



Professional Expertise Distilled

www.SoftGozar.com

Telerik WPF Controls Tutorial

Create powerful WPF applications using Telerik controls with the help of real-world examples

Daniel R. Spalding

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.SoftGozar.com

Telerik WPF Controls Tutorial

Create powerful WPF applications using Telerik controls with the help of real-world examples

Daniel R. Spalding

[PACKT] enterprise 
PUBLISHING professional expertise distilled
BIRMINGHAM - MUMBAI

Telerik WPF Controls Tutorial

Copyright © 2014 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 author, 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 2014

Production Reference: 1140214

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

ISBN 978-1-78217-652-7

www.packtpub.com

Cover Image by Tony Shi (shihe99@hotmail.com)

www.SoftGozar.com

Credits

Author

Daniel R. Spalding

Project Coordinator

Aboli Ambardekar

Reviewers

Dr. Brian Finnegan

Jiri Pik

Bill Youngman

Proofreader

Maria Gould

Indexers

Mehreen Deshmukh

Rekha Nair

Acquisition Editors

Amarabha Banerjee

Harsha Bharwani

Production Coordinator

Kyle Albuquerque

Content Development Editor

Shaon Basu

Cover Work

Kyle Albuquerque

Technical Editors

Abhishek Kanade

Menza Mathew

Copy Editors

Sayanee Mukherjee

Laxmi Subramanian

About the Author

Daniel R. Spalding is a software consultant who works with many companies in the Delaware Valley area of Pennsylvania. His software has been used by several companies as their primary source for running their businesses. He has consulted with several large firms on software architecture and product prototypes using Visual Studio and C#. He started his business in 1996, but worked for Bell Atlantic for the first 5 years of his career where he was in charge of the Lotus Notes Center of Excellence for Philadelphia.

He has also been an adjunct professor for the last 19 years at both Drexel University and Peirce College working with students in all aspects of computing from software development to networking certification.

I would like to thank my wife Cindy, and my family, for putting up with the time and effort this book took to complete. I would also like to thank the technical reviewers, Bill Youngman and Brian Finnegan, for volunteering their time to review my work, and fix all my mistakes.

About the Reviewers

Dr. Brian Finnegan is an associate professor of Information Technology at Peirce College where he teaches courses in database management, human-computer interaction, and programming.

Jiri Pik is a finance and business intelligence consultant working with major investment banks, hedge funds, and other financial players. He has architected and delivered breakthrough trading, portfolio and risk management systems, and decision-support systems across industries.

His consulting firm, WIXESYS, provides their clients with certified expertise, judgment, and execution at the speed of light. The power tools of WIXESYS include revolutionary Excel and Outlook add-ons available at <http://spearian.com>.

Bill Youngman graduated from the University of Kansas in 1983 with his BA in Psychology and after that spent almost 10 years as a radar/computer technician in the U.S. Navy. Upon leaving the Navy, he moved to Philadelphia where he began his career as an application developer with a multimedia company.

He spent the next 20 years as a consultant working in the healthcare, pharmaceutical, insurance, and financial industries working on web applications and computer-based training systems before finally ending up at PJM Interconnection where he has been working since 2008.

While at PJM, he earned his Master's degree in Software Engineering at Pennsylvania State University and is currently a solutions architect in the System Planning and Applied Solutions Applications group where he supports PJM's system planning efforts for the electrical utility grid for the Mid-Atlantic region as well as working with the Applied Solutions group researching and developing new technologies at PJM.

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](https://twitter.com/PacktEnterprise) on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Getting Started with Telerik RadControls	7
Prerequisites	10
Downloading the Telerik trial software	10
Creating the first Telerik project	13
Telerik demo project	15
Summary	17
Chapter 2: Telerik Editors and How They Work	19
Database setup	20
RadAutoCompleteBox with data binding	22
Binding to a System.Data.DataTable	23
Binding to the CustomerEntity object	26
RadMaskedInput – currency, phone, and zip	27
RadMaskedInputCurrency	27
RadMaskedInputText – phone and zip	28
RadMaskedInput – e-mail and validations	30
SpellChecker – TextBox	31
RadSpellChecker with VS TextBox	32
RadSpellChecker with RadRichTextBox	33
RadSpellChecker with RadDataGrid	34
Summary	36
Chapter 3: Data Entry and Validation of Telerik Controls	39
Database setup	40
RadComboBox	40
RadComboBox with DataTable	41
RadComboBox with a generic list	43

RadSpreadsheet	44
Dynamic validation	49
Summary	56
Chapter 4: Layout Organization and Display Functionality	59
Database setup	60
RadTabControl	60
RadTabControl with DataTable	61
RadTabControl with a generic list	64
RadTabControl with an XML file	64
RadBook	67
Summary	71
Chapter 5: Navigation and Dynamic Event Handling	73
Database setup	74
Additional prerequisites	74
New WPF concepts	74
BaseWindow	75
The AppRequest persistence class	78
The UserEntity class	80
The BarItem class	83
RadOutlookBar	84
RadOutlook with GenericList, DataBinding, and database security	85
RadOutlookBar using generic list binding with XML security	88
RadMenu	89
Summary	94
Chapter 6: Telerik Scheduling and Object Bound Loading	97
New object-oriented concepts	98
The IGanttTask interface class	98
The CommonTask class	99
RadGanttBar	100
RadGanttView with user task filtering	101
RadGanttView displaying the tasks with a summary task	104
Summary	106
Index	107

Preface

This book aims to demonstrate the use of the Telerik RadControls within a Windows Presentation Foundation (WPF) application. The book will work with four aspects of RadControls:

- Data Entry Controls
- Navigation Controls
- Scheduling Controls
- Layout Organization Controls

The book will also review several key aspects of loading these controls using data objects and XML serialization. The last feature the book will cover is how to validate data within RadControls and Visual Studio controls.

What this book covers

Chapter 1, Getting Started with Telerik RadControls, discusses the process of installing the Telerik controls, then verifying the installation and making sure the controls are loaded into Visual Studio and the Control ToolBar. By the end of the chapter, the Telerik controls should be available in Visual Studio and be ready for use in a WPF project.

Chapter 2, Telerik Editors and How They Work, reviews the listed controls and discusses how to load the values and properties in a bound and unbound technique. The spell-check control is added to other controls to allow for spell-check on several controls. By the end of this chapter, the reader should be able to use the selected controls in both a bound and unbound mode. In addition, the reader will be able to implement the spell-check controls for use with the accompanying text editing controls.

Chapter 3, Data Entry and Validation of Telerik Controls, reviews the selected controls and discusses the creation and loading of these controls. The chapter also discusses how to validate data in each control, and how to allow an object class to act as the validation tool for these controls. By the end of this chapter, the reader should be able to load the data entry controls and validate data within the control for the correct data. In addition, the reader should be able to use object attributes to validate data within any control.

Chapter 4, Layout Organization and Display Functionality, discusses the Telerik container controls and how to efficiently design with the container controls. The chapter also discusses how to load this information in a dynamic format from either the database or from configuration files. By the end of this chapter, the reader should be able to design the Telerik container controls and allow the controls to be built in a dynamic format.

Chapter 5, Navigation and Dynamic Event Handling, reviews the Telerik navigation controls and how to take a DataSet or object list and bind the information to the controls. The chapter will also cover how to handle dynamic creation of events from data. By the end of the chapter, the reader should be able to create a collection of objects or a DataSet and generate entries inside the Telerik navigation controls. The reader should also be able to dynamically populate the events for handling navigation based on the user selection.

Chapter 6, Telerik Scheduling and Object Bound Loading, reviews the scheduling controls from Telerik and discusses how to load these controls from data in a database. This chapter will also discuss how to take information from controls and load the values into the new data repository. By the end of the chapter, the reader should be able to create and load these controls from any data repository, and take the information from the controls to load into another data repository.

What you will need for this book

You will need to download the Telerik RadControls trial version from the Telerik website at <http://www.telerik.com/products/wpf/download.aspx>. It is recommended to download the MSI installer. You would also have to register yourself with Telerik to allow access to the MSI file.

The reader must have at least Visual Studio 2010 Express installed to take advantage of the book's examples. The projects that are referred to in the book are in the .NET Framework 4.5 version. Please make sure to set the projects to the 4.0 or 4.5 Framework to ensure the best results. The operating system used in this book is Windows 7 Ultimate, but these examples should also run on Windows 7 or Windows 8.

The database manager for the projects in this book will be SQL Server 2008 Express and it also uses XML files. The Telerik installation process will install the SQL Server as well. The book's code examples will include both a SQL Server backup file and the SQL Server MDF file for use within the application.

Who this book is for

This book is targeted at developers and architects who currently have a working knowledge of WPF, but are looking to work with the Telerik RadControls to build a new application. The book reviews the Telerik RadControls as well as techniques for using the controls. Also, it discusses advanced techniques for working with the WPF Window DataContext property.

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: "...RadAutoCompleteBox, both in a Data Bound and Object Bound mode".

A block of code is set as follows:

```
private void checkDataType_Click(object sender, RoutedEventArgs e) {
    try {
        if (checkDataType.IsChecked == true) {
            RadComboCustomer.DisplayMemberPath = "FullName";
            RadComboCustomer.SelectedValuePath = "Id";
            RadComboCustomer.ItemsSource = cust.Fetch().DefaultView;
        }
        else {
            cust.FetchList();
            this.DataContext = cust;
        }
    }
    catch (Exception ex) {
        // log your error
    }
}
```

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



```
void Application_StartUp(object sender, StartupEventArgs e)
{
    //1. Create UserEntity instance
    user = new ObjectLayer.Util.UserEntity();



    //2. Set the type of security
    ObjectLayer.Util.UserEntity.SecurityType type = GetSecurityType(Config("SecurityType", "DB"));
    AppRequest = new Request();

    try
    {
        //3. Authenticate the user
        if (type == ObjectLayer.Util.UserEntity.SecurityType.AD)
        {
            _user.UserADGroups.Add("Order");
            _user.FetchXMLMenuItems();
            _user.Authenticate(GetUser(Environment.UserName));
        }
        else
        {
            _user.UserName = "dantest01";
            _user.UserCheckPwd = "dantest01";
            _user.Authenticate();
        }

        //3. Set the user to the Request class property for the CurrentUser
        AppRequest.CurrentUser = _user;
        //4. Pass the Request class to the instance of the Window
        MenuWindow mainWindow = new MenuWindow(AppRequest);
        if (_user.UserLoggedIn)
        {
            mainWindow.Show();
        }
    }
}
```

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: "Right-click on the **Databases** tree option and select **Restore Database**."

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for 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.

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-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

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

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

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

Questions

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

www.SoftGozar.com

1

Getting Started with Telerik RadControls

During my history as a developer and architect, I have always worked under the assumption that the controls in Visual Studio would be enough to develop any application. The reason I always worked with the common controls in Visual Studio was that these controls are updated automatically with the .NET framework the moment the version of the .NET framework is installed; so, there is never a problem of having to maintain the control version. Third-party controls can tend to be expensive, sometimes have a steep learning curve, and can be expensive to maintain throughout the life cycle of the application. When I needed more functionality, I extended the Visual Studio version of the control to meet the needs of the application. This approach works fine if the application is meant for a small audience such as an internal company application or small company application, where costs need to be contained as much as possible. This scenario needs to be decided by the developer in conjunction with the potential client.

If the plan for the application includes selling it to a wider audience, or if the application needs additional functionality without the extra work of extending each control, then using third party controls is a great way to cut the development time of the application while gaining additional functionality required in the application. Telerik RadControls can be a great option for this type of development. Telerik offers a large suite of .NET controls for WinForms, ASP.NET, and WPF.

This book will cover Telerik RadControls for WPF. RadControls can cut development time by 50 percent over the task of extending the existing Visual Studio standard controls, and since the controls are styled, the look and feel of the application will be enhanced by the use of RadControls.

This book expects that you already have experience with WPF and Visual Studio. You should also have some experience working with SQL Server and SQL. This experience will be helpful when deploying the database from the downloads in the book. The main focus of the book, along with RadControls, will be the use of object classes to bind a data object to the WPF controls. Here are the WPF concepts you should be familiar with in order to work with this book:

- Object Oriented Design as well as the concept of the DataContext property in a WPF form. If you need to do a quick review of WPF and DataContext, you can review the following page at the <http://msdn.microsoft.com/en-us/library/ms752347.aspx> site. Also, if you are new to or lack the knowledge of object oriented design, review this website before starting on the book: <http://cplusplus.about.com/od/introductiontoprogramming/a/aboutoop.htm>.
- DataContext binding information to support Telerik RadControls. These projects are available for download from the <http://www.packtpub.com/support> website along with the SQL Server database to support the following projects:
 - A class library that has object classes to support object data binding using the DataContext property in the WPF.
 - A class library that will act as the data layer for the application. This library will be called directly when the DataContext property uses a data object to bind. If the DataContext property uses an object from this class library, the class object will call the data layer.

Please make sure to download these files before reviewing the chapters in this book since the libraries are a major portion of the work with RadControls.

Telerik RadControls for WPF offer several different categories of RadControls for use within WPF:

- **Data Management:** This category provides the editable options to display data from a data store, which include options such as a GridView and TreeListView
- **Data Visualization:** This category provides the static options to display data, which include options such as a BarCode or Chart
- **Editors:** This category provides the data entry options to gather input from the user, which include options such as a ComboBox or MaskedInput

- **Layouts:** This category provides the display options such as a `Book` or `TileList`
- **Navigation:** This category provides the options to create navigation in the application such as `PanelBar` or `OutlookBar`
- **Interactivity:** This category provides the options to handle special media
- **Scheduling:** This category provides the options to create scheduling information with `Timeline` and `ScheduleView`
- **Framework:** This category provides the options for working with data with `PersistenceFramework` and `EntityFrameworkDataSource`

`RadControls` give the developer a great option to create a fully functional, well-styled, and full-featured application.

This book discusses the advantages of working with Telerik `RadControls` for WPF. The design of this book is a step-by-step methodology for working with selected Telerik controls. Each chapter discusses a different type of `RadControl`, such as `Editors`, and breaks down how to use the control in both bound and unbound methodologies.

The book will use C# as the language for the projects. We refer to VB.NET but the examples will be in C# only. If you are not familiar with C#, but still want to use the book, there is a great website for translating C# to VB.NET, which can be found at http://www.harding.edu/fmccown/vbnet_csharp_comparison.html. Telerik also has a site for handling the translation, which can be found at <http://converter.telerik.com/>. This site allows you to take a snippet of code and translate the code to VB.NET or C# depending on your preference.

Since this book is meant to be a tutorial for `RadControls`, we will stay away from selecting a design pattern for the use of the controls, and focus on the use of the controls. The choice of a design pattern should be based on the best option for the type of application you are developing within WPF. The options for using a pattern are discussed but we have decided to remove the discussion of design patterns. The **Model View View Model (MVVM)** pattern is the most commonly used design pattern for WPF applications. If you want to understand the pattern further, please review the following page:

[http://msdn.microsoft.com/en-us/library/gg405484\(v=pandp.40\).aspx](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx).

Prerequisites

The reader must have at least Visual Studio 2010 Express installed to take advantage of the book's examples. The projects that are referred to in the book are in the .NET framework 4.5 version. Please make sure to set the projects to the 4.0 or 4.5 framework, to ensure the best results. The example operating system is Windows 7 Ultimate, but these examples should run on Windows 7 or 8.

The sample projects in this book will use both, a SQL Server 2008 Express database and XML files, as data sources for the examples. The database and XML files will be included with the sample projects. The Telerik installation process will install the SQL Server as part of the complete installation. The book consists of projects to support the `DataContext` information in WPF. These projects, along with the database files, can be downloaded from Packt Publishing's website at <http://www.packtpub.com/support>.

The book will cover the 2013.2.0724 version of Telerik RadControls. At the time of the development of this book, this version was the latest version of `RadControls`.

At the time of writing this book, the current version of Telerik RadControls was Version 2013.2.0724.

The license for Telerik RadControls is a 30-day free trial license. The reader will **not** be able to create an installation of any application without purchasing the Telerik license.

Downloading the Telerik trial software

The first step to start using Telerik RadControls is to download the trial for WPF from the Telerik website. There are over 55 controls from Telerik with this license. The following is the link:

<http://www.telerik.com/products/wpf/download.aspx>

It is recommended that you download the MSI installer as shown in the following screenshot:

Download RadControls for WPF

PRODUCTS > WPF CONTROLS > DOWNLOAD WPF CONTROLS

The fully functional 30-day trial gives you access to:

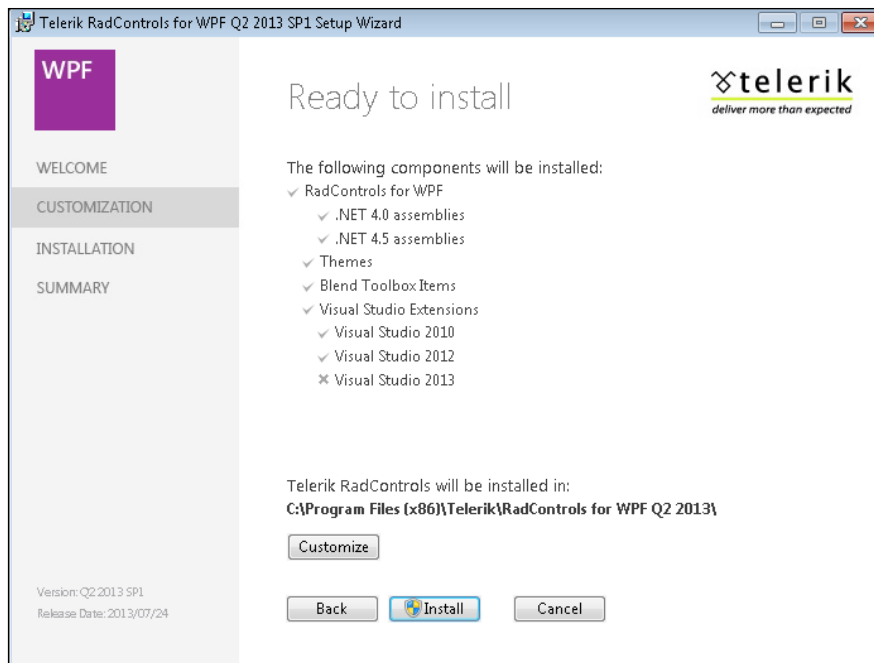
- 55+ WPF UI controls for .NET 4.0/4.5 desktop applications
- 10 ready-to-use themes, including Windows8Touch and Windows8 theme
- Over 300 demos with source code
- 72h turn-around support (24h with a purchase)

Download RadControls for WPF
Bootstrap Installer EXE

Your firewall blocks the bootstrap installer? Download the MSI installer (requires registration).

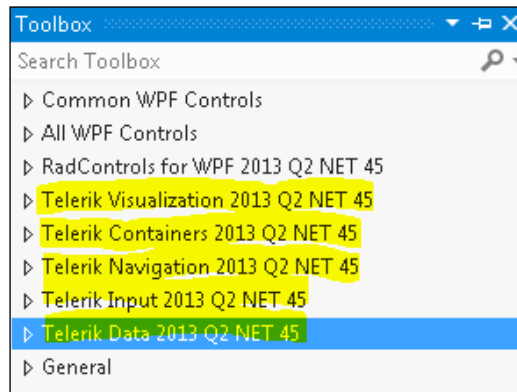
The download requires registration with Telerik to allow access to the MSI file. Fill in the information and the site will redirect you to the download page and begin to download the demo MSI file.

Once the file is downloaded you can start the installation. The Telerik installation will display the typical installation menu. Select the default installation making sure that you select your version of Visual Studio as part of the installation.



Once your version of Visual Studio has been selected, you can customize the location of the installation as you see fit. The installation will not take more than 10 minutes, but the process will require an Internet connection to verify the installation with the Telerik licensing site.

The next step is to do a simple verification to see whether the installation of the Telerik controls has been completed. The first step will be to create a new WPF project in Visual Studio 2012. Once the project has been created, the Telerik controls should display in the Visual Studio Toolbox as shown in the following screenshot:



If there is a problem with the installation and the controls are not displaying, you can right-click on the **Toolbox**, select **Choose Toolbox items**, and navigate to the Telerik installation to select the Telerik DLL libraries. If the Telerik libraries are not installed within the selected Telerik folder, contact the Telerik support site at <http://www.telerik.com/support.aspx> and enter a support ticket. Telerik does have very good technical support and will contact you with a ticket number for your issue.

Telerik also has an excellent blog for contacting Telerik support or other Telerik developers to discuss any issues that you may have as part of the development process. The Telerik blog is located at <http://blogs.telerik.com/>. The blog is broken down by the type of control or issue and can be very helpful when you develop a problem during your working process.

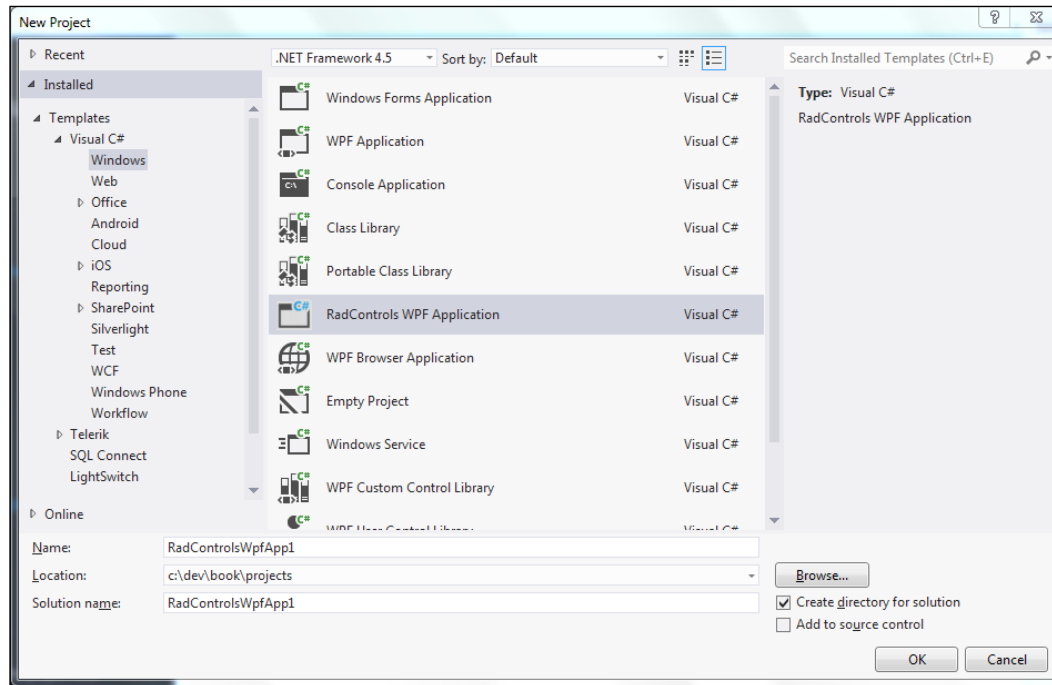
If you need more than those 30 days, which the Telerik trial license allows to work with RadControls, you can contact Telerik to extend the licensing. Telerik will often give you up to 90 days to allow for a full evaluation of RadControls.

Creating the first Telerik project

The next step in the installation process is to create a WPF project within Visual Studio to handle the final verification that the Telerik installation has been successful.

First, open Visual Studio by right-clicking on the desktop icon or start menu option and select **Run as Administrator**. This will allow you to access the registry during the creation of the WPF project. The Telerik installation needs to access the registry to set up the controls in the Toolbox.

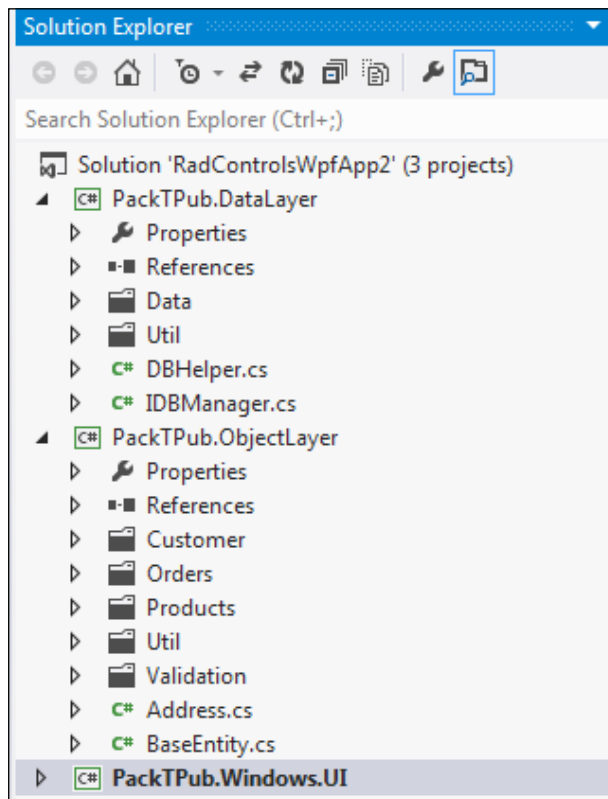
Second, create a RadControls WPF/C# project in Visual Studio. To do this, create a new project, select C# as the language, and the .NET framework should be 4.5. Please refer to the following screenshot:



The Telerik project wizard should display in such a way that it allows you to select the controls you want to use within the project. This project will only require the Telerik.Windows.Controls and the Telerik.Windows.Controls.Input assemblies. If you do not see the Telerik project wizard, the installation is not correct. In this case, please uninstall the Telerik controls and retry the installation. If this problem persists, create a support ticket with Telerik at <http://support.telerik.com>.

Once the project has been created, open the `MainWindow.xaml` file in the designer within Visual Studio, then open the `Toolbox` and drag the `RadMaskedEditTextInput` control from the Telerik Input 2013 Q2 NET 45 option in the `Toolbox`. Once this control is dragged to the `MainWindow.xaml` (the `Mask` property should already be set to `a20`) run the project from the debugger. The project should build without an issue, and warn you that the Telerik controls are in demo mode.

Now that you have been able to create a project with Visual Studio and Telerik, the next step is to add the projects from the book's website to the solution for the WPF project. First, you right-click on the solution and select **Add Project** from the **File** menu option. Then, go to the location of the projects that you have downloaded from the Packt Publishing website at www.packtpub.com, and select the PackTPub data layer project. The namespace should be `PackTPub.DataLayer`. Arrange the hierarchy for the projects so that the `PackTPub.DataLayer` project should be highest in the compile list. The next project is the `PackTPub.ObjectLayer` project, then the Telerik `PackTPub.Windows.UI` project. The solution should now look like the following screenshot:

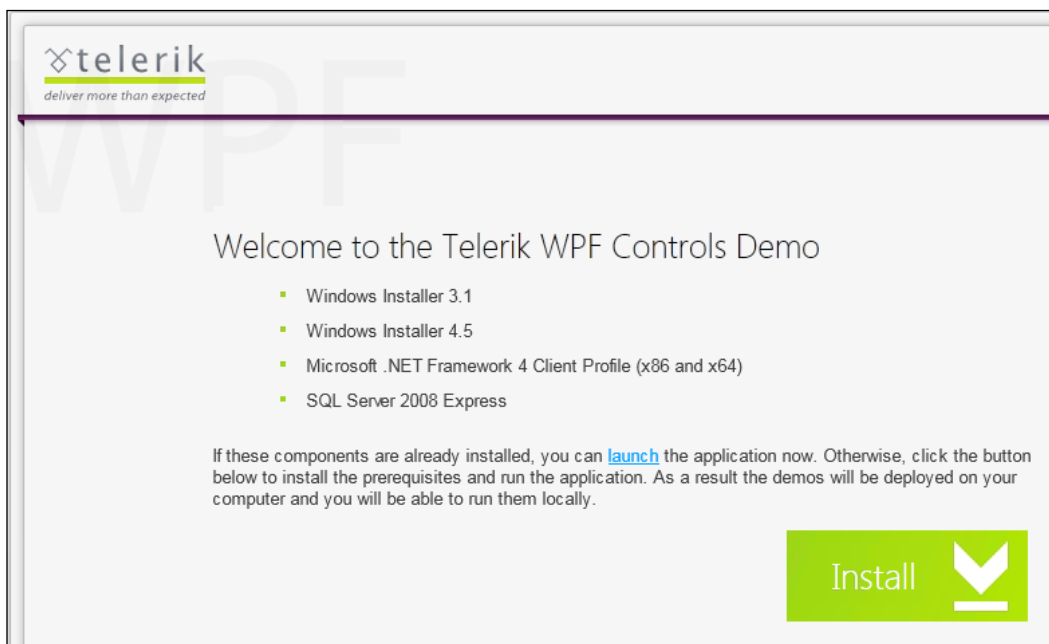


There should be three projects in the solution. Once you have added the projects to the solution, make sure to rebuild the project and verify that the projects are complete. If there are any compile errors, please try to redownload the files and step through the process again. If these problems persist, please contact www.packtpub.com/support with the issue.

As a reminder, since the Telerik controls are demo versions, you will not be able to create an installation or deployment for these examples. The Telerik installation will not pass the licensing to the installation and the application will fail.

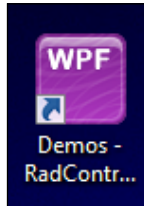
Telerik demo project

Telerik offers a complete Visual Studio project as a demo of all `RadControls` and how to use the controls within WPF. You can download the demo project and solution from the Telerik website at <http://demos.telerik.com/wpf/>.

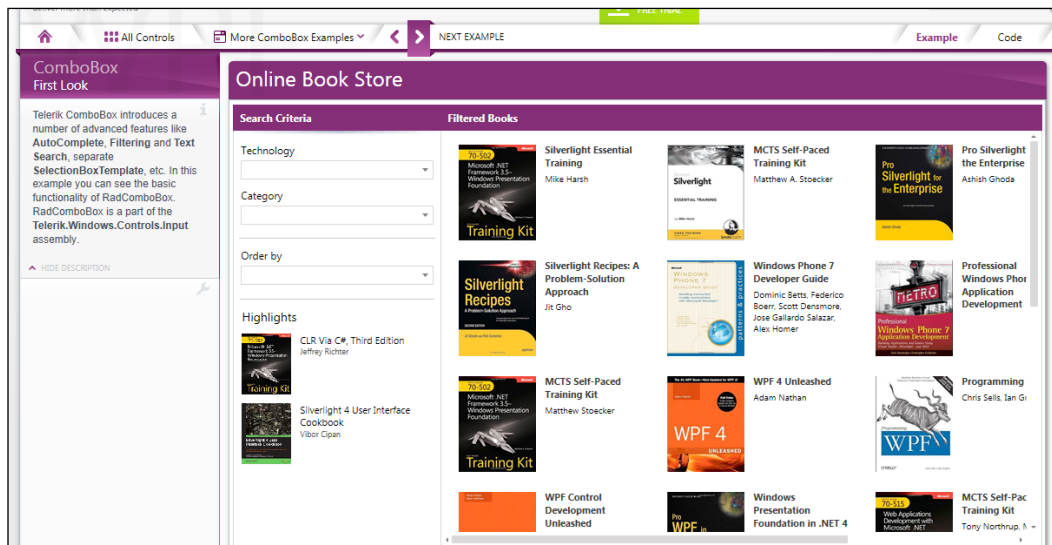


Click on the **Install** button to download the `Setup.exe` file from the Telerik site. The installation process will also install SQL Server 2008 Express as noted in the previous section. Please make sure that your computer has enough space to install the SQL Server instance. If you already have SQL Server 2008 Express installed, select the launch link instead to bypass the demo project installation.

The demo installation will create the following icon on your desktop:



You can use this icon to launch the demo solution. This demo is executable and will display examples of each control within the `WPF RadControl` library. Each control will have an XAML and a C# example to illustrate how to use the control. The examples are often displayed using the MVVM design pattern, but the examples are still capable of displaying how to use each control even if you are not familiar with the MVVM pattern. Here is an example of how the demo is formatted:



Each XAML page displays a well-skinned page with the controls as an example. This page is an example of using a **ComboBox** to filter the data. There are a few points to review on this page:

- On the upper-right corner, you will notice the **Example** and **Code** toggles. These toggles will allow you to view the code for the example displayed.
- The **More <control> Examples** option allows you to review additional examples of the selected control by clicking on a drop-down list option of your choice.
- The **NEXT EXAMPLE** button allows you to review each example one by one rather than trying to determine which example to review.

Summary

This chapter has demonstrated how to create a WPF project with the Telerik controls included within the Visual Studio `Toolbox`. Make sure that you have the capability to create a WPF project and add controls to a WPF form. The subsequent chapters rely on this functionality.

You should also have a complete Telerik solution with two supporting projects from the book downloads. This solution should be able to be built in Visual Studio without any compilation errors.

Please make sure to download the SQL Server database files, the supporting class libraries, and ensure that SQL Server has been installed. The subsequent chapters will rely on the capability to retrieve data from the SQL Server database to populate the controls and the class libraries to pass the information to the WPF application and the controls.

If you are light on experience with WPF, the concept of `DataContext` binding to an object, object-oriented design, or C# here is the list of the web links from the previous sections of the chapter for your review. Please make sure that you have a reasonable understanding of these concepts when you review the websites mentioned in the previous sections of this chapter. Here is the list of web links again for your review:

- For VB.NET to C# comparison: http://www.harding.edu/fmccown/vbnet_csharp_comparison.html
- To convert C# to VB.NET: <http://converter.telerik.com/>
- For the Model View ViewModel (MVVM) design pattern: <http://vortexwolf.wordpress.com/2011/11/27/wpf-and-silverlight-design-patterns/>

- For object-oriented design overview: <http://cplusplus.about.com/od/introductiontoprogramming/a/aboutoop.html>
- For WPF and DataContext overview: <http://msdn.microsoft.com/en-us/library/ms752347.aspx>
- To download the SQL Server database and class libraries: <http://www.packtpub.com/support>

You should also be able to review the Telerik Demo project and review the examples from Telerik. Even though most of the examples use the MVVM design pattern, the examples can still be helpful in working with the controls not covered in this book.

Now that we have set up the projects for the book, let us discuss the topics in the next chapter. We will be covering the simple data entry controls available from Telerik, such as the RadAutoComplete and the RadMaskedEdit controls. We will also be looking at the RadSpellCheck feature within Telerik to allow us to check spelling within any text controls.

2

Telerik Editors and How They Work

In this chapter, we will discuss the use of Telerik controls in WPF and how to make these controls handle the tasks for which they were designed. The reason I am starting with these editing controls is that they are simple to use; then, I will develop the concepts that will be used later on in the book.

The book will be working with the concepts of Data versus Object binding of Telerik controls. The reason for using both of these concepts is to cover both bindings since the architecture you may be using could use either the `System.Data` .NET libraries for `DataSet` and `DataTable`, or use custom-developed class objects. The `DataTable` class can often be used for sending data through a web service. This methodology allows for changes to the returning data without making specific **Web Service Definition Language (WSDL)** changes. The class object can act as a data contract in a WCF web service. The controls we will start with will be the editor controls as listed:

- `RadAutoCompleteBox`: This control will be used both in the Data bound and Object bound mode
- `RadMaskedInput`: This control will be described using currency, phone, e-mail, and zip code masks
- `RadSpellChecker`: This control acts in collaboration with other data entry controls to verify the spelling of the text inside the associated control

The first goal of this chapter is for you to understand these controls and how to use them to create an effective application. This chapter will also introduce the database and the two supporting class libraries for the book. These files are available at the publisher's website located at <http://support.packtpub.com>. Please make sure you have downloaded the files before moving forward in the chapter.

The second goal of this chapter will be to set up the example SQL Server database. This database will be used throughout the book; so once you have set up the database, you will be set for the rest of the book. There may be some additional data scripts to load more example data, but these scripts will be mentioned in each chapter. This chapter will not require any additional data from the current database.

Database setup

The next step, before reviewing the controls in the chapter, will be to make sure that the database is set up for the Telerik projects. In this chapter, you will be required to create the database in SQL Server; then create a configuration setting in the `App.config` file of your WPF User Interface project.

The first step will be to make sure you have downloaded the SQL Server database files from the PacktPub support website, you will have two separate options to install the database in SQL Server.

The first option will be to take the `PackTPubOrders.bak` file and restore the SQL Server database on your local SQL Server instance. The steps to do this are as follows:

1. Open the **SQL Server Mgmt Studio (SSMS)**.
2. Connect to the SQL Server instance you wish to use for the database.
3. Right-click on the **Databases Tree** option and select **Restore Database**.
4. Type in the name `PackTPubOrders` as the database name.
5. Select the name of the file from the location where you saved the book downloads.
6. Click on the **Restore** button and the database should display in the list of databases.

The second option will be to take the `PackTPub.mdf` and `PackTPub.ldf` files and attach these files to a new SQL Server database. The steps to do this are as follows:

1. Open **SQL Server Mgmt Studio (SSMS)**.
2. Connect to the SQL Server instance you wish to use for the database.
3. Right-click on the **Databases Tree** option and select **Attach**.
4. Select the MDF database file from the location where you saved the book downloads.

Once the database is set up, you will need to create a `ConnectionStrings` setting in your `App.config` file to match the database instance that you created previously in the SQL Server instance.

The first step will be to create an `App.Config` file. If you do not already have the configuration file within your current project, right-click on the project and select **Add File** from the menu. Select the **Application Configuration** option from the pull-down menu and an `App.config` file should be created.

Within the `App.Config` file, you will need to create a section called `connectionStrings`. The following connection string example should be used if you have created a specific SQL Server username for the database. The section should look like this:

```
<connectionStrings>
  <add connectionString="Server=<SERVERNAME>;database=<DATABASE>;User ID=<USERNAME>;Password=<PASSWORD>" name="TELERIKDB"/>
</connectionStrings>
```



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.

If you are using your Windows login, then you can use this configuration setup:

```
<connectionStrings>
  <add connectionString="Server=<SERVERNAME>;database=<DATABASE>;Trusted_Connection=true" name="TELERIKDB"/>
</connectionStrings>
<startup>
```

Once you have this configuration setting complete, you are ready to start working with the database.

If you are using another type of security or you are not sure which option to choose, please use the link <http://www.connectionstrings.com/sqlconnection/>. This site has several different options for connection strings and should have the string that you will need to access SQL Server.

Now that the configuration is set up, you will need to test the connection to ensure everything is correct. You will want to use the current solution created in the first chapter. If you did not create a solution, refer back to the first chapter. The code for the `MainWindow.xaml.cs` file should look like the following code snippet.

```
using PackTPub.DataLayer;
using PackTPub.DataLayer.Data;
using PackTPub.DataLayer.Util;
```

In the `Window_Loaded` event code place the following snippet:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    IDbManager dbMgr = new DBManager(this.Config("TELERIKDB", "TELERIKDB"));
    string query = "SELECT * FROM CUSTOMER";

    try
    {
        dbMgr.Open();
        dbMgr.ExecuteReader(CommandType.Text, query);

        while (dbMgr.DataReader.Read())
        {
            _custList.Add(cust);
        }
    }
    catch (Exception ex)
    {
        throw;
    }
    finally
    {
        dbMgr.Dispose();
    }
}
```

Once you have created the code for the `MainWindow.xaml.cs` class, run the solution through the debugger in Visual Studio. Make sure you place a breakpoint at the `IDbManager` line of the `Window_Loaded` method. Check to make sure the code does not cause an exception.

RadAutoCompleteBox with data binding

Now that the SQL Server connection is set up, we can start with the first control. The `RadAutoCompleteBox` control works in a very similar way to the `AutoComplete` search box, which you will see on the `google.com` site. The first step to working with the `RadAutoCompleteBox` control will be to familiarize yourself with the main properties and events that are associated with this control.

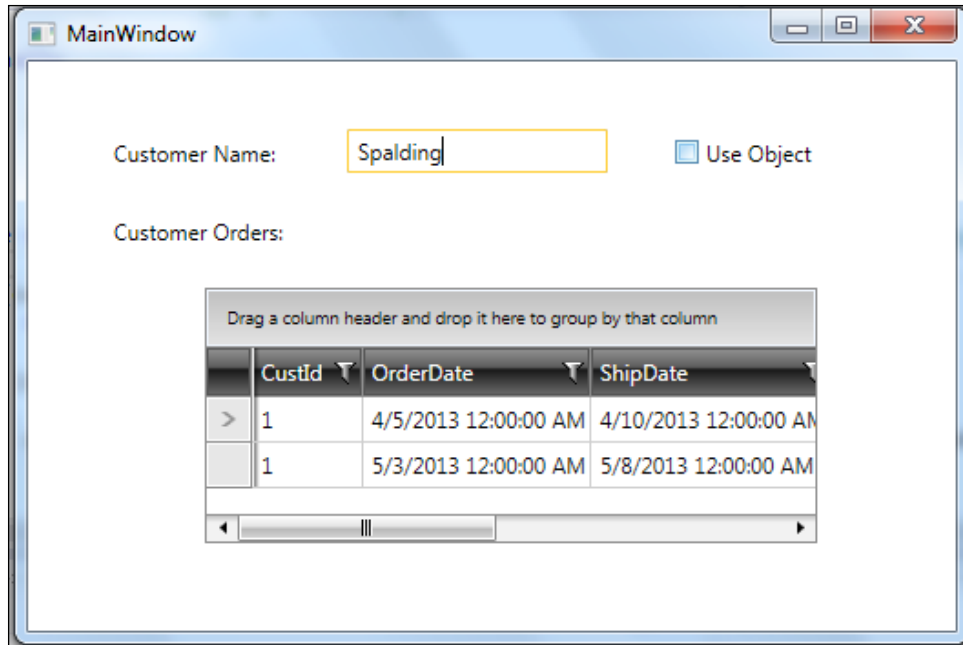
- The events are as follows:
 - `SelectionChanged`: This event is fired when the user changes the selection from within the `RadAutoCompleteBox` list. This event can assist in changing other information based on the selection by the user.
 - `SearchTextChanged`: This event is fired as the user types within the `RadAutoCompleteBox` list.
- The main properties are as follows:
 - `ItemSource`: This property sets the searchable values within the `RadAutoCompleteBox` list. Telerik recommends that the list object be of the `ObservableCollection` class.
 - `TextSearchMode`: This property determines how the text searching is performed within the `RadAutoCompleteBox` list. The options are `Contains`, `ContainsCaseSensitive`, `StartsWith` (the default setting), and `StartsWithCaseSensitive`.
 - `SelectionMode`: This property determines whether the user can select one option from the `RadAutoCompleteBox` option list or multiple options. The default is `Single`.

Now that you are familiar with the main events and properties, let's start to use this control.

Binding to a System.Data.DataTable

The first exercise will be to incorporate the `RadAutoCompleteBox` control and create a useable sample on the `MainWindow.xaml` file. We will be using `RadGridView` to verify the selection information from the `RadAutoCompleteBox` control, and a checkbox to determine how `RadGridView` will be populated. `RadGridView` will populate with customer orders based on the selection of a customer from the `RadAutoCompleteBox` control.

The first step will be to add the RadAutoCompleteBox control and RadGridView to your MainWindow.xaml window. Place the RadAutoCompleteBox control above RadGridView on your MainWindow.xaml file, and a checkbox next to the RadAutoCompleteBox control, as shown in the following screenshot:



There are several key properties that are set in the RadAutoCompleteBox control that we need to focus on in order to understand how the RadAutoCompleteBox control works:

- **TextSearchMode:** This is set to *Contains*. There are four settings for the *TextSearchMode* property:
 - *Contains*: If the typed text matches any portion of the loaded values
 - *ContainsCaseSensitive*: If the typed text matches any portion of the loaded values based on the case of the text
 - *StartsWith*: If the typed text matches the beginning of the loaded values
 - *StartsWithCaseSensitive*: If the typed text matches the beginning of the loaded values based on the case of the text
- **TextSearchPath:** This is set to *LName*. *LName* is the property in the *CustomerEntity* class that we want to focus the search on.

- SelectionMode: This is set to Single. There are two settings for this property:
 - Single: The RadAutoCompleteBox list will display a single option based on the text that is entered
 - Multiple: The RadAutoCompleteBox list will display a list of options based on the text that is entered
- WatermarkContent: This is set to Enter a Customer Last Name. This value will display in the RadAutoCompleteBox control when the window is loaded.

The next step will be to gather the customer information from the database to populate the ItemSource property on the RadAutoCompleteBox control. Inside the Window_Loaded event code, you will need to add the following code:

```
using PackTPub.ObjectLayer;
```

The following is the Window_Loaded event code:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Create an instance of the CustomerEntity from the
    cust = new PackTPub.ObjectLayer.Customer.CustomerEntity();

    try
    {
        if (checkUseObject.IsChecked == true)
        {
            cust.FetchList();
            this.DataContext = cust;
        }
        else
        {
            // Retrieve the customer information from the database using a DataTable and add it to the ItemSource property
            // in the customerComp AutoCompleteBox control.
            customerComp.ItemsSource = cust.Fetch().DefaultView;
        }
    }
    catch (Exception ex)
    {
        //TODO: Add logging to this system
    }
    finally
    {
        //TODO: Add clean up of any objects
    }
}
```

Notice in the example there are two different options. The option we want to use will be the DataTable option. The DataTable object will be bound to the ItemSource property of RadAutoCompleteBox named customerComp. We are using the DefaultView property from the DataTable option since the DataTable option cannot be directly bound to the ItemsSource property.

Once the code is in place, create a breakpoint on the line of code creating the instance of the `CustomerEntity` class. The first time you run the code you can move through the `Fetch` method to see how the `DataTable` option is populated. Once `AutoCompleteBox` is populated with customer data, try to select a customer by typing in the first few characters of the customer's last name. Once you select a customer's last name, the orders for that customer should display in the grid.

Binding to the `CustomerEntity` object

Now that we have handled the binding of the `RadAutoCompleteBox` control to the `DataTable` property through the code, the next option will be to have the `DataContext` property of the WPF window handle the binding of the data. This example will have the `Window_Loaded` event generate the `CustomerEntity` class with the `CustList` property populated with a generic list of `CustomerEntity` objects. This property will be bound to the `RadAutoCompleteBox` control using the `Binding` value rather than using the C# code to set the value.

The changes we will be making to the form will be to change the `ItemsSource` property in the `MainWindow.xaml` code. The first example of setting the `ItemsSource` property is to set this property using the `Window_Loaded` event. This example will set the property as follows:

```
<telerik:RadAutoCompleteBox
  x:Name="customerComp"
  HorizontalAlignment="Left"
  Margin="120,47,0,0"
  VerticalAlignment="Top"
  Width="143"
  WatermarkContent="Enter a customer..."
  TextSearchMode="Contains"
  AutoCompleteMode="Suggest"
  SelectionMode="Single"
  TextSearchPath="LName"
  ItemsSource="{Binding CustList}"
  SelectionChanged="customerAutoCompleteBoxSelectionChanged"/>
```

This property setting will be used to populate the control once the `DataContext` property for the window is set within the `Window_Loaded` event. Observe the following line of code in the `Window_Loaded` event:

```
if (checkUseObject.IsChecked == true)
{
    cust.FetchList();
    this.DataContext = cust;
}
```

This line sets the DataContext property of the window to the CustomerEntity object. The AutoCompleteBox control then picks up the CustList generic list and uses that data within the AutoCompleteBox control.

RadMaskedInput – currency, phone, and zip

The next control I will review is the RadMaskedInput control. This control is excellent for formatting data to make data entry easier for the user. This section will review the simple format of the RadMaskedInput control. The examples will be using the Mask property in the RadMaskedInput control to display how to create a mask for currency, phone number, and zip code data.

RadMaskedInputCurrency

The first step will be to create a new window for displaying the RadMaskedInput information. Once you have created the new window, add a RadMaskedInputCurrency control to the window. The setup for the control should look like the following screenshot:

```
<telerik:RadMaskedCurrencyInput
    HorizontalAlignment="Left"
    Height="33"
    Margin="183,65,0,0"
    Mask="#6.2"
    IsCurrencySymbolVisible="True"
    Culture="en-US"
    VerticalAlignment="Top"
    Width="161"/>
```

The properties that you should pay close attention to are highlighted in the example XAML information.

- **Mask:** This property controls the format of the information within the textbox. The mask "#6.2" means that the user can enter six numbers before the decimal and two numbers after the decimal.
- **Culture:** This property determines the currency to be displayed within the textbox. The default value is "en-US".
- **IsCurrencySymbolVisible:** This property displays a culture-based currency symbol. The default value for this property is "True".

Once you have the setup for the control correct (based on the previous example) try to test the control by running the application. The mask should allow you to input six numbers before the decimal and two numbers after the decimal. There should be a dollar sign in front of the entered data. Notice that if you change the `Culture` property of the control, the currency symbol will change to reflect the correct currency type. The control will also include a red cross symbol inside the control on the right side. This allows the user to clear the data from the control. The control properties also support binding to allow you to set up the controls based on configuration or database information.

RadMaskedInputText – phone and zip

The control we will demonstrate next will be the `RadMaskedInputText` control. This control can be used to format the text values of a textbox to allow a user to view the information in a familiar format. The data that is saved will not – or at least should not – be formatted. The first example will be to format the phone number. The control XAML should look like the following screenshot:

```
<telerik:RadMaskedTextInput
  HorizontalAlignment="Left"
  Height="37"
  Margin="206,232,0,0"
  VerticalAlignment="Top"
  Width="193"
  Mask="(###)##-####"/>
```

This format will support phone numbers in the U.S. after the mid 1960s. The phone numbers before that would use the first two characters to identify the area of the number. Let's setup the mask to allow the first two characters to be letters rather than numbers. The `Mask` property should then look like the following screenshot:

```
<telerik:RadMaskedTextInput
  HorizontalAlignment="Left"
  Height="37"
  Margin="206,232,0,0"
  VerticalAlignment="Top"
  Width="193"
  Mask="(###)aa#-####"/>
```

Now we have a mask that should support any options for a U.S. phone number. The next example will be to create a mask to support a U.S. zip code. The mask should support the nine-character format used in the U.S. The mask should look like the following screenshot:

```
<telerik:RadMaskedTextInput
  HorizontalAlignment="Left"
  Height="37"
  Margin="206,295,0,0"
  VerticalAlignment="Top"
  Width="193"
  Culture="en-US"
  Mask="#####-####"/>
```

The `Culture` property is available in this control, but the change in the value of this property will not assist if the user information is from Canada. The Canadian zip code is in a six-character format, so the mask will need to reflect this information based on the culture setting in the Visual Studio project, or user input such as the country. This mask should look like the following screenshot:

```
<telerik:RadMaskedTextInput
  HorizontalAlignment="Left"
  Height="37"
  Margin="206,295,0,0"
  VerticalAlignment="Top"
  Width="193"
  Culture="en-CA"
  Mask="a#a-#a#"/>
```

RadMaskedInput – e-mail and validations

The examples we will be discussing next will show us how to use the RadMaskedInput controls to act as validation for the values entered in the textbox. The purpose of this section is to provide a sample of how to extend the control for additional functionality without additional code. The validation of the data is handled at the class object level by adding an attribute to the Class property. We will review the RadMaskedEdit control, how to code the property associated to the control to pick up the validation, and how to display the validation message on the screen. The first step will be to add a RadMaskedTextInput control to your current masked edit window. This control will be bound to the Product class in the PacktPub.ObjectLayer project. The binding should be on the following properties:

```
<telerik:RadMaskedTextInput
  HorizontalAlignment="Left"
  Height="37"
  Margin="206,232,0,0"
  AllowInvalidValues="True"
  EmptyContent="Enter Product Code"
  Value="{Binding ProdCode, Mode=TwoWay, ValidatesOnExceptions=True}"
  VerticalAlignment="Top"
  Width="193"
  Mask=""/>
```

Once you have added these property settings, you can then add a combobox to select the product from the list. The selected product will be displayed in the three RadMaskedTextInput controls by binding the product object to the window's DataContext property. The following screenshot is an example:

```
public partial class MaskedEditWindow : Window
{
    public MaskedEditWindow()
    {
        InitializeComponent();
        this.DataContext = new Product();
    }
}
```

In the Product class in the PackTPub.ObjectLayer.Products.Product class file, the RegularExpression class allows only numbers and letters to be entered as values:

```

[StringLength(5, MinimumLength = 5, ErrorMessage = "The Product Code must be exactly 5 characters. ")]
[RegularExpression("[a-zA-Z0-9]+$", ErrorMessage = "Only numbers and letters allowed !")]
public string ProdCode
{
    get { return _prodCode; }
    set {
        Validator.ValidateProperty(value, new ValidationContext(this, null, null) { MemberName = "ProdCode" });
        _prodCode = value; }
}

```

Once you have completed this code, you can try to run the example. Start to enter data inside the masked edit textbox and you should see the selection generate the information in the input boxes. The next step to test this functionality would be to try and enter invalid data in these textboxes. If the data is invalid, you should see a message displayed with the information from the attribute on the property.

SpellChecker – TextBox

The next control we will be reviewing is the `RadSpellChecker` control. This control is a wrapper control that is associated with other controls to implement spell checking within the associated control. The `RadSpellChecker` control works best with the input controls from Telerik, but can be associated with the standard Visual Studio input controls. The first associated control we will review is the `TextBox` control from the Visual Studio standard `System.Windows.Controls` library. We will review the `RadRichTextBox` control and `RadDataGrid` from the Telerik control library. The reason we need to review the `TextBox` control is to allow you to understand how to implement the spell check control with standard controls as well as the Telerik controls. There is a key property and method that is set in the `RadSpellChecker` control that we need to focus on, in order to understand how the `RadSpellChecker` control works.



The `SpellCheckMode` property is used to determine the mode the spell checker handles while spell checking. There are two modes: `WordByWord` and `AllWordsatOnce`. The `WordByWord` mode allows the user to select a word and check the spelling. The `AllWordsatOnce` mode reviews all of the text within the current control.

The `RadSpellChecker` control requires that the WPF project has a reference for `Telerik.Windows.Documents.Proofing`. This library contains the `RadSpellChecker` control.

RadSpellChecker with VS TextBox

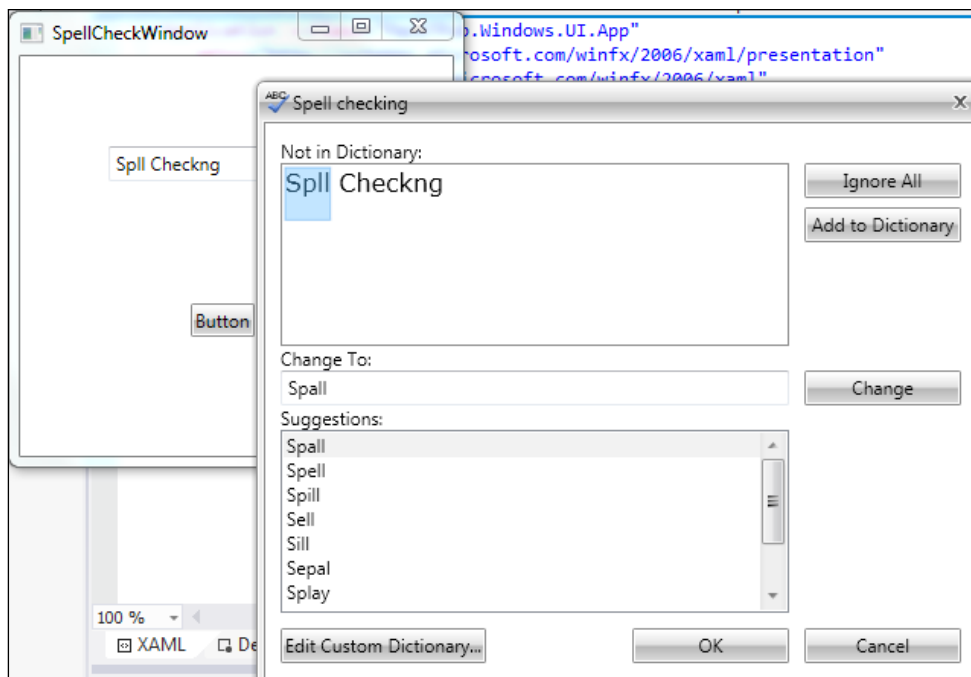
We will review the RadSpellChecker control by associating the standard TextBox control from Visual Studio. The first step will be to create a new window named RadSpellWindow and add this window to the project. Once the window has been created, you can drag a TextBox from the Toolbox on to the new window, and then drag over a RadButton control. Here are the names of the controls:

- TextBox: Name this control textCheckData
- RadButton: Name this control RadButton

Once the controls are in place on your window, you will need to move to the code view and set up the RadButton click event as follows:

```
private void RadButton_Click(object sender, RoutedEventArgs e)
{
    RadSpellChecker.Check(textCheckData, SpellCheckingMode.AllAtOnce);
}
```

The code should display the Telerik spell checking modal popup window. The window should highlight any spelling issues as shown:



The `RadSpellChecker` class has a static method named `Check`. This method takes two parameters, the control to be spell checked, and the `SpellCheckingMode` enumeration mentioned earlier in this section. The `Check` method will handle the spell checking of the text value inside the control. Any spelling issues within the control will cause the spell checking modal pop up to display. If there are no spelling issues, the method will execute but not respond.

RadSpellChecker with RadRichTextBox

The next control we will associate with the `RadSpellChecker` control is the Telerik `RadRichTextBox` control. The `RadRichTextBox` control is an extension of the Visual Studio `TextBox` control, but one of the properties Telerik has added to the `TextBox` is a Boolean property called `IsSpellCheckingEnabled`. This property then implements the `IControlSpellChecker` interface class, and allows the `RadSpellChecker` control to be implemented.

The next step will be to add a `RadRichTextBox` control to the current `SpellCheckWindow` control in the project. Name the `RadRichTextBox` control `textSpellCheck`. You can then set up the `RadRichTextBox` control to display the data as large as you would like to display it. The following step would be to add another `RadButton` to the window. This button will check the spelling on the `RadRichTextBox` control. This button is optional, but will make validating the text in the `RadRichTextBox` control easier. Once you have added the button, the next step will be to add the code in the following figure. If you decide not to create the button, you can add the code to the previous button click event from the `TextBox` example from the previous section.

```
private void buttonSpellCheck_Click(object sender, RoutedEventArgs e)
{
    RadSpellChecker.Check(textSpellCheck, SpellCheckingMode.AllAtOnce);
}
```

The next example we will document will work with the `WordByWord` mode of the spell checking method. Let's add another button for the `RadRichTextBox` and add the text into the `Content` property, `WordByWord`, for example. The next step will be to add the following code into the click event of the new button:

```
private void buttonSpellCheck_Click(object sender, RoutedEventArgs e)
{
    RadSpellChecker.Check(textSpellCheck, SpellCheckingMode.WordByWord);
}
```

The `WordByWord` mode will load each word into the spell check modal window for review by the user. This mode would work similar to how Microsoft Word would evaluate the text of a document, rather than the entire content of the control.

The mode you should use depends on the use of the text control. If the text control is meant to be used for a small amount of text, then the `AllAtOnce` mode would be a better option since all the text data would be evaluated at once. If the text control is meant to enter a large amount of data, then `WordByWord` will be easier to work with for the user. The decision is based on the user and the content, but I believe these are good rules of thumb to follow for the mode to use for the `SpellChecker` method.

RadSpellChecker with RadDataGrid

The next section in this chapter will discuss the use of the `RadSpellChecker` control within a Telerik `RadDataGrid` control. The Telerik `RadDataGrid` control gives you two different options for handling the spell checking within the grid.

- The first option allows the user to enable the spell checking manually. A button to check the spelling is displayed below the text in the grid cell to request the spell check.
- The second option is to automate the spell checking process. The spell check modal displays when the user leaves the current cell within the grid.

Let's get started with the next control. We'll create two columns within a `RadDataGrid` control. The first column will be an example of the manual spell checking setup and the second will demonstrate the automated spell check.

Add a `RadDataGrid` control to the `SpellCheckWindow` class we created earlier in the chapter. Once the `RadDataGrid` control is added, you will need to connect the `RadDataGrid` control to the `fetch` method of the `Order` class to load data within the grid. The code should look like the following screenshot:

```
public partial class SpellCheckWindow : Window
{
    private Order _order;

    public SpellCheckWindow()
    {
        InitializeComponent();
        _order = new Order();
        _order.FetchList(1);
        this.DataContext = _order.OrderList;
    }
}
```

The next step should be to add the Telerik spell check information to the RadDataGrid properties. The XAML for the RadDataGrid control should look like this:

```
<DataGrid
    telerik:DataGridSpellCheckHelper.IsSpellCheckingEnabled="True"
    x:Name="dataGridSpellCheck"
    AutoGenerateColumns="False"
    CanUserAddRows="False">
</DataGrid>
```

Once you have set up the RadDataGrid xaml file, you will then need to add a Telerik RadRichTextBox property inside the Grid section of the RadDataGrid control. The XAML should look like the following code snippet. Note that the IsSpellCheckingEnabled property is equal to False.

```
<telerik:RadRichTextBox
    x:Name="radRichTextBox1"
    VerticalScrollBarVisibility="Hidden"
    Height="100"
    BorderThickness="0" IsReadOnly="True"
    IsSpellCheckingEnabled="False"
    IsSelectionMiniToolBarEnabled="False"
    IsSelectionEnabled="False" IsContextMenuEnabled="False"
    IsHitTestVisible="False"
    telerik:XamlDataProvider.Source="{Binding OrderInstructions, Mode=OneWay}"/>
```

The first column will require a RadButton property below the RadRichTextBox property. This button will be the way to have the user check the spelling within the data cell. The XAML button should look like this:

```
<telerik:RadButton Grid.Row="1" Content="Check Spelling"
    x:Name="spellcheckRTBButton" Click="spellcheckRTBButton_Click" />
```

The C# code for the button should look like this:

```
private void spellcheckRTBButton_Click(object sender, RoutedEventArgs e)
{
    DataGridSpellCheckHelper.CheckChildControl(this.dataGrid, <the name of your RadRichTextBox>);
}
```

Notice that the method I am using in the preceding section is the `CheckChildControl` method. There are two parameters used in this method, the parent control, which in our case is the `RadDataGrid` control, and the control to be evaluated by the spell checker, which in this example is the `RadRichTextBox` control. When the user clicks the button within the `RadDataGrid` control, the cell's text will be evaluated and the Telerik spell check modal will be displayed.

The second column will be set up for automated spell checking so we will do the same setup of the XAML in the column, except that we will add the `Telerik:DataGridSpellHelper.IsSpellCheckEnabled` property to `DataGridTemplateColumn` and set the column spell checking to `True`. The XAML for the column should look like this:

```
<DataGridTemplateColumn Header = "Automatic spell checking" ~
width = "*" telerik:DataGridSpellCheckHelper.IsSpellCheckingEnabled="True">
```

Notice that the Telerik reference is created in the header of the XAML document when Telerik is included as a reference in the WPF project.

Once you have finished the XAML setup, you can test the `DataGrid` spell checking property. The first column will have the button display to check the spelling when you double click on the cell. The second column should automatically evaluate the spell checking once you move from the data grid.

Summary

In this chapter, you should now have the database set up and configured to work with the system. All the example data should now be available and can be displayed in the examples.

This chapter also covered three controls that you should now be able to include in a WPF project. Let us review these controls again:

- `RadAutoCompleteBox`: You should now be able to create a Telerik `AutoComplete` control, and bind the data to the control by using a `DataTable` object or a class object. You should also be able to handle the selected index event once the user selects an option from the `AutoComplete` control.
- `RadMaskedEdit`: You should now understand how to use the `RadMaskedEdit` controls to handle formatting text for the database, as well as handling validation on the information in the text control. This validation should be handled using attributes on a class.

- `RadSpellChecker`: Once you have finished the chapter, you should be able to use the `RadSpellChecker` static class to work with the following controls:
 - `TextBox`: You should be able to use the spell checker with the standard `TextBox` control from Visual Studio. You should be able to set up the spell checker within a button control and you should understand the two different modes of the spell checker: `WordByWord` and `AllContent`.
 - `RadRichTextBox`: You should be able to use the spell checker on the Telerik `RadRichTextBox` control using the `IsSpellCheckingEnabled` property setting on the Telerik control.
 - `RadDataGrid`: You should be able to create a `RadDataGrid` and use the spell checker functionality in either an automated or manual process.

In this chapter, we also reviewed the concept of validation using property attributes to handle the invalid message and format of the valid information. This concept will be discussed in further detail in the next chapter. We will start to create classes to handle the validation of the WPF controls using data binding and the property attribute.

3

Data Entry and Validation of Telerik Controls

In this chapter, we will review and enhance the validation information from the previous chapter. If you remember, in the validation of the `RadMaskedEdit` control, we created an attribute on the class property to handle validation so that the information entered within the control was based on the format in the attribute and correct. This chapter will further elaborate on this concept of creating validation for each editor type of control.

This chapter will also enhance the concepts of the `Data` versus `Object` binding of the Telerik controls. The binding works with the validation to create the link between the form and the class object. We will add some additional classes to the base portion of the project to allow for the attribute to be used with the class objects. This validation class can be enhanced to include many different types of validation and can be included within any WPF project. This chapter will review the following controls:

- `RadComboBox`: This control will be reviewed using the `DataTable` and `List<T>` bindings with validation
- `RadSpreadsheet`: This control will be reviewed using the `DataTable` and `List<T>` bindings and validation within the spreadsheet control
- `Dynamic validation`: This control will review the examples of the dynamic validation through class objects

The first goal of this chapter is to work with the `RadComboBox` and `RadSpreadsheet` controls using both `DataTable` and `Object` bindings. Understanding the flexibility for binding data classes as well as object binding allows for multiple options when connecting your application to all data sources.

The second goal of this chapter will be to further explore the dynamic validation concept using attribute-validation examples. We will start to add additional validation from different sources such as business logic (like an e-mail address) to enhance the use of the validation attributes.

Database setup

The next step, before reviewing the controls in the chapter, will be to make sure that the database is set up for the Telerik projects. The last chapter discussed the `PackTPub` database setup to support the examples used in the previous chapter. This chapter is supported by the same database information, so nothing new is required for this chapter.

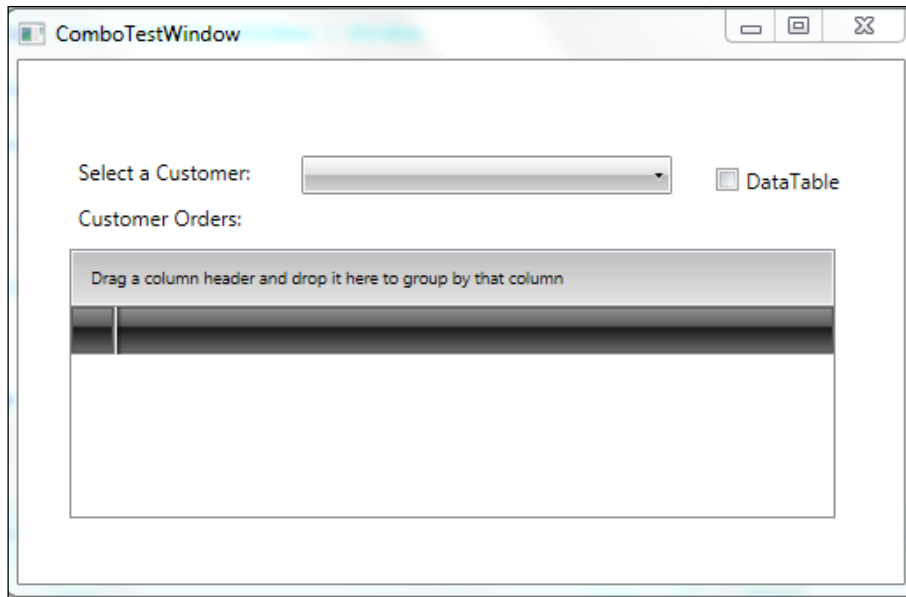
You should make sure that you can access the database. If you're not sure, or you do not remember whether you have set up the database, please refer to the *Database setup* section of *Chapter 2, Telerik Editors and How They Work*.

RadComboBox

The first control I will review in this chapter will be `RadComboBox`. This control extends `ComboBox` from the Visual Studio standard toolbox by taking the base as `ComboBox` and adding additional styles and data features. The first example I will review will be to set the binding of the control's list values using both a `DataTable` from the `System.Data` namespace and a list of customer class objects from the object layer of the example solution. The first step will be to create a new window in the current WPF project (name this new window `Chap3Window.xaml`). Once you have created the new window, you will need to add the following controls:

- `RadComboBox`: Name the combobox `RadComboCustomer`
- `RadDataGrid`: Name the grid `RadGridOrders`
- `CheckBox`: Name the checkbox `checkDataType`
- `Labels`: Add labels to describe the controls:
 - The label for the combobox should say **Select a Customer:**
 - The label for the checkbox should say **DataTable:**
 - The label for `DataGrid` should say **Orders for Customer:**

The final appearance of the window should be the following screenshot:



The next step will be to create the code to work with the data objects for populating the combobox with the customer information.

RadComboBox with DataTable

The first example of populating the combobox with customer information will be to use `DataTable` from the `System.Data` namespace. The reason for using `DataTable` to populate the combobox is twofold:

- `DataTable` offers an option to gather data using a common interface for the information.
- `DataTable` can be used in a web service call to allow the data to be passed in a common format without making any WSDL changes.

The reason for the checkbox is that I want to use this control to determine the type of data that will be loaded inside the combobox. This will be accomplished by clicking on the checkbox.

The first step will be to create an instance of the `Customer` class from the `PackTPub.Objects` project. This code should be added to the click event of the `checkDataType` checkbox. You will need to add a `using` statement to the beginning of your code to refer to the `PackTPub.Objects` project. Once the instance of the `Customer` class is created, you need to use the `checkDataType` value to determine which data object to display in the combobox. If the `checkDataType` property is `IsChecked == true`, we will populate the combobox with the `DataTable` object. The code should appear like the following screenshot:

```
private void checkDataType_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (checkDataType.IsChecked == true)
        {
            RadComboCustomer.DisplayMemberPath = "FullName";
            RadComboCustomer.SelectedValuePath = "Id";
            RadComboCustomer.ItemsSource = cust.Fetch().DefaultView;
        }
        else
        {
            cust.FetchList();
            this.DataContext = cust;
        }
    }
    catch (Exception ex)
    {
        // log your error
    }
}
```

Let's review the code in the preceding image. The `DataTable` version handles the binding by setting the `DisplayMemberPath` property to the column in `DataTable`, which should be displayed in the combobox. The `SelectedValuePath` property is set to the column in `DataTable`; this will be used as `SelectedValue` or the ID value for the query. The `ItemsSource` property is set to `DefaultView` of `DataTable`. The reason for using `DefaultView` rather than the actual `DataTable` is that the `ItemsSource` property requires an object that implements the `IEnumerable` interface class.

RadComboBox with a generic list

The next method for populating the RadComboBox control on the window will be to use a generic list of objects; in this case it will be Customer objects. Using the code from the previous example for loading DataTable, you will need to code the else condition of the if statement for when the checkDataType checked value is false. The CustomerEntity instance variable can now be used to retrieve the list of CustomerEntity objects. The code should appear like the highlighted portion of the following screenshot:

```
private void checkDataType_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (checkDataType.IsChecked == true)
        {
            RadComboCustomer.DisplayMemberPath = "FullName";
            RadComboCustomer.SelectedValuePath = "Id";
            RadComboCustomer.ItemsSource = cust.Fetch().DefaultView;
        }
        else
        {
            cust.FetchList();
            this.DataContext = cust;
        }
    }
    catch (Exception ex)
    {
        // log your error
    }
}
```

Let's review the code after the else statement. This time we execute the query by using the FetchList method from the CustomerEntity class. This method generates a generic list of CustomerEntity objects. We will then bind the instance of the CustomerEntity class to the DataContext property of the WPF window. The RadComboBox control has the ItemsSource property bound to the CustList property within the CustomerEntity class, as shown in the following code snippet:

```
ItemsSource="{Binding CustList}"
```

Now that the code is set, let's run the debugger to verify that the code works. You can set a breakpoint at the if statement in the event method.

Once you have verified that the code is working, the next step will be to take the information from the combobox and retrieve the customer orders based on the selected customer value in the combobox. You will need to create an event handler for the `SelectedValueChanged` event from the combobox. When the user selects a customer from the combobox, the event will fire and gather the customer ID from the selected combobox value and query the database for the orders. The code should appear like the following screenshot:

```
private void RadComboCustomer_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Order order = new Order();
    order.FetchList(Convert.ToInt32(RadComboCustomer.SelectedValue));
    RadGridOrders.ItemsSource = order.OrderList;
}
```

Notice the difference between the `RadComboBox` example and the `RadAutoComplete` example from *Chapter 2, Telerik Editors and How They Work*. Since the `RadComboBox` text and value are bound to the object, we are not worried about the type of binding objects. The `RadAutoComplete` control binds the object directly, so retrieving the value from the selected information requires knowledge of the type of bound object. The `RadComboBox` control does not require that level of knowledge. You can simply convert the `SelectedValue` property to an integer and pass the integer to the `Order` class method's `FetchList` to retrieve the list of orders for the selected customer.

Now that we have finished this control, you should be able to populate a `RadComboBox` with both `DataTable` and a generic list of objects. You should also be able to gather the information from `RadComboBox` to be used for later processing within your WPF window.

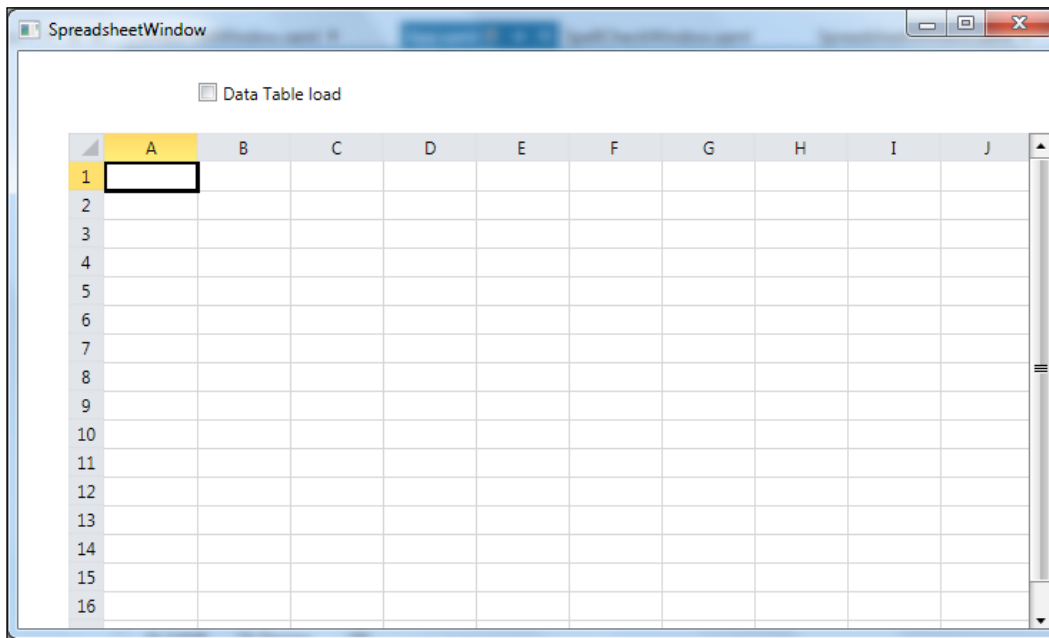
RadSpreadsheet

The next Telerik WPF control I will review will be the `RadSpreadsheet` control. This control is very powerful since it allows the user to work with data in a way similar to working with an Excel spreadsheet. This section will review binding database information to the `RadSpreadsheet` control. Again, I will be working with `DataTable` and a generic list of class objects to demonstrate the power of `RadSpreadsheet`.

The first step will be to make sure that you have the reference libraries you need in your WPF project. Go to the `References` section of your project, and then right click to add new references. Here are the libraries from the Telerik folder that you will need to add:

- Telerik.Windows.Controls.Spreadsheet
- Telerik.Windows.Documents.Spreadsheet
- Telerik.Windows.Documents.Spreadsheet.FormatProviders.OpenXml

The next step will be to create a new window to display the spreadsheet and test the code for this section. Go to your WPF project in the PackTPub solution, add a new WPF window, and name the window `SpreadsheetWindow.xaml`. Once you have created the new window, drag a `RadSpreadsheet` control from `Toolbox` on to the window. Name the spreadsheet control `RadSSControl` to match the code examples. The next step will be to add a checkbox control and name this control `checkDataTable`. The checkbox will determine which type of data object will be bound to the spreadsheet control. The window should appear like the following example:



Once you have finished setting up the interface in the window, you will need to set up the code to work with the `RadSpreadsheet` control. The `using` statements required to work with the code are as follows:

```
using PackTPub.ObjectLayer.Customer;  
using Telerik.Windows.Documents.Spreadsheet.FormatProviders.OpenXml.Xlsx;  
using Telerik.Windows.Documents.Spreadsheet.FormatProviders;  
using Telerik.Windows.Documents.Spreadsheet;  
using Telerik.Windows.Documents.Spreadsheet.Model;
```

The first using statement will incorporate the class object for the example. The Spreadsheet.Model library allows us to work with the Workbook class. This Workbook class is the key to binding data to the RadSpreadsheet control. The RadSpreadsheet control does not have data binding in the traditional definition. The way we will be populating the data from our database information to the spreadsheet control will be by creating an instance of a Workbook class, and then populating that object with the data. The first step in creating the code to support the spreadsheet will be to create an instance of the CustomerEntity class in the SpreadsheetWindow instance method. This line of code should be placed right below the InitializeComponent() method call. The code should appear like the following screenshot:

```
public partial class SpreadsheetWindow : Window
{
    CustomerEntity cust;

    public SpreadsheetWindow()
    {
        InitializeComponent();
        cust = new CustomerEntity();
    }
}
```

The next step will be to create the code to determine which data object will be used to load the spreadsheet control. You will need to create an event handler for the Click event of the checkDataTable checkbox control. The Click event will then determine how to load the spreadsheet control. The event handler code should appear like the following screenshot:

```
private void CheckBox_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (checkDataTable.IsChecked == true)
        {
            RadSSControl.Workbook = ConvertDataTable(cust.Fetch());
        }
        else
        {
            cust.FetchList();
            RadSSControl.Workbook = ConvertList(cust.CustList);
        }
    }
    catch (Exception ex)
    {
        //Log error
    }
}
```

Notice the two method calls, `ConvertDataTable` and `ConvertList`, in the `checkDataTable` click event code. These methods will take the data object from the `CustomerEntity` class and convert the object to a `Workbook` class that can be loaded into the spreadsheet control. The next step will be to include these new methods inside the `SpreadsheetWindow` class. Let's review what each of these methods does for this window:

- `ConvertDataTable`: This method takes `DataTable` from the `CustomerEntity` `Fetch` method, loops through the column names to create the headers for the spreadsheet, and then loops through the rows of the `DataTable` to load the data. The method creates a new `Worksheet` class from the `Workbook` class and adds cells to the worksheet. The workbook is then returned to the event method and applied to the spreadsheet control.

```
private Workbook ConvertDataTable(System.Data.DataTable custTable)
{
    Workbook workbook = new Workbook();
    try
    {
        if (custTable.Rows.Count > 0)
        {
            int rowCnt = 0;
            int col = 0;

            workbook.Worksheets.Add();
            Worksheet worksheet = workbook.ActiveWorksheet;

            foreach (System.Data.DataColumn column in custTable.Columns)
            {
                worksheet.Cells[col, rowCnt].SetValue(column.ColumnName);
                rowCnt++;
            }

            col++;

            foreach (System.Data.DataRow row in custTable.Rows)
            {
                for (int i = 1; i < row.ItemArray.Length; i++)
                {
                    worksheet.Cells[col, i].SetValue(row.ItemArray[i].ToString());
                }

                col++;
            }
        }
    }
    catch (Exception ex) { throw; }
    return workbook;
}
```


- **ConvertList:** This method takes the generic list of `CustomerEntity` objects from the `CustomerEntity FetchList` method, loops through the list, takes the property name to load the headers of the spreadsheet, loops through the data objects to gather the values in the properties of the object, and then loads the spreadsheet cells. The method creates a new `Worksheet` class from the `Workbook` class and adds cells to the worksheet. The workbook is then returned to the event method and applied to the spreadsheet control.

```
private Workbook ConvertList(List<CustomerEntity> custList)
{
    Workbook workbook = new Workbook();
    try
    {
        if (custList.Count > 0)
        {
            workbook.Worksheets.Add();
            Worksheet worksheet = workbook.ActiveWorksheet;
            bool headers = false;
            int rowCnt = 0;
            int col = 1;
            int headCol = 0;
            foreach (CustomerEntity cust in custList)
            {
                foreach (var prop in
                    cust.GetType().GetProperties())
                {
                    int curCnt = 0;
                    Type type = prop.GetType();
                    if (prop.GetType().
                        AssemblyQualifiedName == "System.Collections.
                        Generic.List")
                        continue;
                    if (!headers)
                        worksheet.Cells[headCol,
                            rowCnt].SetValue(prop.Name);
                    worksheet.Cells[col,
                        rowCnt].SetValue(prop.GetValue
                            (cust, null).ToString());
                    rowCnt++;
                }
                headers = true;
                rowCnt = 0;
                col++;
            }
        }
    }
    catch (Exception ex) { throw; }
    return workbook;
}
```

Now that the code is complete, you can try to test the code. Place a breakpoint at the creation of the `Workbook` instance and run the debugger. When you click on the `checkDataTable` object, the `DataTable` portion of the `if` statement should be executed. The spreadsheet should load with the customer data from the database. When you remove the check from `checkDataTable`, the `List<CustomerEntity>` `else` condition should be executed. You should notice a difference in the data that is loaded into the spreadsheet control. Since the `CustomerEntity` class has properties for `CreatedBy`, `CreatedDate`, `LastUpdatedBy`, and `LastUpdatedDate`, these properties are loaded into the spreadsheet.

Now that we have finished the `RadSpreadsheet` control, you should be able to populate a `RadSpreadsheet` control with both `DataTable` and a generic list of objects. You should also be able to understand the differences between an object loading and a `DataTable` load.

Dynamic validation

The next section of this chapter deals with checking of data within a Telerik or standard control using custom validation. This section will not be specific to Telerik controls—it can be used with any control. I will create examples with Telerik controls, but I will also include standard Visual Studio controls.

The setup for this type of validation starts with the `PackTPub.Object` project. Inside of the project is a folder named `Validation`. All the classes for handling the validation are stored within this folder. The design of this validation is to use a property attribute to handle the validation for that property. The control that is bound to the property using the `DataContext` property from the WPF `Window` class will pick up the validation and display the attribute message.

The beginning of this validation setup is the `IValidationRule` interface class. This interface class has one implementation method called `Validate`. This method—for those of you not familiar with interface classes—will be defined in each validation class that will inherit with the interface class. The method will appear like the following code:

```
public interface IValidationRule
{
    void Validate(object value, out bool isValid,
        out string errorMessage);
}
```

The object value parameter is the `DataContext` value from the control on the WPF window. The output parameters `isValid` and `errorMessage` determine if the data inside the control is valid; if the data is not valid, the `errorMessage` parameter is displayed.

The next step will be to create a class to act as the validation attribute's check to verify the data. The example Visual Studio project has two example classes to demonstrate this concept. The `ComboValidationAttribute` and `TextValidationAttribute` classes demonstrate how to set up a validation attribute class. The first aspect of these classes, you will want to notice, is the inheritance. The following is an example:

```
public class TextValidationAttribute : Attribute, IValidationRule
```

There are two classes that the `TextValidationAttribute` class inherits from in the previous example:

- `Attribute`: This class allows the class to be used as a property attribute
- `IValidationRule`: This interface class determines the setup for the `Validate` method call

The next piece of code we want to review is the actual `Validate` method in the `TextValidationAttribute` class. The class has a property called `MinLength`; this property determines the length of the text value that is valid in the `Validate` method. This property value is passed to the class using the `class` attribute. There is an example of this attribute setup in the next section of this chapter. The `Validate` method will review the current property value for two checks: if there is a value in the property and if the length matches or exceeds the `MinLength` property. The following is an example:

```
public void Validate(object value, out bool isValid,
    out string errorMessage)
{
    isValid = false;
    errorMessage = "";
    if (value != null && value.ToString()
        .Length >= MinLength)
        isValid = true;
    if (!isValid)
        errorMessage = value + " is not equal to
            or longer than " + MinLength;
}
```

The `errorMessage` parameter is returned to the class, and then to the WPF window to display to the user. You can format the message as you see fit to make the information display a user-friendly message.

The following is an example of the `TextValidationAttribute` class defined with a property in the `Product` class in the `PackTPub.Objects` project:

```
[TextValidation(MinLength = 10)]
public string LastName
```

The attribute name `TextValidation` matches the class that was created in the example. The `MinLength` property sets the property in the `TextValidation` class, and the value is used to determine if the length of the string is long enough to pass the validation test.

The next example of the validation will be to create a check for a combobox control. I use the combobox validation to determine if a correct value has been selected inside the combobox control. The combobox validation works in a fashion similar to the textbox validation. The attribute for the combobox is set above the property to be bound from the associated control on the WPF window. The selected value of the combobox is then passed into the `Validate` method to verify that a correct value has been selected. The following is an example of the `ComboBoxValidation` attribute:

```
[ComboBoxValidation(DefaultValue = 0)]
public int CustomerType
```

The default value should be the default item in the combobox that you are validating. When you are loading the combobox items, a default item should be selected inside the combobox. This item will act as the default. I chose the number zero since this number would not match type values in a database table. Once the attribute is set, we need to review the `Validate` method from the `ComboBoxValidation` class to understand how this method works:

```
public void Validate(object value, out bool isValid, out string
errorMessage)
{
    int result = 0;
    isValid = false;
    if (value != null)
        isValid = Int32.TryParse(value.ToString(), out result);
    errorMessage = "";
    if (!isValid)
        errorMessage = "- *";
    else if (isValid & result == _defaultValue)
    {
        errorMessage = "- *";
        isValid = false;
    }
}
```

The `Validate` method in this example checks to see if the value is an integer. Once an integer has been passed, the next check is to determine if the integer value is not equal to the default value. If any of these conditions is false, the method returns the `errorMessage` parameter with the user correction. Notice that the message is only an asterisk in this example. You are more than welcome to incorporate your own message into this method.

Now that we have the validation in place, let's create a window to use as an example for validating the information in a WPF window. Go ahead and create a new window in your WPF project and name the window `ValidationTestWindow.xaml`. Add a textbox and a combobox to the window. Once you have completed adding the controls to the window, we need to add the properties for `CustomerEntity` to the textbox and the combobox. This setup will cause the class to bind to the objects. The following is how the binding should appear in the textbox:

```
<TextBox
    x:Name="textLName"
    Text="{local:ValidationBinding Path=LName}"
    HorizontalAlignment="Left"
    Height="23"
    Margin="19, 91, 0, 0"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Width="120"/>
```

This is how the combobox binding should be done:

```
<telerik:RadComboBox
    x:Name="comboTestValue"
    SelectedValue="{local:ValidationBinding Path=ComboTest}"
    HorizontalAlignment="Left" Margin="19, 37, 0, 0"
    VerticalAlignment="Top"
    Width="250"
    Grid.ColumnSpan="2"/>
```

The `Path=value` in the XAML code should be set to the `CustomerEntity` class property that is to be bound to the selected control. In the case of `TextBox`, the `LName` property in the `CustomerEntity` class will be bound to `textLName` `TextBox`. Once you have the binding set up in the controls, the next step will be to create an instance of the bound object and set the window's `DataContext` property to that object. The following is how the code should appear in the window's instance method:

```
CustomerEntity cust;

public ComboTestWindow()
{
    InitializeComponent();
    cust = new CustomerEntity();

    cust.FName = "Dan";
    cust.LName = "Spalding";
    cust.Email = "test@aol.com";
    cust.FetchList();
    this.DataContext = cust;
}
}
```

The preceding code creates an instance of the `CustomerEntity` class, populating the properties necessary, and then binding the instance of the class to the `DataContext` property of the window. When you execute the debugger to run through the code, the window will display the messages for the validation. When you enter/select the correct values into the controls, the message should disappear from the window. Now that we have the validation running on the common Visual Studio controls, let's test this concept on the Telerik controls. Add the `RadComboBox` control to the current validation window, and then take the binding from the current combobox and add that binding to `RadComboBox`, as shown in the following example:

```
<telerik:RadComboBox
    x:Name="comboTestValue"
    SelectedValue="{local:ValidationBinding Path=ComboTest}"
    HorizontalAlignment="Left"
    Margin="19,37,0,0"
    VerticalAlignment="Top"
    Width="250"
    Grid.ColumnSpan="2"/>
```

Once you have completed the binding to the `RadComboBox` step, try to run the system through the debugger. The results should match the original window's results.

This methodology for validation using the WPF binding context can be used with any control – Telerik or otherwise. The Telerik's `RadMaskedEdit` control works in a similar fashion, but a validation method is required to be placed in the set method of each property to be validated. I feel that this validation is easier to use and allows you to incorporate business rules as a part of the validation, if necessary. You can also expand the validation types if you feel there is anything missing from the current classes.

The next example will be on how to incorporate a new validation (e-mail validation) into the current list of checks. The object layer project has a class file called `Email.cs` that you can incorporate into the current project. Here are the steps to incorporate this new class:

1. Right-click on the **PackTPub.ObjectLayer** project and click on **Add New Folder**.
2. Enter the folder name as `Util`.
3. Right-click on the **PackTPub.ObjectLayer** project and click on **Add Existing Item**.
4. In the **Explorer** window, select the `Email.cs` file and place it in the `Util` folder.

Once the `Email` class is incorporated into the `PackTPub.ObjectLayer` project, we can start to create the class to validate the e-mail address using the property attribute. Let's start by creating a class in the `Validation` folder of the `PackTPub.ObjectLayer` project called `EmailValidationAttribute.cs`. This class should be set up like the following example:

```
public class EmailValidationAttribute : Attribute, IValidationRule
{
    public EmailValidationAttribute()
    { }

    public void Validate(object value, out bool isValid, out string errorMessage)
    {
        isValid = false;
        if (value != null)
            isValid = Email.ValidEmailAddress(value.ToString());

        errorMessage = "";

        if (!isValid)
            errorMessage = "- *";
    }
}
```

The code in the example uses the `ValidEmailAddress` method in the `Email` class to determine if the text value of the property is correctly formatted as an e-mail address. The setup of the `Validate` method matches the examples that we created earlier with the `Text` and `Combo` attributes. The design of this validation allows you to incorporate additional business rules without a lot of additional coding.

The next step will be to incorporate the new e-mail validation into one of the current classes in the project. Go to the `CustomerEntity` class in the `Customer` folder. The next step will be to add the `using` statement to include the attributes:

```
using PackTPub.ObjectLayer.Validation;
```

Once the `using` statement has been added, you can start to add the attributes to the properties. The `CustomerEntity` class has a property named `Email`. You can now add the validation attribute to the property, as shown in the following example:

```
[EmailValidation]
public string Email
{
    get { return _email; }
    set { _email = value; }
}
```

The next step will be to add the assembly information to the window namespaces for allowing the validation to display on the window. The `PackTPub.Window.UI` project already has the `Validation` classes included in it; you will only need to add the assembly reference to the top of the window XAML, as shown in the following line of code:

```
xmlns:local="clr-namespace:PackTPub.Windows.UI.Validation"
```

This reference will allow you to include the validation binding when you create new controls on your window. The next step will be to create `TextBox` on `ValidationTestWindow.xaml` and add the binding to `TextBox`, as shown in the following example:

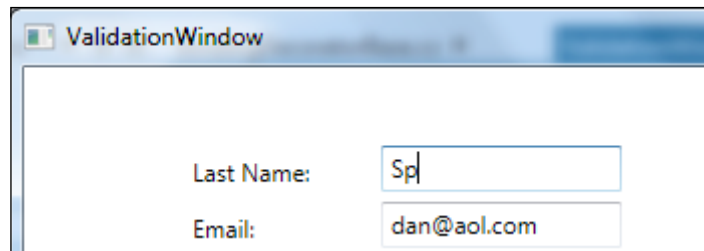
```
<TextBox
x:Name="textEMail"
Text="{local:ValidationBinding Path=Email}"
HorizontalAlignment="Left"
Height="23"
Margin="19,91,0,0"
TextWrapping="Wrap"
VerticalAlignment="Top"
Width="120"/>
```

Notice the highlighted portion of the XAML code; the local reference matches the namespace addition we made in the assembly information. The binding reference picks up the information for the assembly, and it will allow the attribute's validation and messages to display in the window.

Now that the validation is in place, here is how the form will appear if the data is invalid:



You will notice the red outline around the textbox with the **Last Name:** label when the data is invalid. As you type data into the textbox, the validation will check the data against the logic built into the `Validate` method. The following screenshot is an example of the validation after the data is considered as valid:



Once the data is valid, the red outline is removed. You can test this on the e-mail textbox as well. Remove `.com` from the `dan@aol.com` value, and the red outline should appear. I hope you have found this methodology for validation helpful. I think that this form of validation allows you to incorporate business rules in an easier fashion than the Telerik method.

Summary

In this chapter, I covered two controls and a validation concept that can be used with any input control to validate the data entry information. Let's review these controls again:

- `RadComboBox`: You should now be able to create a Telerik `RadComboBox` control, and bind the data to the control by using `DataTable` or a class object. You should also be able to handle the selected index event once the user selects an option from the `RadComboBox` control.

- `RadSpreadsheet`: You should now understand how to use the `RadSpreadsheet` control to handle the formatting of the information from either a generic list or `DataTable`. You should be able to load the spreadsheet with either type of data object.

The other WPF concept that I covered was the Dynamic Validation of any data input controls – Telerik or otherwise. The concept of this validation uses the `DataContext` property and the binding class object to determine the validation for the selected control. This type of validation saves a tremendous amount of work in coding the validation into each window. The concept also allows for sharing the validation across several WPF projects, since the library could be used in several solutions. Telerik's masked edit controls also apply a similar concept, but there are a few differences that should be noted. These differences are as follows:

- The Telerik validation can only be used with Telerik controls
- The Telerik validation requires coding in the `set` method of the property rather than using a property attribute
- The Telerik solution is not as portable

The biggest feature I want to reinforce with the validation is portability. This methodology allows you to implement the validation in any WPF user interface. If you have a Silverlight application as well as a WPF application that supports your users, you can have the same validation for both the applications without any additional coding.

In the next chapter, we will be working with the layout and design functionality that Telerik's `RadControls` provides. I will specifically cover the `RadTab` control and `RadBook` from Telerik. I will discuss how to load these controls from the database and objects to create a more dynamic interface.

4

Layout Organization and Display Functionality

In this chapter, we will demonstrate and discuss the use of the Telerik container controls. At this point in the book, we have worked with standalone controls for data entry on WPF windows. This chapter will start to work with container controls. These are the controls that will handle the display of other controls. We will review the best uses for the container controls we cover, then we will discuss the options for dynamically populating these controls through the C# code, and either database or XML configuration.

I have selected two controls that I feel give the best examples of the concepts that we want to cover. We will be focusing our attention on the following controls:

- RadTabControl
- RadBook

We will be focusing our efforts on dynamically loading these controls to make the user interface capable of external configuration for allowing dynamic content. This concept will allow you to develop applications that require very few changes once the initial development is complete. The concept would be to allow the power users of the system to add information as needed to meet the user demand without any additional coding for the development team.

This chapter will also continue discussions on the concepts of Data versus Object binding of the Telerik controls. We will demonstrate the loading of the controls from both the database and the configuration files to give the system the flexibility in the handling of the dynamic information. I hope this chapter gives you a real-world perspective on the use of these container controls, and how to handle the processing of the controls in multiple data interfaces.

Database setup

The next step, before reviewing the controls in the chapter, will be to make sure that the database is set up for the Telerik projects. The last chapter discussed the PackTPub database setup to support the examples used in the previous chapter. This chapter is supported by the same database information, so there is nothing required for this chapter.

You should make sure that you can access the database. If you're not sure, or do not remember if you have set up the database, please refer to the *Database setup* section of *Chapter 2, Telerik Editors and How They Work*.

This chapter will also require the XML files to be stored on your machine. If you haven't done so already, create a folder called `LoadConfig` inside the `PackTPub` WPF project's `bin` folder. This folder will be read at the loading of the system to gather the required information for generating the `RadTabControl` information.

RadTabControl

The first control I will review in this chapter will be the `RadTabControl` control. This control extends the `TabControl` control from the Visual Studio standard toolbox. The first example that I will review will be to set the binding of the control's tabs using both a `DataTable` object from the `System.Data` namespace and a list of class objects from the object layer of the example solution.



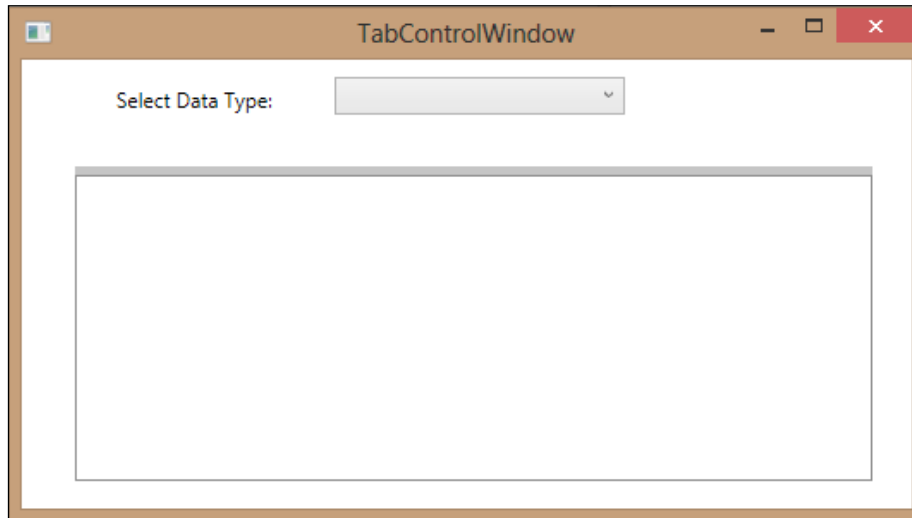
The `DataContext` property in the `RadTabControl` control only allows for population of the tabs and not the content in the container.

The first step will be to create a new window in the current WPF project. Name this new window `Chap4TabWindow.xaml`. Once you have created the new window, you will need to add two controls:

- `RadTabControl`: Name the `RadTabControl` control `RadTestTab`
- `ComboBox`: Name the `ComboBox` control `comboDataType`

Add labels to describe the controls. The label for the checkbox should say: **Select Data Type:**.

The final look of the window should be like the following screenshot:



The next step will be to create the code to work with the data objects to populate the tab control with the layout for the order processing.

RadTabControl with DataTable

The first example of populating the `RadTabControl` control with the customer tab information will be to use a `DataTable` object from the `System.Data` namespace. The reason for using a `DataTable` object to populate the combobox is twofold:

- The `DataTable` object offers an option to gather the data using a common interface for the information.
- `DataTable` can be used in a web service call to allow data to be passed in a common format without making any WSDL changes.

The reason for the combobox is that I want to use this control to determine the type of data that will be loaded inside the tab control. This will be accomplished by selecting the option on the combobox to determine the type of data to be loaded.

The first step will be to create an instance of the `Customer` class from the `PackTPub.Objects` project. This code should be added to the `SelectedIndexChanged` event of the `comboDataType` combobox. You will need to add a `using` statement to the beginning of your code to reference the `PackTPub.Objects` project. Once the instance of the `Customer` class is created, you need to use the `comboDataType` value to determine which data object to display in the tab control. If the `comboDataType` property is `SelectedValue == "Table"`, we will populate the tab control with the `DataTable` object. The code should look like the following screenshot:

```
private void comboDataType_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (comboDataType.SelectedIndex == -1)
        return;

    string filePath = System.AppDomain.CurrentDomain.BaseDirectory + "\\XML\\";

    try
    {
        switch (comboDataType.SelectedValue.ToString())
        {
            case "XML":
                _list = _cust.fetchXmlTabs(filePath + _cust.GetType().Name + ".xml");
                radTestTab.DisplayMemberPath = "HeaderText";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _list;
                break;
            case "Table":
                _list = null;
                _table = _cust.fetchDataTabs();
                radTestTab.DisplayMemberPath = "TabTitle";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _table.DefaultView;
                break;
            case "List":
                _list = _cust.fetchTabs();
                radTestTab.DisplayMemberPath = "HeaderText";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _list;
                break;
        }
    }
    catch (Exception ex)
    {
        //TODO: Add logging to your code, empty exceptions are BAD!
    }
}
```

Let's review the code in the preceding screenshot. The `DataTable` version handles the binding by setting the `DisplayMemberPath` property to the column in the `DataTable` object that should be displayed in each tab on the `TabControl` control. The `ItemsSource` property is set to `DefaultView` of the `DataTable` object. The reason for using `DefaultView` rather than the actual `DataTable` is that the `ItemsSource` property requires an object that implements the `IEnumerable` interface class. The SQL Server table for the tab information also has the default selected tab set as a Boolean value. We take the table and determine which tab is selected, and then set the `SelectedIndex` property on the `TabControl` control to that index. The method for determining the `SelectedIndex` property looks like the following screenshot:

```
private int FetchSelectedTab()
{
    int index = 0;

    if (_list != null)
    {
        foreach (ObjectLayer.Util.TabItemEntity item in _list)
        {
            if (item.IsSelected)
                return index;

            index++;
        }
    }
    else if (_table != null)
    {
        foreach (System.Data.DataRow rowData in _table.Rows)
        {
            if (Convert.ToBoolean(rowData["TabSelected"]))
                return index;

            index++;
        }
    }

    return 0;
}
```

This method determines which type of value is available (whether the generic list or the `DataTable` object), and then loops through the values to determine which tab index should be selected on the `TabControl` control. If the method cannot find a selected tab, the default is the first tab or index 0.

RadTabControl with a generic list

The next method for populating the `RadTabControl` control on the window will be to use a generic list of objects (in this case it will be the `TabItem` objects). Using the code from the previous example, for loading the `DataTable` object, you will need to code the `else` condition of the `if` statement for when the `comboBoxType` checked value is `false`. The `Customer` instance variable can now be used to retrieve the list of the `TabItem` objects. The code should look like the highlighted portion of this code:

```
case "List":
    _list = _cust.fetchTabs();
    radTestTab.DisplayMemberPath = "HeaderText";
    radTestTab.SelectedIndex = FetchSelectedTab();
    radTestTab.ItemsSource = _list;
    break;
```

Let's review the code in the `List` section of the `switch` statement. This time we execute the query by using the `fetchTabs()` method from the `CustomerEntity` class. This method generates a generic list of `TabItem` objects. We will then bind the instance of the `TabItem` class to the `ItemsSource` property of the `RadTabControl` control. `RadTabControl` has the `DisplayMemberPath` property bound to the `HeaderText` property within the `TabItem` class as shown:

```
<telerik:RadTabControl x:Name="radTestTab" HorizontalAlignment="Left"
Margin="31,62,0,0" VerticalAlignment="Top" Height="183" Width="464"
DisplayMemberPath="HeaderText">
</telerik:RadTabControl>
```

RadTabControl with an XML file

The last step will be to create the tabs by using an XML file to gather the information. There are several reasons for using an XML file, as opposed to a database, for this type of configuration. The following are the reasons:

- The system can be moved without any changes in the system configuration
- The system becomes database-independent
- The configuration can be installed using an MSI installer without access to a database

The XML files for the `PackTPub` system are stored in the project's `bin\XML` folder. The `BaseEntity` class has four methods to serialize and deserialize XML files. The following are the methods:

- `Serialize`: This method generates an XML file from a single class object

- **Deserialize:** This method generates objects from the XML file
- **SerializeList:** This method generates an XML file from a generic list of objects
- **DeserializeList:** This method generates a list of class objects from an XML file

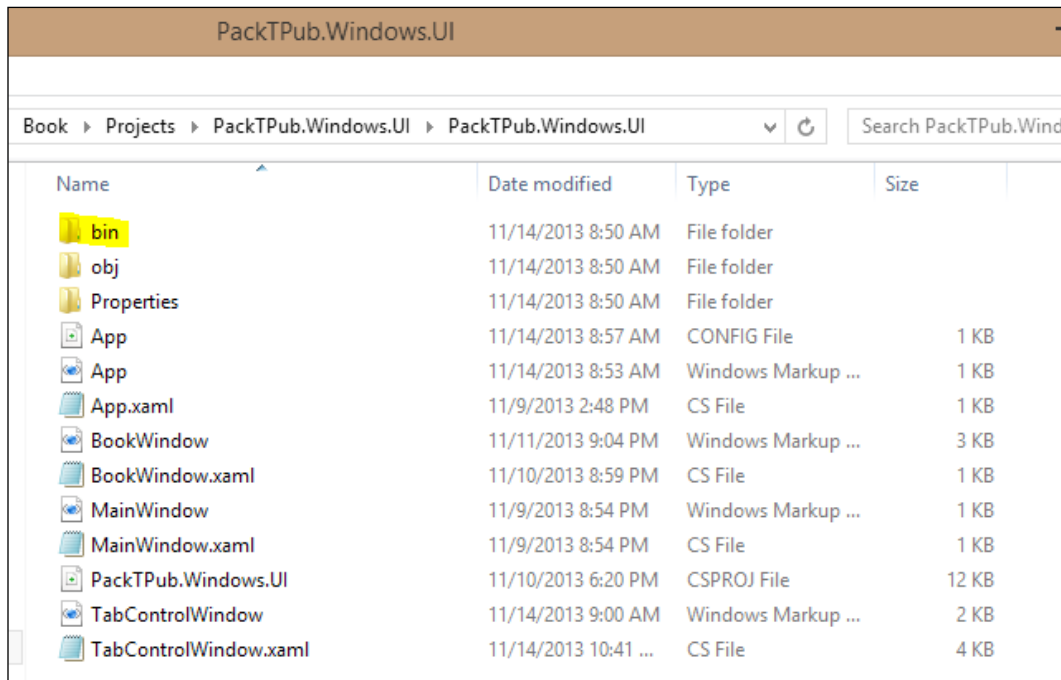
Serialization is the concept of taking a class and generating an XML file based on the class information. **Deserialization** reverses that process and creates the object from the XML file. Here is a link if you want to understand more about serialization; you can review the web page at <http://support.microsoft.com/kb/815813>. The following example reviews the simple process of serializing a class object to XML:

```
private void comboDataType_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (comboDataType.SelectedIndex == -1)
        return;

    string filePath = System.AppDomain.CurrentDomain.BaseDirectory + "\\XML\\";

    try
    {
        switch (comboDataType.SelectedValue.ToString())
        {
            case "XML":
                _list = _cust.fetchXmlTabs(filePath + _cust.GetType().Name + ".xml");
                radTestTab.DisplayMemberPath = "HeaderText";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _list;
                break;
            case "Table":
                _list = null;
                _table = _cust.fetchDataTabs();
                radTestTab.DisplayMemberPath = "TabTitle";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _table.DefaultView;
                break;
            case "List":
                _list = _cust.fetchTabs();
                radTestTab.DisplayMemberPath = "HeaderText";
                radTestTab.SelectedIndex = FetchSelectedTab();
                radTestTab.ItemsSource = _list;
                break;
        }
    }
    catch (Exception ex)
    {
        //TODO: Add logging to your code, empty exceptions are BAD!
    }
}
```

Now that we understand the concept and reasons for using serialization, let's review the code for this portion of the example. The first difference we should discuss is the line of the preceding code, the `try` statement. This statement creates the path to the XML files. The `BaseDirectory` property is the executable path for the system. If you want to locate this directory on your system, go to the directory where you have the WPF project stored on your drive. Inside the project folder where the XAML files are located, you will see a `bin` folder, as shown in the following screenshot:



Inside the `bin` folder will be a folder named `XML`. You will see a file called `CustomerEntity.xml`. This XML file has the data for tab configuration.

Inside the event code for the combobox, the switch statement has an option for XML. We will call the `fetchXMLTabs` method with the path to the XML file. Unlike the other examples that used the database configuration from `App.Config`, we need to pass the path of the XML file location to the method. Once we have passed the file information, the code for binding the tab information matches the `List` code. Since the serialization process returns a generic list of `TabItem` objects, the code will match the `List` example with the exception of the path, of course.

Now that the code is set, let's run the debugger to verify that the code works. You can set a breakpoint at the switch statement in the event method.

One aspect of this method I would like to point out is `//TODO:`, which I marked inside the catch portion of the `try` statement. You will notice that I have not added any true error handling for the system. You should determine how you plan to handle any exceptions. It's a very poor coding practice to have the `catch` block in the `try` statement be empty.

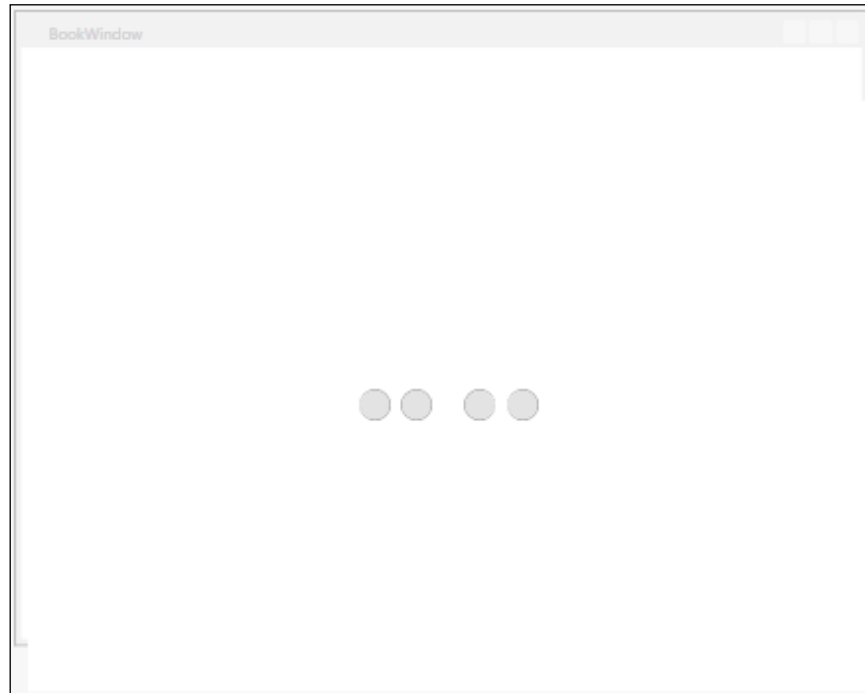
RadBook

The next Telerik WPF control I will review will be the `RadBook` control. This control is excellent for creating documentation for systems or classes. This section will review the binding of the database information to the `RadBook` control. Again, I will be working with a `DataTable` object and a generic list of class objects to demonstrate the power of `RadBook`. We will also be reviewing the important properties and events of `RadBook` to help you understand how to make the paging processing work within the control.

First, let's review the important properties and events that are associated with the `RadBook` control. These properties are as follows:

1. Events:
 - `FoldActivated`: This event fires when the user hovers `fold` with the mouse
 - `FoldDeactivated`: This event fires when `fold` is not active
 - `PageChanged`: This event fires when the page set is changed
2. Main properties:
 - `ItemSource`: This property sets the binding page values within the `RadBook` control. The recommended setup for the pages is to use `DataTemplate` for the page setup. The binding is set up within `DataTemplate`. Telerik recommends that the list object should be of the `ObservableCollection` class.
 - `LeftPageTemplate`: This property determines which `DataTemplate` is to be displayed on the left side of the book.
 - `RightPageTemplate`: This property determines which `DataTemplate` is to be displayed on the right side of the book.
 - `PageFlipMode`: This property determines how the pages are moved using the keyboard. The options are `SingleClick`, `DoubleClick`, or `None`.
 - `IsKeyboardNavigationEnabled`: This property determines if the keyboard can be used to have the book page instead of using the page control.

Now that you are familiar with the main events and properties, which we will be covering, let's start working with the `RadBook` control. The next step in the process will be to create a new window in our WPF project and name the window `BookWindow.xaml`. Once you have created the new window, the next step will be to add a `RadBook` control to the window from the toolbox. The window design should look like the following screenshot:



Once you have finished setting up the interface in the window, you will need to set up the code to work with the `RadBook` control. The using statement required to work with the code is as follows:

```
using PackTPub.ObjectLayer.Products;
```

The first using statement will incorporate the class object for the example. The first step in creating the code to support the book will be to create an instance of the Product class in the BookWindow instance method. This line of code should be placed right below the InitializeComponent() method call. The code should look like the following screenshot:

```

/// <summary>
/// Interaction logic for BookWindow.xaml
/// </summary>
public partial class BookWindow : Window
{
    private Product _prod;

    public BookWindow()
    {
        InitializeComponent();
        _prod = new Product();

        try
        {
            radTestBook.ItemsSource = _prod.FetchList()
        }
        catch (Exception ex)
        {
            //TODO: Make sure to log your exception
        }
    }
}

```

The next step will be to create the buttons to flip the pages when the user clicks the button bar below the book control. The RadBook control has integrated commands to handle the movement of the pages using any button control. Our example will use a RadButton control from Telerik, but any button control will work. Here is an XAML example of one of the buttons:

```

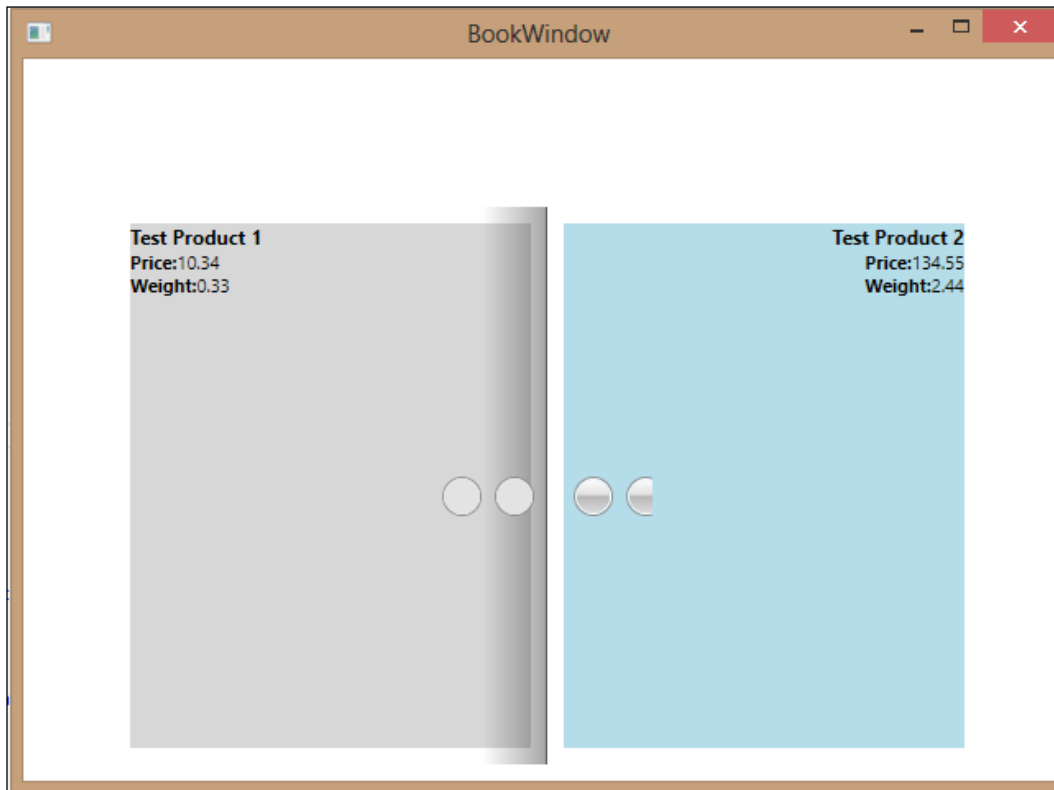
<telerik:RadButton Width="24"
    Height="24"
    Command="telerik:BookCommands.FirstPage"
    CommandTarget="{Binding ElementName=radTestBook}"
    CornerRadius="12"
    ToolTipService.ToolTip="First Page">

```

Notice the two properties in the button, `Command` and `CommandTarget`, in the `RadButton` xaml code. The `CommandTarget` property binds the `RadBook` control on the window to the button. The `Command` property uses the command from the Telerik `RadBook` command to move to the first page of the book. There are four options for `BookCommands`. They are as follows:

- `FirstPage`: This option moves to the first page of the bound book control
- `LastPage`: This option moves to the last page of the bound book control
- `NextPage`: This option moves to the next page of the bound book control
- `PrevPage`: This option moves to the previous page of the bound book control

The book should create five pages, one for each product from the database. Once the book is loaded, you should be able to move through the pages. The following screenshot is an example of how the book should look:



At this point you should have the pages loaded. Use the buttons on the right to move forward and the buttons on the left to navigate backward in the book. The first button on the left will navigate you to the first page, and the last button on the right will navigate you to the last page.

Summary

In this chapter, I have covered two container controls that can be used to display the information in an attractive presentation for users to enter the data. Let's review these controls again:

- `RadTabControl`: You should now be able to create a Telerik `RadTabControl` control, and bind the data to the control by using a `DataTable` or a class object. You should also be able to serialize and deserialize XML for the class objects to load the `RadTabControl` control.
- `RadBook`: You now should understand how to use the `RadBook` control to handle formatting of information from a generic list. You should also be able to navigate through the book using the Telerik book commands with the `RadButton` control.

During this chapter, we discussed the concept of serialization of XML into class objects through C#. Serialization and Deserialization concepts allow you, as the developer, to create XML files that map to your object classes. This concept can allow you to save information from your class object into a file for use later in processing, or bypass the database entirely and store the object class information in XML files.

As an architect, I like to use XML configuration files to build the system controls, similar to the concept of creating `RadTabControl`, as shown in this chapter. I prefer to allow the users to then populate the information in the control as they see fit for their business. My concept is to give the users as much power as they need, and eliminate as much re-coding as possible.

This chapter also covered two container controls from Telerik, `RadTabControl` and `RadBook`. A great way to divide information for user input is by using `RadTabControl`. The design of `RadTabControl` allows you to break up the information into logical pieces for the user to review as they see fit. `RadBook` is a great container for documentation such as a user guide or help documentation. The presentation of the `RadBook` control allows the user to move through the information in a logical format.

In the next chapter, we will be working with the navigation functionality that the `RadControls` from Telerik provide. Specifically, I will cover the Outlook Bar and Menu from Telerik. I will discuss how to load these controls from the database and objects to create a more dynamic interface.

5

Navigation and Dynamic Event Handling

In this chapter, we will demonstrate the use of the Telerik navigation controls and how to configure Telerik controls for dynamic event handling. The navigation controls are used to allow users to navigate through the system to other portions loaded in the system control by using links. We will review the best uses of the navigation controls which we want to cover, then we will discuss the options for dynamically populating these controls through the C# code, and either the database or the XML configuration.

I have selected two controls that I feel give the best examples of the concepts that we want to cover. We will be focusing our attention on the following controls:

- RadOutlookBar
- RadMenu

We will be concentrating our efforts on dynamically loading these controls to make the user interface capable of external configuration to allow for dynamic content. This chapter will also focus on loading these controls based on the security information of the application-authenticated user. The purpose of this design is to allow the options inside the navigation to be determined by the current access level of the user. We will discuss different security methods for handling this information using both Active Directory user groups, and database security where the database is loaded with the access information.

We will also demonstrate the loading of the controls from both the database and configuration files to give the system flexibility in the handling of the dynamic information.

Database setup

The next step, before reviewing the controls in the chapter, will be to make sure that the database is set up for the Telerik projects. The last chapter discussed the `PackTPub` database setup to support the examples used in the previous chapter. This chapter is supported by the same database information and so there is nothing required for this chapter.

You should make sure that you can access the database. If you're not sure, or do not remember if you set up the database, please refer to the *Database setup* section of *Chapter 2, Telerik Editors and How They Work*.

This chapter will also require the XML files to be stored on your machine. If you haven't done so already, create a folder called `LoadConfig` inside the `PackTPub` WPF project's `bin` folder. This folder will be read during the loading of the system to gather the required information for generating the `RadMenu` and `RadOutlookBar` information.

Additional prerequisites

This chapter will require you to have additional resources set up to work with the security code examples. The security for this chapter will have two different types, Active Directory and database security. In order to use the Active Directory examples, you will need to have an Active Directory domain created within the network, which you are using in your development environment. You will also need to add the library, `System.DirectoryServices`, as a reference to your project. The security example uses the `DirectoryServices` library to gather Active Directory information for the Windows domain user on the current system. The code example will gather the group and authentication information for the user's domain account. Once you have added the reference and created the Active Directory domain, you should be able to work with the code examples for this chapter.

New WPF concepts

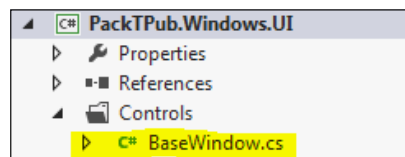
In the first four chapters, we covered the very simple concepts of binding `DataTable` or a list of objects to a Telerik control. In this chapter, we will start to move to more complex concepts such as extending the WPF window class, persisting a class throughout the life of the application (a class to map the Telerik navigation items), and creating a user object for application authentication. The next section will review these concepts and discuss how to use these concepts in a true application.

BaseWindow

The first concept we want to cover is extending the WPF `Window` class to include application-specific information for each window. We want to extend the WPF `Window` class to allow for creating methods and properties that can be accessed throughout our application. Extending the WPF `Window` class allows us to do the following for our application:

- Create a method for building the navigation control based on our `UserEntity` class
- Create a method for logging exceptions that can be used in each subsequent window
- Create a method for handling the loading of a new window based on the selected item in the Telerik navigation control

The next step is to set up the `BaseWindow` class to incorporate the class into each subsequent window class we create in the WPF project. Inside the current WPF project, create a folder called `Controls`, and then create a class called `BaseWindow` in that folder. The result of this process should look like the following screenshot:



Once we've created the `BaseWindow` class, we need to inherit the WPF `Window` class, as shown in the following screenshot:

```
using System.ComponentModel;
using System.Reflection;
using System.Runtime.Remoting;

using Telerik.Windows.Controls;

namespace PackTPub.Windows.UI.Controls
{
    public class BaseWindow : Window
    {
    }
```

The first step is to extend the `Window` class to the `BaseWindow` class. The example for this change is on the last line of the preceding example. We will also need to add the three `using` statements for use in the new `BaseWindow` class. Now that we have the `BaseWindow` class, we will now want to include the class in all the windows we create in the WPF project. Let's create a new window called `OutlookBarWindow.xaml`. This window will have the base class of the `Window` class. We will want to change this in two places to allow for our new base window class to be included. The first change will be in the XAML file. The following is an example:

```
<src:BaseWindow
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:PackTPub.Windows.UI.Controls"
    xmlns:telerik="http://schemas.telerik.com/2008/xaml/presentation"
    Title="OutlookBarWindow" Height="518" Width="693">
    <Grid>
```

Now, we replace the `<Window>` tag with the `<src:BaseWindow>` tag. The `src` reference is created in the second highlighted namespace reference of the preceding code. This reference allows us to use the `BaseWindow` class that we created as the base class for this window. The next steps will be to create the inheritance in the `xaml.cs` file, as shown in the following screenshot:

```
public partial class OutlookBarWindow : BaseWindow
{
    private Request _req;

    public OutlookBarWindow()
    {
        InitializeComponent();
    }

    public OutlookBarWindow(Request req)
        : this()
    {
        //1. Set the local Request instance to the global instance
        req = req;

        //2. Load the RadOutlook bar with the user menu items.
        radOutlookTest = LoadRadOutlook(radOutlookTest, _req.CurrentUser.UserMenuItems);
    }
}
```

Notice the highlighted portion of the preceding code where we have replaced the `Window` reference with the `BaseWindow` class. If you try to do a build now, you should not get any compile errors from the build process. The most common compile error is the lack of the `src` reference in the namespace references of the XAML file. If you get a "namespace prefix `src` not defined" compile error, please make sure that you review the second screenshot in this section for how to create the reference in the XAML file.

The `BaseWindow` class also has methods for generating the Telerik Navigation control information and other Telerik controls. These methods are as follows:

- `LoadRadOutlook`: This method generates a `RadOutlookBar` control with `RadTreeView` for the links from a list of `BarItem` objects
- `LoadRadMenu`: This method generates `RadMenu` from a list of the `BarItem` objects
- `treeSubItem_Click`: This method is meant to be an event handler to take the click event from the navigation control and open a window based on the selected item in the navigation control
- `RefreshGrid`: This method reloads `RadDataGrid` with a list of generic objects
- `LogException`: This method is used to handle the logging and user display of any error that occurs during the execution of the system

These methods are designed to be common methods that could be used on any WPF window class in your application. The purpose of this `BaseWindow` class is to create methods and properties that could be reused throughout your application.

The AppRequest persistence class

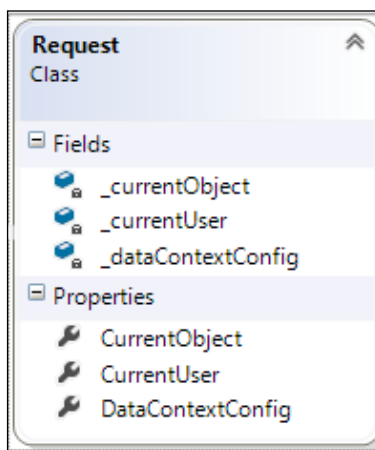
The next new concept we should cover is the idea of persisting application information during the life of the application session. Inside of the WPF project for this book is a class file named `Request.cs`. An instance of this class is created when the application is started by creating an instance in the `Application_Startup` method, as shown below:

```
void Application_Startup(object sender, StartupEventArgs e)
{
    //1. Create UserEntity instance
    _user = new PackTPub.ObjectLayer.Util.UserEntity();
    AppRequest = new Request();
    _user.UserName = "dantest01";
    _user.UserCheckPwd = "dantest01";

    try
    {
        //2. Authenticate the user
        _user.Authenticate();

        //3. Set the user to the Request class property for the CurrentUser
        AppRequest.CurrentUser = _user;
        //4. Pass the Request class to the instance of the Window
        //OutlookBarWindow mainWindow = new OutlookBarWindow(AppRequest);
        BookWindow mainWindow = new BookWindow();
        if (_user.UserLoggedIn)
        {
            mainWindow.Show();
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

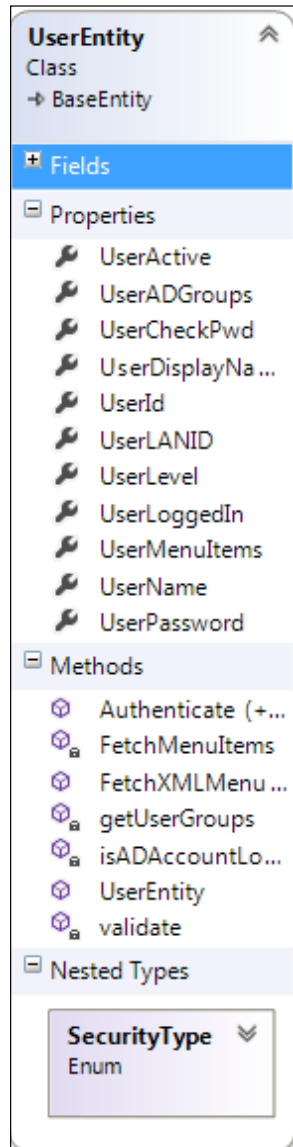
Notice the highlighted line that creates the instance of the `AppRequest` class. This class is a public class that allows this instance to be persisted throughout the application. When we create a new window, we also add and pass the instance of the `AppRequest` class into the instance of the window to allow for any possible changes to the class instance in each window. This instance is passed in the constructor of the `Window` class. We create a second instance method with the `AppRequest` class as a parameter in the method. We then add `: this()` to the end of the method call, as shown in the following screenshot:



This setup for the instance method creates a call to the default instance method to load the `InitializeComponent` method.

The UserEntity class

The last new concept of this chapter will be the user class called `UserEntity`. This class will be used by the application to authenticate the user based on information in the database or in Active Directory. We will use the `UserEntity` class to persist the user information during the life of the application. The following is the class diagram of the `UserEntity` class:



The `Authenticate` methods are the overloaded methods that allow the two types of security, database and Active Directory, in the application. The database version of the `Authenticate` method reads the database to gather the user information from it, and then compares the information to the user input to authenticate the user. There are three exceptions created to work with the authentication process:

- `DisabledException`: This exception is thrown when the user is disabled in the database
- `ExceedsLimitException`: This exception is thrown if the user exceeds the amount of allotted attempts to enter his/her password
- `NotPermittedException`: This exception is thrown if the username does not exist or the password is incorrect

The second `Authenticate` method uses Active Directory to determine the user access. The method takes the `UserPrincipal` object from the `Windows` class. This class is populated through the .NET framework to gather the current Windows domain user information. The method first determines if the current user is locked out of the Windows domain. The method sets the `UserLoggedIn` property to `false` and returns control to the application. If the user account is not locked, the system gathers the group information from Active Directory as well as the user's name, information, and e-mail for notification purposes.

The `FetchMenuItems` and `FetchXMLMenuItems` methods retrieve a list of `BarItem` objects based on the user-level security. This list is later used to populate the `RadNavigation` controls.

The `FetchMenuItems` method queries the database using the user level from the user database information to determine the access list for the current user. The method loops through the access list, generating the `BarItem` classes to later be loaded into the navigation control.

The `FetchXMLMenuItems` method retrieves the access list from an XML file in the `bin` folder of the application. The method takes the path of the XML file and generates an `XDocument` object to load the XML information. The reason we are using an `XDocument` object rather than an `XMLDocument` object is that we plan to use a `Linq` query to retrieve the proper list information based on the user's group information. The `Linq` query takes the group information and queries the `XDocument` object for the user items in the XML file. This XML file is a serialized version of the `BarItem` list of objects. The method then returns the list of the `BarItem` objects to the calling program for loading into the proper control.

The `Request` class has a property for the `UserEntity` class. This property is set during the authentication process. If the user is not authenticated, the system will simply shut down and not allow access to the main window. The following screenshot is the code for this process:

```
public void Authenticate()
{
    IDbManager dbMgr = new DBManager(this.Config("TELERIKDB", ""));
    //dbMgr.ConnectionString = Config.getValue("DBConnStr", "");
    string query = "SELECT * FROM Users WHERE UserName = @id";

    try
    {
        dbMgr.Open();
        dbMgr.CreateParameters(1);
        dbMgr.AddParameters(0, "@id", _userName);

        dbMgr.ExecuteReader(System.Data.CommandType.Text, query);

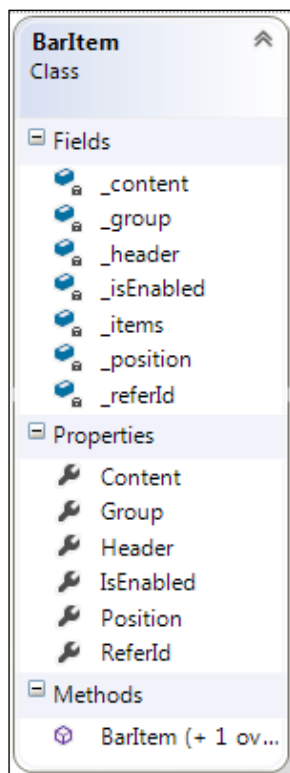
        if (dbMgr.DataReader.Read())
        {
            _userLevel = dbMgr.DataReader.GetInt32(4);
            _userActive = dbMgr.DataReader.GetBoolean(3);
            _userPassword = dbMgr.DataReader.GetString(2);
            _userId = dbMgr.DataReader.GetInt32(0);
        }
        else
            _userLoggedIn = false;

        validate();
    }
    catch (ExceedsLimitException ex1)
    {
        throw (ex1);
    }
    catch (NotPermittedException exp)
    {
        throw (exp);
    }
    catch (DisabledException exd)
    {
        throw (exd);
    }
    catch (Exception ex)
    {
        throw;
    }
}
```

Notice that when we call the `authenticate` method, the system checks if the user information should be logged into the system. The `authenticate` method is an overloaded method. There are two different options based on the type of security we plan to use in the system. The first method reads the database to determine if the user has access to the system and if the password is correct. The second option uses Active Directory to read the user group information to determine if the user is in an Active Directory group that has access to the system.

The BarItem class

The next aspect of the system we want to review is the `BarItem` class. The design of this class is set up to be used by the `RadOutlookBarItem` and `RadMenuItem` classes. The reason for this setup was to match the properties of the Telerik item classes. The following is a screenshot of the class design:



The main properties include `Header`, `Position`, and `ReferId`. The `Header` property is set up to match the `Header` property from the `RadOutlookBarItem` and `RadMenuItem` classes. The `Position` property orders the items on the system for proper display, and the `ReferId` property creates the structure of the view. The `ReferId` values are set up so that if the value of the `ReferId` property is zero, this item becomes the header, otherwise the rest of the items become the subitems of the main item.

Now that we've covered the new concepts within this chapter, let's start to work with the navigation containers for Telerik. The next sections of this chapter will cover the `RadOutlookBar` and `RadMenu` controls using the concepts from this section to load these controls based on the level of user access.

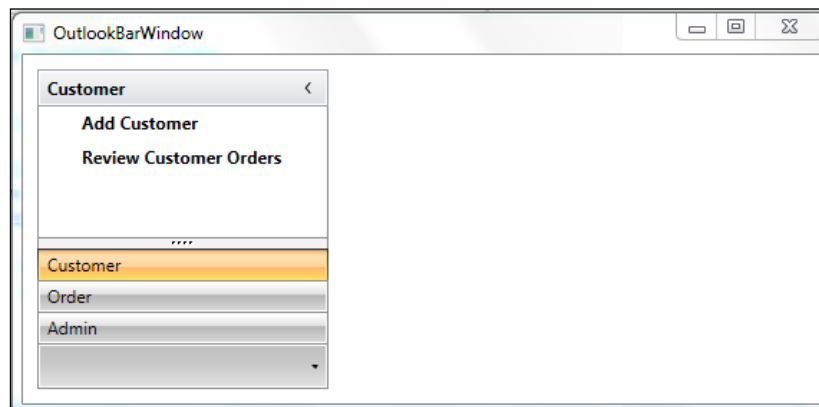
RadOutlookBar

The first control I will review in this chapter will be the `RadOutlookBar` control. This control is unique to the Telerik toolset. The standard Visual Studio control library does not contain a control like the `RadOutlookBar` control. The first example I will review will be to set the binding of the control's tabs using both a list of class objects from the object layer, and the database to populate the list of objects and authenticate the user.

The first step will be to create a new window in the current WPF project; name this new window `Chap5OutlookWindow.xaml`. Once you have created the new window, you will need to add the `RadOutlookBar` control:

- `RadOutlookBar`: Name the outlook bar `RadTestOutlook`

The final appearance of the window should be the following screenshot:




Notice that the `RadOutlookBar` control is already loaded with the options. This is the fully functioning example, but gives you an idea of how the window should look once we've finished this section.

RadOutlook with GenericList, DataBinding, and database security

This is the first example of populating the RadOutlookBar control with the menu links from the database. The concept we are demonstrating is that the menu links inside the RadOutlookBar control will be populated based on the user that is authenticated into the system. In this example, we will be using the database security. This means that the database will store the username, password, and security link information for the RadOutlookBar control.

Since we don't have a login form to gather the username and password, we will create the user information within the App.xaml.cs file instead. This will allow us to simulate the login process and authenticate the user in the system.

 This should not be done for a production application—this setup is for demonstration purposes only. Hard-coding a username and password is always a bad practice in a real-world application.

The following screenshot shows how the code should look inside the App.xaml.cs file:

```
void Application_StartUp(object sender, StartupEventArgs e)
{
    //1. Create UserEntity instance
    user = new ObjectLayer.Util.UserEntity();

    //2. Set the type of security
    ObjectLayer.Util.UserEntity.SecurityType type = GetSecurityType(Config("SecurityType", "DB"));
    AppRequest = new Request();

    try
    {
        //3. Authenticate the user
        if (type == ObjectLayer.Util.UserEntity.SecurityType.AD)
        {
            _user.UserADGroups.Add("Order");
            _user.FetchXMLMenuItems();
            _user.Authenticate(GetUser(Environment.UserName));
        }
        else
        {
            _user.UserName = "dantest01";
            _user.UserCheckPwd = "dantest01";
            _user.Authenticate();
        }

        //3. Set the user to the Request class property for the CurrentUser
        AppRequest.CurrentUser = _user;
        //4. Pass the Request class to the instance of the Window
        MenuWindow mainWindow = new MenuWindow(AppRequest);
        if (_user.UserLoggedIn)
        {
            mainWindow.Show();
        }
    }
}
```

Let's review the preceding code. The first step is to create an instance of the `UserEntity` class. The next step is to determine the type of security the system will be using to authenticate the user. The `app.config` class has a setting for the type of security, as shown in the following line of code:

```
<add key="SecurityType" value="DB"/>
```

This key value is passed to the system and used by the system to determine which type of security is used in the `App.xaml.cs` class. The value is read by the system; it sets an enum property from the `UserEntity` class called `SecurityType`. This enumeration is then evaluated to determine the security setup for the system. The next step is to set the `UserName` and `UserChkPwd` properties. Once these properties are set, we can call the `Authenticate` method to verify the user information against the database. The next step will be to set the `CurrentUser` property in the application request class. This object is passed to each window to persist base application information to each window.

The final step will be to create an instance of the `OutlookBarWindow` class to open the window. Notice the code for the instance; we are passing the `AppRequest` object to the window using the instance method of the window. The following screenshot is of the instance code:

```
private Request _req;

public OutlookBarWindow()
{
    InitializeComponent();
}

public OutlookBarWindow(Request req)
    : this()
{
    //1. Set the local Request instance to the global instance
    req = req;

    //2. Load the RadOutlook bar with the user menu items.
    radOutlookTest = LoadRadOutlook(radOutlookTest, _req.CurrentUser.UserMenuItems);
}
```

Now that the window code is set up, let's review the `LoadRadOutlook` method from the `BaseWindow` class. This method requires two values: the `OutlookBar` object to be loaded and a generic list of `BarItem` objects. The `BarItem` class takes the information from the database and loads the `RadOutlookBar` control using `BarItems` to create the `RadOutlookBarItem` objects to load the `RadOutlookBar` control. The `LoadRadOutlook` method also creates a `RadTreeView` control of the options within each `RadOutlookBarItem` object.

The database version then handles the gathering `BarItem` and loads the `RadOutlookBar` control using a method called `FetchMenuOptions`. This method takes `BarItems` from the database query and creates a `RadOutlookBar` object with all the menu options for the current user.

```
private void FetchMenuItems()
{
    IDbManager dbMgr = new IDbManager(this.Config("TELERIKDB", ""));
    _userMenuItems = new List<BarItem>();
    BarItem item;
    string query = "select BarItemText, b.BarItemId, b.ReferBarItemId from BarItem b, Users u, BarItem_User ub " +
        "where b.BarItemId = ub.BarItemId " +
        "and u.UserLevelId = ub.UserLevelId " +
        "and UserName = @id " +
        "order by b.BarItemId";

    try
    {
        int pos = 1;

        dbMgr.Open();
        dbMgr.CreateParameters(1);
        dbMgr.AddParameters(0, "@id", _userName);

        dbMgr.ExecuteReader(System.Data.CommandType.Text, query);

        while (dbMgr.DataReader.Read())
        {
            item = new BarItem(dbMgr.DataReader["BarItemText"].ToString(), "");
            item.IsEnabled = true;

            if (dbMgr.DataReader["ReferBarItemId"] == System.DBNull.Value)
            {
                item.Position = Convert.ToInt32(dbMgr.DataReader["BarItemId"]);
                item.ReferId = 0;
                pos++;
            }
            else
            {
                item.Position = pos;
                item.ReferId = Convert.ToInt32(dbMgr.DataReader["ReferBarItemId"]);
            }

            _userMenuItems.Add(item);
        }
    }
    catch (Exception ex)
    {
    }
}
```

This method gathers the menu information based on the current username and generates a generic list of the `BarItem` classes. This list is used by the `BaseWindow` class to load the `RadOutlookBar` control.

RadOutlookBar using generic list binding with XML security

The next method for populating the `RadOutlookBar` control on the window will be to use serialized XML to populate the generic list of objects to load the `RadOutlookBar` control. The XML file has the group information and pulls the `BarItem` objects based on the user's group information. The example we will be using has the group information hard-coded into the group list, but this is just an example. You should not hard-code the groups in a real-world application. Here is the method for gathering the menu items from the XML file:

```
public void FetchXMLMenuItems()
{
    try
    {
        XDocument xdoc = XDocument.Load("c:\\dev\\book\\UserMenuItems.xml");

        List<BarItem> fullList = (List<BarItem>) Deserialize<BarItem>(xdoc);

        var userMenuItems = fullList.Where(p => p.Group.Contains(_userADGroups[0]));

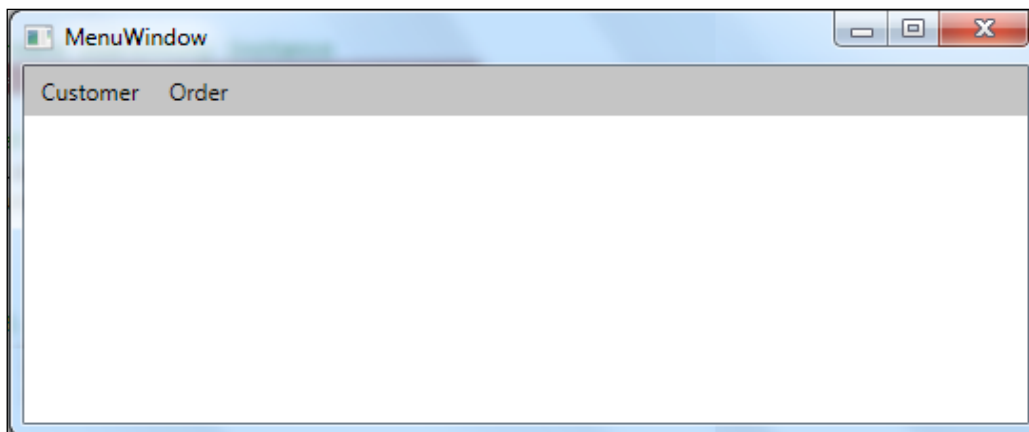
        foreach (var barItem in userMenuItems)
            _userMenuItems.Add(barItem);
    }
    catch (System.IO.FileNotFoundException fex)
    {
        throw;
    }
}
```

Notice that we first load the `XDocument` object, then deserialize the `XDocument` object to gather the list of `BarItem` objects. Once the full list is loaded, we use a `Linq` query to gather the items based on the user group for the current user. Now that we have that new list, we can populate the `UserMenuItems` property for use in the application. The application then calls the `LoadOutlookBar` method to generate the `RadOutlookBar` control again.

RadMenu

The next Telerik WPF control I will review will be the RadMenu control. This control will be used to create a menu system at the top of a WPF window to create access to the requested information based on the current authenticated user. The design of the `app.xaml.cs` class will be the same as the work we did in the RadOutlookBar control's setup, but we will add one additional method to generate the RadMenuItem objects to load the menu on the new window.

The first step will be to create a new window for the menu. This window will contain only one control, and that control will be the RadMenu control from the toolkit. The new window should look like the following screenshot:



The menu should be empty when you add the control from the Toolkit. The window shown in the preceding screenshot is an example of what it will look like after being loaded using the example code.

This example will use Active Directory rather than the database to handle the security. The first change will be to set the `app.config` setting for `SecurityType` to AD, rather than DB, as shown in the following line of code:

```
<add key="SecurityType" value="AD"/>
```

This change will cause the system to use the Active Directory security setup instead of the database setup. The `App.xaml.cs` code picks up the `SecurityType` setting, and then determines the security call to the `UserEntity` class. In this case, the `Authenticate` method for Active Directory is called. The code for the Active Directory portion of the security system is highlighted in the following screenshot:

```
void Application_StartUp(object sender, StartupEventArgs e)
{
    //1. Create UserEntity instance
    user = new ObjectLayer.Util.UserEntity();

    //2. Set the type of security
    ObjectLayer.Util.UserEntity.SecurityType type = GetSecurityType(Config("SecurityType", "DB"));
    AppRequest = new Request();

    try
    {
        //3. Authenticate the user
        if (type == ObjectLayer.Util.UserEntity.SecurityType.AD)
        {
            _user.UserADGroups.Add("Customer");
            _user.FetchXMLMenuItems();
            //user.UserLoggedIn = true;
            _user.Authenticate(GetUser(Environment.UserName));
        }
        else
        {
            _user.UserName = "dantest01";
            _user.UserCheckPwd = "dantest01";
            _user.Authenticate();
        }

        //3. Set the user to the Request class property for the CurrentUser
        AppRequest.CurrentUser = _user;
        //4. Pass the Request class to the instance of the Window
        MenuWindow mainWindow = new MenuWindow(AppRequest);
        if (_user.UserLoggedIn)
        {
            mainWindow.Show();
        }
    }
}
```

The first difference we need to notice is that no username or password is set for Active Directory security. Since we expect the user to already be authenticated in Active Directory, we use the `UserPrincipal` object to determine if the user is currently logged in to the domain. If the user is not logged in to the domain, an exception will be triggered. We add the `Order` group to the list of groups to force the system to pick up the security information from the XML file. There are three options for the groups to test the security. They are as follows:

- Order
- Customer
- Admin

The `UserEntity` class has three methods that support the Active Directory security. They are as follows:

- `Authenticate`: This method takes `UserPrincipal`, which is an object populated with the current Windows domain information, and populates the `UserEntity` object with the user information.
- `isADAccountLocked`: This method queries Active Directory to determine if the current user's account is locked. If so, the user is denied access to the system by setting the `UserLoggedIn` property to `false`.
- `getUserGroups`: This method queries Active Directory for the group information for the current user. This method populates the `UserADGroups` property.

The following is the example code of these methods from the `UserEntity` class:

```
public void Authenticate(UserPrincipal userPrincipal)
{
    // 1. Check to see if the user account is locked in AD
    if (isADAccountLocked(userPrincipal))
    {
        this.UserLoggedIn = false;
        return;
    }

    // 2. Gather the AD information for the current user
    this.UserDisplayName = userPrincipal.GivenName + " " + userPrincipal.Surname;
    this.UserLANID = userPrincipal.SamAccountName;
    // 3. Gather the AD groups the user is associated with in AD.
    this.UserADGroups = getUserGroups(userPrincipal);
}

private bool isADAccountLocked(UserPrincipal userPrincipal)
{
    DirectoryEntry dEntry = new DirectoryEntry(userPrincipal.DistinguishedName);

    return Convert.ToBoolean(dEntry.InvokeGet("IsAccountLocked"));
}

private List<string> getUserGroups(UserPrincipal userPrincipal)
{
    List<string> groups = new List<string>();
    foreach (Principal p in userPrincipal.GetGroups())
    {
        groups.Add(p.Name);
    }

    return groups;
}
```

There are three steps in the `Authenticate` method to gather the domain information for the current user. First we verify that the user's account is not locked. If the account is locked, we force the user out of the system. Next we gather the user information from the `UserPrincipal` object to populate the `UserEntity` properties. Lastly, we generate a list of the Active Directory groups the user belongs to in the domain. This list is used later to set up the security for the `RadMenu` control. If you would like to read further about the `UserPrincipal` class from the .NET framework, here is an excellent article from MSDN:

[http://msdn.microsoft.com/en-us/library/system.directoryservices.accountmanagement.userprincipal\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.directoryservices.accountmanagement.userprincipal(v=vs.110).aspx)



You must include the `System.DirectoryServices` and `System.DirectoryServices.AccountManagement` libraries in your `PackTPub.ObjectLayer` project.

The next step will be to create the code for loading the `RadMenu` objects to display the menu information based on the current user. We will be using the same method from the `RadOutlookBar` information to gather the `BarItem` object list, but this time we will call the `LoadRadMenu` method to create the menu based on the `BarItem` object list from the `UserEntity` class. The following screenshot is the example code for the `MenuWindow` class:

```
/// <summary>
/// Interaction logic for MenuWindow.xaml
/// </summary>
public partial class MenuWindow : BaseWindow
{
    private Request _req;

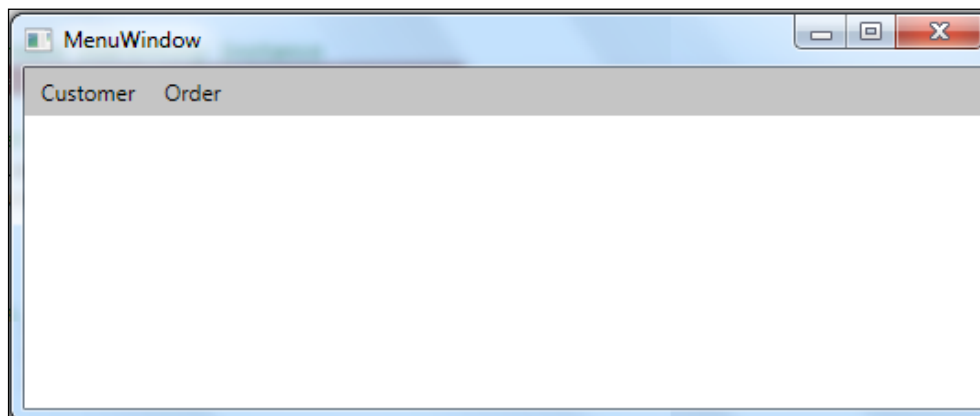
    public MenuWindow()
    {
        InitializeComponent();
    }

    public MenuWindow(Request req)
        : this()
    {
        //1. Set the local Request instance to the global instance
        _req = req;

        //2. Load the Menu bar with the user menu items.
        radMenuTest = LoadRadMenu(radMenuTest, _req.CurrentUser.UserMenuItems);
    }
}
```

Notice that the code for the `RadMenu` control is the same as the code for the `RadOutlookBar` control. This setup is by design. We want a consistent process for handling user security so that our choice of navigation should not impact the security setup for the application. The `LoadRadMenu` method takes the list of `BarItem` objects and creates the `RadMenuItem` objects to be loaded into the `RadMenu` control. This method takes the list of `BarItem` objects and creates `RadMenuItem` to display the heading information, then creates a menu subitem based on the level of the `BarItem` class. The `ReferId` property determines the level of the menu item to generate the menu in the correct format.

Once we have the menu loaded, the window should load like the following screenshot:



Notice that the menu now displays the user-level category information exactly like the `RadOutlookBar` example from the previous section in the chapter. The menu should display two categories in the main links of the menu and two options under each main menu link. If you would like to test this concept further, you can change the group in the `App.xaml.cs` file on line 40. The options for the groups are Admin, Order, and Customer. The Admin group should return three main menu links, including an Admin option as the last option. The Customer group should only create one main menu link.

Summary

In this chapter, I covered two navigation controls that can be used to display user access information based on the access level of the current user who has logged in to the application. The navigation controls are loaded with the navigation links based on the group information from the user authentication process. Let's review these controls again:

- `RadOutlookBar`: You should now be able to create a Telerik `RadOutlookBar` control and bind the data to the control by using a database query or an XML file to gather the information. You should also be able to create a `Linq` query to gather information from an XML file to create a list of class objects to load the `RadOutlookBar` control.
- `RadMenu`: You should now understand how to use and load the `RadMenu` control to handle formatting the information from a generic list with Active Directory security. You should also be able to understand how to create a class to handle the loading of both the controls with the common information.

In the course of this chapter we discussed the concept of serialization of XML into class objects through C#. Serialization and deserialization concepts allow you, as the developer, to create XML files that map to your object classes. This concept can allow you to save information from your class object into a file for use later in processing, or bypass the database entirely and store the object class information in XML files.

This chapter also introduced the concepts of securing an application using two different methods, database information and Active Directory. The database security demonstrated how the user SQL Server tables store the username, password, and user-level information, as well as access the information for the application. The Active Directory example showed us how we can create access levels (based on the AD groups which are assigned to a domain user) in the Active Directory.

As I discussed in the previous chapter, I like to use the XML configuration files to build for security information, similar to the concept in this chapter for creating the `RadOutlookBar` and `RadMenu` controls. I like to allow the users to then populate the information in the control as they see fit for their business. My concept is to give the users as much power as they need, and eliminate as much re-coding as possible.

This chapter also covered two navigation controls from Telerik, `RadOutlookBar` and `RadMenu`. The `RadOutlookBar` control is a nice way to create a fancy presentation of your system navigation. The `RadOutlookBar` control gives the user a familiar navigation if they are already acquainted with Outlook, and gives you flexibility in the presentation since you can add any control inside the `RadOutlookBarItem` class. We used a `RadTreeView` control to display the navigation items, but this is just an example. You are more than welcome to try other controls inside the `RadOutlookBarItem` class. The `RadMenu` control looks like this typical Windows menu, so it's very familiar to most Windows users.

In the next chapter, we will be working with the `Scheduling` functionality that `RadControls` from Telerik provides. I will specifically cover the Gantt chart and the calendar from Telerik. I will discuss how to load these controls from the database and objects to create a more dynamic interface.

www.SoftGozar.com

6

Telerik Scheduling and Object Bound Loading

In this chapter, we will demonstrate and discuss the use of the Telerik scheduling control, `RadGanttView`, and how to load this control for acquiring the data from a database. At this point in the book, we have worked with standalone and container controls for data entry on WPF windows. The `RadGanttView` control allows users to work with scheduling tracking information that is based on the system. We will review a method for setting up a class to populate the `RadGanttView` control, and then we will discuss the options for dynamically populating this control through the C# code from the database.

We will be focusing our efforts on loading the information into the `RadGanttView` control from the database based on the currently authenticated user and the overall tasks with summary categorization. We will discuss the organization of this information and the use of the `IGanttTask` interface class from Telerik.

We will also demonstrate the loading of the control from the database to give the system flexibility in the handling of the dynamic information. I hope this chapter gives you a real-world perspective on the use of this scheduling control.

New object-oriented concepts

In the first five chapters, we covered very simple object-oriented concepts of creating classes and simple class inheritance. In this chapter, we will start to move to more complex concepts such as using an interface class to inherit a base for a class.

An interface class is a required blueprint for a class. The methods and properties of an interface class cannot be used alone, but they determine how all classes that inherit this class should be designed. When an interface class is inherited, the Visual Studio compiler will compare the interface class to the new class. If the new class does not include all the public methods and properties of the interface class, the compiler will consider this as an error. Here is a link from MSDN to give you further information: <http://msdn.microsoft.com/en-us/library/ms173156.aspx>.

The IGanttTask interface class

The first aspect we want to cover is using the `IGanttTask` interface as a secondary base for our new `CommonTask`. The `IGanttTask` class is the basis for all the objects that can be loaded into the `RadGanttView` control.

The first step will be to create the proper references for the interface class. This is shown in the following screenshot:

```
using PackTPub.DataLayer;
using PackTPub.DataLayer.Data;

using Telerik.Windows.Controls.GanttView;
using Telerik.Windows.Controls.ScheduleView;
using Telerik.Windows.Core;

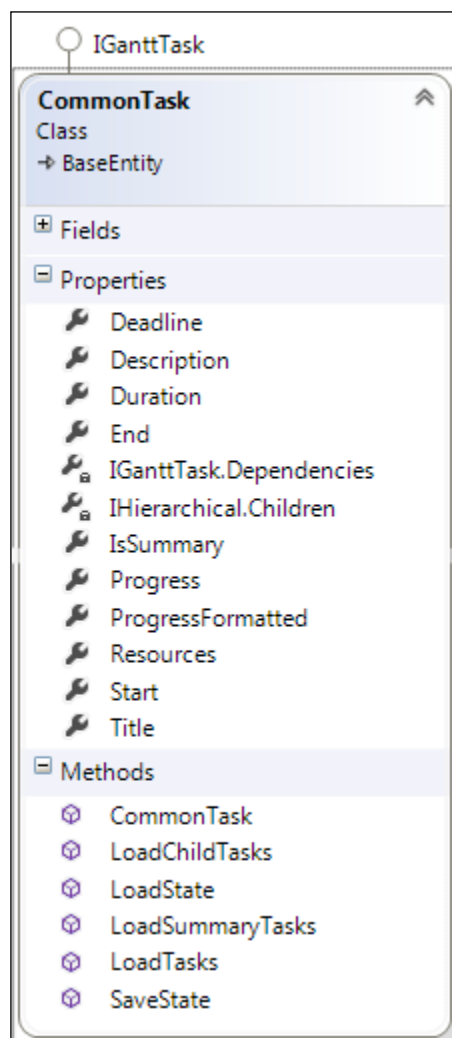
namespace PackTPub.ObjectLayer.Util
{
    public class CommonTask : BaseEntity, IGanttTask
    {
    }
}
```

When we create a new `CommonTask` class, we use the `BaseEntity` and `IGanttTask` classes as inheritance for the `CommonTask` class. We want to include the data-layer assemblies to allow database access, and we want the `RadGanttView` and `ScheduleView` assemblies to allow for the `CommonTask` class to include the proper information. Notice that we have the `BaseEntity` class listed before the `IGanttTask` class in the inheritance list. This alignment is required by the C# compiler. C# does not support multiple base class inheritance.

You can inherit from one base class, but all other inheritance classes must be the interface classes. Here is a great link to read over and understand C# inheritance: http://www.tutorialspoint.com/csharp/csharp_inheritance.htm.

The CommonTask class

The next aspect of the system we want to review is the `CommonTask` class. The design of this class is set up to be used by the `RadGanttView` control. The reason for this setup was to match the properties of the Telerik item classes. The following is a screenshot of the class design:



Notice that the `BaseEntity` class is identified under the `CommonTask` class name, but the `IGanttTask` interface class is marked above the class in the diagram. This design shows how the interface class can be implemented in multiple classes.

The only property in the `CommonTask` class that is not in the `IGanttTask` interface is the `IsSummary` property. We will use this property to display a hierarchy in the `RadGanttView` control. The `RadGanttView` control allows the developer to create a tree view-like structure for displaying the summary tasks at the top of the tree and the subtasks under the summary task. The database tables created for this chapter were designed with this hierarchy in mind.

Now that we've covered the new concepts within this chapter, let's start to work with the `RadGanttView` control for Telerik. The next sections of this chapter will cover this control using the concepts from this section to load the control for user-based information and an overall view of the tasks in the `RadGanttView` control.

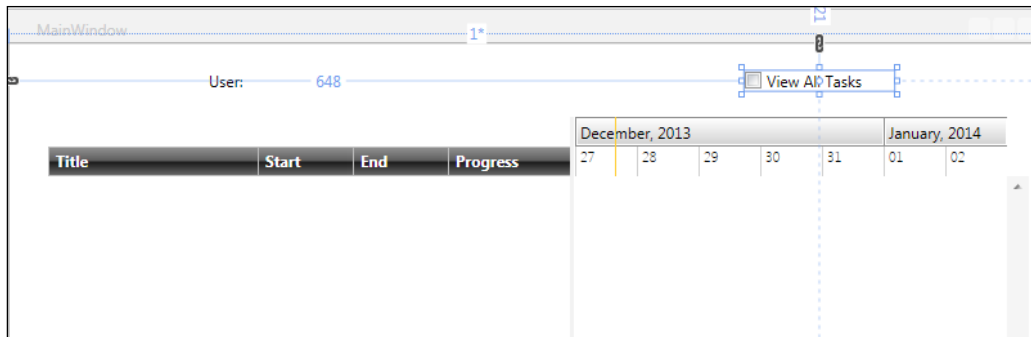
RadGanttBar

The Telerik control that I will review in this chapter will be the `RadGanttView` control. This control is unique to the Telerik toolset. The standard Visual Studio control library does not contain a control like the `RadGanttView` control. The first example I will review will be to set the binding of the control's `TasksSource` property. This loads the control with tasks based on the user assigned to these tasks for giving the illustration of the `RadGanttView` control to look like a Microsoft Project.

The first step will be to create a new window in the current WPF project (name this new window `Chap6GanttWindow.xaml`). Once you have created the new window, you will need to add the following controls to it:

- `RadGanttView`: Name the gantt view `radGanttExmaple`
- `RadComboBox`: Name the combobox `radComboUser`
- `CheckBox`: Name the checkbox `checkAllTasks`
- `Label`: Set the content to **User**:

The final appearance of the window should be like the following screenshot:



The purpose of the checkbox is to determine the manner in which we will load the RadGanttView control. If the checkbox is checked, we will load all the tasks with a summary task example. Otherwise, we will load the RadGanttView control based on the user information from the combobox.

RadGanttView with user task filtering

The first example of populating the RadGanttView control will be to create a filtering of the current tasks in the RadGanttView control based on the current authenticated user. We will accomplish this by populating the combobox on the window with a list of users from the database. We will use this list to load the RadGanttView control with the tasks assigned to the selected user.

The first step in the code will be to query the database for the current users in the database and load that list into the combobox. We will determine whether to load the combobox based on the checkbox on the window. The following screenshot is the Window_Loaded event from the C# code, as an example:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        if (checkAllTasks.IsChecked == false)
        {
            radComboUser.ItemsSource = new UserEntity().FetchUsers();
            radComboUser.SelectedValuePath = "UserId";
            radComboUser.DisplayMemberPath = "UserName";
        }
        else
            loadGanttControl(false);
    }
    catch (Exception ex)
    {
        //TODO: Don't forget your logging!
    }
}
```

Let's review the code for the `Window_Loaded` event. The `loadGanttControl` method will be used to load the data into the `RadGanttView` control based on the checkbox. If you need a refresher on how to load the combobox, please refer to *Chapter 2, Telerik Editors and How They Work*. The `FetchUsers` method returns a generic list of user objects to be loaded into the combobox from the database.

Now that we've reviewed how to load the combobox, next we need to create the code to handle the selection of the user from the combobox. This code will be very similar to the code from *Chapter 2, Telerik Editors and How They Work*, where we loaded the combobox with customer information and then loaded `RadGridView` with the `Order` information from the database. The following example will take `UserId` from the combobox and query the database for the current tasks:

```
private void radComboUser_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    try
    {
        loadGanttControl(true);
    }
    catch (Exception ex)
    {
        //TODO: Don't forget your exception logging!
    }
}

private void loadGanttControl(bool user)
{
    if (user)
    {
        int userId = Convert.ToInt32(radComboUser.SelectedValue);

        radGanttExample.TasksSource = new CommonTask().LoadTasks(userId);
    }
    else
        radGanttExample.TasksSource = new CommonTask().LoadSummaryTasks();
}
```

Let's review the preceding code; the first step is to gather `SelectedValue` from the combobox and convert that value to an integer to be passed to the `LoadTasks` method in the `CommonTask` class. The `LoadTasks` method queries the database based on the selected `UserId` and passes back a generic list of `CommonTask` objects. Since these objects inherit the `IGanttTask` interface, the list can be attached to the `TasksSource` property of the `radGanttExample` control on the window. The following is the code for the `LoadTasks` method:

```
public ObservableCollection<CommonTask> LoadTasks(int userId)
{
    _children = new ObservableCollection<CommonTask>();
    CommonTask comm;
    IDbManager dbMgr = new DBManager(this.Config("TELERIKDB", "TELERIKDB"));
    string query = "select TaskTitle, TaskStartDate, TaskEndDate, TaskProgress, UserName " +
        "from Tasks t, Users u, TaskResources tr " +
        "where t.TaskId = tr.TaskId " +
        "and u.UserId = tr.TaskUserId " +
        "and u.UserId = @id";

    try
    {
        dbMgr.Open();
        dbMgr.CreateParameters(1);
        dbMgr.AddParameters(0, "@id", userId);
        dbMgr.ExecuteReader(System.Data.CommandType.Text, query);

        while (dbMgr.DataReader.Read())
        {
            comm = new CommonTask();
            comm.Start = Convert.ToDateTime(dbMgr.DataReader["TaskStartDate"]);
            comm.End = Convert.ToDateTime(dbMgr.DataReader["TaskEndDate"]);
            comm.Title = dbMgr.DataReader["TaskTitle"].ToString();
            comm.Progress = Convert.ToDouble(dbMgr.DataReader["TaskProgress"]);

            _children.Add(comm);
        }
    }
    catch (Exception ex)
    {
        throw;
    }

    return _children;
}
```

The key component of this method is the creation of `ObservableCollection` of the `CommonTask` objects. The reason for using `ObservableCollection` rather than a simple generic list is that `ObservableCollection` implements the `INotifyCollectionChanged` interface class. This allows you to create event handling through WPF binding. When an object in the collection changes, you can create an event handler to determine what should be done (with the change) to that object. The rest of the code is based on the examples that we created in *Chapter 2, Telerik Editors and How They Work*.

RadGanttView displaying the tasks with a summary task

The next method for populating the RadGanttView control on the window will be to display the tasks with a summary task. This example will take the advantage of the TreeView display in the RadGanttView control. The display we want to replicate is the view in Microsoft Project, where a summary task is displayed above the tasks that the task summarizes. The first step is to set up the XAML for the window to bind the properties of the CommonTask class to the radGanttExample control on the window, as shown in the following screenshot:

```
<telerik:RadGanttView x:Name="radGanttExample" HorizontalAlignment="Left"
    Margin="26,56,0,0" VerticalAlignment="Top" Width="844" Height="381">
  <telerik:RadGanttView.Columns>
    <telerik:TreeColumnDefinition Header="Title" MemberBinding="{Binding Title}" Width="240">
      <telerik:TreeColumnDefinition.CellTemplate>
        <DataTemplate>
          <TextBox Text="{Binding Title}" />
        </DataTemplate>
      </telerik:TreeColumnDefinition.CellTemplate>
    </telerik:TreeColumnDefinition>
    <!--<telerik:ColumnDefinition ColumnWidth="175" Header="Title" MemberBinding="{Binding Title}"/>-->
    <telerik:ColumnDefinition ColumnWidth="75" Header="Start" MemberBinding="{Binding Start}"/>
    <telerik:ColumnDefinition ColumnWidth="75" Header="End" MemberBinding="{Binding End}"/>
    <telerik:ColumnDefinition MemberBinding="{Binding ProgressFormatted}" Header="Progress" Width="100"/>
  </telerik:RadGanttView.Columns>
</telerik:RadGanttView>
```

Notice the highlighted areas of the XAML code for the control in the preceding screenshot. The TreeColumnDefinition template allows the control to use the IsSummary property to create a tree structure of the tasks. If the task object is a summary task, the tasks after that point are treated as subtasks until the next summary task is found in the collection. The rest of the binding handles the properties of the CommonTask class to display the property information in the control.

Now that we have the RadGanttView control set up for the summary task, we need to create the code to return ObservableCollection to display the summary task inside the control. The CommonTask class has two methods that support the summary task structure. These two methods are as follows:

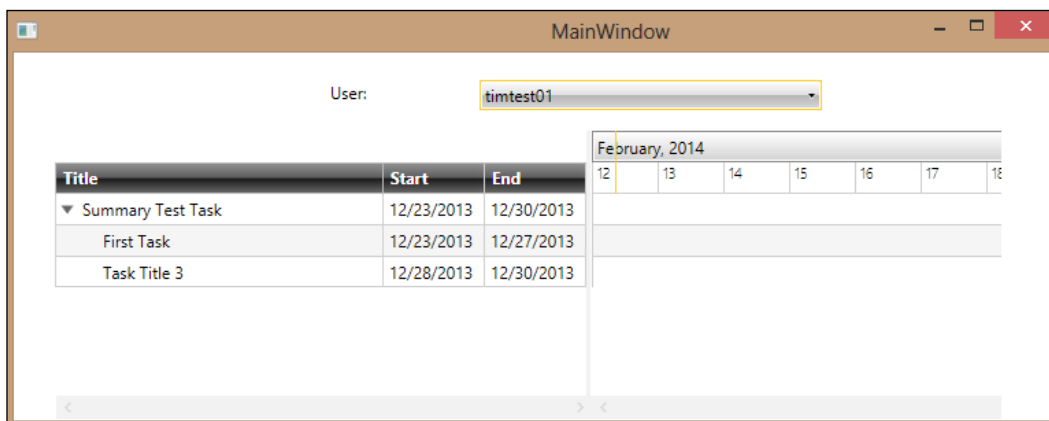
- LoadSummaryTasks: This method queries the database for summary tasks. This method returns ObservableCollection of the CommonTask objects, similar to the LoadTasks method.
- LoadChildTasks: This method queries the database for tasks that are the children of the summary task. This method is called from the LoadSummaryTasks method to gather the child tasks of the current summary task. The method creates ObservableCollection of the child tasks to be loaded into the CommonTask property, Children.

The code will create the hierarchy of tasks that we require to load the `RadGanttView` control with the proper structure for the summary tasks. The next step is to load the information into the `RadGanttView` control. The following screenshot shows the difference in the coding for the summary tasks. We are no longer concerned with the user, but we want all the tasks to display the summary information.

```
private void radComboUser_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    try
    {
        loadGanttControl(true);
    }
    catch (Exception ex)
    {
        //TODO: Don't forget your exception logging!
    }
}

private void loadGanttControl(bool user)
{
    if (user)
    {
        int userId = Convert.ToInt32(radComboUser.SelectedValue);
        radGanttExample.TasksSource = new CommonTask().LoadTasks(userId);
    }
    else
    {
        radGanttExample.TasksSource = new CommonTask().LoadSummaryTasks();
    }
}
```

Please notice the last line of the `loadGanttControl` method. We use the `LoadSummaryTasks` method to populate the `TasksSource` property of the `radGanttExample` control. This code setup will display the tasks in the hierarchical structure shown below:



The image displays the new, hierarchical structure of the tasks based on the `IsSummary` task property. You should also notice that the summary task can be expanded and collapsed with the small tree control to the left of the summary task text. This method for loading the tasks would be great for the overall project manager to review all the tasks, rather than the original example, which would be user-based.

Summary

In this chapter, we have covered the scheduling control `RadGanttView`. We have reviewed how to load this control using both a user-based model as well as a summary-tasks-based model. We also discussed how to use the database to load the `RadGanttView` control with a hierarchical display to show the summary tasks.

In this chapter, we discussed and elaborated further on the use of an interface class and how you must implement an interface class if you choose to inherit the class information. Remember that the inheritance of an interface class requires you to implement each method and property in the class. This setup is different from inheriting a base class. You simply inherit the class, and all the properties and methods become available in the inheriting class. We also learned that you can not only inherit from one base class in `C#`, but you can also inherit from as many interface classes as you wish.

Last but not least, we discussed what the differences between `ObservableCollection` and a generic list are, and why you might want to use `ObservableCollection`. The `INotifyCollectionChanged` interface class is implemented by `ObservableCollection`. This interface allows you to create an event handler to work with any changes to the collection while the information is loaded inside of a control or window. This difference makes the handling changes to the collection more straightforward than a normal generic list.

Well, we've come to the end of the last chapter in the book. I hope you have found the information contained within this book to be helpful and informative. I have tried to create exercises that I felt reflected real-world examples so that the book would be relevant to any work you may be doing in your professional development. Thank you for your patronage and I hope you will read my next book, coming soon!

Index

A

Application Configuration option 21
Application_Startup method 79
AppRequest class
 about 79
 used, for application information
 persisting 78, 79
App.xaml.cs class 86
authenticate method 83

B

BarItem class
 about 83
 design 84
 properties 84
BaseDirectory property 66
BaseEntity class 98
BaseWindow class
 methods 77
 setting up 75
 WPF Window class, inheriting 75-77
BaseWindow class methods
 LoadRadMenu method 77
 LoadRadOutlook method 77
 LogException method 77
 RefreshGrid method 77
 treeSubItem_Click method 77
BookCommands options
 FirstPage 70
 LastPage 70
 NextPage 70
 PrevPage 70
BookWindow method 69

C

CheckChildControl method 36
checkDataType property 42
Click event 46
comboDataType combobox 62
comboDataType property 62
ComboValidationAttribute 50
Command property 70
CommandTarget property 70
CommonTask class
 about 99
 IsSummary property 100
 RadGanttView control 100
ConvertDataTable method 47
ConvertList method 48
CurrentUser property 86
CustList property 26, 43
CustomerEntity class 26, 49, 55, 64
CustomerEntity FetchList method 48
CustomerEntity object
 binding to 26, 27

D

database
 RadBook control 67-71
 RadTabControl control 60-67
 setting up 20-22, 40, 60
DataContext property 8, 27, 43, 52
DataTable
 using, for populating RadComboBox 41, 42
DataTable class 19
DataTable object
 used, for combobox populating 61-63
DataTable portion 49

DataTable property 26
Deserialization 65
DirectoryServices library
 used, for Active Directory information gathering 74
DisabledException 81
DisplayMemberPath property 42, 63, 64
dynamic validation
 about 49
 example 51
 setup 49
 TextValidationAttribute class 50
 using 52-55
Dynamic Validation control 39

E

errorMessage parameter 50, 52
event method 43
ExceedsLimitException 81

F

FetchMenuItems method 81
FetchMenuOptions method 87
fetchTabs() method 64
FetchXMLMenuItems method 81
fetchXMLTabs method 66

H

Header property 84
HeaderText property 64

I

IEnumerable interface class 63
IGanttTask interface class
 about 98
 references, creating for 98
InitializeComponent() method 69, 79
INotifyCollectionChanged class 103
Install button 16
instance method 79
IsSpellCheckingEnabled property 35
ItemSource property 23, 42, 63
ItemsSource property

L

LoadChildTasks method 104
loadGanttControl method 102, 105
LoadRadMenu method 77
LoadRadOutlook method 77
LoadSummaryTasks method 104
LoadTasks method 102
LogException method 77

M

MainWindow.xaml.cs class 22
Mask property 14
MinLength property 51
Model View ViewModel (MVVM)
 pattern 9
More <control> Examples option 17

N

NEXT EXAMPLE button 17
NotPermittedException 81

O

object generic list
 used, for RadTabControl populating 64
object-oriented concepts
 about 98
 CommonTask class 99, 100
 IGanttTask interface class 98
ObservableCollection class 23, 67
OutlookBarWindow class 86

P

Position property 84
Product class 30

R

RadAutoCompleteBox control
 about 19
 events 23
 properties 23
 working with 22

RadAutoCompleteBox control events

SearchTextChanged 23
SelectionChanged 23

RadAutoCompleteBox control properties

ItemSource 23
SelectionMode 23-25
TextSearchMode 23, 24
TextSearchPath 24
WatermarkContent 25

RadBook control

about 67
events 67
properties 67, 68
working with 69-71

RadBook control events

FoldActivated 67
FoldDeactivated 67
PageChanged 67

RadBook control properties

IsKeyboardNavigationEnabled 67
ItemSource 67
LeftPageTemplate 67
PageFlipMode 67
RightPageTemplate 67

RadButton property 35**RadComboBox control**

about 39, 40
ComboTestWindow 41
DataTable, using 41, 42
new window, controls 40
object generic list, using 43, 44

RadDataGrid

RadSpellChecker control,
associating with 34-36

RadGanttView control

about 100, 101
populating, user task filtering used 101-103
tasks, displaying with summary task 104,
105

RadMaskedEdit control 53**RadMaskedInput control**

about 19, 27
RadMaskedInputCurrency 27, 28
RadMaskedInputText control 28, 29
RadSpellChecker control 31
reviewing, with VS TextBox 32, 33
used, for data validating 30, 31

RadMaskedInputCurrency control

about 27, 28
Culture property 28
IsCurrencySymbolVisible property 28
Mask property 28

RadMaskedInputText control 28, 29**RadMenu control**

about 89
working 89-93

RadOutlookBar control

about 84
database security 85-87
generic list, binding with XML security 88
populating, with menu links 85-87

RadRichTextBox

RadSpellChecker control,
associating with 33, 34

RadSpellChecker control

about 19, 31
associating, with RadDataGrid 34-36
associating, with RadRichTextBox 33, 34
associating, with VS TextBox 32, 33

RadSpreadsheet control

about 39, 44-46
ConvertDataTable method 47-49
new window, creating 45

RadTabControl control

about 60, 61
DataTable object used, for combobox
populating 61-63
populating, object generic list used 64
populating, XML file used 64-67

RadTreeView control 86**ReferId property 84****RefreshGrid method 77****RegularExpression class 30****Request class 82****S****SearchTextChanged event 23****SelectedIndexChanged event 62****SelectedIndex property 63****SelectedValueChanged event 44****SelectedValuePathV property 42****SelectedValue property 44****SelectionChanged event 23**

- SelectionMode property 23
- Serialization 65
- Spellchecker method 34
- SpellCheckMode property 31
- SpellCheckWindow class 34
- SpreadsheetWindow class 47
- SQL Server Mgmt Studio (SSMS) 20
- System.Data.DataTable
 - binding to 23-26

T

- TabItem class 64
- TasksSource property 100
- Telerik
 - demo project, installing 15-17
 - navigation containers, working with 84-93
 - project, creating 13-15
 - Telerik dynamic event handling controls 73
 - Telerik navigation controls 73
 - Telerik project
 - database, setting up 20-22
 - Telerik RadControls
 - about 7
 - advantages 9
 - categories 8
 - prerequisites 10
 - WPF trial, downloading 10-12
 - Telerik RadControls categories
 - Data Management 8
 - Data Visualization 8
 - Editors 8
 - Framework 9
 - Interactivity 9
 - Layouts 9
 - Navigation 9
 - Scheduling 9
 - Telerik trial software
 - downloading 10-12
 - URL 10
 - Telerik WPF control
 - RadBook 67-71
 - RadTabControl 60-67
 - TextSearchMode property 23
 - TextValidationAttribute class 50, 51
 - TreeColumnDefinition template 104
 - treeSubItem_Click method 77

U

- UserADGroups property 91
- UserChkPwd property 86
- UserEntity class 75
 - about 80
 - authentication process exceptions 81
 - methods, for Active Directory security 91
 - used, for user information persisting 80-83
- UserEntity class methods
 - Authenticate 91
 - getUserGroups 91
 - isADAccountLocked 91
- UserLoggedIn property 81, 91
- UserMenuItems property 88
- UserName property 86
- UserPrincipal object 81
- user task filtering
 - creating, to populate RadGanttView 101-103

V

- Validate method 54, 56
- Validation classes 55
- ValidEmailAddress method 54
- VS TextBox
 - RadSpellChecker control, associating with 32, 33

W

- Web Service Definition Language changes.
 - See WSDL changes
- Window class
 - extending 75
- Window_Loaded event 22, 26, 101, 102
- Window_Loaded method 22
- Windows Presentaion Foundation *See* WPF
- Workbook class 47
- Worksheet class 48
- WPF
 - concepts 8
 - Telerik RadControls 7
 - Telerik RadControls advantages 9
 - Telerik RadControls categories 8

WPF concepts

AppRequest persistence class 78, 79

BarItem class 83, 84

BaseWindow class 75-77

UserEntity class 80-83

Window class, extending 75

WSDL changes 19**X****XDocument object**

using 81

XML file

used, for RadTabControl populating 64-66



Thank you for buying Telerik WPF Controls Tutorial

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.

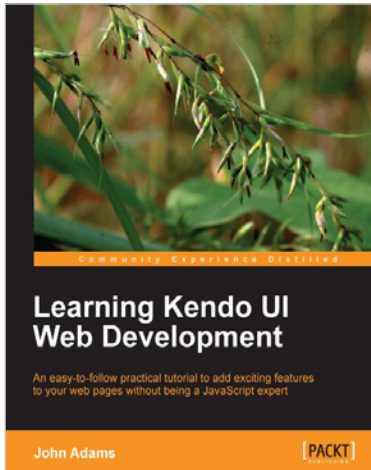
About Packt Enterprise

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.

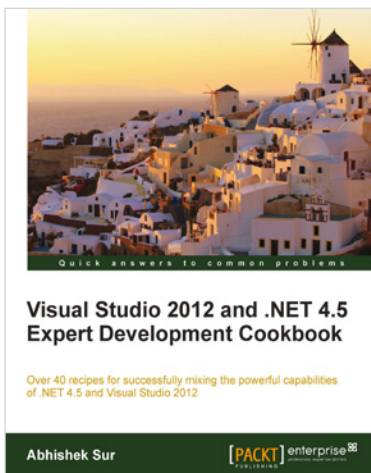


Learning Kendo UI Web Development

ISBN: 978-1-84969-434-6 Paperback: 288 pages

An easy-to-follow practical tutorial to add exciting features to your web pages without being a JavaScript expert

1. Learn from clear and specific examples on how to utilize the full range of the Kendo UI tool set for the web.
2. Add powerful tools to your website supported by a familiar and trusted name in innovative technology.
3. Learn how to add amazing features with clear examples and make your website more interactive without being a JavaScript expert.



Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

ISBN: 978-1-84968-670-9 Paperback: 380 pages

Over 40 recipes for successfully mixing the powerful capabilities of .NET 4.5 and Visual Studio 2012

1. Step-by-step instructions to learn the power of .NET development with Visual Studio 2012.
2. Filled with examples that clearly illustrate how to integrate with the technologies and frameworks of your choice.
3. Each sample demonstrates key concepts to build your knowledge of the architecture in a practical and incremental way.

Please check www.PacktPub.com for information on our titles

www.SoftGozar.com



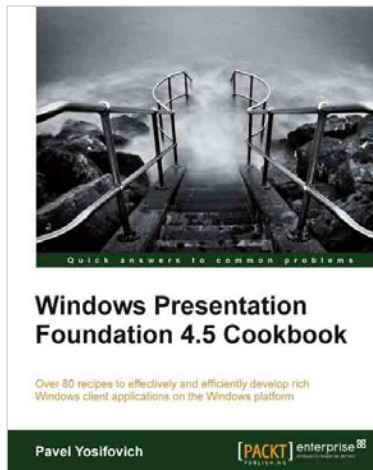
ASP.NET Web API

Build RESTful web applications and services on the .NET framework

ISBN: 978-1-84968-974-8 Paperback: 224 pages

Master ASP.NET Web API using .NET Framework 4.5 and Visual Studio 2013

1. Clear and concise guide to the ASP.NET Web API with plentiful code examples.
2. Learn about the advanced concepts of the WCF-windows communication foundation.
3. Explore ways to consume Web API services using ASP.NET, ASP.NET MVC, WPF, and Silverlight clients.



Windows Presentation Foundation 4.5 Cookbook

ISBN: 978-1-84968-622-8 Paperback: 464 pages

Over 80 recipes to effectively and efficiently develop rich Windows client applications on the Windows platforms

1. Full of illustrations, diagrams, and tips with clear step-by-step instructions and real world examples.
2. Gain a strong foundation of WPF features and patterns.
3. Leverage the MVVM pattern to build decoupled, maintainable apps.

Please check www.PacktPub.com for information on our titles

www.SoftGozar.com