double UnitPrice { get; set; }
    [DataMember]
public int UnitsInStock { get; set; }
}

```

To load the data from the `Products.xml` file, we will be using the repository pattern. The `ProductRepository` class contains two methods. The `GetProduct` method will return a single product based on its ID, and the `GetProducts` method will return a collection of all the products defined in the `Products.xml` file. We will use XLINQ to load the data from `Products.xml` into a collection of products.



Note: The code uses the extension methods we previously defined. The `GetProducts` method calls the `GetProducts` method, and then uses LINQ to filter the returned collection, based on the product ID passed into the method.

```

using System.Collections.Generic;
using System.Linq;
using System.Web.Hosting;
using System.Xml.Linq;
using Chapter9.Common;
using Chapter9.Model;

namespace Chapter9.Service.Model
{
    public IList<Product> GetProducts()
    {
        return this.LoadProducts();
    }

    private IList<Product> LoadProducts()
    {
        string dataFilePath =
            HostingEnvironment.MapPath("~/App_Data/products.xml");
        XDocument document = XDocument.Load(dataFilePath);
    }
}

```

```
var result = from d in document.Root.Descendants("Product")
select new Product
{
    ProductId = d.Element("ProductID").Value,
    ProductName = d.Element("ProductName").Value,
    Supplier = d.Element("Supplier").Value,
    Category = d.Element("Category").Value,
    UnitPrice = d.Element("UnitPrice").GetDoubleValue(),
    UnitsInStock = d.Element("UnitsInStock").GetIntValue()
};

return result.ToList();
}

public Product GetProduct(string productId)
{
    IList<Product> products = this.LoadProducts();
    var result = products.Where(
        q => q.ProductId == productId).SingleOrDefault();
    return result;
}
}
```

To load the `Products.xml` file, we use the class `HostingEnvironment` to map the path to the XML file. Any type of .NET applications (WPF, ASP.Net, Windows Services, and so on) can be used to host a WCF service. So it's important that you develop the service in a host-independent way.

```
string dataFilePath =
    HostingEnvironment.MapPath("~/App_Data/products.xml");
XDocument document = XDocument.Load(dataFilePath);
```

Now that we have defined the data access code for our service, we will switch gears and take a look at how to build SOAP and REST-based services using WCF. Both of the services we are about to define will use the product repository we just created.

Building a SOAP service

Silverlight only supports a subset of the Windows Communication Foundation (WCF) features that are normally available with .NET. To create a Silverlight-enabled WCF service using Visual Studio, add a new item and select the correct template under the Silverlight node. Name the new service `ProductSoapService`. After the service is created, there are a few things that you should modify.

1. Create an interface for your service, and name it `IProductSoapService`, and add a service contract attribute to the class:

```
[ServiceContract(Namespace = «http://mydomain»)]
public interface IProductSoapService
```

2. Add the following using statements to the interface:

```
using System.Collections.Generic;
using System.ServiceModel;
using Chapter9.Model;
```

3. Now, define the contents of the interface to include a `GetProduct` and `GetProducts` method. Add an operation contract attribute to each method.

```
[OperationContract]
Product GetProduct(string productId);
```

```
[OperationContract]
IList<Product> GetProducts();
```

4. Update the `ProductSoapService` class so that it implements the interface you just created:

```
public class ProductSoapService : IProductSoapService
```

5. Make sure the soap service class has the `AspNetCompatibility` and that the `RequirementsMode` is set to `Allowed`:

```
[AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Allowed)]
```

6. Add the following using statements to the top of the class:

```
using System.Collections.Generic;
using System.ServiceModel.Activation;
using Chapter9.Model;
using Chapter9.Service.Model;
```

7. Update the `GetProducts` and `GetProduct` methods to call the product repository:

```
public Product GetProduct(string productId)
```



```
{  
    public IList<Product> GetProducts()  
    {  
        ProductRepository repository = new ProductRepository();  
        return repository.GetProducts();  
    }  
}
```

Before we try calling the service from Silverlight, make sure the `web.config` for the WCF service project has the necessary end-point configuration settings defined. You should make sure the end point is using the service interface `IProductSoapService`.

```
<system.serviceModel>  
  <behaviors>  
    <serviceBehaviors>  
      <behavior name=»»>  
        <serviceMetadata httpGetEnabled=»true» />  
        <serviceDebug include ExceptionDetailInFaults=»false» />  
      </behavior>  
    </serviceBehaviors>  
  </behaviors>  
  <bindings>  
    <customBinding>  
      <binding name=»Chapter9.Service.Services.Soop.  
        ProductSoapService.customBinding0»>  
        <binaryMessageEncoding />  
        <httpTransport />  
      </binding>  
    </customBinding>  
  </bindings>  
  <serviceHostingEnvironment aspNetCompatibilityEnabled=»false»  
    multipleSiteBindingsEnabled=»true» />  
  <services>  
    <service name=»Chapter9.Service.Services.Soop.  
      ProductSoapService»>  
      <endpoint address=»»binding=»customBinding  
        » bindingConfiguration=»Chapter9.Service.Services.Soop.  
        ProductSoapService.customBinding0»  
        contract=»Chapter9.Service.Services.Soop.  
        IProductSoapService» />  
      <endpoint address=»mex»binding=»mexHttpBinding»  
        contract=»IMetadataExchange» />  
    </service>  
  </services>  
</system.serviceModel>
```

8. In order to use the WCF SOAP service we have just created, you need to add a service reference to the service. To do this:
 - Right-click on your Silverlight project and select Add service reference.
 - Click the discover button to find the WCF services available in same solution as the Silverlight application.
 - The service wizard will create a set of proxy classes that can be used to call the service. Instead of using the service wizard, you can alternatively use the WCF client library directly, and manually create the proxy classes used to load the products returned from the service.

After the wizard has finished generating the proxy classes, you can use them to call the WCF service. You can do this by creating an instance of the `ProductSoapServiceClient` class, and subscribing to the completed and async methods. Each operation method defined in your WCF service contract will have a completed and async method generated. To call the service method, invoke its async method. When the call to service is done, the completed method will be called. In the completed method, you can check to see whether the call was a success. If it was successful, set the item source of the DataGrid to the returned product collection.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Chapter9.ProductSoapService;

namespace Chapter9.Views
{
    public partial class CallSoapService : Page
    {
        public CallSoapService()
        {
            InitializeComponent();

            this.Loaded += new RoutedEventHandler(CallSoapService_Loaded);
        }

        void CallSoapService_Loaded(object sender, RoutedEventArgs e)
        {
            ProductSoapServiceClient serviceClient = new
                ProductSoapServiceClient();
            serviceClient.GetProductsCompleted
                += new EventHandler<GetProductsCompletedEventArgs>
                    (serviceClient_GetProductsCompleted);
        }
    }
}
```

```
        serviceClient.GetProductsAsync();
    }
    void serviceClient_GetProductsCompleted(object sender,
        GetProductsCompletedEventArgs e)
    {
        if (e.Error == null)
        {
            this.ProductList.ItemsSource = e.Result;
        }
    }
}
}
```

Building a REST service

Starting with the release of the .NET Framework 3.5 (.NET 4.0 adds additional support), it is possible to easily build a REST-based service using WCF. Representational State Transfer (REST) is an architectural style, based on the concept of accessing resources using the HTTP protocol. Each resource is uniquely identified using a URI (`http://.../products`), and can be accessed using one of the HTTP verbs (GET, POST, PUT, or DELETE).



To learn more about REST and JSON see the following links:

REST: <http://msdn.microsoft.com/en-us/netframework/dd547388>

JSON: <http://msdn.microsoft.com/en-us/library/bb299886.aspx>

Let's walk through building a WCF REST (HTTP)-based data service and consuming it in our Silverlight application. We will use the same WCF service application we have previously created.

1. Add a new item to the service project by selecting the WCF service template. Name the new service `ProductRestService`.
2. Add an interface to the project named `IProductService`, add the Service Contract attribute above the interface.

```
[ServiceContract]
public interface IProductRestService
```

3. Add the following using statements:

```
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Web;
using Chapter9.Model;
```

4. Add the `GetProduct` and `GetProducts` methods to the interface. Add the operation contract and web get attributes to each method. Make sure you see the `Name` parameter for the operation contract. Additionally, define the `UriTemplate` and response format for each method.

```
[OperationContract (Name = "GetProducts")]
[WebGet (UriTemplate = «/», ResponseFormat =
    WebMessageFormat.Json)]
IList<Product> GetProducts ();

[OperationContract (Name = «GetProduct»)]
[WebGet (UriTemplate = «/product/{productId}»,
    ResponseFormat = WebMessageFormat.Json)]
Product GetProduct (string productId);
```

5. Update the `Product Rest Service` so that it implements the `IProductService` interface.

```
public class ProductRestService : IProductRestService
```

6. Add the following using statements to the class:

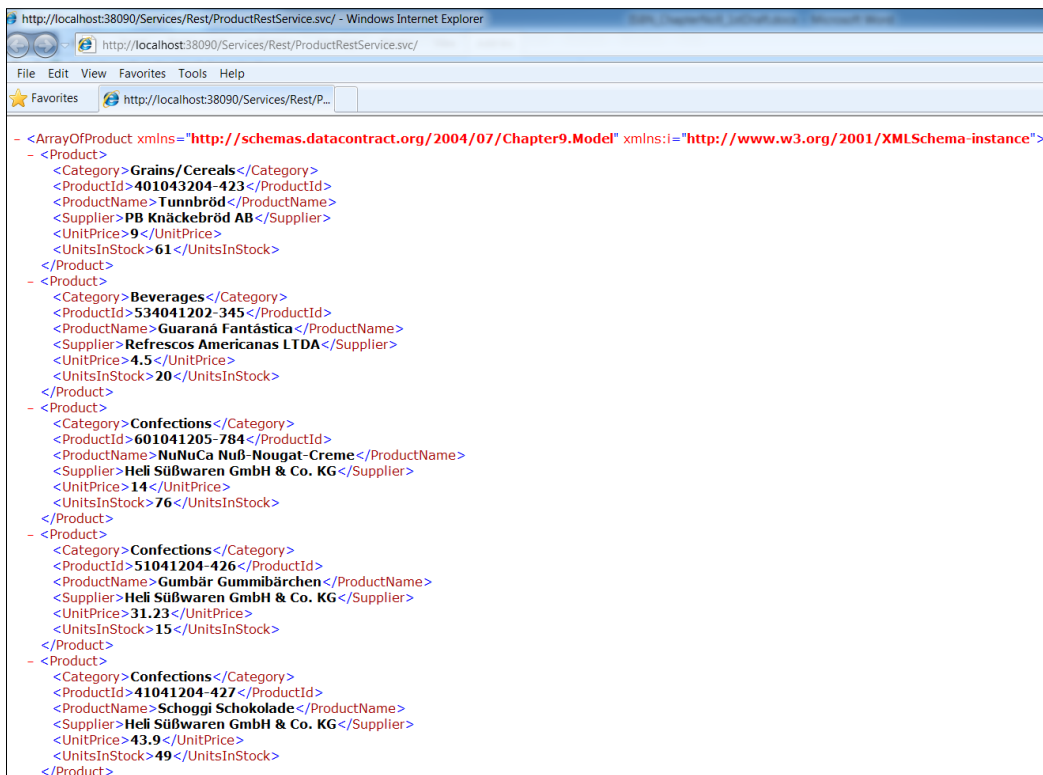
```
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Web
```

7. Update the `GetProducts` and `GetProduct` methods to call the product repository:

```
public Product GetProduct (string productId)
{
    public IList<Product> GetProducts ()
    {
        ProductRepository repository = new ProductRepository ();
        return repository.GetProducts ();
    }
}
```

8. The service class for our REST service looks very similar to its corresponding SOAP counterpart. The only major difference is that we do not need to add the `AspNetCompatibilityRequirements` attribute, because we will be using the `WebClient` class to call the REST service.

9. There are a couple other things we need to define for our REST service to work. Switch to the markup for the SVC file, and add the following:
`Factory="System.ServiceModel.Activation.WebServiceHostFactory"`
10. This defines the factory that WCF will use to process calls to your REST service. Change the response format for both `GetProduct` and `GetProducts` to XML. Then load your very favorite web browser, navigate to the following URI to display the default (/) URI for the REST service:
`http://localhost:38090/Services/Rest/ProductRestService.svc/`
11. The Port # for the service maybe different, because Visual Studio picks a dynamic port to load the WCF service project. Using the default URI (/) for the service, you should see a list of products returned from the service. If we change the URI to `product/534041202-345`, we will see a single product displayed.



12. After testing our REST service using a web browser, make sure you change the format to JSON in the service interface. Our Silverlight application will use the JSON format to deserialize the data returned from the service.
13. To call our REST service, we will use the `WebClient` class by passing in the URI for the service. When the service call is completed, we use the `DataContractJsonSerializer` class to deserialize the JSON data returned from the class into a collection of products.

```

void CallRestService_Loaded(object sender,
    System.Windows.RoutedEventArgs e)
{
    Uri serviceUri = new Uri(«http://localhost:38090/Services/Rest/
        ProductRestService.svc/»);

    WebClient client = new WebClient();
    client.OpenReadCompleted += new OpenReadCompletedEventHandler
        (client_OpenReadCompleted);
    client.OpenReadAsync(serviceUri);
}

void client_OpenReadCompleted(object sender,
    OpenReadCompletedEventArgs e)
{
    DataContractJsonSerializerserializer =
        newDataContractJsonSerializer(typeof(ICollection<Product>));
    ICollection<Product> data = (ICollection<Product>)serializer.ReadObject
        (e.Result);
    this.ProductList.ItemsSource = data;
}

```

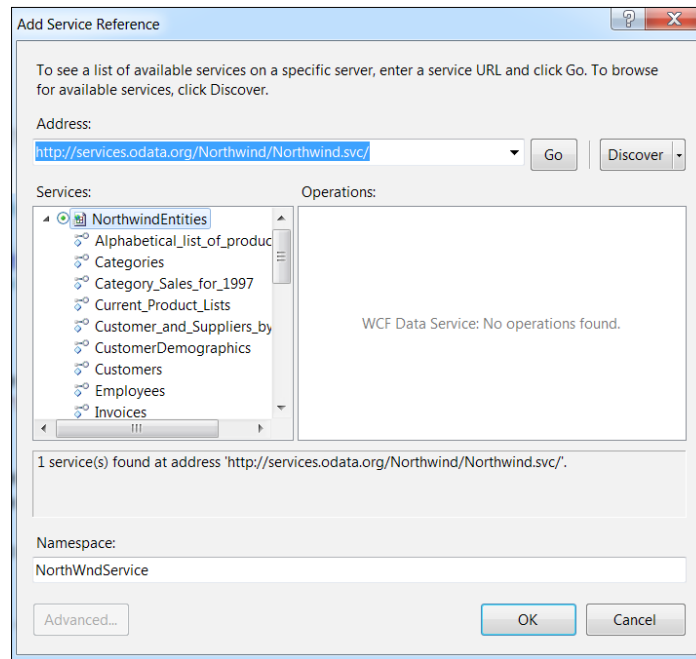
Exploring OData data services

The Open Data Protocol (OData) is a protocol based on the principles of REST for building queryable data services. It is built on the concepts behind the Atom publishing Protocol (AtomPub) and the Java Script Object Notation (JSON) data format. It provides a standard way for data consumer and producers to interact with each other.

The core data type in OData is a feed, which is defined as a collection of entry types. An entry is a structured record that includes a key and a list of properties. Properties can be defined as simple (e.g., string) or complex (e.g., address) types. An entry can have related entries named links and be part of a hierarchy. For example to access a feed of customer entries, we would use the following URI (`http://...//OData.svc/Customers`).

A number of OData-supported client libraries are available that make it easier to call OData producers. In Silverlight, you can use the WCF Data Service client library to query an OData source using LINQ. The Language Integrated Query (LINQ) language is a set of extensions introduced with .NET 3.5 that allows you to query a data source easily. LINQ can be used to query any data source that implements the IQueryable interface such as IEnumerable Collections, WCF Data Services, Entity Framework, and so on.

To use the WCF Data Service client library, add a service reference using the **Visual Studio | Add Service Reference** wizard. This wizard will generate the service context and proxy classes you will use in your Silverlight application as in the following image:



For a more in-depth overview of OData AND a list of consumers and producers, see the OData portal at <http://www.odata.org>.

Building an OData service

We will be using the WCF Data Service framework to host our custom OData service. WCF Data Services is a framework that allows you to publish an OData-based service easily by defining an Entity Data Model (EDM). You can easily define an EDM by using the Entity Framework designer in Visual Studio or using attributes to define a custom EDM model class.

To create a custom entity model class, all we need to do is decorate a class with the following properties: `EntityTypeMapping` and `DataServiceKey`. The `EntityTypeMapping` attribute is used to map a property of an entity to the metadata element for the feed.

```

DataServiceKey is used for defining the key used for identifying an
entity.using System.Data.Services.Common;

namespace Chapter9.Service.Model
{
    [EntityTypeMapping("ProductName",
        SyndicationItemProperty.Title,
        SyndicationTextContentKind.Plaintext, true)]
    [DataServiceKey("ProductId")]
    public class ProductRecord
    {
        public string ProductId { get; set; }
        public string ProductName { get; set; }
        public string Supplier { get; set; }
        public string Category { get; set; }
        public double UnitPrice { get; set; }
        public int UnitsInStock { get; set; }
    }
}

```

The product record repository class contains the `IQueryable` data source that will be hosted by WCF Data Service. The class uses `XLINQ` to load a collection of product records. The `Products` property returns the loaded product collection as an `IQueryable` data source as follows:

```

using System.Collections.Generic;
using System.Linq;
using System.Web.Hosting;
using System.Xml.Linq;
using Chapter9.Common;

namespace Chapter9.Service.Model
{

```



```
public class ProductRecordRepository
{
    public ProductRecordRepository()
    {
        this.LoadProducts();
    }

    public IQueryable<ProductRecord> Products
    {
        get
        {
            return productRecordList.AsQueryable();
        }
    }

    private List<ProductRecord> productRecordList;
    private void LoadProducts()
    {
        string dataFilePath = HostingEnvironment.MapPath
            ("~/App_Data/products.xml");
        XmlDocument document = XmlDocument.Load(dataFilePath);
        var result = from d in document.Root.Descendants("Product")
        select new ProductRecord
        {
            ProductId = d.Element("ProductID").Value,
            ProductName = d.Element("ProductName").Value,
            Supplier = d.Element("Supplier").Value,
            Category = d.Element("Category").Value,
            UnitPrice = d.Element("UnitPrice").GetDoubleValue(),
            UnitsInStock = d.Element("UnitsInStock").GetIntValue()
        };
        productRecordList = result.ToList();
    }
}
```

To host our custom IQueryable data source using WCF Data Services, we need to create a class that inherits from the `DataService<T>` base class, where the generic type `<T>` contains one or more IQueryable properties. The `InitializeService` method is used for defining the service's behaviors and the security (read/write) access for the entries returned from the service.

```
using System.Data.Services;
using System.Data.Services.Common;
using Chapter9.Service.Model;

namespace Chapter9.Service.Services.OData
{
    public class ProductODataService :
        DataService<ProductRecordRepository>
    {
        public static void InitializeService
            (DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("*", EntitySetRights.AllRead);
            config.SetServiceOperationAccessRule
                ("*", ServiceOperationRights.All);
            config.MaxResultsPerCollection = 100;
            config.DataServiceBehavior.MaxProtocolVersion =
                DataServiceProtocolVersion.V2;
        }
    }
}
```

In our Silverlight application, we can consume the OData service we just built by adding a service reference to the hosted WCF Data Service. To call the service, we need to pass the URI for the service to the service data context, and then asynchronously invoke the data service query by calling its `begin execute` method. When the async call completes, we can get the results returned from the service call, and then bind it to a grid defined in our view. You will need to make sure that you set the grid's item source on the UI thread by invoking the UI dispatcher as follows:

```
Uri dataSourceUri = new Uri("http://localhost:38090/Services/OData/
    ProductODataService.svc/");
ProductDataSource context = new ProductDataSource(dataSourceUri);
IEnumerable<ProductRecord> results;
var query = (from p in context.Products
    where p.UnitPrice > 25
    select p);
DataServiceQuery<ProductRecord> dataServiceQuery =
    query as DataServiceQuery<ProductRecord>;
```

```
dataServiceQuery.BeginExecute((asyncResult) =>
{
    results = dataServiceQuery.EndExecute(asyncResult);
    Dispatcher.BeginInvoke(() =>
    {
        this.ProductList.ItemsSource = results.ToList();
    });
}, null);
```

We will need to use LINQ to define the query we will be sending to the OData service. When the service gets invoked, the LINQ query is converted into the correct URI for calling the service.

```
var query = (from p in context.Products
             where p.UnitPrice > 25 select p);
```

The LINQ query above will be converted into the following URI: .../
ProductDataService.svc/Products()?\$filter=UnitPrice%20gt%2025.0.
The query will only return products that have a unit price that is greater than 25.

Consuming an external service

Throughout this chapter, we have looked at how to build and consume our own custom services. While in most cases, you will build your own service, there are times that you will need to consume one or more external services hosted by a third-party vendor or deploy in the cloud-based platform.

To consume an external service, we follow the same process we have been using to call our own custom services. The only extra step is to make sure the external service we are calling has a client policy file deployed that allows access for all domains or allows access to the host domain for our Silverlight applications.

In the following sample, we will be calling the NorthWind OData service hosted by the OData portal (<http://www.odata.org>). The NorthWind service is hosted at the following URI: <http://services.odata.org/Northwind/Northwind.svc>.

Add a service reference to the externally hosted ODataService. The service wizard will generate the `NorthWindEntities` class and the service proxy for the NorthWind service. Using the `NorthWindEntities` class, set up a LINQ query to get a list of customers who have a title of owner. To call the service invoked, begin the execute method of the data service class. When the call to the external service is complete, the end execute method will be called to return a collection of customers. Using the UI dispatcher, return the execution to the UI thread, and set the item source of the CustomerList data grid.

As we did in the previous OData sample, we will need to use the WCF Data Service client and LINQ to query the NorthWndService.

```
void NorthWndOData_Loaded(object sender, RoutedEventArgs e)
{
    Uri dataSourceUri = new Uri("http://services.odata.org/Northwind/
    Northwind.svc/");
    NorthwindEntities context = new NorthwindEntities(dataSourceUri);
    IEnumerable<Customer> results;
    var query = (from c in context.Customers
        where c.ContactTitle == "Owner"
        select c);
    DataServiceQuery<Customer> dataServiceQuery = query as
    DataServiceQuery<Customer>;
    dataServiceQuery.BeginExecute((AsyncResult) =>
    {
        results = dataServiceQuery.EndExecute(AsyncResult);
        Dispatcher.BeginInvoke(() =>
        {
            this.CustomerList.ItemsSource = results.ToList();
        });
    }, null);
}
```

The LINQ query returns a list of customers from the NorthWnd data service that has a contact title of Owner. When the query is invoked, the following URI is sent to the externally hosted service: `http://services.odata.org/Northwind/Northwind.svc/Customers()?$filter=ContactTitle%20eq%20'Owner'`.

One of the powerful attributes of the REST and OData protocol is that it is possible to test the service call by just using a web browser. If you want to directly test out your own custom service or external service, you can use your favorite web browser to test out the service's URI configuration.

Summary

In this chapter, we discussed the different data access strategies we could use for building a Silverlight application. We looked at how to use the basic HTTP classes included in the Silverlight runtime. Additionally, we discussed the advanced techniques for building and consuming SOAP, REST, or OData-based services, and how the Silverlight security policy system works for allowing cross-domain calls to external hosted resources.

In the next chapter, we will look into how to build Silverlight dashboards hosted in SharePoint. We will examine how to create a Silverlight web part, and use the SharePoint client library to access SharePoint list data, and display it in Silverlight.

10

Building Dashboards in SharePoint and Silverlight



This chapter is taken from *Microsoft Silverlight 4: Building Rich Enterprise Dashboards* (Chapter 10) by Todd Snyder, Joel Eden, Ph.D., Jeff Smith, Matthew Duffield.

This chapter will introduce you to the features included in SharePoint 2010 for hosting Silverlight dashboard applications. We will explore how to set up a Silverlight web part, and use the SharePoint Silverlight Client Object Model to communicate with data hosted in SharePoint.

In order to run the samples for this chapter, you will need to have a SharePoint 2010 development environment set up. If you don't have one available, follow the steps below for setting up SharePoint 2010 on Windows 7. If you are not familiar with SharePoint 2010, spend some time getting up to speed before tackling the samples in this chapter.

In this chapter, we will cover the following topics:

- Overview of SharePoint
- Building a SharePoint Silverlight Dashboard
- SharePoint Data Access Strategies

Overview of SharePoint

Microsoft SharePoint 2010 is a portal platform that includes features for building content, collaboration, document management, and business intelligence applications. Prior to SharePoint 2010, it was possible to host a Silverlight application by manually creating a SharePoint hosted page or web part that included the HTML object tag. SharePoint 2010 introduces a Silverlight web part and Client Object Model, which makes it easier for Silverlight applications to access data hosted in SharePoint.

Setting up SharePoint

Before we can walk through how to host a Silverlight dashboard application in SharePoint and use the Client Object Model, we need to set up a version of SharePoint 2010 to use. There are two versions of SharePoint 2010: SharePoint 2010 Foundations and SharePoint Server 2010. Since our main focus is going to be on how to use the Client Object model, we will be setting up SharePoint Foundation 2010 on Windows 7.

If you need to have all the features enabled for SharePoint, or you are building a production machine, you will need to install it on a Windows 2008 64-bit machine. Even for Windows 7, you will need to have the 64-bit version installed.

In order to set up SharePoint to run on Windows 7, you will need to do the following:

1. Download SharePoint 2010 Foundation from the following:
`http://www.microsoft.com/downloads/en/details.aspx?FamilyID=49c79a8a-4612-4e7d-a0b4-3bb429b46595&displaylang=en`
2. After the file is complete, follow the given steps to install it on Windows 7:
 - Copy the downloaded file to a directory on your machine e.g., `c:\SharePointInstall`.
 - Open a command prompt, and run the following: `c:\SharePointInstall\SharePoint /extract:c:\SharePointInstall`; this will extract the installation files from the download. If you download the SharePoint Server 2010 edition, run this command line instead: `c:\SharePointInstall\Office Server /extract:c:\SharePointInstall`.

- Using a text editor (e.g., Notepad), open the installation configuration file `config.xml` located under `c:\SharePointInstallation\files\Setup\config.xml`, and add the following line to the configuration section: `<Setting Id="AllowWindowsClientInstall" Value="True"/>`.
- Run and install the prerequisite for SharePoint 2010 Foundation.
- At the Command prompt, type the following `c:\SharePointInstall\PrerequisiteInstallerFiles\FilterPack\FilterPack.msi` to install the Microsoft FilterPack 2.0.
- You will need to install the following items also: Microsoft Sync Framework, SQL Server NativeClient, and Windows Identify PrerequisiteInstallerFiles\FilterPack\FilterPack.msi.
- Open a command prompt, and run the following script. Note: all the parameters should be on the same line. You can also manually enable these features by running the Turn Windows Features On and Off under **Control Panel\Program and Features**.

The following script will enable the necessary Windows and IIS features required for running SharePoint 2010. It's important that this script is run successfully. If you run into issues, SharePoint may not install or run correctly. If you download the source code for this chapter, you will find a `SharePointSetup.txt` file that contains the script below you need to run to set up IIS to properly run SharePoint.

```
start /w pkgmgr /iu:IIS-WebServerRole;IIS-WebServer;
IIS-CommonHttpFeatures;
IIS-StaticContent;IIS-DefaultDocument;IIS-DirectoryBrowsing;
IIS-HttpErrors;
IIS-ApplicationDevelopment;IIS-ASPNET;IIS-NetFxExtensibility;
IIS-ISAPIExtensions;IIS-ISAPIFilter;IIS-HealthAndDiagnostics;
IIS-HttpLogging;IIS-LoggingLibraries;IIS-RequestMonitor;
IIS-HttpTracing;IIS-CustomLogging;IIS-ManagementScriptingTools;
IIS-Security;IIS-BasicAuthentication;
IIS-WindowsAuthentication;IIS-DigestAuthentication;
IIS-RequestFiltering;IIS-Performance;
IIS-HttpCompressionStatic;IIS-HttpCompressionDynamic;
IIS-WebServerManagementTools;IIS-ManagementConsole;
IIS-IIS6ManagementCompatibility;
IIS-Metabase;IIS-WMICompatibility;
WAS-WindowsActivationService;WAS-ProcessModel;
WAS-NetFxEnvironment;WAS-ConfigurationAPI;WCF-HTTP-Activation;
WCF-NonHTTP-Activation
```


3. Run the SharePoint installation (C:\SharePointInstall\setup.exe), and select the standalone installation.
4. When the installation is finished, make sure you install the SQL Server 2008 KB 970315 x64 hotfix: <http://support.microsoft.com/kb/970315>.
5. When the hotfix is being installed, run the SharePoint 2010 Configuration Wizard. After the wizard is complete, you should be able to load the default SharePoint site.

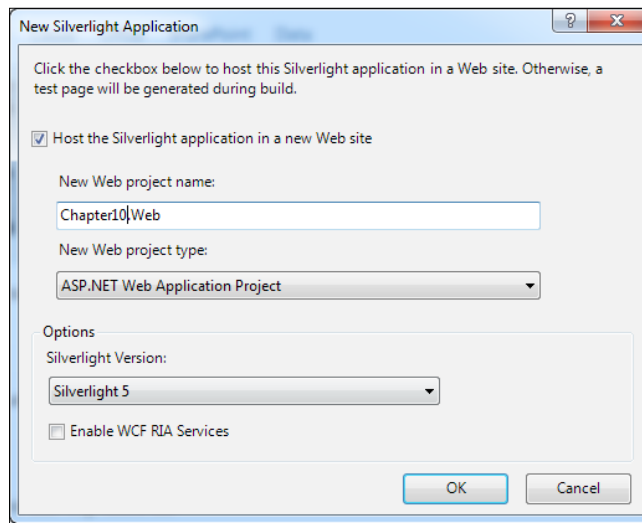
If you experience any issues when creating the SharePoint 2010 configuration database, you may need to delete the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Shared Tools\Web Server Extensions\14.0\Secure\FarmAdmin` registry key and rerun the SharePoint configuration wizard. Additional details about setting up SharePoint 2010 on Windows 7 can be found here: [http://msdn.microsoft.com/en-us/library/ee554869\(offic.14\).aspx](http://msdn.microsoft.com/en-us/library/ee554869(offic.14).aspx)

Building a Silverlight web part

Now that our SharePoint development environment is set up and ready to go, let's build a simple "hello world" style Silverlight web part. Later on in the chapter, we will talk about how to create a Silverlight dashboard and connect it to SharePoint.

To begin building our Silverlight web part, you will need to complete the following steps:

1. Open Visual Studio 2010, and create a new Silverlight Navigation Application. When you are prompted to create a website, uncheck the **Host the Silverlight application in a new Web site** check box. We do not need a hosting application, because we will be using SharePoint to host the application.

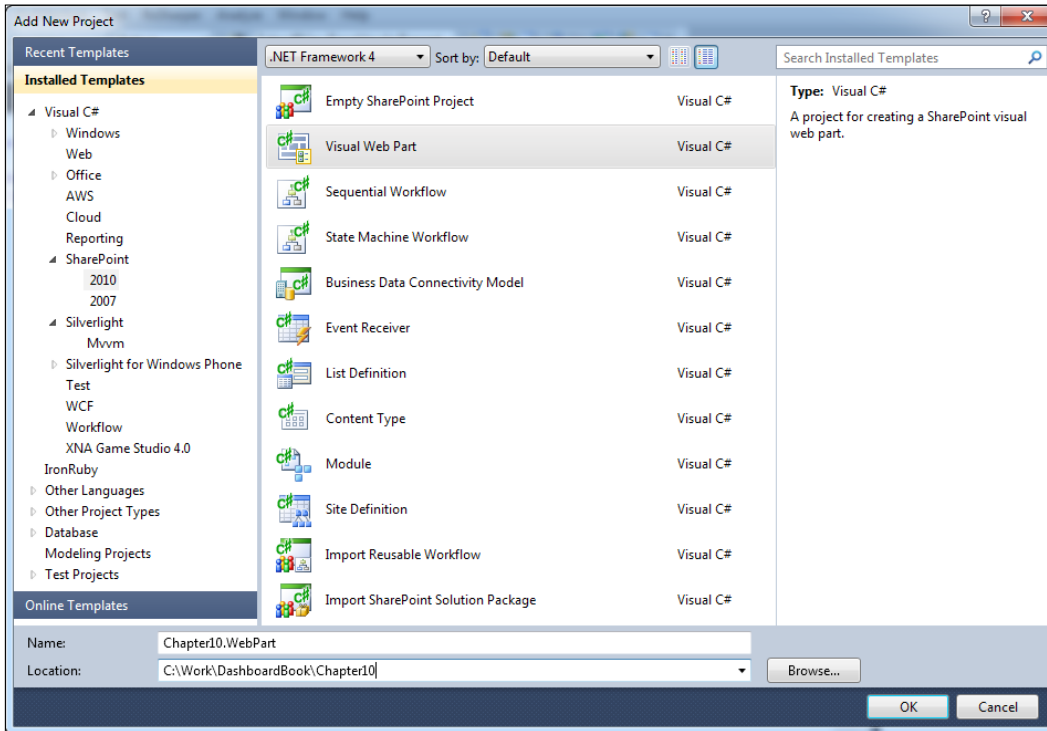


2. Open the `view\home.xaml` in the Silverlight application, and add a text block that says "Hello SharePoint".

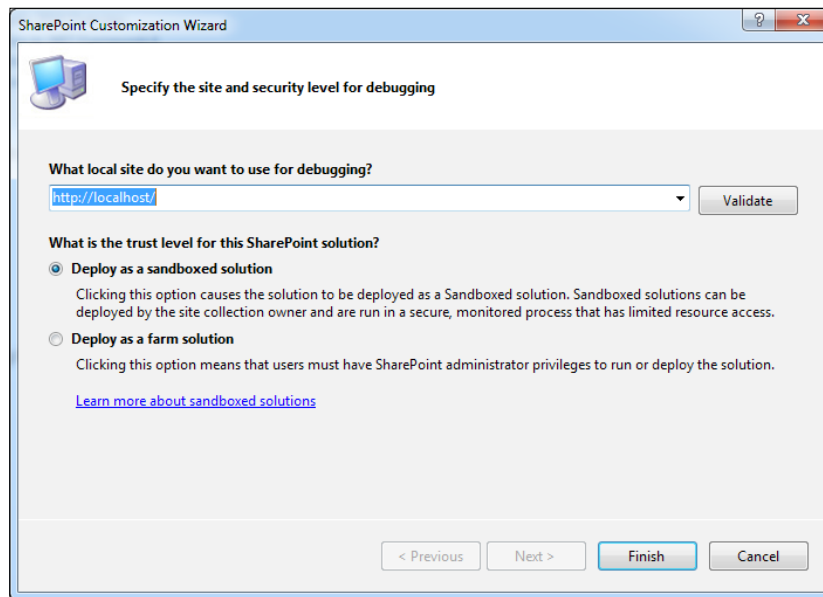
```
<Grid x:Name="LayoutRoot" >  
    <TextBlock Text="Hello SharePoint" >  
        Margin="10, 10, 0, 0" />  
</Grid>
```

3. To use the Silverlight application you just created in SharePoint, you need to create a SharePoint project using Visual Studio 2010. Make sure you are running Visual Studio under elevated Administrator permissions. This is necessary so we can deploy and debug the application under SharePoint.
4. To load Visual Studio as an administrator, save your work, and close Visual Studio, then right click on the Visual Studio 2010 shortcut and select **Run as Administrator**.

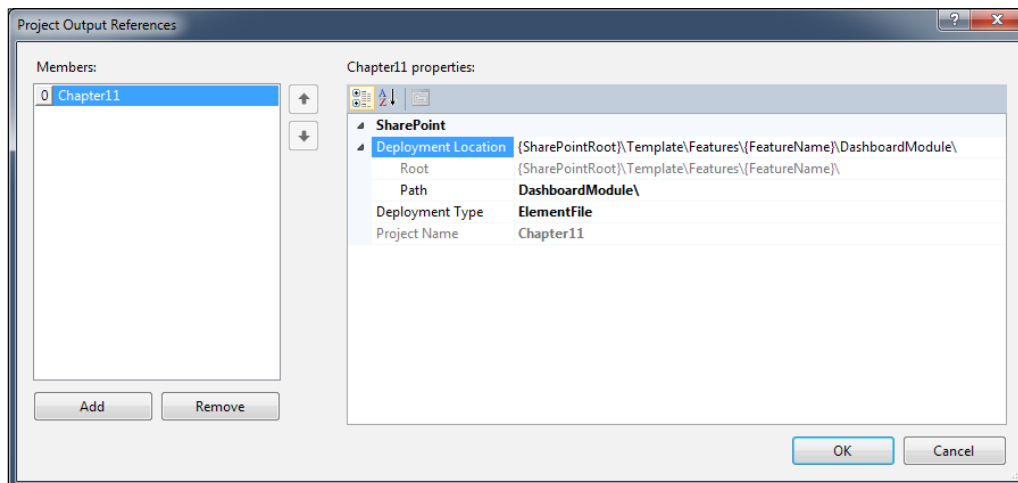
5. Back in Visual Studio, reload your Silverlight project, and add a new SharePoint 2010 (select **Empty SharePoint Project**) Project as follows:



6. When the SharePoint customization wizard is displayed, select **Deploy as a sandboxed solution**.



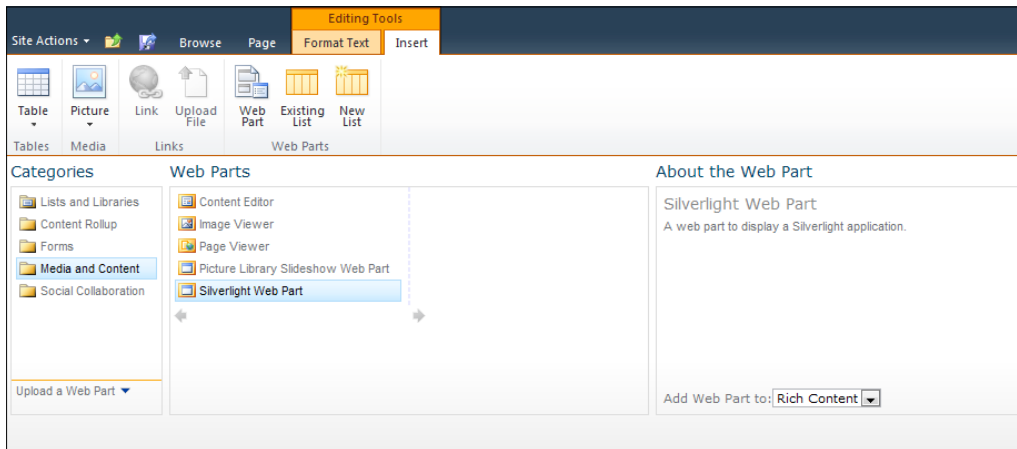
7. After the wizard is complete, right-click on the SharePoint project in the solution explorer, and add a new SharePoint module item.
8. Right-click on the newly added module file and select properties. When the property window is displayed, click on the ellipse button (...) in the **Project Output Reference** property.



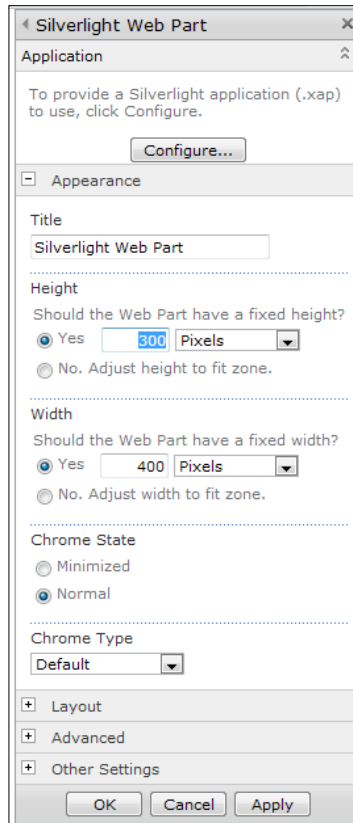
9. Click the **Add** button to add a new entry. Expand the deployment location property. Now change the **Deployment Type** to ElementFile, and set the **Product Name** to the name of your Silverlight project.
10. Expand the module element and delete the `Sample.txt` file. Open the `Elements.xml` file, and remove the file listing for `Sample.txt`.
11. Add a new entry for your Silverlight application XAP file. The XAP will be deployed to a document library named Silverlight. Either create the document library before running this solution, or change the deployed path to an existing document library.

```
<?xmlversion="1.0" encoding="utf-8"?>
<Elementsxmlns="http://schemas.microsoft.com/sharepoint/>>
  <ModuleName=>DashboardModule<>
    <FilePath=>DashboardModule\Chapter10.xap<>
      Url=>Silverlight/Chapter10.xap<> />
    </Module>
  </Elements>
```

12. At this point, our Silverlight application is ready to be deployed. Right-click on the project, and set it as the **Startup Project**, and then right click and select **Properties**. When the property dialog appears, click on the **SharePoint** tab. Make sure the **Enabled Silverlight debugging** (instead of **Script debugging**) check box is checked. This will allow you to debug the Silverlight application running under SharePoint. Run the application by pressing *F5*.
13. Now that your Silverlight application has been deployed to SharePoint, let's add it to the default SharePoint page by adding a Silverlight web part (located under the Media and Content Category), and set the path to the URI for your application XAP file for example, `Silverlight/Chapter10.xap`.



- When you display your Silverlight application for the first time, it will appear squished. To modify the width and height of the web part, click on edit web part, and modify the width to be 400.



Using the Client Object Model

To fully take advantage of hosting your Silverlight application in SharePoint, you need to use the SharePoint Client Object Model. The Client Object Model offers a rich API for interacting with SharePoint and content (data) it is hosting. Under the covers, the Client Object Model calls the SharePoint Window Communication Foundation (WCF) services. It is possible to call these WCF services directly, but the Client Object Model offers you a much easier way to interact with SharePoint.

To use the Client Object Model, you need to reference the Silverlight Client Object Model assemblies (`Microsoft.SharePoint.Client.Silverlight.dll` and `Microsoft.SharePoint.Client.Silverlight.Runtime.dll`). These assemblies can be found in the `%ProgramFiles%\Common Files\Microsoft Shared\web server extensions\14\TEMPLATE\LAYOUTS\ClientBin` folder on your SharePoint development machine.

To use the SharePoint Client Object Model, you will need to get the current `SharePoint.ClientContext.Current` context. Using this context, you can access the website, and list data in SharePoint. The following sample gets the names of all the lists in SharePoint. The code uses the client context to query SharePoint. A LINQ expression is defined that includes the list title and its fields. SharePoint will only return the fields defined in the query. When the query is complete, it will load the `listInfo` collection we defined; this is the collection we will use later to access the returned data from SharePoint.

```
private IEnumerable<List>listInfo;
void SharePointLists_Loaded(object sender, RoutedEventArgs e)
{
    ClientContext clientContext = ClientContext.Current;
    ListCollection collList = clientContext.Web.Lists;
    listInfo = clientContext.LoadQuery(
        collList.Include(
            list =>list.Title,
            list =>list.Fields.Include(
                field =>field.Title).Where(
                    field =>field.Required == true && field.Hidden != true));
    clientContext.ExecuteQueryAsync(onQuerySucceeded, onQueryFailed);
}
```

The `onQuerySucceeded` method will be called when the query operation has completed.

If an error occurs while the query is executing, the `onQueryFailed` method will be called. If the query is successful, we need to use the UI dispatcher so that the result is marshalled to the UI thread. In the `DisplayInfo` method, we loop through the `listInfo` collection we set up previously and create a collection of `spLists` that will bind to the data grid.

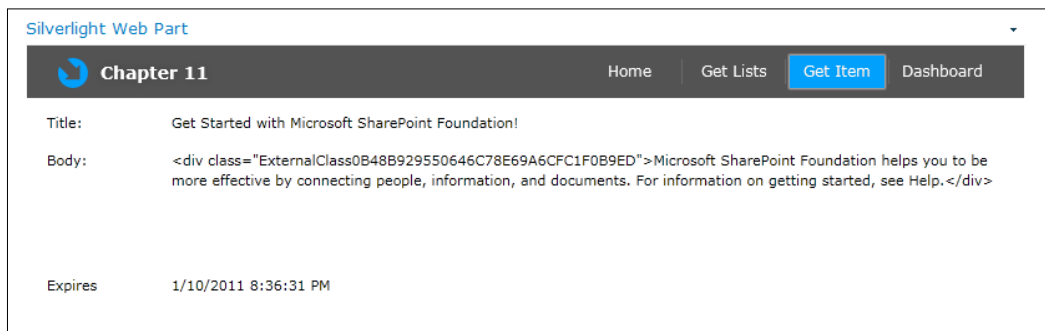
```
private void onQuerySucceeded(object sender,
    ClientRequestSucceededEventArgs args)
{
    this.Dispatcher.BeginInvoke(DisplayInfo);
}
```

```

private void DisplayInfo()
{
    IList<spList>listData = new List<spList>();
    foreach (List oList in listInfo)
    {
        listData.Add(new spList { Title = oList.Title });
    }
    this.ListGrid.ItemsSource = listData;
}
private void onQueryFailed(object sender,
    ClientRequestFailedEventArgs args)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        MessageBox.Show("Request failed. " + args.Message + "\n" +
            args.StackTrace);
    });
}
}

```

In the next sample, we will use the client object model to retrieve the announcement list from SharePoint. The following screenshot shows the retrieved announcement displayed in a Silverlight web part:



The following code retrieves a SharePoint list using its title. The `ClientContext.Current` method uses the current user credentials to log into SharePoint and gain access to the host SharePoint website the Silverlight web part is currently running under.

```

ClientContext clientContext = ClientContext.Current;
oWebsite = clientContext.Web;
this.oList = oWebsite.Lists.GetByTitle("Announcements");

```


To get the items from a list, we need to set up a CAML Query and define the fields to return from the list to get the field using the following pattern item `{{FieldName}}`. CAML is an XML query language used by SharePoint to filter data returned from a list. The following CAML query has a row limit set to 100 to restrict the query to only return 100 rows of data:

```
CamlQuery camlQuery = new CamlQuery();
camlQuery.ViewXml = "<View><RowLimit>100</RowLimit></View>";
this.collListItem = oList.GetItems(camlQuery);
clientContext.Load(collListItem,
items =>items.Include(
    item =>item.Id,
    item =>item.DisplayName,
    item => item["Title"],
    item => item["Body"],
    item => item["Expires"],
    item =>item.HasUniqueRoleAssignments));
clientContext.ExecuteQueryAsync(onQuerySucceeded, onQueryFailed);
```

When the query is returned from executing, we load the data returned in a class named `spAnnouncement` by looping through the collection loaded by the CAML query. We then notify the view that the announcement is ready to be displayed.

```
public class spAnnouncement
{
    public class spAnnouncement
    {
        public string Title { get; set; }
        public string Body { get; set; }
        public string Expires { get; set; }
    }

    spAnnouncement announcement = new spAnnouncement();
    foreach (ListItem item in this.collListItem)
    {
        announcement.Title = item["Title"].ToString();
        announcement.Body = item["Body"].ToString();
        announcement.Expires = item["Expires"].ToString();
    }
}
```

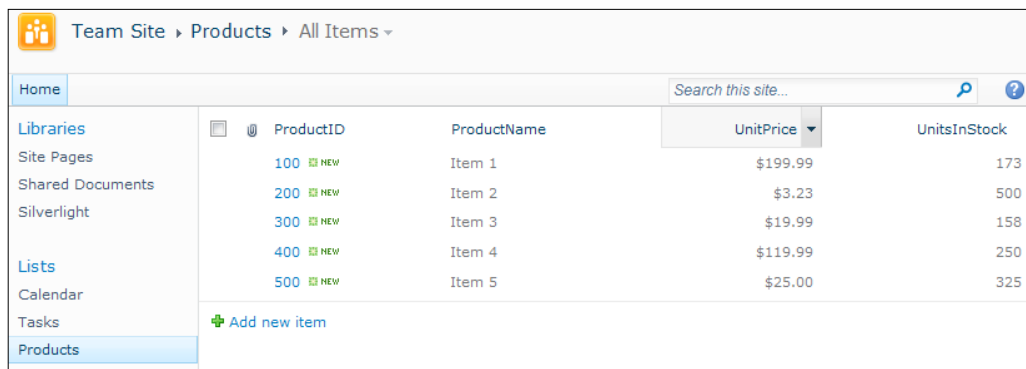
Building a SharePoint Silverlight dashboard

Now that we have a good understanding of what the SharePoint platform offers and its support for hosting Silverlight applications, let's walk through building a Silverlight dashboard application that we will host in a SharePoint 2010 website.

Setting up our data source

To keep things simple, we will be using a SharePoint list as our sample's data source. In most real world scenarios, you would want to use the Business Data Catalog or deploy your own custom WCF services to the SharePoint server, and use SQL Server 2008 or another RDBMS to host your dashboard's data.

To create the data source for the dashboard, add a new list to SharePoint named products. After the new list is created, rename the title field to **Product Id**, you will still need to use the title field to access the product ID data. Add a **ProductName** (string), **UnitPrice**(Currency), and **UnitsInStock** (Numeric) to the products list. Add one or more rows of data to the list using the SharePoint UI.

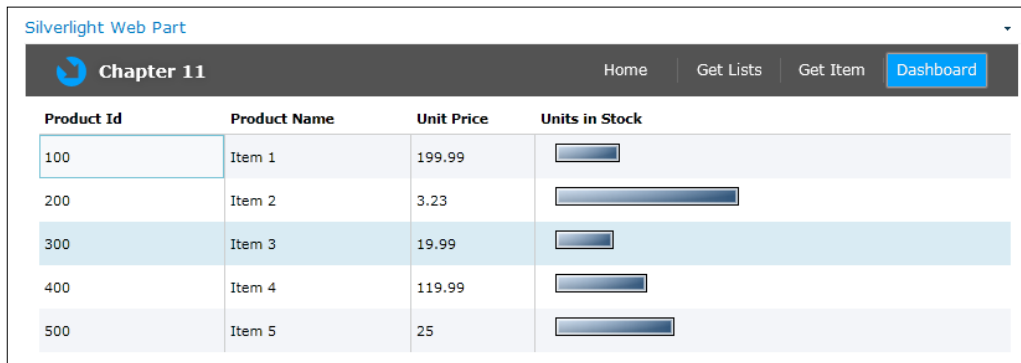


The screenshot shows a SharePoint 2010 interface for a list named 'Products'. The breadcrumb path is 'Team Site > Products > All Items'. The list view includes a search bar and a navigation pane on the left with categories like Libraries, Lists, and Tasks. The main content area displays a table with five columns: ProductID, ProductName, UnitPrice, and UnitsInStock. Each row represents a product item with a 'NEW' status indicator. An 'Add new item' link is visible at the bottom of the list.

ProductID	ProductName	UnitPrice	UnitsInStock
100	Item 1	\$199.99	173
200	Item 2	\$3.23	500
300	Item 3	\$19.99	158
400	Item 4	\$119.99	250
500	Item 5	\$25.00	325

Building our dashboard

We will be building a simple dashboard that displays the data from the SharePoint list as read-only text. The number of units available will be displayed using a bar chart. This will enable the end user to see quickly what products have the highest and lowest number of items in stock.



The screenshot shows a Silverlight Web Part titled "Chapter 11" with a navigation menu containing "Home", "Get Lists", "Get Item", and "Dashboard". Below the menu is a table with four columns: "Product Id", "Product Name", "Unit Price", and "Units in Stock". The table contains five rows of data, each with a corresponding horizontal bar chart representing the "Units in Stock" value.

Product Id	Product Name	Unit Price	Units in Stock
100	Item 1	199.99	10
200	Item 2	3.23	50
300	Item 3	19.99	15
400	Item 4	119.99	12
500	Item 5	25	45

1. To build our dashboard application, we will be using the MVVM pattern and SharePoint Client Object Model we discussed in the previous section. To get started, let's define a base view model class that has the code for sending property notification, and a helper method for calling the view's dispatcher.

```
using System;
using System.ComponentModel;
namespace Chapter10.ViewModel
{
    public class BaseViewModel : INotifyPropertyChanged
    {
        private IView view;
        public BaseViewModel(IView view)
        {
            this.view = view;
        }
        public void CallDispatcher(Action action)
        {
            this.view.ViewDispatcher.BeginInvoke(action);
        }
        public event
        PropertyChangedEventHandler
        PropertyChanged;
        protected void
```

```

        SendChangedNotification(stringpropertyName)
        {
            if (this.PropertyChanged != null)
            {
                this.PropertyChanged
                    (this,
                     new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}

```

2. The next step is to create the product class we will load with the data returned from SharePoint. We must create a child collection for the Units in Stock so we can bind the result to the bar chart.

```

using System.Collections.Generic;
namespace Chapter10.Model
{
    public class Product
    {public class Product
    {
        public string ProductId { get; set; }
        public string ProductName { get; set; }
        public double UnitPrice { get; set; }
        public IList<NameValueItem>UnitsInStock { get; set; }
        public void AddUnitsInStock(int value)
        {
            IList<NameValueItem> data = new List<NameValueItem>();
            data.Add(new NameValueItem {Name = «UnitsInStock»,
                Value = value});
            this.UnitsInStock = data;
        }
    }
}
public class NameValueItem
{public class NameValueItem
{
    public string Name { get; set; }
    public int Value { get; set; }
}
}
}

```

3. Set up the view for the Dashboard so it contains a data grid. Add the following text columns to the grid: Product ID, Product Name, and Unit Price. Add a template column that will display the bar chart for the number of units in stock.

```
<navigation:Page x:Class="Chapter10.Views.ShareDashboard"
  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"
  xmlns:navigation="clr-namespace:System.Windows.Controls;
  assembly=System.Windows.Controls.Navigation"
  xmlns:Controls="clr-namespace:System.Windows.Controls;
  assembly=System.Windows.Controls.Data" d:DesignWidth="640"
  d:DesignHeight="480"
  xmlns:dv="clr-namespace:System.Windows.Controls.
  DataVisualization.Charting;assembly=System.Windows.Controls.
  DataVisualization.Toolkit"
  xmlns:chartingPrimitivesToolkit="clr-namespace:System.Windows.
  Controls.DataVisualization.Charting.Primitives;
  assembly=System.Windows.Controls.DataVisualization.Toolkit"
  xmlns:datavis="clr-namespace:System.Windows.Controls.
  DataVisualization;assembly=System.Windows.Controls.
  DataVisualization.Toolkit" Title="ShareDashboard Page">
  <Grid x:Name="LayoutRoot">
    <Controls:DataGrid x:Name="ProductsGrid"
      ItemsSource="{Binding Products}"
      Margin="5, 5, 0, 0"
      HorizontalAlignment="Left"
      VerticalAlignment="Top"
      Style="{StaticResourceDataGridStyle}" >
      <Controls:DataGrid.Columns>
        <Controls:DataGridTextColumn Header="Product
          Id" IsReadOnly="True" Binding="{BindingProductId}"
          Width="150" />
        <Controls:DataGridTextColumn Header="Product
          Name" IsReadOnly="True" Binding="{BindingProductName}"
          Width="150" />
        <Controls:DataGridTextColumn Header="Unit
          Price" IsReadOnly="True" Binding="{BindingUnitPrice}"
          Width="100" />
        <Controls:DataGridTemplateColumn Header="Units in Stock"
          Width="100*" >
          <Controls:DataGridTemplateColumn.CellTemplate>
```

```

<DataTemplate>
  <Grid Height=>30>VerticalAlignment=>Center>
    Margin=>5,5,0,0>
    <dv:ChartVerticalContentAlignment=>Center>
      HorizontalContentAlignment=>Center>
      <dv:Chart.LegendStyle>
        <StyleTargetType=>datavis:Legend>
          <Setter Property=>Width> Value=>0/>
          <Setter Property=>Height> Value=>0/>
        </Style>
      </dv:Chart.LegendStyle>
      <dv:Chart.Series>
        <dv:BarSeriesItemsSource=>
          {BindingUnitsInStock}>
          IndependentValueBinding=>{Binding Name}>
          DependentValueBinding=>{Binding Value}>
          Height=>40> >
        </dv:BarSeries>
      </dv:Chart.Series>
      <dv:Chart.Template>
        <ControlTemplateTargetType=>dv:Chart>
          <chartingPrimitivesToolkit:EdgePanel
            x:Name=>ChartArea> Height=>40>
            <GridCanvas.ZIndex=>-1> />
          </chartingPrimitivesToolkit:EdgePanel>
        </ControlTemplate>
      </dv:Chart.Template>
      <dv:Chart.Axes>
        <dv:LinearAxis Orientation=>x>
          Opacity=>0>MinWidth=>1> Minimum=>0>
          Maximum=>1000> />
        <dv:LinearAxis Orientation=>y>
          Opacity=>0>MinWidth=>1> />
        <dv:CategoryAxis Orientation=>y>
          Opacity=>0> />
        </dv:Chart.Axes>
      </dv:Chart>
    </Grid>
  </DataTemplate>
</Controls:DataGridTemplateColumn.CellTemplate>
</Controls:DataGridTemplateColumn>
</Controls:DataGrid.Columns>
</Controls:DataGrid>
</Grid>
</navigation:Page>

```

4. In the code behind for the view, implement the `IView` interface. This is used for exposing the UI dispatcher to the view model. Make sure you set the data context for the view to an instance of its view model.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using System.Windows.Threading;
using Chapter10.ViewModel;

namespace Chapter10.Views
{
    public partial class ShareDashboard : Page, IView
    {
        {public ShareDashboard()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler
                (ShareDashboard_Loaded);
        }

        void ShareDashboard_Loaded(object sender, RoutedEventArgs e)
        {
            this.DataContext = new DashboardViewModel(this);
        }

        public Dispatcher ViewDispatcher
        {
            get { return this.Dispatcher; }
        }

        // Executes when the user navigates to this page.
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }
    }
}
```

5. In the view model, create a new method named `LoadData`. Inside of the method, create the following CAML query. This query will retrieve the listed fields from the Products List stored in SharePoint.

```
ClientContext clientContext = ClientContext.Current;
oWebsite = clientContext.Web;
this.oList = oWebsite.Lists.GetByTitle(«Products»);

CamlQuery camlQuery = new CamlQuery();
camlQuery.ViewXml = «<View><RowLimit>100</RowLimit></View>»;
```

```

this.collListItem = oList.GetItems(camlQuery);
clientContext.Load(collListItem,
    items =>items.Include(
        item =>item.Id,
        item => item[«Title»],
        item => item[«ProductName»],
        item => item[«UnitPrice»],
        item => item[«UnitsInStock»],
        item =>item.HasUniqueRoleAssignments));
clientContext.ExecuteQueryAsync(onQuerySucceeded, onQueryFailed);
    }

```

6. Add a new property to the view model named `Products`, and set it up so it notifies the view when its value changes. We will set this property when the product list is successfully retrieved from SharePoint.

```

private IList<Product> products = null;
public IList<Product> Products
{
    get { return this.products; }
    set
    {
        if (this.products != value)
        {
            this.products = value;
            this.SendChangedNotification(
                «Products»);
        }
    }
}

```

7. Add the `OnQuerySucceeded` and `OnQueryFailed` methods to the view model. The `OnQuerySucceeded` method will be called when the query execution is completed. `OnQueryFailed` will be called if an error occurs when the query is executed.
8. Add the `DisplayInfo` method that will be called when the query execution is completed successfully. This method will loop through the list items returned from SharePoint and create a collection of products that will bind to the dashboard.

```

private void DisplayInfo()
{
    IList<Product> products = new List<Product>();
    foreach (ListItem item in this.collListItem)

```



```
{
    Product product = new Product
    {
        ProductId = item[«Title»].ToString(),
        ProductName = item[«ProductName»].ToString(),
        UnitPrice = double.Parse(item[«UnitPrice»].ToString())
    };
    product.AddUnitsInStock(int.Parse(item[«UnitsInStock»].
        ToString()));
    products.Add(product);
}
this.Products = products;
}
```

SharePoint Data Access Strategies

When working with SharePoint, there are several different approaches you can take for accessing data. While it fairly easy to set up a SharePoint list, it is usually not the best option for a production application; especially if you have a large amount of data or a complex data structure. In such cases, you should use a RDMBS database, such as SQL server, and build custom Windows Communication Foundation (WCF) services that expose your application data to your Silverlight Dashboard.

If you are using SharePoint Office Server 2010, you can use the Business Connectivity Services (BCS) to set up number of external data sources, including RDMBS, AS400 Links, or external Web Services. To access the BCS, you need to use the SharePoint Object Model from your custom Windows Communication Foundation service.

Summary

In this chapter we gave an overview of how to use SharePoint 2010 to host Silverlight dashboard applications. We walked through how to setup SharePoint 2010 on Windows 7, and introduced you to building Silverlight web parts and use the client object model to retrieve data from SharePoint. Whether you are building a dashboard or other line of business application, the combination of SharePoint and Silverlight in a powerful platform to utilize.

11

Working with 3D Characters



This chapter is taken from *3D Game Development with Microsoft Silverlight 3 Beginner's Guide* (Chapter 4) by Gastón C. Hillar.

In order to create a nice 3D scene for a game, we must be able to work with 3D DCC tools and then be able to load the models in our application. This seems to be an easy task, involving just a few steps. However, it involves many file format conversions that usually generate a lot of incompatibilities. There is always trouble just around the corner when working with 3D meshes. Hence, we must use the right tools and procedures to achieve our desired result.

In this chapter, we will take 3D elements from popular and professional 3D DCC tools and we will show them rendered in real-time on the screen. By reading it and following the exercises we will learn to:

- ◆ Understand how to work in a 3D world that is shown in a 2D screen
- ◆ Take advantage of 3D DCC tools to create 3D models for our games
- ◆ Prepare the 3D elements to be loaded into our games
- ◆ Understand hardware and software real-time rendering processes
- ◆ Control transformations applied to meshes and 3D elements

The second remake assignment

So far, we have been working with raster and vector based sprites in 2D scenes. We were able to use a good object-oriented design to generalize the most common tasks related to sprite management. However, our first goal is to develop 3D scenes using Silverlight 3. How can we load and display 3D characters in a 3D space using Silverlight?

We can do this by exploiting the powerful features offered by many 3D DCC tools to create and export **3D models** to the file formats that are compatible with the Silverlight **3D engine**. Then, we can combine these models with a good object-oriented design and we will be able to use similar principles to the ones learned for raster and vector based sprites, but working in a 3D space. It is time to begin working with 3D games, in particular a 3D space invaders game.



The key to success is preparing the 3D models before exporting them to the required file formats. We must understand how 3D DCC tools work in order to create compatible models for our games.


Time for Action – exporting a 3D model without considering textures

The vector-based prototype of the remake was indeed successful. You have signed your first contract to develop a new remake! This time, you will have to create a 3D remake. Your first assignment is to work with a 3D digital artist to choose a 3D model for the spaceship. In order to do so, you have to watch the model being rendered, and rotate it in the 3D space. This will allow you and the 3D digital artist to decide whether the spaceship is suitable or not for this new game.

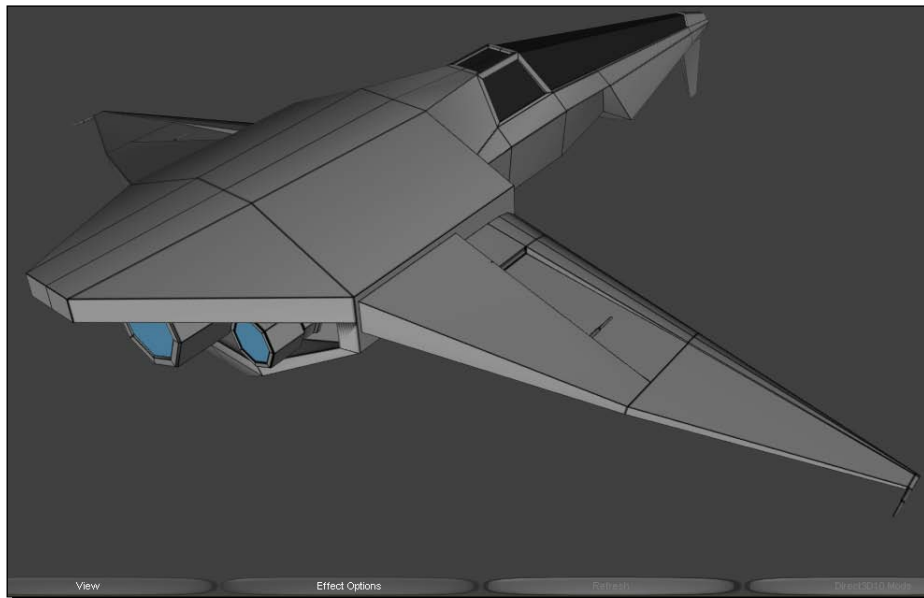
The 3D digital artist has been creating 3D models for DirectX games. Therefore, he is used to working with the DirectX .x file format. As you do not know the appropriate 3D engine to use with Silverlight in order to load the model, your first tests will be done using an XBAP WPF application and an XAML 3D model. This will allow you to interact with the 3D model.

First, we are going to convert the model to the XAML file format using an open source 3D DCC tool:

1. Download the spaceship model in Direct X .x format from XNA Fusion's website (<http://www.xnafusion.com>). This website offers many 3D models with low polygon counts, with a Creative Commons License (<http://creativecommons.org/licenses/by/3.0>), appropriate for usage in games using **real-time rendering**. The link for the spaceship preview <http://www.xnafusion.com/?p=97> and the link for downloading the compressed (.zip) file with the model and the textures is http://www.xnafusion.com/wp-content/uploads/2009/02/ship_06.zip. Save all the uncompressed files and folders in a new folder (C:\Silverlight3D\Invaders3D\3DModels\SPACESHIP01). The decompression process will create two new folders: Models and Textures.

 This 3D model, called Ship_06, was created by Skonk (e-mail: skonk@xnafusion.com). You are free to use, modify and share the content providing credit is given to the author.

2. You can preview, zoom, and rotate the DirectX .x format using the DirectX Viewer included in DirectX Software Development Kit. You can download and install it from <http://www.microsoft.com/downloads/details.aspx?FamilyID=24a541d6-0486-4453-8641-1eee9e21b282&displaylang=en>. Once installed, you can preview the .x files, as shown in the following screenshot, by double clicking on them in Windows Explorer:

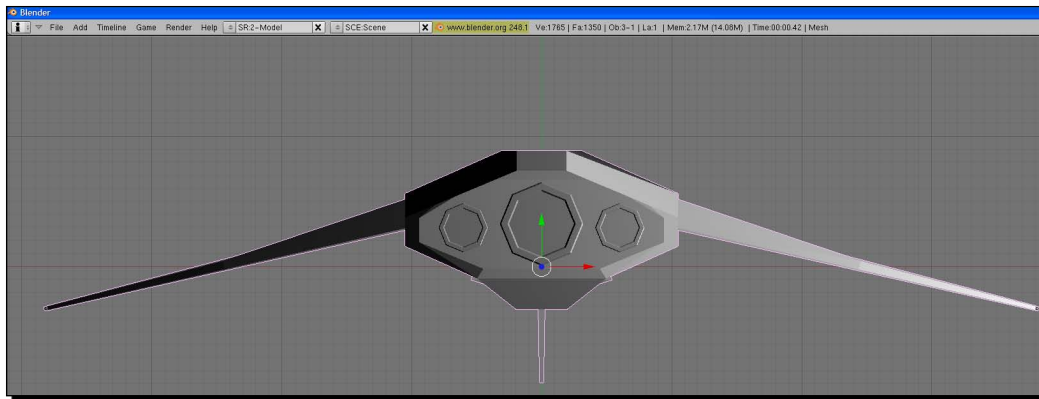


3. If you do not have it yet, download and install Blender (<http://www.blender.org/download/get-blender/>).

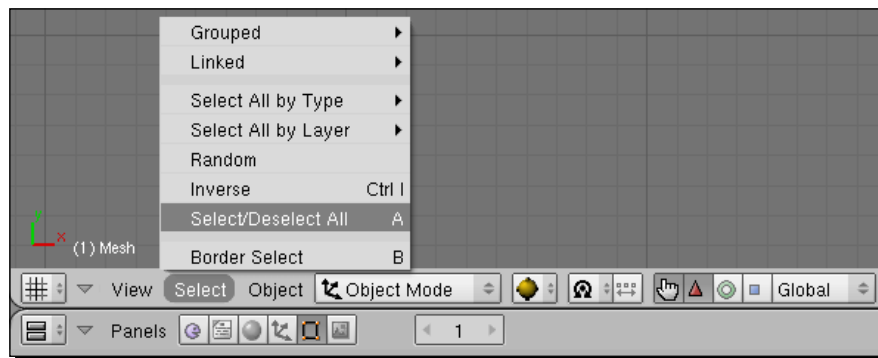


Blender is an excellent open source 3D DCC tool available for all major operating systems. It is distributed under the GNU General Public License (<http://www.blender.org/education-help/faq/gpl-for-artists/>). The creation of 3D architectural visualizations of buildings, interiors, and environmental scenery using Blender is described in depth in *Blender 3D Architecture, Buildings, and Scenery* by Allan Brito, Packt Publishing.

4. Download the latest version of the XAML Exporter for Blender from <http://xamlexporter.codeplex.com/>. For example, one of the latest versions is <http://xamlexporter.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=25481#DownloadId=63694>. This script is developed by TheRHogue and released under the **Microsoft Public License (Ms-PL)**.
5. Save the downloaded Python script (the file with the .py extension) to Blender's scripts folder. By default, it is C:\Program Files\Blender Foundation\blender\scripts. The filename for the version v0.48 of this script is xaml_export.py. On some installations, the location for Blender's scripts folder could also be C:\Documents and Settings\\Application Data\Blender Foundation\Blender\blender\scripts.
6. Start Blender. Right-click on the cube (represented by a square in the view). Press *Del* and click on **Erase selected object(s)**. This step is necessary because we do not want the cube in our new 3D model.
7. Select **File | Import | DirectX (.x)...** Browse to the folder that holds the .x file (C:\Silverlight3D\Invaders3D\3DModels\SPACESHIP01\Models) and select the file to import, Ship_06.x. Then, click on **Import DirectX**. The spaceship mesh's default top view will appear on the screen, as shown in the following screenshot:

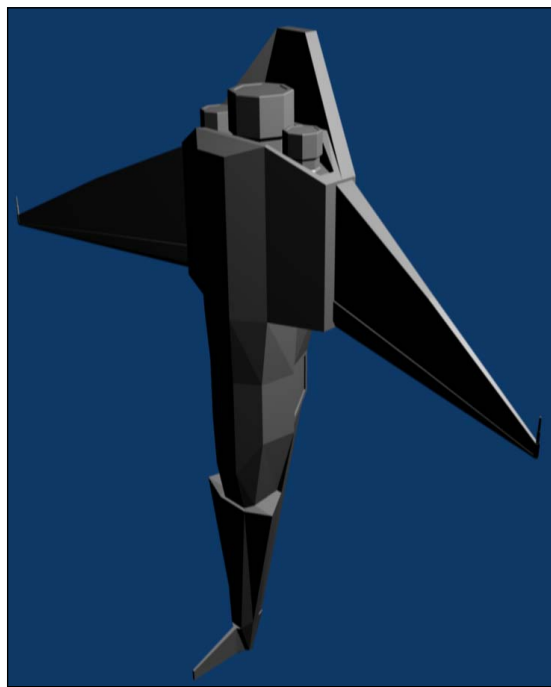


8. Save the model in Blender's native format. It will be useful for further format conversions. Select **File | Save...** Browse to the folder created to hold the new models (C:\Silverlight3D\Invaders3D\3DModels\SPACESHIP01) and enter the desired name, *Ship01.blend*. Then, click on **Save As**. Now, the model is available to be loaded in Blender without the need to follow the aforementioned steps.
9. Click on **Select | Select/Deselect All** in the menu that appears at the bottom of the 3D model. There will be no elements selected on the viewport, as shown in the following screenshot:



10. Click on **Select | Select/Deselect All** again. Now, you will see all the elements selected on the viewport. These two steps are necessary to ensure that all the elements are going to be exported in the new file format.
11. Now, select **File | Export | XAML (.xaml)...** in the main menu. The default folder will be the same as the one used in the previous step. Hence, you will not need to browse to another folder. Enter the desired name, *Ship01.xaml*. Then, click on **Export Xaml**. Now, the model is available as an XAML 3D model.

12. Select **View | Camera** in the menu that appears at the bottom of the 3D model. The spaceship mesh will appear as seen by the active **camera**. Select **Render | Render current frame** in the main menu and a new window will appear showing the spaceship rendered by Blender's internal render engine, without using the texture to envelope the mesh, as shown in the following screenshot:



What just happened?

We used Blender's import and export capabilities to convert an existing 3D model designed to work with DirectX or XNA Framework to the new XAML 3D file format. We imported the DirectX .x model into Blender and then we exported it to XAML.

We used the default camera to render the model without textures on a 2D screen. The rendering process takes a few seconds to show the results, because it is focused on offering an accurate scene. When we render a scene using 3D DCC tools like Blender or 3D Studio Max, they take the necessary time to offer the best possible 2D image that represents the 3D scene, according to the rendering technique used.

However, 3D games need to show many 3D models performing animations on the screen in **real-time**. They require many successive rendering processes per second, in order to generate many 2D images, representing the 3D scene, per second. A 3D game needs more than 30 frames per second (FPS) to create a fluid animation.

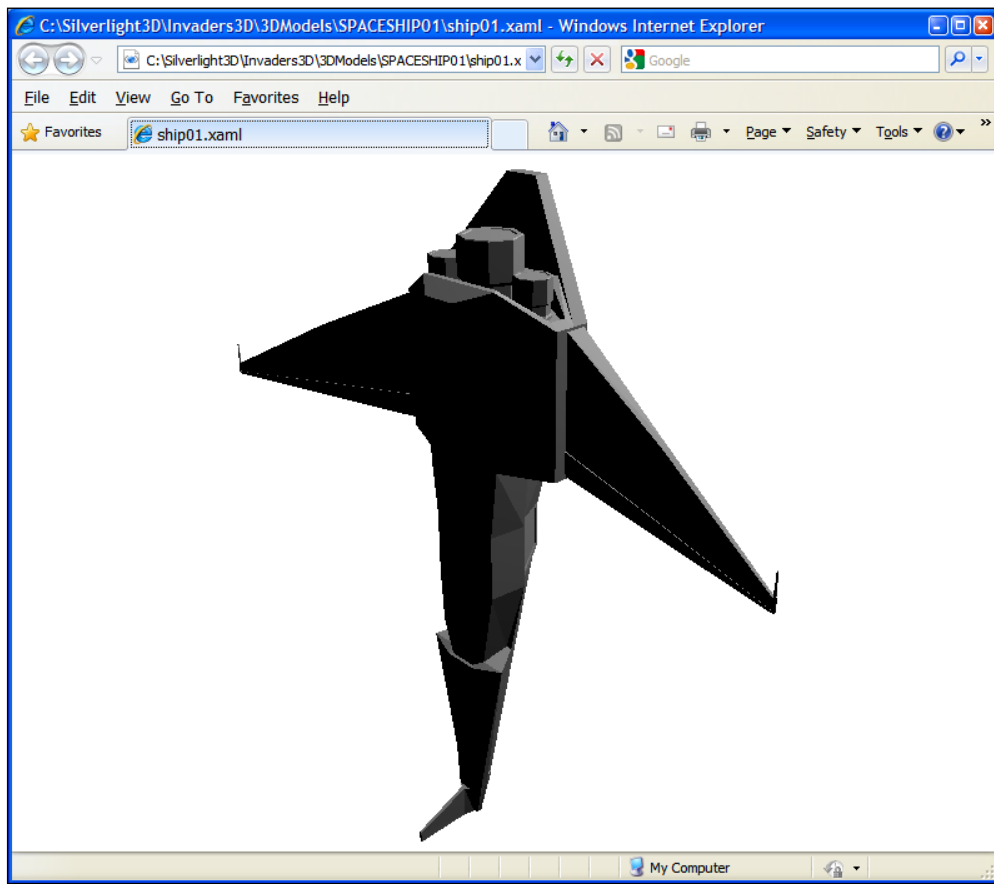


For this reason, the real-time rendering engines used for 3D games require 3D models with a lower polygon counts than the ones used to create realistic 3D scenes in 3D DCC tools. On one hand, this reduces the rendering process' accuracy, but on the other hand, it allows the game to show a fluid animation.

The animation speed is often more important than the scene's definition in a 3D game. A game showing excellent scenes but running at less than 5 FPS does not make sense. It will make the player quit the game as soon as he notices the low animation speed. A 3D game needs to provide a real-time response to the player.

XAML 3D models

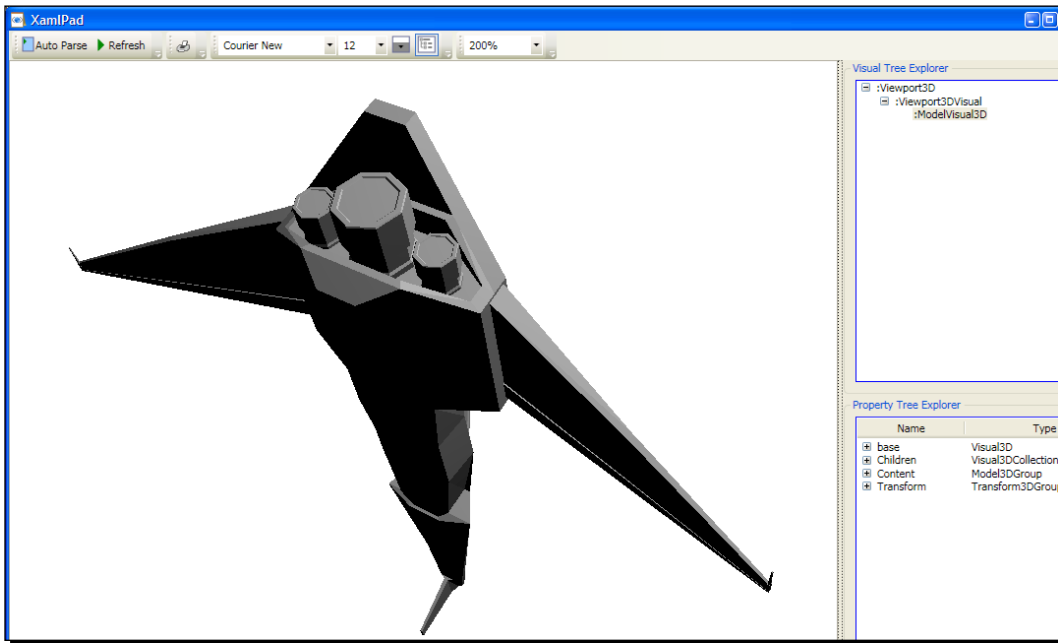
If we open the previously exported XAML 3D model (`Ship01.xaml`) using Internet Explorer, it will show us the 3D spaceship rendered in the browser's window, as shown in the following screenshot:



This happens because the XAML 3D model includes a definition for a 3D viewport and a camera targeting the model (the spaceship). Internet Explorer is performing the rendering process necessary to display a 3D model defined using XAML in a 2D screen.

We can display an XAML 3D model in any WPF application. We can create a 3D model using XAML directives. However, this does not make sense when working with 3D games. We would need a few days to create the spaceship model writing XAML code. As previously explained for 2D art assets, drawing is easier than writing XAML code. 3D modelling is also easier and more efficient than writing XAML code to define the 3D meshes that represent the model. Besides, you are working with a great 3D digital artist that will provide you with the necessary 3D models for your games.

We can also preview an XAML 3D model copying and pasting the XAML code in XamlPad, as shown in the following screenshot:

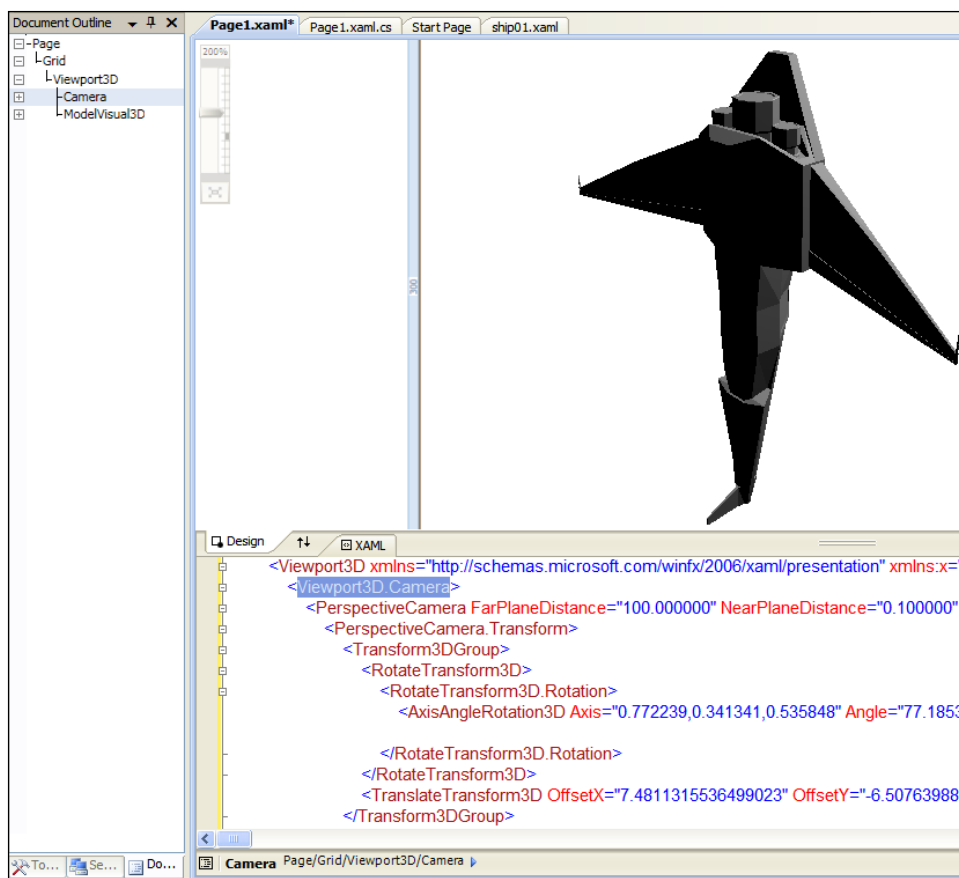


The creation of XAML 3D models by writing XAML code is described in depth in *3D Programming for Windows Three-Dimensional Graphics Programming for the Windows Presentation Foundation* by Charles Petzold, Microsoft Press.

Time for action – from DCC tools to WPF

Now, we are going to display the XAML 3D model exported from Blender in an XBAP WPF application:

1. Create a new C# project using the **WPF Browser Application** template in Visual Studio or Visual C# Express. Use `3DInvadersXBAP` as the project's name.
2. Open the file that defines the XAML 3D model using Notepad or any other text editor. Select all the content and copy it to the clipboard.
3. Open the XAML code for `Page1.xaml` (double-click on it in the **Solution Explorer**) and paste the previously copied XAML 3D model after the line that begins defining the main Grid (`<Grid>`). You will see the spaceship appear in the page in the designer window. You can understand how an XAML 3D model is defined and inserted in a `Viewport3D` container navigating through the document's outline, as shown in the following screenshot:



4. Build and run the solution. The default web browser will appear showing the spaceship rendered in the 2D window. Resize the web browser's window and the model will scale. This time, it is running an XBAP WPF application.

What just happened?

We showed the spaceship rendered in a viewport just copying and pasting the XAML 3D model definition previously exported from a 3D DCC tool.

One of the great advantages of WPF applications is that we can preview the XAML 3D model in design-time. We can also see the changes while we modify the properties in XAML code.

XBAP WPF applications with 3D content

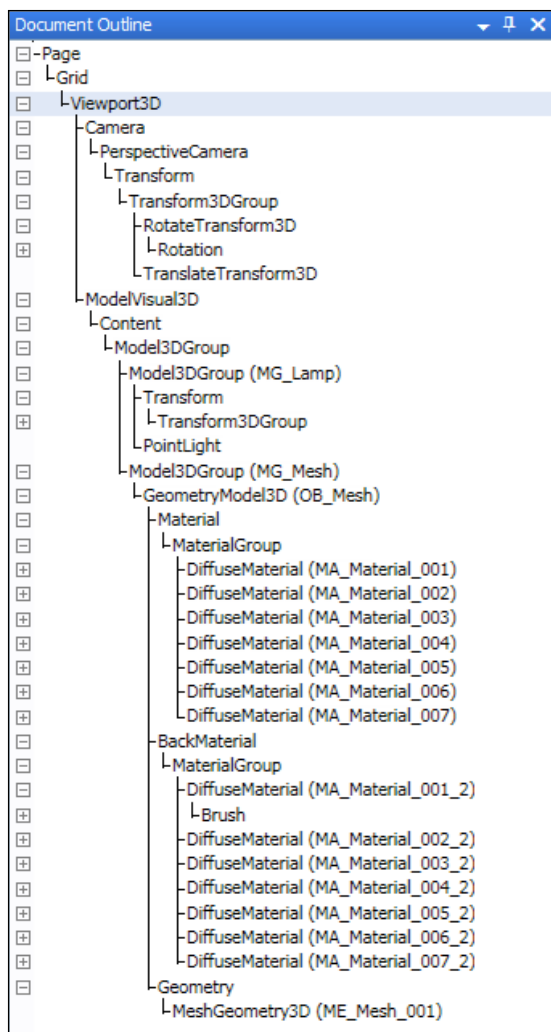
The key is the `Viewport3D` element (`System.Windows.Controls.Viewport3D`). It allows the definition of 3D elements like cameras, models, meshes, lights, materials, and 3D transforms, among others.

This line defines the `Viewport3D` element. The XAML export filter created it:

```
<Viewport3D xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

Inside the `Viewport3D`, we can find the `Viewport3D.Camera` element that defines a camera targeting the model and the `ModelVisual3D` element (`System.Windows.Media3D.ModelVisual3D`) that, in this case, defines many transformations, lights, materials, brushes and meshes.

We did not need to write a single line of code to preview and display the 3D model exported from the 3D DCC tool. The tool created the necessary XAML code to describe the viewport, a camera and the model's components, as shown in the following screenshot presenting a more complete document's outline:



Time for action – displaying a 3D model in a 2D screen with WPF

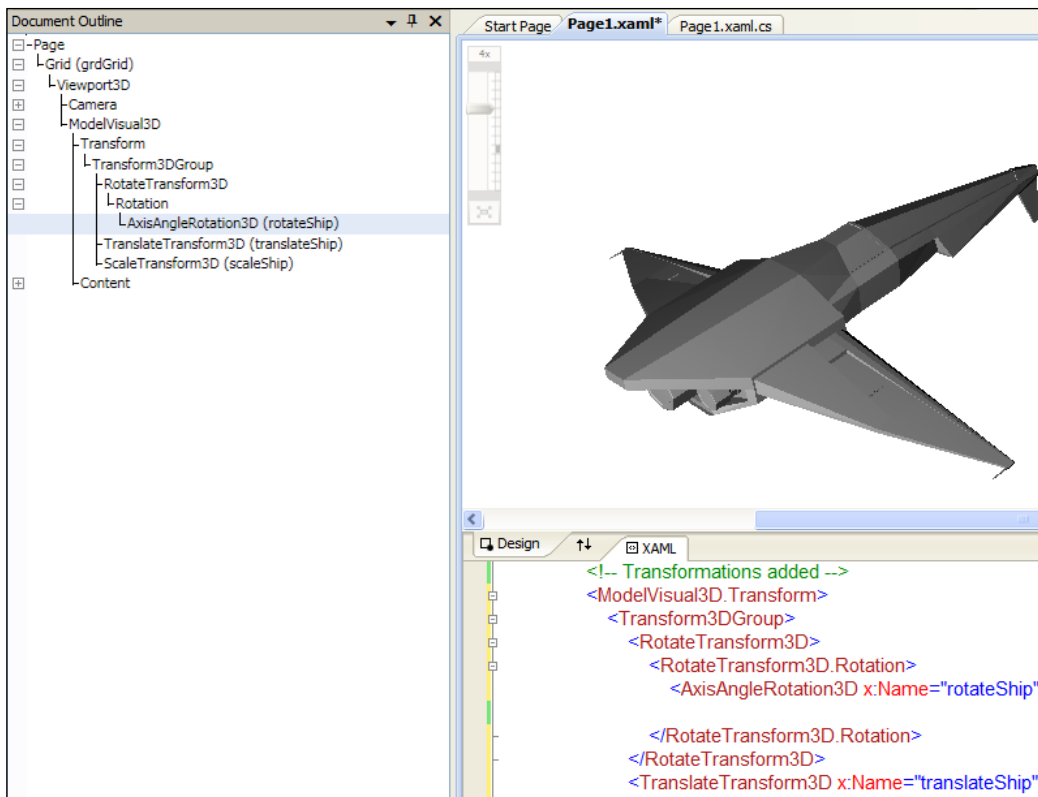
The 3D digital artist is still waiting to see the spaceship from different angles. He needs to know if the model is appropriate for your game. You want to see the ship moving and rotating in the screen. In order to do this, we must add some transformations and some code to control them. We will add both XAML and C# code:

1. Stay in the 3DInvadersXBAP project.

2. Open the XAML code for Page1.xaml and add the following lines of code after <ModelVisual3D> (we are adding transformations for the ModelVisual3D element):

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <RotateTransform3D>
      <RotateTransform3D.Rotation>
        <AxisAngleRotation3D x:Name="rotateShip" Axis="1 0 0"
          Angle="100" />
      </RotateTransform3D.Rotation>
    </RotateTransform3D>
    <TranslateTransform3D x:Name="translateShip" OffsetX="0.0"
      OffsetY="0.0" OffsetZ="0.0" />
    <ScaleTransform3D x:Name="scaleShip" ScaleX="0.5"
      ScaleY="0.5" ScaleZ="0.5" />
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

3. You will see the spaceship rotated and scaled down, as shown in the following screenshot:



4. Change the definition of the `Grid` element by the following (the grid must be focusable and we are defining an event handler to capture the keys pressed):

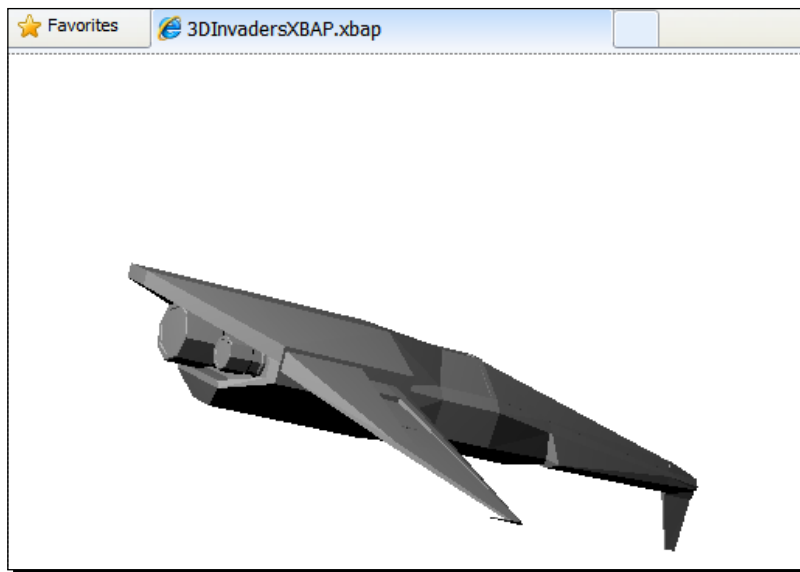
```
<Grid x:Name="grdGrid" KeyDown="Grid_KeyDown" Focusable="True">
```
5. Now, expand `Page1.xaml` in the Solution Explorer and open `Page1.xaml.cs`—the C# code for `Page1.xaml`. (double-click on it). We need to add an event handler to change the values of some properties for the previously defined transformations according to the keys pressed.

6. Add the following lines of code in the public partial class `Page1 : Page`, to program the event handler for the `KeyDown` event:

```
private void Grid_KeyDown(object sender, KeyEventArgs e)
{
    switch (e.Key)
    {
        // Move the ship in the X; Y and Z axis
        case Key.Left:
            translateShip.OffsetX -= 0.05f;
            break;
        case Key.Right:
            translateShip.OffsetX += 0.05f;
            break;
        case Key.Up:
            translateShip.OffsetY += 0.05f;
            break;
        case Key.Down:
            translateShip.OffsetY -= 0.05f;
            break;
        case Key.Z:
            translateShip.OffsetZ -= 0.05f;
            break;
        case Key.X:
            translateShip.OffsetZ += 0.05f;
            break;
        // Rotate the ship
        case Key.G:
            rotateShip.Angle -= 1;
            break;
        case Key.H:
            rotateShip.Angle += 1;
            break;
    }
}
```

7. Add the following line of code to the constructor after the line `InitializeComponent();` (we set the focus to the grid to capture the keyboard events):

```
grdGrid.Focus();
```
8. Build and run the solution. The default web browser will appear showing the spaceship 3D model rendered in the 2D screen. Use the cursor movement keys and the Z and X keys to move the spaceship in the X, Y, and Z axis. Use the G and H keys to rotate the spaceship. You will see the spaceship moving and rotating in real-time inside the web browser's viewport as shown in the following screenshot:



What just happened?

The 3D digital artist could move and rotate the spaceship in real-time. You have decided that the spaceship is suitable for this new game. The same code base could be used for many other 3D models in order to watch them from different angles.

We changed the definition of the `Grid` element to allow it to be focusable (`Focusable="True"`), because we wanted to capture the keys pressed by the user. Besides, we defined an event handler for the `KeyDown` event.

Understanding the 3D world

We added tree transforms for the `ModelVisual3D` that contains the meshes that define the 3D model. They transform the model, not the camera. However, the model is viewed through the camera defined in the `Viewport3D` container. This is one of the main differences between the 2D world and the 3D world.

In a 2D scene, we can easily understand dimensions, because we work with pixels and a great pixel grid.

In a 3D scene, the active camera defines an eye for the models. Hence, when this scene is rendered in a 2D screen, we can see a part of the entire 3D world through the camera's lens. The camera changes the perspective for the 3D models that compose the 3D world. Hence, we work with relative dimensions, because what is seen in the 2D screen can change according to the camera used and its properties.

In this case, we apply the transforms to the model and we are keeping the camera stationary. We defined a `Transform3DGroup` to group the three transforms.

The `ScaleTransform3D`, named `scaleShip`, enables us to change its `ScaleX`, `ScaleY`, and `ScaleZ` properties to shrink or stretch the meshes that compose the model. This is done in the following line:

```
<ScaleTransform3D x:Name="scaleShip" ScaleX="0.5" ScaleY="0.5"
    ScaleZ="0.5" />
```

Initially, we scale down the spaceship proportionally to 50% of its original size.

The `TranslateTransform3D`, named `translateShip`, enables us to move the model's meshes through the 3D space. We can do this changing its `OffsetX`, `OffsetY`, and `OffsetZ` properties in order to change its position in the X; Y and Z axis. This is done in the following line:

```
<TranslateTransform3D x:Name="translateShip" OffsetX="0.0"
    OffsetY="0.0" OffsetZ="0.0" />
```

The `RotateTransform3D` is a little more complex, because it adds a `Rotation` and an `AxisAngleRotation3D` named `rotateShip`. It enables us to rotate the model's meshes through its X-axis. We can do this by changing its `Angle` property to rotate the model around its defined central point. This is done in the following lines:

```
<RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="rotateShip" Axis="1 0 0" Angle="100"
      />
  </RotateTransform3D.Rotation>
</RotateTransform3D>
```


The central points can be defined in the `RotateTransform3D` element, using the `CenterX`, `CenterY`, and `CenterZ` properties. In this case, we defined a single rotation, around the X-axis. However, we can add more `RotateTransform3D` groups to define new rotations around different axes. The `Axis` property specifies the axis that will rotate according to the value assigned to the `Angle` property. It requires three binary numbers separated by a space, a 1 indicates that the axis represented in the position (X; Y and Z) should rotate. For example, the following line indicates that the rotation should be done in the X-axis

```
<AxisAngleRotation3D x:Name="rotateShip" Axis="1 0 0" Angle="100" />
```

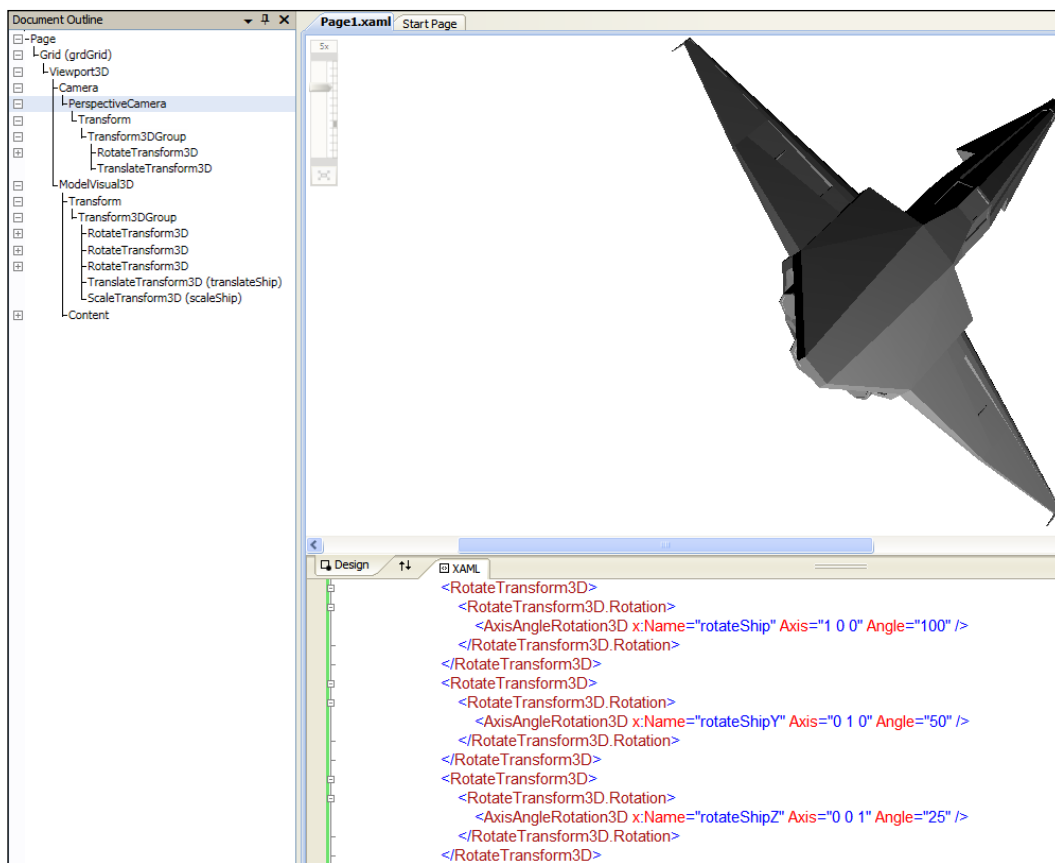
If we want to rotate in the Y-axis, we should change `Axis="1 0 0"` by `Axis="0 1 0"`.

We can define many `RotateTransform3D` groups, like in the following lines of code, in which we create three independent rotations: `rotateShipX`, `rotateShipY`, and `rotateShipZ`:

```
<RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="rotateShipX" Axis="1 0 0"
      Angle="100" />
  </RotateTransform3D.Rotation>
</RotateTransform3D>
<RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="rotateShipY" Axis="0 1 0" Angle="50"
      />
  </RotateTransform3D.Rotation>
</RotateTransform3D>
<RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="rotateShipZ" Axis="0 0 1" Angle="25"
      />
  </RotateTransform3D.Rotation>
</RotateTransform3D>
```



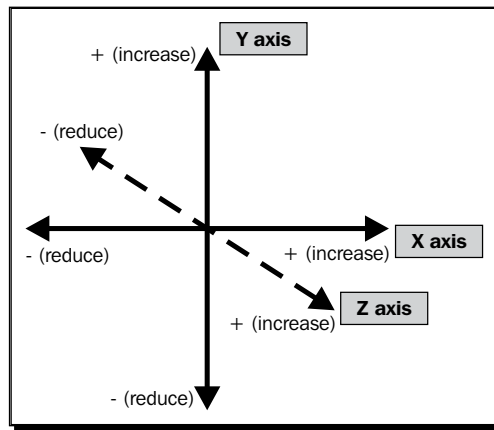
One of the simplest ways of mastering the 3D world and understanding how 3D models move in the 3D space is defining transformations and changing the values of their properties. Using a WPF application and XAML code, we can see a preview in real-time, while we change the rotation values, as shown in the following screenshot that presents the spaceship with the three aforementioned rotations defined:



X, Y, and Z in practice

As previously explained, the 2D world uses a bi-dimensional coordinate system. The 3D world uses a three-dimensional coordinate system. It adds the Z-axis. However, WPF also changes the way the Y-axis works in the 3D world, because the model moves up when the Y-axis increases.

The following diagram illustrates the way the X-coordinate, Y-coordinate, and Z-coordinate values work to display elements in a 3D scene:



The following list explains the previous diagram in detail:

- ◆ If we want to move right, we must increase the X-coordinate's value
- ◆ If we want to move left, we must reduce the X-coordinate's value
- ◆ If we want to move up, we must increase the Y-coordinate's value
- ◆ If we want to move down, we must reduce the Y-coordinate's value
- ◆ If we want to move front, we must increase the Z-coordinate's value
- ◆ If we want to move back, we must reduce the Z-coordinate's value



Nevertheless, we must be very careful, because the three-dimensional coordinate system used in our games will be relative to the active camera. The camera's position and its target in the 3D space will set the baselines for the three-dimensional coordinate system. A camera can view the three-dimensional coordinate system from any direction.

The code programmed in the `KeyDown` event handler takes into account the key that is pressed and reduces or increases the value of the `OffsetX`, `OffsetY`, or `OffsetZ` properties for the translate transform.

It also changes the `Angle` value of the previously explained X-axis rotate transform.

GPU 3D acceleration

Real-time rendering of 3D scenes is a very complex process. On one hand, we have a screen capable of showing 2D images (X and Y), but on the other hand, we have 3D models in a 3D world (X, Y, and Z). There is a very easy to understand asymmetry problem. Therefore, in order to show the 3D scene in a 2D screen, a rendering process must create a 2D image

in a specific resolution that shows the portion of the whole 3D world seen by the lens of an active camera. The screen can display the resulting 2D image.

The rendering process for a single frame requires thousands of complex mathematics operations. We need a performance of at least 30 FPS to create a responsive 3D game. For this reason, the real-time rendering process will require hundreds of thousands of mathematics operations per second.

Therefore, there is specialized hardware dedicated to accelerating real-time 3D rendering processes. We have already talked about GPUs.

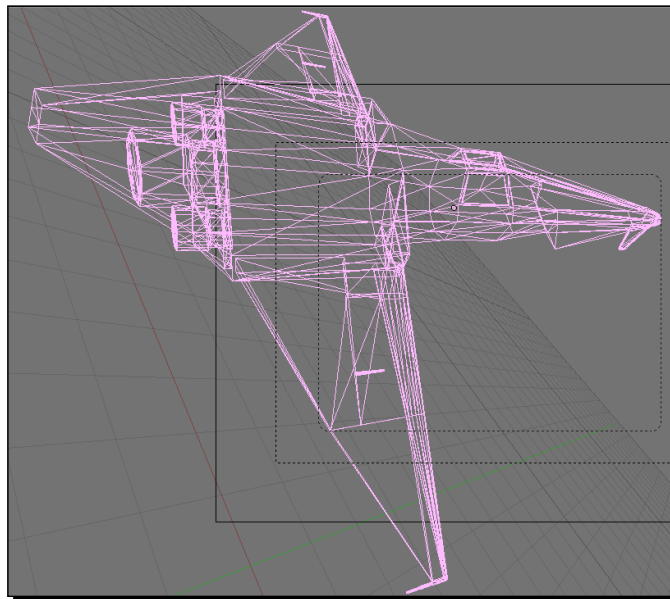
WPF and XBAP WPF applications take advantage of the presence of GPUs to perform real-time 3D rendering processes. Hence, they can offer great performance for 3D games that need to render complex models.




So far, Silverlight 3 does not offer the possibility to use a GPU to perform real-time 3D rendering processes. When working with Silverlight, we will have to use software based rendering. This means that the rendering process will run on the CPU(s) and their available processing cores. Hence, if we need more power for our game, we can take advantage of XBAP WPF applications' advanced capabilities.

Understanding meshes

In the following diagram, we can see the wire-mesh view of the 3D model that represents the spaceship:



This mesh defines the spaceship using many primitive elements (points, lines, triangles, and polygons). We can use materials and textures to paint and envelope the different faces. As we can see, this is a wire-mesh with a low polygon counts. This is very important, because 3D DCC tools are able to work with meshes with hundreds of thousands of polygons. However, they would require too much processing time to successfully render them in real-time. We must remember that we must show at least 30 FPS.

 We can understand how to work with meshes using 3D DCC tools. They will provide us an interactive experience with the 3D models and we will be able to use this knowledge in developing 3D games that interact with meshes and models.


We are going to work with 3D DCC tools in order to create the 3D models and their meshes for our games.

Time for action – using other XAML exporter for DCC tools


The 3D digital artist has to develop new models for the game. However, he is going to develop them using 3D Studio Max. He wants to see a few spheres in the scene to check whether the XAML exporter he found works fine or not.

Now, we are going to convert the model to the XAML file format using an open source XAML exporter for 3D Studio Max:

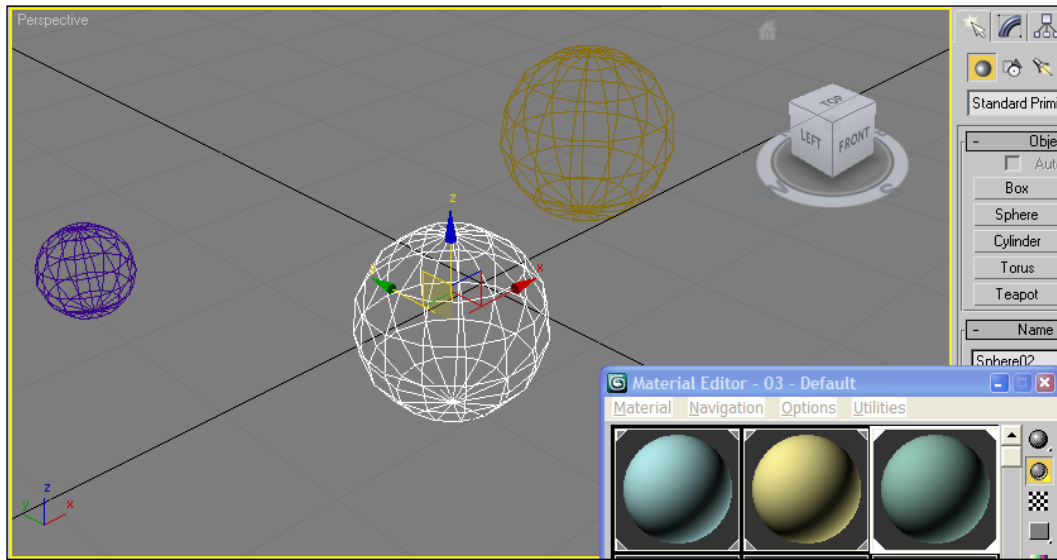
- 1.** Download the most recent release of the XAML exporter for 3D Studio Max from <http://max2xaml.codeplex.com/>.

 Timmy Kokke (Sorskoot) developed the XAML exporter for 3D Studio Max as an open source project in CodePlex.

- 2.** Save all the uncompressed files in a new folder (C:\Silverlight3D\Invaders3D\3DModels\MAX_XAML_EXPORTER). The decompression process will create three files: XamlExport.ms, Main.ms, and Utils.ms.
- 3.** Now, start 3D Studio Max.

 3D Studio Max is commercial software. However, you can download a free fully functional 30-day trial for non-commercial use from Autodesk's website: <http://usa.autodesk.com/>

4. Add three spheres using 15 segments and assign each one a material, as shown in the following screenshot:



5. It is very important to assign a material to each element, because if there is an element without a material, the script used to export to XAML will not work. If you are facing problems, remember that the 3D digital artist left the `spheres.max` file in the following folder (C:\Silverlight3D\Invaders3D\3DModels\SPHERES).
6. Save the file using the name `spheres.max` in the aforementioned folder.
7. Select **MAXScript | Run script...** Browse to the folder in which you decompressed the export scripts (C:\Silverlight3D\Invaders3D\3DModels\MAX_XAML_EXPORTER) and choose `Main.ms`. Then, click on **Open**. A new window will appear showing the title **Max2Xaml**.
8. Click on **Export**. Browse to the same folder in which you saved the original 3D Studio Max model and enter `spheres.xaml` in the **Name** textbox. Then, click on **Save**. The exporter will show the XAML output. Now, the model is available as an XAML resource dictionary.

What just happened?

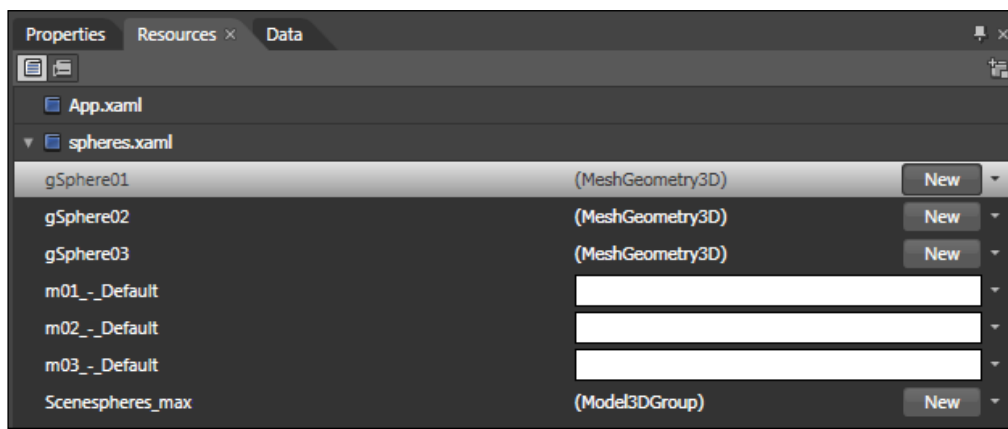
We used an open source XAML exporter to create an XAML 3D model from a 3D Studio Max file. However, we cannot preview the model using a web browser or XamlPad, as previously done with the spaceship.

This XAML exporter creates a resource dictionary with all the data for the meshes and materials. Hence, there is no Viewport3D definition. We will have to work a bit harder to include the meshes in a 3D scene.

Time for action – adding 3D elements and interacting with them using Expression Blend

Now, we are going to add the 3D elements exported from 3D Studio Max to our existing XBAP WPF application:

1. Open the project 3DInvadersXBAP in Expression Blend.
2. Select **Project | Add existing item....** Choose the previously exported `spheres.xaml` and click on **Open**.
3. Click on the **Resources** panel and expand `spheres.xaml`. You will see three `MeshGeometry3D`, three `MaterialGroup`, and a `Model3DGroup` listed, as shown in the following screenshot:



4. Now, open the XAML code for `Page1.xaml` and add the following lines of code after the `Page` definition (we are merging the resource dictionary that contains the definition for the 3D elements in the main page):

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="spheres.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Page.Resources>
```



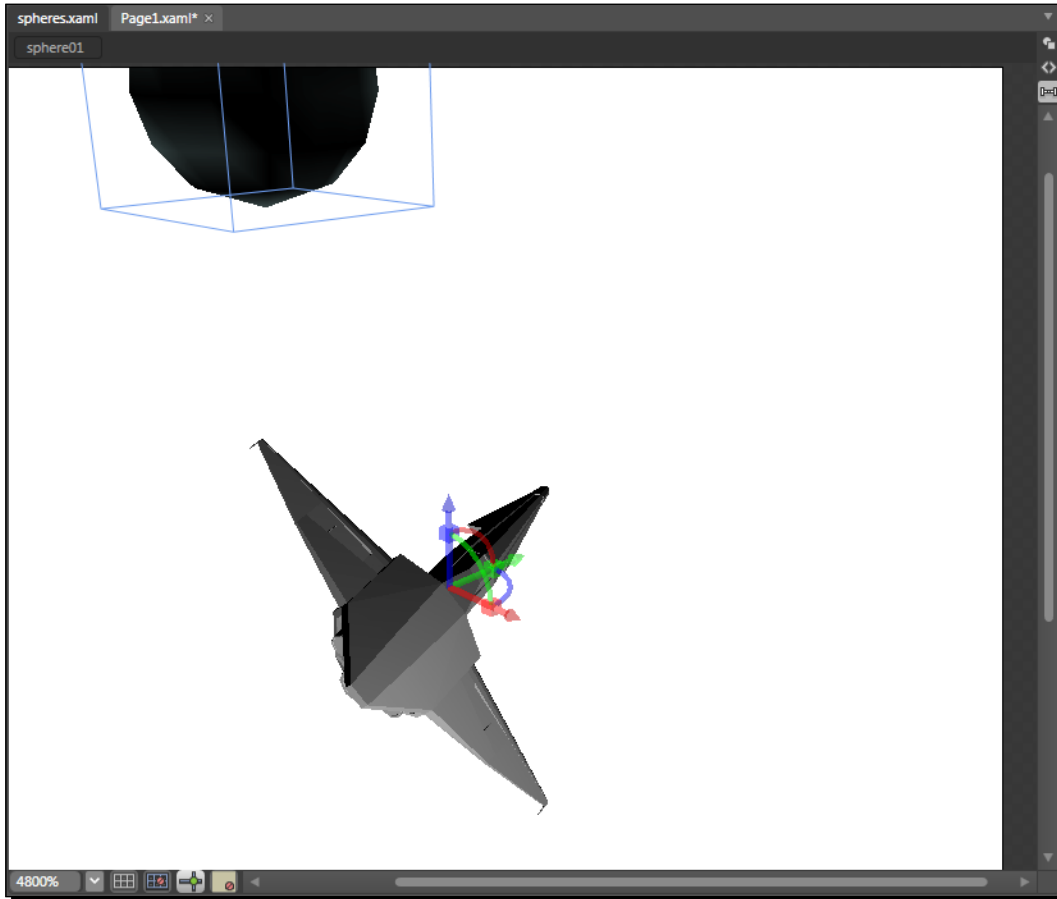
You can also do the previous step without adding any code, by right-clicking on the `Page` element and selecting **Linking to Resource Dictionary**, `spheres.xaml`.

5. Add the following code after the line `</Viewport3D.Camera>` that ends the definition for the camera element (we are adding a new `GeometryModel3D` element for each sphere, using the names `sphere01`, `sphere02`, and `sphere03`):

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <GeometryModel3D x:Name="sphere01" />
  </ModelVisual3D.Content>
</ModelVisual3D>
<ModelVisual3D>
  <ModelVisual3D.Content>
    <GeometryModel3D x:Name="sphere02" />
  </ModelVisual3D.Content>
</ModelVisual3D>
<ModelVisual3D>
  <ModelVisual3D.Content>
    <GeometryModel3D x:Name="sphere03" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

6. Click on `Viewport3D` under **Objects and Timeline** and expand its children. Select `sphere01` go to its **Properties** tab.
7. Expand **Miscellaneous**, click on **Geometry** and select **Local resource | gSphere01** from the context menu that appears. This step assigns the `MeshGeometry3D` resource defined in the resource dictionary.

8. Expand **Materials**, click on **Material** and select **Local resource | m01_-_Default** from the context menu that appears. This step assigns the `MaterialGroup` resource defined in the resource dictionary. Now, you will see a new sphere in the scene, as shown in the following screenshot:



9. Go back to the XAML code and you will see that new code was added to the `GeometryModel3D` element. The line that defines this element will be this:

```
<GeometryModel3D x:Name="sphere01"
    Geometry="{DynamicResource gSphere01}"
    Material="{DynamicResource m01_-_Default}" />
```
10. Repeat the steps 6 to 9 for the other two spheres, `sphere02` and `sphere03`, assigning them the `MeshGeometry3D gSphere02` and `gSphere03`, and then the `MaterialGroup m02_-_Default` and `m03_-_Default`.

What just happened?

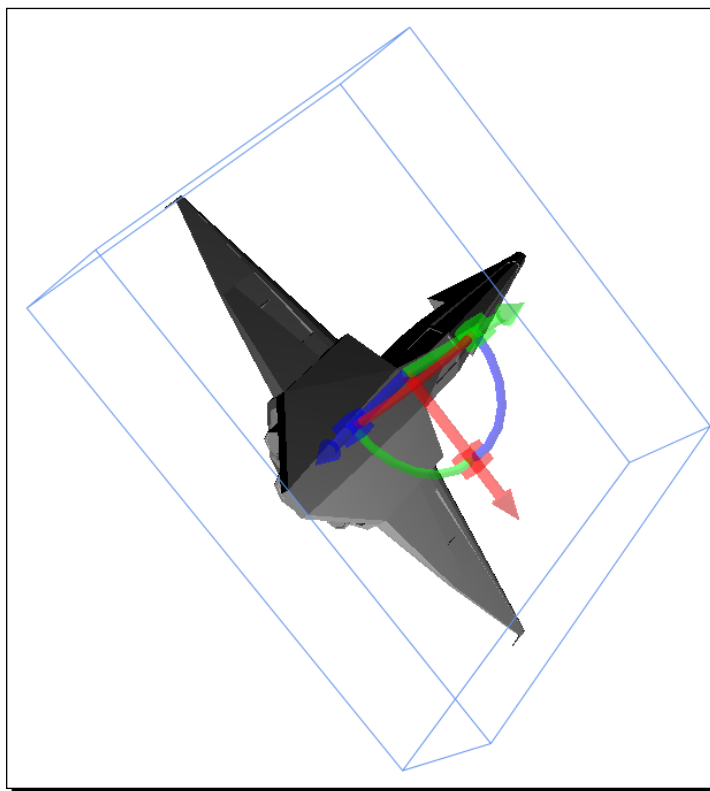
We added the XAML 3D models created in 3D Studio Max to our 3D scene in an XBAP WPF application.

First, we added the resource dictionary to the solution. Then, we created new `GeometryModel3D` elements for each sphere. Finally, we assigned the mesh definitions and the materials to each sphere.

Interacting with 3D elements using Expression Blend

Expression Blend is very helpful when we need to interact with 3D elements contained in a `Viewport3D`. We can see and change properties for the 3D elements that generate the scene. Therefore, we can also learn how things work in the 3D space without the need to write XAML code.

Visual Studio shows a preview of the scene, but it does not allow the same interaction than this tool. Expression Blend allows us to interact visually with the 3D elements, offering the possibility to design the scenes placing and moving elements. For example, we can rotate the spaceship using the mouse, as shown in the following diagram:



Silverlight and the 3D world

So far, we have been adding 3D models to an XBAP WPF application. We exported the models from Blender and 3D Studio Max and we were able to include them in a 3D scene. However, we want to do this using Silverlight, which does not have official support for 3D XAML models. How can we create 3D scenes using real-time rendering in Silverlight 3?

We can do this using a 3D graphics engine designed to add software based real-time rendering capabilities to Silverlight. We have two excellent open source alternatives for this goal:

- ◆ Kit3D (<http://www.codeplex.com/Kit3D>). It is developed by Mark Dawson. Matches the `System.Windows.Media.Media3D` namespace from WPF. It offers a subset of WPF 3D capabilities and it offers a very fast rendering pipeline. Its main drawback is that it does not offer a mechanism to load meshes. Therefore, you have to create the meshes using C# code. It is a good alternative for 3D games that use simple basic meshes like boxes, cubes, and spheres.
- ◆ Balder (<http://www.codeplex.com/Balder>). Einar Ingebrigtsen leads its development team. It is intended to be used in games, for this reason it uses a model similar than the one found in XNA Framework. It offers many features that make it simple to begin developing games with this engine. It offers the possibility to load models from many popular file formats. It is an excellent alternative for 3D games that need to show many 3D models designed using DCC tools.



We will use Balder to add 3D real-time rendering capabilities to Silverlight for our games. However, in some cases, you may find Kit3D to be a very useful alternative.

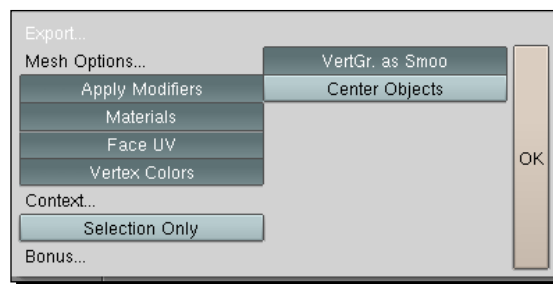
Time for action – exporting a 3D model to ASE

So far, Balder does not offer support for XAML 3D models. However, it works fine with the ASE (3D Studio Max ASCII Scene Exporter) file format. 3D Studio Max offers the possibility to export ASE files from a 3D scene. However, the spaceship model is now in Blender format.

First, we are going to install a script to allow Blender to export models to the ASE format and then we will save the spaceship in the new file format which is compatible with Balder:

1. Download the latest version of the Goofos ASE export script for Blender from http://www.katsbits.com/html/tools_utilities.htm. For example, one of the latest versions is http://www.katsbits.com/files/blender/goofosASE-2.44v0.6.10b_9sept07.zip. This script is developed by Goofos and released under the GNU GPL license.

2. Decompress the downloaded ZIP file and copy the Python script (the file with the .py extension) to Blender's `scripts` folder. By default, it is `C:\Program Files\Blender Foundation\blender\scripts`. The file name for the version 6.10b of this script is `goofosASE-2.44v0.6.10b_9sept07.py`.
3. Restart Blender and open the spaceship model (previously saved in Blender's native format as `Ship01.blend` in `C:\Silverlight3D\Invaders3D\3DModels\SPACESHIP01`).
4. Now, select **File | Export | ASCII Scene (.ase) v0.6.10**. The default folder will be the same used in the previous step. Hence, you will not need to browse to another folder. Enter the desired name, `Ship01.ase`. Then, click on **Export ASCII Scene**. A dialog box will appear.
5. Click on **Selection only** to deselect this option. This will tell the script to export all the elements in the scene and not just the selected ones, as shown in the following screenshot:



6. Click on **OK**. Now, the model is available as an ASE 3D model, ready to be loaded in Silverlight using Balder.


What just happened?

We used Blender's export capabilities to convert an existing 3D model to the ASE file format, using the Goofos ASE export script.

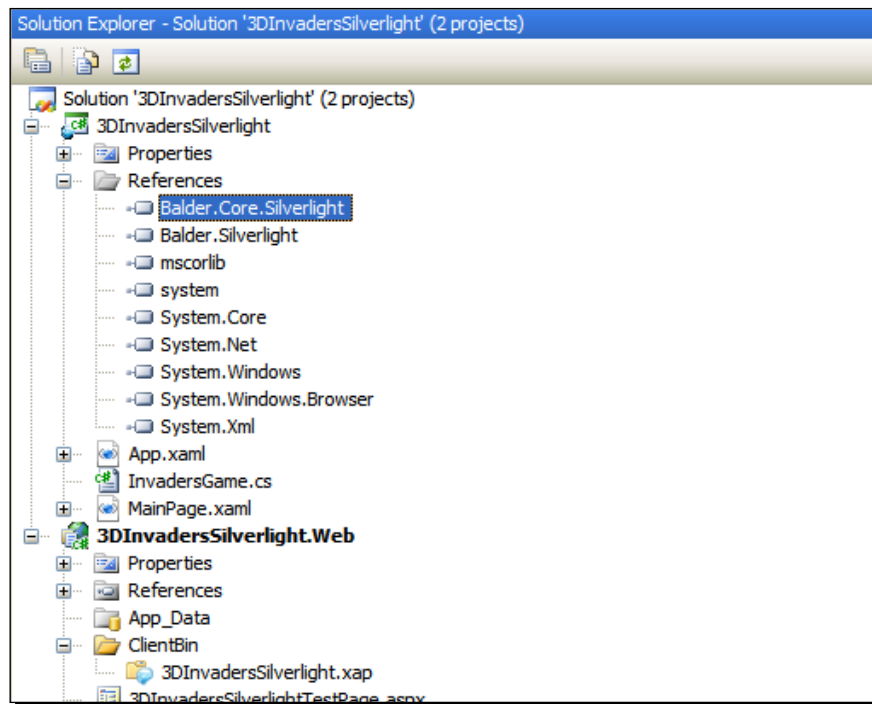
Time for action – installing Balder 3D engine

Now, we are going to create a new Silverlight application adding the necessary references to use the Balder 3D engine:

1. Download the most recent release of Balder from <http://www.codeplex.com/Balder>.

 Balder is a very active project. Thus, it regularly releases new versions adding additional features and fixing bugs. Sometimes, a new version can introduce changes to classes or methods. In the following examples, we will use version 1.0.

2. Save all the uncompressed files in a new folder (C:\Balder).
3. Create a new C# project using the **Silverlight Application** template in Visual Studio or Visual C# Express. Use `3DInvadersSilverlight` as the project's name.
4. Select **File | Add Reference...** and add the following DLLs from Balder's folder:
 - ◆ `Balder.Core.Silverlight.dll`
 - ◆ `Balder.Silverlight.dll`
5. Now, the project will list the references to the aforementioned Balder's DLLs in the **Solution Explorer**, as shown in the following screenshot:



What just happened?

We downloaded Balder and we added the necessary references to use it in a Silverlight project. The previously explained steps are the only ones required to access Balder's components and services in any new Silverlight application.

Time for action – from DCC tools to Silverlight

Now, we are going to display the ASE 3D model exported from Blender in a Silverlight application with Balder's help:

1. Stay in the `3DInvadersSilverlight` project.
2. Create a new folder in `3DInvadersSilverlight` (the main project that will generate the XAP output file). Rename it to `Assets`.
3. Right-click on the previously mentioned folder and select **Add | Existing item...** from the context menu that appears.
4. Go to the folder in which you saved the 3D model in the ASE format (`C:\Silverlight3D\Invaders3D\3DModels\SPACESHIP01`). Select the ASE file and click on Add.
5. Click on the ASE file added in the previous step. Change its **Build Action** property to `Resource`.

6. Open the XAML code for `MainPage.xaml` (double-click on it in the **Solution Explorer**) and replace the existing code with the following:

```
<UserControl x:Class="_3DInvadersSilverlight.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="1366" Height="768" >
  <Grid x:Name="LayoutRoot" Background="White" >
  </Grid>
</UserControl>
```

7. Create a new class, `InvadersGame`.
8. Add the following lines of code at the beginning (as we are going to use many Balder's classes and interfaces):

```
using Balder.Core;
using Balder.Core.FlatObjects;
using Balder.Core.Geometries;
using Balder.Core.Lighting;
using Balder.Core.Math;
```

9. Replace the new `InvadersGame` class declaration with the following (it has to be a subclass of `Game`):

```
public class InvadersGame : Game
```

- 10.** Add the following lines to define two private variables:

```
// The spaceship's mesh
private Mesh _ship;
// A light
private Light _light;
```

- 11.** Override the `Initialize` method to change some properties on the main viewport defined by Balder:

```
public override void Initialize()
{
    base.Initialize();
    Display.BackgroundColor = Colors.White;
    Viewport.XPosition = 0;
    Viewport.YPosition = 0;
    Viewport.Width = 1366;
    Viewport.Height = 768;
}
```

- 12.** Override the `LoadContent` method to load the spaceship's mesh, create a light, and define the main camera's target:

```
public override void LoadContent()
{
    base.LoadContent();
    _ship = ContentManager.Load<Mesh>("ship01.ase");
    Scene.AddNode(_ship);

    _light = new OmniLight();
    _light.Position.X = 0;
    _light.Position.Y = 0;
    _light.Position.Z = -30;
    _light.ColorAmbient = Colors.Red;
    _light.ColorDiffuse = Colors.Purple;
    _light.ColorSpecular = Colors.Magenta;
    Scene.AddNode(_light);
    Camera.Target.X = 0;
    Camera.Target.Y = 0;
    Camera.Target.Z = 15;
}
```

- 13.** Override the `Update` method to leave it ready to add some scene management code in it later:

```
public override void Update()
{
    base.Update();
    // TODO: Add code to update the scene
}
```

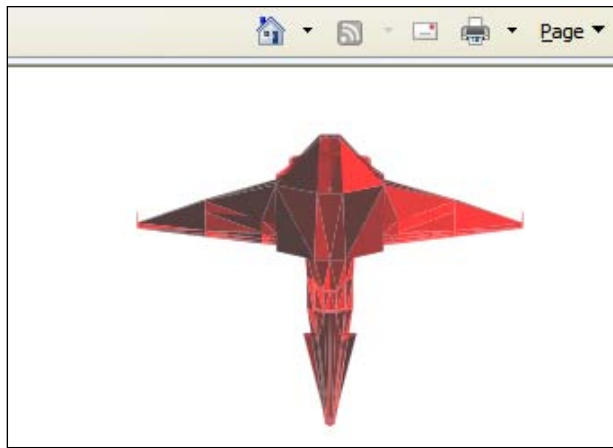
- 14.** Now, expand `App.xaml` in the **Solution Explorer** and open `App.xaml.cs`—the C# code for `App.xaml`. We need to add some code to the `Startup` event handler to initialize Balder with our game class.

- 15.** Add the following lines of code at the beginning (as we are going to use many Balder's classes and interfaces):

```
using Balder.Core.Runtime;
using Balder.Silverlight.Services;
```

- 16.** Add the following lines of code to the event handler for the `Application_Startup` event, after the line `this.RootVisual = new MainPage();`:
`TargetDevice.Initialize<InvadersGame>();`

- 17.** Build and run the solution. The default web browser will appear showing the spaceship colored in red, as shown in the following screenshot:



What just happened?

The 3D digital artist is very happy because the spaceship can be loaded and shown using Silverlight. Now, he trusts you and he will continue to work on exciting 3D models for your game. However, do not be quiet, because you still have to learn many things about cameras and lights.

We showed the spaceship rendered in a Balder's viewport. We had to work a bit more than with the XAML 3D model in the XBAP WPF application. However, we could load and render a 3D model previously exported from a 3D DCC tool using the ASE file format.

One of the drawbacks of Silverlight 3D applications using Balder is that we cannot preview the 3D model during design-time. However, we can experiment with an XBAP WPF application and then, we can work with Silverlight.

Displaying a 3D model in a 2D screen with Silverlight

Once we have the model in the ASE file format and Balder is installed and added as a reference, the steps to load and show a 3D model in Silverlight are the following:

1. Add the ASE model to the `Assets` folder. It must be built as a `Resource`.
2. Override the game class's `LoadContent` method to load the model, creating a new mesh using the `ContentManager`, as done in the following line:

```
_ship = ContentManager.Load<Mesh>("ship01.ase");
```
3. Add the new mesh to the scene using the `AddNode` method:

```
Scene.AddNode(_ship);
```
4. Override the game class's `Update` method to manage the scene, the cameras, the lights, and the meshes:

Before following those steps, we must be sure that we have configured the main viewport, the main camera, and the necessary lights.

In order to start the engine's run-time, Balder requires just one line of code, such as the following one:

```
TargetDevice.Initialize<InvadersGame>();
```

We must replace `InvadersGame` by the main game class (it must be a `Balder.Core.Game` subclass).

Using 3D vectors

Balder works with 3D vectors (`Balder.Core.Math.Vector`) to define positions in the 3D space. For example, the common `Position` property is a 3D vector, with the following fields: `X`; `Y` and `Z`.

We used Balder's 3D vectors to define the position for the light:

```
_light.Position.X = 0;  
_light.Position.Y = 0;  
_light.Position.Z = -30;
```

And, we used another 3D vector to specify the main camera's target:

```
Camera.Target.X = 0;  
Camera.Target.Y = 0;  
Camera.Target.Z = 15;
```

These vectors allow many complex math operations, like translations and transforms. They will simplify the code needed to control perspectives and cameras.

Have a go hero – working with multiple 3D characters

The project manager that hired you wants to see your recent work. He is very happy to know you are able to work with existing 3D models in your new games in XBAP WPF and in Silverlight.

He asks you to prepare a new XBAP WPF application showing a model of a car. He wants the main camera to rotate around the car.

But wait, he also wants a Silverlight version of the same application. You can do it using Balder as the 3D engine.

Remember the elapsed time technique learned when working with 2D characters. You can also use it with 3D models.

Once you finish these new applications, you can add a motorbike and make some effects using different colors in the lights.

Pop quiz – 3D models and real-time rendering

1. Real-time rendering is more efficient when working with meshes that have:
 - a. Millions of polygons
 - b. Hundreds of thousands of polygons
 - c. A low polygon count (less than 2,000 polygons per mesh)

2. Silverlight 3 allows us to load:
 - a. The XAML 3D models using a third party 3D engine—Pentium.
 - b. The ASE 3D models using a third party 3D engine—Balder.
 - c. The 3DS 3D models using Silverlight 3 native controls.
3. A 3D vector represents:
 - a. Three fields: X, Y, and Z.
 - b. Three fields: W, X, and Y.
 - c. Three dimensions using 2 fields: 2D (X and Y) and 3D (X, Y, and Z).
4. Expression Blend allows us to:
 - a. Load and interact with 3D models in XAML 3D format in WPF applications.
 - b. Load and interact with 3D models in 3DS 3D format in WPF applications.
 - c. Import .X 3D models and export them as Silverlight 3D mesh open format.
5. When rendering a 3D scene in a 2D screen:
 - a. We can see the whole 3D scene in 3D.
 - b. We can see a portion of the whole 3D world simulated in a 2D screen, as seen through a specific camera.
 - c. We can see the whole 3D world as polygons without textures.

Summary

We learned a lot in this chapter about 3D models, meshes, and other elements. Specifically, we imported, exported and prepared 3D models using 3D DCC tools in order to load them in our applications. We showed models in XBAP WPF using XAML and in Silverlight applications using the services provided by a specialized 3D engine. We rendered and transformed the meshes in real-time using both hardware-based and software-based rendering processes. We understood the usage of a Silverlight 3D engine and we began controlling some aspects of the 3D elements.

Pop quiz answers

1	2	3	4	5
c	b	a	a	b

Index

Symbols

3D elements

interacting with, Expression Blend used
491

3D model

converting, to XAML file format 469, 471
displaying, in 2D screen with Silverlight
498

exporting 468

3D vectors

3D models, with real-time rendering 499,
500

about 499

multiple 3D characters, working with 499

3D world 481, 482

3D world, in Silverlight

3D model, displaying in 2D screen 498

3D model, exporting to ASE 492, 493

3D vectors, using 499

about 492

ASE 3D model, displaying 495, 497

Balder 3D engine, installing 493

`_context.ExecuteQueryAsync` method 327

`_context.Web.CurrentUser.Id` property 330

`[DataContract]` attribute 88

`[DataMember]` attribute 88

A

`AddBackgroundMusic` method 412

`AddFieldChoicesToComboBoxes` method
334

`AddFieldChoicesToComboBox` method 336

`AddImage` method 401

`AddItem` method 329

`AddItemToList` method 319, 321, 328, 329

`AddObject` method 288

ADO.NET Data Services

about 279, 280

locked-down services 281

used, for persisting data 286-290

`AlreadyRead` property 238

Application Programming Interface (API)
428

ASE 3D model

displaying, in Silverlight application
495-497

ASP.NET application

business object, creating 86-88

`AspNetCompatibilityRequirements`
attribute 93

asset library, SharePoint 2010

content, adding to 364-366

creating 362

creating, steps 363

structures, browsing 367, 368

`AssetsBrowserWebPart` class 374

`Atom Publishing Protocol (AtomPub)` 279,
439

`Author` property 244

`AutoGenerateColumns` property 220, 238

automatic synchronization 156

`AutoPostBack` property 372

Auto sizing 15

B

`backgroundMusic.Source` property 413

Balder 492

Balder 3D engine

installing 493

- basicHttpBinding** 94
- BeginExecute** method 282
- BeginSaveChanges** method 289-291
- binaryHttpBinding** 94
- BindingBase** class 185
- BindingBase** properties
 - converters, replacing with 184-188
- Binding** class 116, 187
- Binding Expression** 116
- BindingsSource** control 132
- BitmapImage** class 379
- Blender**
 - downloading 470
 - installing 470
- boolean** property 242
- BrowserHttpStack**
 - versus ClientHttpStack 272
- business object**
 - creating 86

C

- CakeService.svc.cs** class 90
- CellTemplate** property 242
- change-aware** collection type
 - building 206
 - building, steps 206-208
- class**
 - AssetsBrowserWebPart 374
 - BitmapImage 379
 - Projection 402
- Click** event 149
- ClientAccessPolicy.xml** file 429
- client_DownloadStringCompleted** method 422
- ClientHttpStack**
 - advantages, over BrowserHttpStack 272
 - using, as default 273
 - working with 270-272
- CLR namespace**
 - mapping, to XML namespace 143
- code snippets** 114
- ColumnWidth** property 220
- command area** 38
- Computer** class 288, 290
- computerCollection** 283

- Computer** instances 283
- ComputerInventoryEntities** type parameter 278
- ComputerLoadCompleted** callback method 283
- ConnectAndAddItemToList** method 319, 320, 328
- ConnectAndFillComboBoxes** method 336
- ConvertBack** method 181
- Converter** property 182
- converters** 133
 - about 180
 - replacing, with Silverlight 4 BindingBase properties 184-188
- ConvertStrokesToStrokeInfoArray** method 125
- CreateChildControls** method 374
- Create, Read, Update, and Delete.** *See* **CRUD**
- credentials**
 - passing, to Twitter from trusted Silverlight application 303-307, 313
- Credentials** property 308
- cross-domain** access
 - passing, to Twitter from trusted Silverlight application 303-307, 313
- CRUD**
 - about 286, 316
- CurrencyConverter** class 181
- customBinding** 94
- custom columns**
 - DataGridCheckBoxColumn 238
 - DataGridTemplateColumn 238
 - DataGridTextColumn 238
 - using, in DataGrid 238
- CustomerCakeIdea** business object 127
- CustomerInfo** class 117, 120
- CustomerInfo** object 116

D

- data**
 - applications 86
 - collection 99
 - collection form, creating 99-110

- datainput validating, attributes used 193-195
- datapersisting, ADO.NET Data Services used 286-290
- datapersisting, REST service used 270
- datareading, WCF Data Services used 281-285
- datavalidating, IDataErrorInfo used 195-199
- datavalidating, INotifyDataErrorInfo used 195-199
- datavisualization, customizing 200-206
- submitting, to server 125-129
- validating 111
- Visual State Machine 111
- data access**
 - overview 420
- data access layer, WCF 430, 431, 432**
- data annotations**
 - uses 195
- data binding**
 - about 132, 180
 - ConvertParameter 183, 184
 - creating from Expression Blend 4, steps 173-175
 - creating from Expression Blend 4, working 175, 176
 - defining, terms 132
 - from Expression Blend 4 173
 - images displaying, URL based 184
 - persisting data allowing, modes used 168-172
 - process 132
 - process, steps 181, 182
 - process, working 183
 - to another UI element 148
 - to another UI element, steps 148-150
 - to another UI element, working 151
- data binding, data validation**
 - data object, binding to controls 116-120
- data-bound input**
 - validating, steps 188, 189
 - validating, working 190, 191
- DataContext property 297**
- DataGrid**
 - custom columns, using 237-239
 - data, deleting 225
 - data, displaying 218-223
 - data, filtering 233
 - DataGridCheckBoxColumn, custom column 238
 - DataGridTemplateColumn, custom column 238, 239
 - DataGridTextColumn, custom column 238
 - data, grouping 228, 230
 - data, inserting 226
 - data, paging 235
 - data, sorting 228, 230
 - data, updating 227
 - LoadingRow event 218
 - master-detail, implementing 244-247
 - sorting 229
 - template column, sorting 233
 - validating 248-250
- DataGridCellEditEndedEventArgs parameter 357**
- DataGrid control 218, 247**
- dataGridProjects_CellEditEnded method 355**
- dataGridProjects.SelectedItem property 358**
- data input**
 - data inputvalidating, attributes used 193-195
- data object, data validation**
 - binding, to controls 116-120
 - code snippets 114, 115
 - creating 111-114
- DataPager control 233, 236**
- DataService class 281**
- data templates 155, 180**
- data validation**
 - data, binding 116
 - data input, validating 121-124
 - data object, binding to controls 116-120
 - data object, creating 111-114
- DeleteObject method 290**
- dependency properties 143**

DependencyProperty 116
DirectX Software Development Kit
 downloading 469
 installing 469
Dispatcher.BeginInvoke method 329
Dispatcher class 421
DisplayMode property 236
DockPanel
 about 54, 59
 child element docking order, changing 58
 docking, controlling 55-57
 orientation, changing 58
 working 57
dynamic bindings
 about 145
 creating, steps 145-147
 creating, working 147

E

element binding
 about 148
 without bindings 151
ElementName property 151
ElementStyle property 241
element-to-element binding 148
EndExecute method 283
EndSaveChanges method 289
EntitySetRights enumeration 278
event
 MouseRightButtonDown 402
 OnPreRender 374
ExecuteQueryAsync method 328
Expand method 282
Expression Blend 4
 data binding from 173-176
 used, for generating sample data 176-178
extent 400

F

FallbackValue property 186
Fiddler
 about 424
 URL 424
 using 424

FillBehavior property 403
filtering 236
FirstName property 250
fixed layout
 creating 13
 using, situations 16, 17
 working 14
flash policy file (crossdomain.xml) 426
Flickr
 about 291, 297
 application, building 292-294
 crossdomain.xml file 291
 values, displaying 297
 WrapPanel used 292
flickr.photos.getinfo method 292
flickr.photos.search method 292, 293
fluid layout
 about 10
 columns 14, 15
 creating 11-13
 grid sizing 14, 15
 rows 14, 15
 Star sizing 15
 using, situations 16, 17
 working 13

G

GetByTitle method 329
GetItemId method 358
GetMediaFileType method 401
GetUserTimeLine method 300
GPU 3D acceleration 484

H

HeadersVisibility property 220
HorizontalAlignment property 23
HTTP 420
HTTP classes 420
HTTP POST method 309
HTTPS protocol 420
HttpRequest class 420
HttpResponse class 420

I

IDataServiceConfiguration instance 281
IList interface 208
ImageButton_MouseRightButtonDown
method 402, 406
Image control 239
ImageInfo instance 294
ImageName property 239
InitializeOwner method 135
InitializeService method 278, 281, 443
Ink control 99
InkPresenter controls 100
INotifyCollectionChanged interface 155,
160, 161, 207
INotifyDataErrorInfo interface 199
INotifyPropertyChanged interface 111, 125,
161, 210, 223
interactive animations
working with 401-408
IProductSoapService interface 433
IsReadOnly property 220
ItemsSource property 160, 220, 223, 230, 283
ItemTemplate property 155
IValueConverter interface 211

J

JavaScript Object Notation. *See* JSON
journal navigation
about 59
navigation chrome, removing 63
Title 64
using, need for 64, 65
utilizing 59-61
WindowTitle 64
working 62
JSON
about 88, 439
URL 436
used, for REST service communication 273-
276
JsonDataContractSerializer 88

K

Kit3D 492

L

Language Integrated Query. *See* LINQ
Language property 230
LastName property 193
layout
fixed 10
fluid 10
Line-Of-Business. *See* LOB
LINQ 420
LoadComputer method 284
LoadingRow event 218
LoadProperty method 286
LOB
about 316
as independent WebParts 347-349
expanding, with delete operations 349-352
expanding, with update operations 354-357
item, deleting from list 352
item, updating in list 357-359
LookupId property 330

M

Margin property 22
master-detail implementation 244-247
maximum size 17
media_MediaEnded method 409
meshes
3D elements, adding 488, 489
about 485, 486
XAML exporter, using for DCC tools 486,
487
method
_context.ExecuteQueryAsync 327
AddBackgroundMusic 412
AddFieldChoicesToComboBoxes 334
AddItem 329
AddItemToList 319, 321, 328, 329
ConnectAndAddItemToList 319, 320, 328
ConnectAndFillComboBoxes 336
CreateChildControls 374
dataGridProjects_CellEditEnded 355
Dispatcher.BeginInvoke 329
ExecuteQueryAsync 328
GetByTitle 329

- GetItemId 358
- GetMediaFileType 401
- ImageButton_MouseRightButtonDown 402, 406
- media_MediaEnded 409
- OnConnectSucceeded 320
- OnUpdateFailed 355
- OnUpdateSucceeded 355
- ReturnFieldByInternalName 334
- ShowErrorInformation 327
- MFC (Microsoft Foundation Classes) 132**
- minimum size 17
- Mode property 151, 171, 238**
- MouseRightButtonDown event 402**
- MSDN website**
 - URL 89

N

- NavigateUri property 402**
- navigation pane**
 - components, buttons 17
 - components, content 17
 - content, hosting into specific tabs 22-25
 - creating, from scratch 19-21
 - tab control, using 18
 - using, situation 25, 26
 - working 22
- NetDataContractSerializer 88**
- netTcpBinding 94**
- network security 425**
- NoMoreCustomerSince property 185**

O

- ObservableCollection 283**
- OData**
 - about 439
 - data services, building 441-444
 - external service, consuming 444, 445
 - URL 440
- OData data services**
 - exploring 440
- OData service**
 - building 441-444
- OnConnectSucceeded method 320**
- OneWay bindings 161**

- OnPreRender event 374**
- OnPropertyChanged method 114**
- OnUpdateFailed method 355**
- OnUpdateSucceeded method 355**
- Owner class 185**

P

- PageSize property 235**
- paging 236**
- parameter**
 - DataGridCellEditEndedEventArgs e 357
- persisting data**
 - allowing, data binding modes used 168-172
- PreferredOwner class 186**
- PreferredSince property 187**
- preferred size 17**
- Products.xml file 422**
- progressive disclosure**
 - using, conditions 52, 54
 - working 51
- Projection class 402**
- property**
 - _context.Web.CurrentUser.Id 330
 - about 358
 - AutoPostBack 372
 - dataGridProjects.SelectedItem 358
 - InternalName 354
 - LookupId 330
 - NavigateUri 402
 - propertyShowInTaskbar 28
 - SchemaXml 332
 - SelectedList 397, 398
 - selectedProject.Title 358
 - TargetName 402
- PropertyChanged event 157**

R

- Representational State Transfer. See REST**
- SizeMode property**
 - about 32
 - styles, setting to 32
- ResponseFormat NamedParameters 274**
- REST 436**

REST service
about 436
building 436-439
communicating with, JSON used 273-276
used, for persisting data 270
ReturnFieldByInternalName method 334
Rich Internet Application (RIA) 419
RowHeight property 220

S

sample data
generating, Expression Blend 4 used 176-178
SchemaXml property 332
second remake assignment
3D elements, interacting with 491
3D model, exporting 468
3D world 481
about 468
GPU 3D acceleration 484
meshes 485
three-dimensional coordinate system 483
XAML 3D models 473
XBAP WPF applications with 3D content 476
security policy files, Silverlight
ClientAccessPolicy.xml 426
crossdomain.xml 426
SelectedList property 397, 398
selectedProject.Title property 358
ServerRelativeUrl property 401
ServiceObjects class 87
services
building, with WCF 427, 428
set accessor method 120
SharePoint
themes, changing 414-417
SharePoint 2010
asset library, creating 362
rich media management, improving 362
SharePoint package file 392
ShowErrorInformation method 327
ShowInTaskbar property 28
Silverlight
about 419

basicHttpBinding 94
binaryHttpBinding 94
core networking classes 420
customBinding 94
data access overview 420
data object, binding 116-120
data validating, IDataErrorInfo used 195-199
data validating, INotifyDataErrorInfo used 195-199
features, merging 209-215
netTcpBinding 94
security policy files 426
validation error, scenarios 120
Silverlight 3
ClientHttpStack 270-272
Silverlight 4
supported audio formats 413
supported video formats 412
Silverlight application
code, cleaning up 162, 163
collections 161
data displaying, steps 134-140
data displaying, working 141-143
data persisting, REST service used 270
enabling to automatically update UI, steps 156-160
enabling to automatically update UI, working 161
Flickr 291
REST service communicating with, JSON used 273-276
single objects 161
Twitter, over REST 298
WCF Data Services, using 276, 278
Silverlight-enabled WCF service
creating 90, 5, 90, 91, 93, 96, 97, 98
Silverlight policy files (ClientAccessPolicy.xml) 426
Silverlight RIA
linking, to Visual Web Part 390-393
themes, modifying 414-417
Silverlight RIA, included in a SharePoint solution
asynchronous operations, working with 327-330

- complex LOB applications, creating 340-344
- data, managing 316
- item insertion, Silverlight Web Part used 324-327
- LOB systems, as independent WebParts 347-349
- LOB systems, expanding with delete operations 349-352
- LOB systems, expanding with update operations 354-357
- multiple Silverlight Web Parts in same page, interacting with 345, 346
- Silverlight Client Object Model, working with 316-323
- specific field information, retrieving 331- 340
- Silverlight RIA, rendered in SharePoint Visual Web Part**
 - creating 376-390
- SOAP service**
 - about 433
 - building 433, 435
- socket-based networking classes 420**
- SortMemberPath property 233**
- Source property 184, 239**
- Star sizing 15**
- startup object 60**
- status bar**
 - about 75
 - adding, steps 75-77
 - implementation guidelines 82, 83
 - other controls, adding 78-80
 - SizeGrip, adding 81
 - using 81, 82
 - working 78
- StringFormat property 186**
- StringLength attribute 194**
- StrokeInfo[] array 127**
- StrokeInfo objects 125**
- submitButton_Click method 127**
- SubmitCakeIdeaCompleted event 128**
- System.ComponentModel namespace 161**
- System.Net.Sockets namespace 420**
- System.Windows.Controls namespace 217**

T

tabs

- about 66
- adding 66-68
- icons, adding 70-72
- implementation guidelines 73, 74
- orientation, changing 70
- using 72
- using, queries 72
- working 69

- Tag property 248**
- TargetName property 402**
- TargetNullValue property 187**
- target property 143**
- TextBlock controls 227, 245**
- Text property 136**
- three-dimensional coordinate system 483, 484**
- Tick event 160**
- ToString method 223**
- Trusted Silverlight application**
 - creating 312
 - credentials, passing 313
 - credentials, passing to Twitter 303-311
 - cross-domain access, passing to Twitter 303-311
- TweetUpdate class 307**
- Twitter**
 - about 298
 - credentials, passing from trusted Silverlight application 303-311
 - cross-domain access, passing from trusted Silverlight application 303-311
 - running, steps 299-302
 - versus Flickr 298
- TwitterUpdate class 300**
- TwoWay bindings 151, 171, 172**

U

UI elements

- collections, binding 152
- collections binding, steps 152-154
- collections binding, working 155

- data obtaining from, steps 163, 166
- data obtaining from, working 167, 168
- UI pattern**
 - implementing, in WPF application 49
- UI pattern, implementing in WPF application**
 - expander control's header label 51, 52
 - steps 49, 50
 - working 51
- UpdateObject method 290**
- using statement 421**

V

- ValidationSummary class 191**
- VerticalAlignment property 24**
- videos**
 - adding 408-411
 - controlling 408-411
 - formats, in Silverlight 4 412
- viewport 400**
- Visual Web Part**
 - about 369
 - adding, in Web page 393-399
 - controls, organizing 399, 400
 - creating 369-375
 - files, reading from assets library 400, 401
 - linking to Silverlight RIA 390-393

W

- WCF**
 - about 420
 - core services 426, 427
 - data access layer 430-432
 - data services 427
 - RIA services 427
 - services, building with 427, 428
 - Silverlight enabled service, creating 90-98
 - working with 429
 - wsHttpBinding 94
- WCF Core Services 427**
- WCF Data Services**
 - about 427
 - used, for reading data 281-285
 - using, with Silverlight 276, 278

- WCF RIA Services 427**
- WebClient**
 - working with 420-423
- WebClient class 420**
- WebGet attribute 89**
- WebRequest.Register method 313**
- window management**
 - about 27
 - contextual window 27
 - general window usage, guidelines 36, 37
 - icon, changing 28
 - minimum screen resolution target, determining 35
 - primary window (top-level) 27
 - program-initiated 27
 - SizeMode property 32
 - ResizeMod, setting 28
 - secondary window(owned) 27
 - states 30
 - system-initiated windows 27
 - Title bar controls 28
 - window borders 28
 - window sizes 29, 34
 - window states 34
 - WindowState property 32
- Windows Communication Foundation. See WCF**
- Windows Media Video. See WMV**
- Windows Presentation Foundation. See WPF**
- WindowStartupLocation property**
 - about 32, 35
 - CenterOwner option 35
 - CenterScreen option 35
 - manual option 35
- WindowState property**
 - about 30
 - maximized state 34
 - minimized state 34
 - normal state 34
 - window positioning, using 35
 - WindowStartupLocation property 35
- WindowState property**
 - about 29
 - styles, setting to 33

wizards

- about 37
- building 37-40
- resizable wizard, designing 47
- types, overview 42
- using, guidelines 48, 49
- using, situations 47
- working 41

wizards, types

- choice page(s) 44
- commit page 44
- follow-up page 46
- Getting started page 43
- progress page 45

WMV 364**WPF**

- about 27
- UI pattern, implementing 49

wsHttpBinding 94**WSP package 392****X****XAML 3D models**

- about 473
- displaying, in XBAP WPF application 475, 476
- features 474

XAML Exporter

- downloading 470

XAML parser 142**XBAP WPF applications**

- 3D elements, adding 488

XBAP WPF applications, with 3D content

- 3D model, displaying in 2D screen 476-480

XLINQ 421, 430**XML namespace**

- mapping, to CLR namespace 143

XmlSerializer class 270



**Thank you for buying
Managing Data and Media in Microsoft Silverlight 4: A mashup
of chapters from Packt's bestselling Silverlight books**

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.

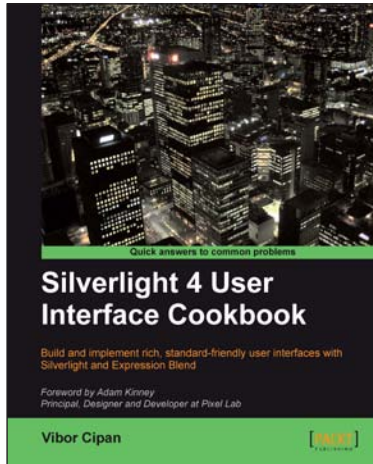
About Packt Enterprise

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.



Silverlight 4 User Interface Cookbook

ISBN: 978-1-847198-86-0 Paperback: 280 pages

Build and implement rich, standard-friendly user interfaces with Silverlight and Expression Blend

1. The first and only book to focus exclusively on Silverlight UI development
2. Have your applications stand out from the crowd with leading, innovative, and friendly user interfaces
3. Detailed instructions on how to implement specific user interface patterns together with XAML and C# (where needed) code, and explanations that are easy-to-understand and follow



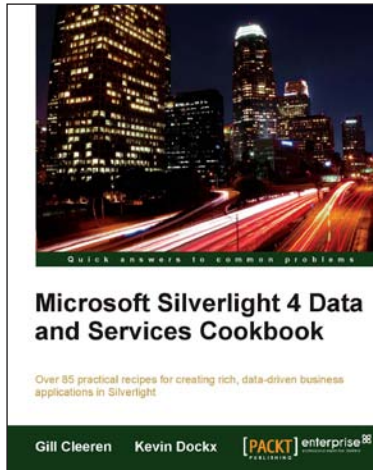
Microsoft Silverlight 4 Business Application Development: Beginner's Guide

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

Please check www.PacktPub.com for information on our titles

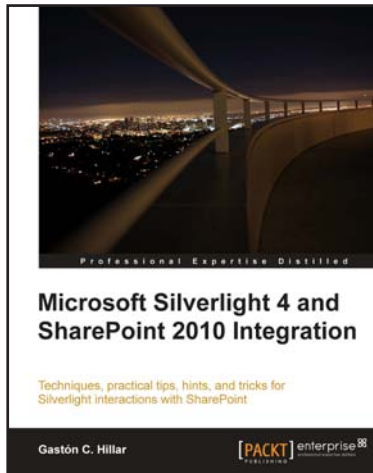


Microsoft Silverlight 4 Data and Services Cookbook

ISBN: 978-1-847199-84-3 Paperback: 476 pages

Over 85 practical recipes for creating rich, data-driven business applications in Silverlight

1. Will get the reader developing applications for processing XML in JDeveloper 11g quickly and easily
2. Self-contained chapters provide thorough, comprehensive instructions on how to use JDeveloper to create, validate, parse, transform, and compare XML documents
3. The only title to cover XML processing in Oracle JDeveloper 11g, this book includes information on the Oracle XDK 11g APIs



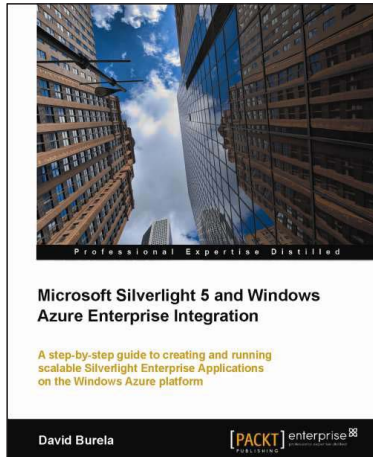
Microsoft Silverlight 4 and SharePoint 2010 Integration

ISBN: 978-1-849680-06-6 Paperback: 336 pages

Techniques, practical tips, hints, and tricks for Silverlight interactions with Sharepoint

1. Develop Silverlight RIAs that interact with SharePoint 2010 data and services
2. Explore the diverse alternatives for hosting a Silverlight RIA in a SharePoint 2010 Page
3. Work with the new SharePoint Silverlight Client Object Model to interact with elements in a SharePoint Site
4. Use Visual Studio 2010's new features to debug Silverlight RIAs that interact with SharePoint 2010

Please check www.PacktPub.com for information on our titles

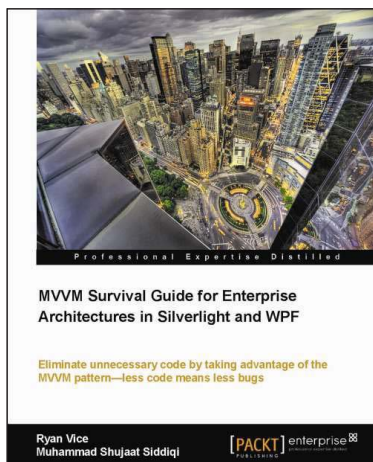


Microsoft Silverlight 5 and Windows Azure Enterprise Integration

ISBN: 978-1-84968-312-8 Paperback: 361 pages

A step-by-step guide to creating and running scalable Silverlight Enterprise Applications on the Windows Azure platform

1. This book and e-book details how enterprise Silverlight applications can be written to take advantage of the key features of Windows Azure to create scalable applications
2. Provides an overview of the Windows Azure platform and how the different technologies can be integrated within your enterprise application
3. Examines ways that distributed asynchronous systems can be created to allow scalable processing



MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF

ISBN: 978-1-84968-342-5 Paperback: 412 pages

Eliminate unnecessary code by taking advantage of the MVVM pattern—less code means less bugs

1. Build an enterprise application using Silverlight and WPF, taking advantage of the powerful MVVM pattern, with this book and e-book
2. Discover the evolution of presentation patterns—by example—and see the benefits of MVVM in the context of the larger picture of presentation patterns

Please check www.PacktPub.com for information on our titles

