



Professional **IIS 7 and ASP.NET** Integrated Programming

Dr. Shahram Khosravi



- Written and tested with both ASP.NET 2.0 and ASP.NET 3.5
- Updates, source code, and Wrox technical support at www.wrox.com

Professional

IIS 7 and ASP.NET Integrated Programming

Dr. Shahram Khosravi



Wiley Publishing, Inc.

Professional

IIS 7 and ASP.NET Integrated Programming

Introduction	xvii
Chapter 1: IIS 7 and ASP.NET Integrated Architecture.	1
Chapter 2: Using the Integrated Configuration System	23
Chapter 3: Managing the Integrated Configuration System from IIS Manager and the Command Line	61
Chapter 4: Managing the Integrated Configuration System with Managed Code	83
Chapter 5: Extending the Integrated Configuration System and Imperative Management API	115
Chapter 6: Understanding the Integrated Graphical Management System.	145
Chapter 7: Extending the Integrated Graphical Management System.	175
Chapter 8: Extending the Integrated Request Processing Pipeline.	289
Chapter 9: Understanding the Integrated Providers Model.	397
Chapter 10: Extending the Integrated Providers Model	445
Chapter 11: Integrated Tracing and Diagnostics	537
Chapter 12: ASP.NET and Windows Communication Foundation Integration in IIS 7	605
Index	651

Professional

IIS 7 and ASP.NET Integrated Programming

Dr. Shahram Khosravi



Wiley Publishing, Inc.

Professional IIS 7 and ASP.NET Integrated Programming

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2008 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-15253-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data:

Khosravi, Shahram, 1963–
Professional IIS 7 and ASP.NET integrated programming / Shahram Khosravi.
p. cm.
Includes index.

ISBN 978-0-470-15253-9 (paper/website)

1. Microsoft Internet information server. 2. Active server pages. 3. Internet programming. 4. Microsoft .NET I. Title.
QA76.625.K555 2007

005.2'768 — dc22

2007032156

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

About the Author

Shahram Khosravi, Ph.D.

Dr. Shahram Khosravi is a senior software engineer, consultant, author, and instructor specializing in ASP.NET, Windows Communications Foundation (WCF), ASP.NET AJAX, Windows Workflow Foundation (WF), IIS 7 and ASP.NET Integrated Programming, ADO.NET, Web services, .NET, and XML technologies such as XSD, XSLT, XPath, SOAP, and WSDL. He also has years of experience in object-oriented analysis, design, and programming, architectural and design patterns, service-oriented analysis, design, and programming, 3D computer graphics programming, user interface design, and usability.

Shahram is the author of the following four books: *Professional ASP.NET 3.5 and .NET 3.5 Programming (ASP.NET Internals plus ASP.NET AJAX, IIS 7.0, Windows Workflow Foundation, and Windows Communication Foundation)*, *ASP.NET AJAX Programmer's Reference with ASP.NET 2.0 or ASP.NET 3.5*, *Professional IIS 7 and ASP.NET Integrated Programming*, and *Professional ASP.NET Server Control and Component Development*. He has written articles on the ASP.NET, ADO.NET, .NET, and XML technologies for the industry's leading magazines such as *Dr. Dobbs' Journal*, *asp.netPRO* magazine, and *Microsoft MSDN Online*.

Credits

Senior Acquisitions Editor

Jim Minatel

Development Editor

Brian MacDonald

Technical Editor

Dan Kahler

Production Editor

Daniel Scribner

Copy Editor

Kim Cofer

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Joseph B. Wikert

Project Coordinator, Cover

Lynsey Osborn

Compositor

Happenstance Type-O-Rama

Proofreader

Nancy Riddiough

Indexer

Robert Swanson

Anniversary Logo Design

Richard Pacifico

Acknowledgments

First and foremost, I would like to thank Jim Minatel, the senior acquisitions editor on the book, for giving me the opportunity to work on this exciting project. Huge thanks go to Brian MacDonald, the book's development editor. I greatly appreciate your input, comments, and advice throughout the process. Thanks Brian, for everything! Special thanks go to Dan Kahler, the book's technical editor. Thanks Dan for all your input and comments. Additional thanks also go to Daniel Scribner, the book's production editor. Thanks also go to Kim Cofer, the copy editor, and Nancy Riddiough, the proofreader.

Contents

Acknowledgments	vii
Introduction	xvii
Chapter 1: IIS 7 and ASP.NET Integrated Architecture	1
Modular Architecture of IIS 7	1
IIS-WebServer	3
IIS-WebServerManagementTools	6
IIS-FTPPublishingService	7
WAS-WindowsActivationService	7
Extensible Architecture of IIS 7	8
IIS 7 and ASP.NET Integrated Request Processing Pipeline	8
IIS 7 and ASP.NET Integrated Configuration Systems	10
IIS 7 and ASP.NET Integrated Administration	11
Building a Customized Web Server	11
Update Dependencies	12
Windows Features Dialog	13
Server Manager	14
Command-Line Setup Option	20
Unattended Setup Option	20
Upgrade	21
Summary	21
Chapter 2: Using the Integrated Configuration System	23
Integrated Configuration System	23
Hierarchical Configuration Schema	24
Distributed Configuration System	26
<location> Tags	28
Include Files	31
<configSections>	32
Protocol Listeners	34
Windows Process Activation Service	34
World Wide Web Publishing Service	35
The Structure of the applicationHost.config File	36
<system.applicationHost>	36
<system.webServer>	45
Summary	60

Contents

Chapter 3: Managing the Integrated Configuration System from IIS Manager and the Command Line **61**

Server Management	61
Internet Information Services (IIS) Manager	62
Application Pools	63
Web Sites	66
Hierarchical Configuration	68
Delegation	73
Command-Line Tool	76
LIST	80
ADD	81
DELETE	81
SET	81
Summary	81

Chapter 4: Managing the Integrated Configuration System with Managed Code **83**

Class Diagrams	83
ConfigurationElement	86
ConfigurationElementCollectionBase<T>	86
ApplicationPool	88
ApplicationPoolProcessModel	89
ApplicationPoolRecycling	90
ApplicationPoolCpu	93
ApplicationPoolCollection	94
Site	95
Binding	96
BindingCollection	97
Application	98
ApplicationCollection	99
VirtualDirectory	100
VirtualDirectoryCollection	101
ConfigurationSection	101
ServerManager	102
Putting It All Together	103
Recipe for Loading a Specified Configuration File	104
Recipe for Accessing the Specified Attribute of a Specified Configuration Section	104
Recipe for Adding or Removing an Element from the Specified Collection Element of a Specified Configuration Section	106
Recipe for Accessing the Configuration Sections in the <system.applicationHost> Section Group	108
Summary	113

Chapter 5: Extending the Integrated Configuration System and Imperative Management API	115
IIS7 and ASP.NET Integrated Configuration Extensibility Model	116
IIS7 and ASP.NET Integrated Declarative Schema Extension Markup Language	117
Adding a Custom Configuration Section	124
IIS7 and ASP.NET Integrated Imperative Management Extensibility Model	134
Representing the Collection Item	136
Representing the Collection Element	136
Representing the Non-collection Element	138
Representing the Outermost Element	139
Putting It All Together	141
Summary	143
Chapter 6: Understanding the Integrated Graphical Management System	145
Module Pages	146
ModuleDialogPage	147
ModuleListPage	147
ModulePropertiesPage	147
Writing a Custom Module Page	148
Tasks	149
Page Navigation	149
Task Forms	150
Wizard Forms	150
The IIS7 Manager Object Model	152
Service	152
ManagementConfigurationPath	154
Connection	155
Navigation Item	156
Navigation Service	156
TaskItem	158
TaskList	163
ModulePageInfo	165
TaskListCollection	166
Putting It All Together	167
Summary	174
Chapter 7: Extending the Integrated Graphical Management System	175
Client-Side Managed Code	175
Custom Module Pages and Task Forms in Action	179

Contents

Proxies	184
ModuleServiceProxy	186
What's PropertyBag Anyway?	189
MyConfigSectionPage	193
Constructor	196
Event Handlers	200
HasChanges Property	201
CanApplyChanges Property	202
OnActivated	202
GetSettings	203
OnWorkerGetSettings	205
OnWorkerGetSettingsCompleted	205
MyConfigSectionInfo	207
InitializeUI	210
ApplyChanges	213
GetValues	214
CancelChanges	215
Adding Support for New Task Items	216
Refreshing	221
MyCollectionPage	229
InitializeListPage	234
OnActivated	235
GetCollectionItems	235
OnWorkerGetCollectionItems	235
OnWorkerGetCollectionItemsCompleted	236
MyCollectionItemInfo	238
MyCollectionItemListViewItem	239
AddItem	239
Adding Support for New Task Items	240
OnListViewBeforeLabelEdit	247
OnListViewAfterLabelEdit	248
OnListViewDoubleClick	251
OnListViewKeyUp	252
OnListViewSelectedIndexChanged	252
Grouping	252
Refreshing	257
MyCollectionItemTaskForm	258
Constructors	262
InitializeComponent	262
OnAccept	265
OnWorkerDoWork	265
OnWorkerCompleted	266

Module	267
Module	267
MyConfigSectionModule	268
Server-Side Managed Code	269
Module Service	270
Module Provider	281
Deployment	283
Summary	287
 Chapter 8: Extending the Integrated Request Processing Pipeline	 289
 Extending the Integrated Pipeline through Managed Code	 289
Managed Handlers	290
Developing Custom Managed Handlers	291
Plugging Custom Managed Handlers into the Integrated Request Processing Pipeline	302
Using the RssHandler HTTP Handler	314
Managed Modules	315
Developing Custom Managed Modules	318
Plugging Custom Managed Modules into the Integrated Request Processing Pipeline	322
Using the UrlRewriterModule HTTP Module	332
Managed Handler Factories	333
Developing Custom Managed Handler Factories	334
Plugging Custom Managed Handler Factories into the Integrated Request Processing Pipeline	336
Extending the Integrated Pipeline with Configurable Managed Components	336
Configuration Support for the URL Rewriting Managed Module	337
Imperative Management Support for the URL Rewriting Managed Module	340
UrlRewriterRule	341
UrlRewriterRules	342
UrlRewriterSection	343
Testing the Managed Classes	344
Graphical Management Support for the URL Rewriter Managed Module	346
Client-Side Managed Code	346
Communications with the Back-End Server	348
UrlRewriterPage	351
UrlRewriterRuleTaskForm	371
UrlRewriterModule	380
Server-Side Managed Code	381
UrlRewriterModuleService	382
UrlRewriterModuleProvider	387
Registering UrlRewriterModuleProvider	389
Configurable UrlRewriterModule	390

Contents

Rewriting Non-ASP.NET URLs	393
Postback Problem with URL Rewriting	393
Summary	396
Chapter 9: Understanding the Integrated Providers Model	397
Why You Need Provider-Based Services	398
The Integrated Providers Model in Action	400
Under the Hood of the Integrated Providers Model	405
ProviderFeature	406
ProviderConfigurationSettings	412
Putting it All Together	415
IProviderConfigurationService	436
Summary	444
Chapter 10: Extending the Integrated Providers Model	445
Recipe	445
Custom Provider Base Class	448
Custom Provider Collection	449
Extending the Integrated Configuration System	450
Extending the Integrated Imperative Management System	454
ProviderSettings	454
ProviderSettingsCollection	455
ProvidersHelper	457
RssSection	460
Implementing the Service Class	462
Implementing Custom Providers	467
SqlRssProvider	467
XmlRssProvider	477
Extending the Integrated Graphical Management System	485
Client-Side Managed Code	493
Server-Side Managed Code	526
Summary	536
Chapter 11: Integrated Tracing and Diagnostics	537
Integrated Tracing Components	537
Tasks Performed from within Your Code	540
Instantiating a Trace Source	540
Adding Trace Events	546
Defining the Conditional Compilation Symbol “TRACE”	550

Tasks Performed from the Configuration File	550
Instantiating and Attaching a Switch	550
Instantiating and Attaching an IisTraceListener	557
Instantiating and Attaching a Trace Filter	562
Putting It All Together	570
Configurable Tracing	578
Runtime Status and Control API	587
ServerManager	589
WorkerProcessCollection	590
WorkerProcess	590
RequestCollection	591
Request	592
ApplicationDomain	593
ApplicationDomainCollection	594
ApplicationPool	595
Site	596
Putting It All Together	596
LogRequest	600
Summary	604
 Chapter 12: ASP.NET and Windows Communication Foundation Integration in IIS 7	 605
Installing the Required Software	605
Bug Report Manager	606
Windows Communication Foundation Service	607
Windows Communication Foundation Endpoint	608
Windows Communication Foundation Service Model	609
Developing a WCF Service	610
Developing a WCF Service Contract	611
Implementing a WCF Service Contract	614
Hosting a WCF Service	617
Administrative Tasks	619
Developing a Windows Communication Foundation Client	625
Adding a Web Reference	625
Using the svcutil.exe Tool	627
Imperative Approach	632
Taking Advantage of ASP.NET and WCF Integration in IIS 7	635
Using Different Bindings	638
Putting It All Together	646
Summary	648
 Index	 651

Introduction

Welcome to *Professional IIS 7 and ASP.NET Integrated Programming*. The deep integration of IIS 7 and ASP.NET provides both IIS 7 administrators and ASP.NET developers with a rich integrated programming environment to implement features and functionalities that were not possible in earlier versions of IIS.

This book provides in-depth coverage of all the major systems that make up the IIS 7 and ASP.NET integrated infrastructure, as follows:

- ❑ IIS 7 and ASP.NET integrated request processing pipeline
- ❑ IIS 7 and ASP.NET integrated configuration system and its associated declarative schema extension markup language
- ❑ IIS 7 and ASP.NET integrated imperative management system
- ❑ IIS 7 and ASP.NET integrated graphical management system
- ❑ IIS 7 and ASP.NET integrated providers model
- ❑ IIS 7 and ASP.NET integrated tracing and diagnostics
- ❑ ASP.NET and Windows Communication Foundation integration in IIS 7

This book not only shows how these major systems work from the inside out and how to use them in your own applications, but also provides comprehensive coverage of the extensibility points of these systems and shows you how to take advantage of them to add support for new features and functionalities.

The discussions of this book are presented in the context of numerous step-by-step recipes and detailed code walkthroughs and in-depth analyses of real-world examples that use these recipes to help you gain the skills, knowledge, and experience you need to use and extend these major systems.

Who This Book Is For

This book is aimed at the ASP.NET developer and IIS 7 administrator who want to learn IIS 7 and ASP.NET integrated programming for the first time. No knowledge of IIS 7 and ASP.NET integrated programming is assumed.

What This Book Covers

This book is divided into 12 chapters as follows:

- ❑ **Chapter 1, “IIS 7 and ASP.NET Integrated Architecture,”** covers the IIS 7 package updates and their constituent feature modules. It shows you five different ways to custom build your own

Web server from the various package updates to decrease the footprint of your Web server. The chapter also provides an overview of the systems that make up the IIS 7 and ASP.NET integrated architecture.

- ❑ **Chapter 2, “Using the Integrated Configuration System,”** discusses the new IIS 7 and ASP.NET integrated configuration system, including the hierarchical structure of its configuration files, the hierarchical relationships among these configuration files, and the notion of the declarative versus imperative schema extension. The chapter also uses numerous examples to walk you through important sections of the new IIS 7 machine-level configuration file named `applicationHost.config`, where you’ll also learn how to override the configuration settings specified in different sections of this file in a particular site, application, or virtual directory.
- ❑ **Chapter 3, “Managing the Integrated Configuration System from IIS 7 Manager and the Command Line,”** shows how to use the IIS 7 Manager and `appcmd.exe` command-line tools to manage the IIS 7 and ASP.NET integrated configuration system. You’ll also learn how the IIS Manager takes the hierarchical nature of the integrated configuration system into account and how you can configure both the IIS 7 Web server and ASP.NET Web applications from the IIS 7 Manager. This chapter also covers the delegation feature of this integrated configuration system.
- ❑ **Chapter 4, “Managing the Integrated Configuration System with Managed Code,”** provides in-depth coverage of those types of the IIS 7 and ASP.NET integrated imperative management system that allow you to manage the IIS 7 and ASP.NET integrated configuration system from managed code. Those types include the `ConfigurationElement`, `ConfigurationElementCollectionBase<T>`, `ApplicationPool`, `ApplicationPoolCollection`, `Site`, `Application`, `ApplicationCollection`, `VirtualDirectory`, `VirtualDirectoryCollection`, `ConfigurationSection`, and `ServerManager`. This chapter also provides step-by-step recipes for using these types and examples where these recipes are used.
- ❑ **Chapter 5, “Extending the Integrated Configuration System and Imperative Management API,”** uses examples to walk you through the XML constructs that make up the IIS 7 and ASP.NET integrated declarative schema extension markup language including `<sectionSchema>`, `<attribute>`, `<element>`, and `<collection>`. It provides a step-by-step recipe for using these XML constructs to extend the integrated configuration system to implement the XML constructs that make up a custom configuration section, including its containing XML element and attributes, Non-collection XML elements and attributes, Collection XML elements and their child add, remove, and clear XML elements and their attributes. The chapter uses this recipe to implement the XML constructs that make up a custom configuration section, including its containing XML element and the associated attributes, a Non-collection XML element, a Collection XML element, and the add, remove, and clear child XML elements.

The chapter also gives you recipes for extending the integrated imperative management API to add support for new imperative management classes that allow managed code to manage the XML constructs making up a configuration section in strongly-typed fashion.

- ❑ **Chapter 6, “Understanding the Integrated Graphical Management System,”** provides in-depth coverage of the integrated graphical management system. This chapter first covers module dialog pages, module list pages, module properties pages, task forms, and wizard forms. It then dives into the IIS 7 Manager’s object model where types such as `IServiceProvider`, `IServiceContainer`, `ManagementConfigurationPath`, `Connection`, `NavigationItem`, and `TaskListCollection` are discussed. The chapter then takes you under the hood where you’ll see for yourself how these types work together.

- ❑ **Chapter 7, “Extending the Integrated Graphical Management System,”** gives you step-by-step recipes for implementing the client-side and server-side code that extend the IIS 7 and ASP.NET integrated graphical management system to add graphical management support for a custom configuration section. The chapter uses these recipes to add support for custom graphical management components that allow users to configure the `<myConfigSection>` configuration section right from the IIS 7 Manager.
- ❑ **Chapter 8, “Extending the Integrated Request Processing Pipeline,”** shows you how to implement your own custom HTTP modules, HTTP handlers, and HTTP handler factories, and plug them into the IIS 7 and ASP.NET integrated request processing pipeline to extend this pipeline to add support for custom request processing capabilities.

The chapter shows you three different ways to plug your custom HTTP modules, HTTP handlers, and HTTP handler factories into the IIS 7 and ASP.NET integrated pipeline: declaratively through a configuration file, graphically through the IIS 7 Manager, and imperatively through managed code.

Finally the chapter shows you how to implement a fully configurable `UrlRewriterModule` HTTP module and plugs the module into the IIS 7 and ASP.NET integrated request processing pipeline.

- ❑ **Chapter 9, “Understanding the Integrated Providers Model,”** begins by showing you the integrated providers model in action. Next, it takes you under the hood where you see for yourself the important roles that the following classes play in the integrated providers model and how you can take advantage of them when you’re implementing your own custom provider-based services:
 - ❑ The `ProviderFeature` abstract base class and its sub. You also learn how to implement a custom provider feature to describe your own custom provider-based service and how to register your custom provider feature with the integrated providers model.
 - ❑ The `ProviderConfigurationSettings` abstract base class and its sub. This chapter also teaches you how to implement a custom provider configuration settings class to describe the configuration settings of the providers of your own custom provider-based service and how to register your custom provider configuration settings class with the integrated providers model.
 - ❑ The `PropertyGrid` control. This chapter walks you through several exercises to help you gain a better understanding of this control, the role it plays in the integrated providers model, and how to customize this control for your own custom provider-based services.
 - ❑ The `AddProviderForm` task form.
 - ❑ The `ProviderConfigurationConsolidatedPage` module list page.
 - ❑ The `IProviderConfigurationService` interface and its standard implementation named `ProviderConfigurationModule`. This chapter shows you how to take advantage of this standard implementation in your own provider-based services.
- ❑ **Chapter 10, “Extending the Integrated Providers Model,”** begins by providing a detailed step-by-step recipe for extending the integrated providers model to implement and to plug your own fully configurable custom provider-based services into this model. It uses this recipe to implement a fully configurable RSS provider-based service and plug it into the integrated providers

model. The RSS provider-based service enables you to generate RSS documents from any type of data store such as SQL Server databases, XML documents, and so on.

- ❑ **Chapter 11, “Integrated Tracing and Diagnostics,”** shows you how to use the IIS 7 and ASP.NET integrated tracing and diagnostics infrastructure to instrument your managed code with tracing. This chapter demonstrates how to emit trace events from within your managed code, how to route these trace events to the IIS 7 tracing infrastructure, and how to configure modules such as Failed Request Tracing to consume these trace events. This chapter uses practical examples to provide in-depth coverage of the `TraceSource` data source, the `SourceSwitch` switch, the `IisTraceListener` listener, the `EventTypeFilter` and `SourceFilter` filters, and how to enable Failed Request Tracing and define new rules in the IIS 7 Manager.

The chapter then uses a real-world example to show you how to make the tracing feature of your managed code fully configurable from configuration files, from managed code, and from the IIS 7 Manager.

The chapter then discusses the Runtime Status and Control API (RSCA), which is an unmanaged API. Next, this chapter provides in-depth coverage of the various types of the integrated imperative management system and uses an example to show you how to use these types to indirectly program against the RSCA unmanaged API from your managed code to access and to manipulate the runtime state of the IIS 7 runtime objects.

The chapter finally discusses the `LogRequest` event of the `HttpApplication` object and implements an HTTP module that registers an event handler for this event where it stores request data in an XML document. This request data provides a powerful diagnostic tool.

- ❑ **Chapter 12, “ASP.NET and Windows Communication Foundation Integration in IIS 7,”** uses a practical example to show you how to use the Windows Communication Foundation Service Model to model the communications of your own components with the outside world and how to take advantage of the deep integration of ASP.NET and WCF services in the IIS 7 environment in your own Web applications. This chapter covers the following topics:
 - ❑ A WCF endpoint and its address, binding, and contract.
 - ❑ WCF service model and its attribute-based, configuration-based, and imperative programming facilities for modeling the communications of your own components with the outside world.
 - ❑ Defining WCF service contracts.
 - ❑ Implementing WCF service contracts.
 - ❑ Adding, updating, removing, and configuring WCF endpoints.
 - ❑ Adding behaviors.
 - ❑ Hosting a WCF service. This chapter shows you how to host your WCF service in IIS 7 to take full advantage of the great features of IIS 7 discussed throughout this book.
 - ❑ Developing WCF clients. This chapter discusses three different ways to develop a WCF client: adding a Web reference, using the `Svcutil.exe` tool, and the imperative approach. This chapter uses each approach to develop a separate WCF client.

This chapter uses a practical example that consists of the following three different applications to demonstrate the deep integration of ASP.NET and WCF services in IIS 7.

What You Need to Use This Book

You'll need the following items to run the code samples in this book:

- ❑ Windows Vista or Windows Server 2008
- ❑ Visual Studio 2005, Visual Studio 2005 Express Edition, Visual Studio 2008, or Visual Studio 2008 Express Edition
- ❑ SQL Server 2005 or SQL Server 2005 Express Edition

You can download free copies of Visual Studio 2005 Express Edition or Visual Studio 2008 Express Edition and SQL Server 2005 Express Edition from <http://msdn.microsoft.com/vstudio/express/>.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Tips, hints, tricks, and asides to the current discussion are offset and italicized, like this.

As for styles in the text:

- ❑ We *highlight* new terms and important words when we introduce them.
- ❑ We show keyboard strokes like this: Ctrl+A.
- ❑ We show filenames, URLs, and code within the text like so: `persistence.properties`.
- ❑ We present code in two different ways:

In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present context, or has been shown before.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at <http://www.wrox.com>. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Introduction

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-15253-9.

Once you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at <http://www.wrox.com/dynamic/books/download.aspx> to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to <http://www.wrox.com> and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

IIS 7 and ASP.NET Integrated Architecture

Internet Information Services 7.0 (IIS 7) is the latest version of Microsoft Web server. IIS 7 has gone through significant architectural changes since the last version. The most notable change for ASP.NET developers is the deep integration of the IIS 7 and ASP.NET framework. This provides both ASP.NET developers and IIS 7 administrators with an integrated programming environment that allows them to implement features and functionalities that were not possible before. The main goal of this chapter is twofold. First, it covers the IIS 7 package updates and their constituent feature modules, discusses five different IIS 7 setup options, and shows you how to use each option to custom-build your own Web server from these package updates. Second, it provides you with an overview of the IIS 7 and ASP.NET integrated architecture and its constituent systems, setting the stage for the next chapters where you'll dive into the details of this integrated architecture and programming framework.

Modular Architecture of IIS 7

The main priority of the Microsoft IIS team for IIS 6.0 was to improve its security, performance, and reliability. For that reason, modularity and extensibility didn't make it to the list of top priorities for IIS 6.0. That said, IIS 6.0 introduced a very important notion: selectively disabling IIS 7 features such as ISAPI extensions and CGI components. One of the main problems with the earlier versions of IIS was that all features of IIS had to be installed and enabled. There was no way to disable features that your application scenario did not need.

IIS 6.0 enables only static file serving by default on a clean install of the Web server. In other words, dynamic features such as ISAPI extensions and CGI components are disabled by default unless the administrator explicitly enables them. Such customization of the Web server allows you to decrease the attack surface of your Web server by giving attackers fewer opportunities for attacks.

Chapter 1: IIS 7 and ASP.NET Integrated Architecture

Disabling unwanted features was the first step toward the customizability of IIS. However, this step didn't go far enough because IIS 6.0 still installs everything, which introduces the following problems:

- ❑ Disabled features consume server resources such as memory, and therefore increase the Web server footprint.
- ❑ Administrators still need to install service packs that address bugs in the disabled features, even though they're never used.
- ❑ Administrators still need to install software updates for the disabled features.

In other words, administrators have to maintain the service features that are never used. All these problems stem from the fact that the architecture of IIS 6.0 is relatively monolithic. The main installation problem with a monolithic architecture is that it's based on an all-or-nothing paradigm where you have no choice but to install the whole system.

IIS 7.0 is modular to the bone! Its architecture consists of more than 40 feature modules from which you can choose. This allows you to install only the needed feature modules to build a highly customized, thin Web server. This provides the following important benefits:

- ❑ Decreases the footprint of your Web server.
- ❑ Administrators need to install only those service packs that address bugs in the installed feature modules.
- ❑ Administrators need to install software updates only for the installed feature modules.

So, administrators have to maintain and service only installed feature modules.

Next, I provide an overview of the IIS 7 feature modules or components. These feature components are grouped into what are known as *functional areas*, where each functional area maps to a specific IIS package update. That is, each package update contains one or more feature modules or components. As you'll see later, you'll use these package updates to custom-build your Web server.

The top-level IIS update is known as IIS-WebServerRole, and contains the updates shown in Figure 1-1. As the name suggests, the IIS-WebServerRole update enables Windows Server 2008 and Windows Vista to adopt a Web server role, which enables them to exchange information over the Internet, an intranet, or an extranet.

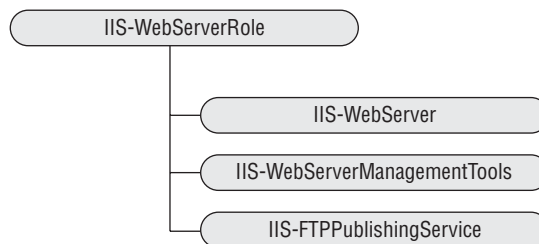


Figure 1-1

IIS-WebServer

The IIS-WebServer update contains five updates as shown in Figure 1-2. As you can see, this update contains the feature modules that make up the core functionality of a Web server.

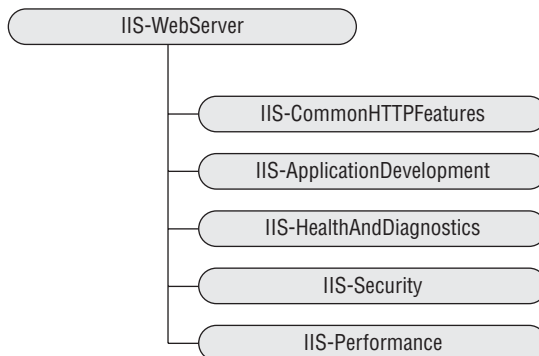


Figure 1-2

IIS-CommonHttpFeatures

The IIS-CommonHttpFeatures update contains the feature modules or components described in the following table:

Feature Module	Description
IIS-StaticContent	Use this module to enable your Web server to service requests for static content. Web site resources with file extensions such as .html, .htm, .jpg, and the like that can be serviced without server-side processing are known as static content.
IIS-DefaultDocument	This module allows you to specify a Web resource that will be used as the default resource when the request URL does not contain the name of the requested resource.
IIS-DirectoryBrowsing	Use this module to enable your Web server to display the contents of a specified directory to end users when they directly access the directory and no default document exists in the directory.
IIS-HttpErrors	Use this module to enable your Web server to support sending custom error messages to end users.
IIS-HttpRedirect	Use this module to enable your Web server to support request redirects.

IIS-ApplicationDevelopment

The IIS-ApplicationDevelopment update contains the feature modules that support different application types as described in the following table:

Feature Module	Description
IIS-ASPNET	Use this module to enable your Web server to host ASP.NET applications.
IIS-NetFxExtensibility	Use this module to enable your Web server to host managed modules.
IIS-ASP	Use this module to enable your Web server to host ASP applications.
IIS-CGI	Use this module to enable your Web server to support CGI executables.
IIS-ISAPIExtensions	Use this module to enable your Web server to use ISAPI extension modules to process requests.
IIS-ISAPIFilter	Use this module to enable your Web server to use ISAPI filter to customize the server behavior.
IIS-ServerSideIncludes	Use this module to enable your Web server to support .stm, .shtm, and .shtml include files.

IIS-HealthAndDiagnostics

The IIS-HealthAndDiagnostics package update contains the feature modules described in the following table:

Feature Module	Description
IIS-HttpLogging	Use this module to enable your Web server to log Web site activities.
IIS-LoggingLibraries	Use this module to install logging tools and scripts on your Web server.
IIS-RequestMonitor	Use this module to enable your Web server to monitor the health of the Web server and its sites and applications.
IIS-HttpTracing	Use this module to enable your Web server to support tracing for ASP.NET applications and failed requests.
IIS-CustomLogging	Use this module to enable your Web server to support custom logging for the Web server and its sites and applications.
IIS-ODBCLogging	Use this module to enable your Web server to support logging to an ODBC-compliant database.

IIS-Security

The IIS-Security package update contains the feature modules described in the following table:

Security Feature Module	Description
IIS-BasicAuthentication	Use this module to enable your Web server to support the HTTP 1.1 Basic Authentication scheme. This module authenticates user credentials against Windows accounts.
IIS-WindowsAuthentication	Use this module to enable your Web server to authenticate requests using NTLM or Kerberos.
IIS-DigestAuthentication	Use this module to enable your Web server to support the Digest authentication scheme. The main difference between Digest and Basic is that Digest sends password hashes over the network as opposed to the passwords themselves.
IIS-ClientCertificateMappingAuthentication	Use this module to enable your Web server to authenticate client certificates with Active Directory accounts.
IIS-IISCertificateMappingAuthentication	Use this module to enable your Web server to map client certificates 1-to-1 or many-to-1 to a Windows security identity.
IIS-URLAuthorization	Use this module to enable your Web server to perform URL authorization.
IIS-RequestFiltering	Use this module to enable your Web server to deny access based on specified configured rules.
IIS-IPSecurity	Use this module to enable your Web server to deny access based on domain name or IP address.

IIS-Performance

The following table describes the performance feature modules:

Performance Feature Module	Description
IIS-HttpCompressionStatic	Use this module to enable your Web server to compress static content before sending it to the client to improve the performance.
IIS-HttpCompressionDynamic	Use this module to enable your Web server to compress dynamic content before sending it to the client to improve the performance.

IIS-WebServerManagementTools

Figure 1-3 presents the Web server management feature modules.

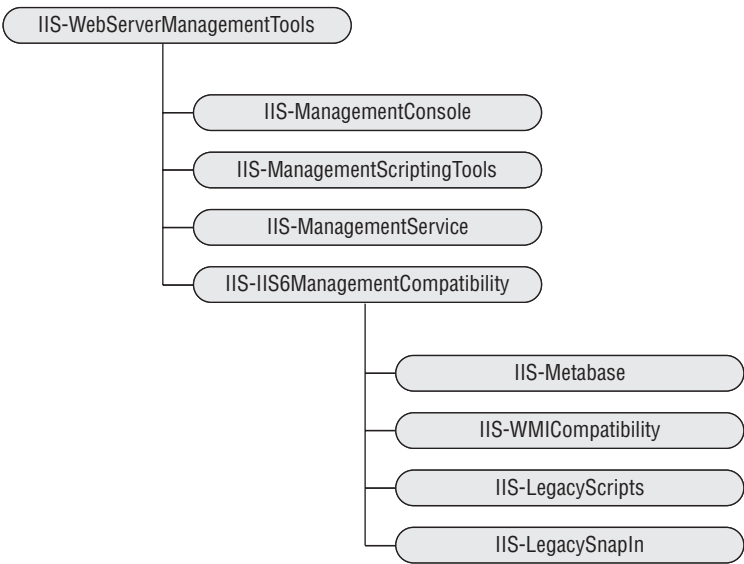


Figure 1-3

The following table describes the feature modules contained in the IIS-WebServerManagementTools update:

Feature Module	Description
IIS-ManagementConsole	This module installs the Web Server Management Console, which allows administration of local and remote IIS Web servers.
IIS-ManagementScriptingTools	Use this module to enable your Web server to support local Web server management via IIS configuration scripts.
IIS-ManagementService	Use this module to enable your Web server to be managed remotely via Web Server Management Console.

The following table presents the feature modules in the IIS-IIS6ManagementCompatibility update:

Feature Module	Description
IIS-Metabase	Use this module to enable your Web server to support metabase calls to the new IIS 7 configuration store.
IIS-WMICompatibility	Use this module to install the IIS 6.0 WMI scripting interfaces to enable your Web server to support these interfaces.
IIS-LegacyScripts	Use this module to install the IIS 6.0 configuration scripts to enable your Web server to support these scripts.
IIS-LegacySnapIn	Use this module to install the IIS 6.0 Management Console to enable administration of remote IIS 6.0 servers from this computer.

IIS-FTPPublishingService

The feature modules contained in the IIS-FTPPublishingService package update are discussed in the following table.

At the time of this writing, Microsoft announced that it would be releasing a significantly enhanced IIS 7 FTP server for Longhorn and (as a separate download) for Vista.

Feature Module	Description
IIS-FTPServer	Use this module to install the FTP service.
IIS-FTPManagement	Use this module to install the FTP Management Console.

WAS-WindowsActivationService

Figure 1-4 presents the feature modules in the WAS-WindowsActivationService package update. These modules provide the base infrastructure for process activation and management.

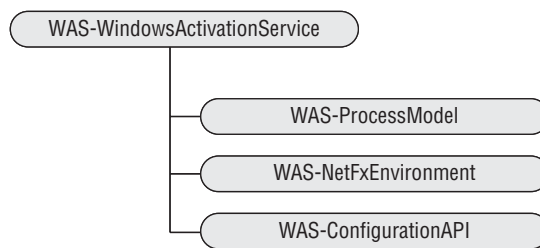


Figure 1-4

Extensible Architecture of IIS 7

IIS 6.0 allows you to extend the functionality of the Web server by implementing and plugging in your own custom ISAPI filter and extension modules. Unfortunately, ISAPI suffers from fundamental problems such as:

- ❑ Because ISAPI is not a convenient or friendly API, writing an ISAPI filter or extension module is not an easy task to accomplish. It can take a lot of time and tends to be error-prone.
- ❑ ISAPI is not a managed API, which means that ASP.NET developers cannot benefit from the rich features of the .NET Framework when they're writing ISAPI filter and extension modules.

IIS 7.0 has replaced ISAPI with a new set of convenient object-oriented APIs that make writing new feature modules a piece of cake. These APIs come in two different flavors: managed and native. The native API is a convenient C++ API that you can use to develop and plug native modules into the core Web server. The managed API, on the other hand, allows you to take full advantage of the .NET Framework and its rich environment. This allows both ASP.NET developers and IIS 7 administrators to use convenient ASP.NET APIs to extend the core Web server.

IIS 7 and ASP.NET Integrated Request Processing Pipeline

Take a look at the request processing model of IIS 6.0 for processing requests for ASP.NET content as shown in Figure 1-5. Notice that this figure contains two different request processing pipelines: IIS 6.0 and ASP.NET. Each request processing pipeline is a pipeline of components that are invoked one after another to perform their specific request processing tasks. For example, both pipelines contain an authentication component, which is called to authenticate the request.

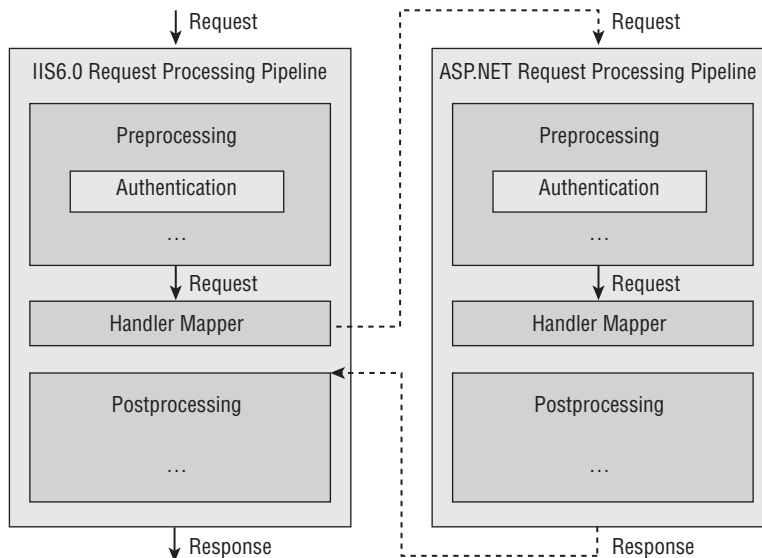


Figure 1-5

As Figure 1-5 shows, the incoming request first goes through the IIS 6.0 pipeline. At some point along this pipeline, IIS 6.0 uses its metabase to map the request to a particular handler. The requests for ASP.NET resources such as ASP.NET pages are mapped to the `aspnet_isapi.dll` handler. This handler then loads the CLR and the target ASP.NET application, if they haven't already been loaded. This is where the ASP.NET request processing pipeline kicks in.

At the beginning of the request, ASP.NET allows the components in its request processing pipeline to register one or more event handlers for one or more ASP.NET application-level events. ASP.NET then fires these events one after another and calls these event handlers to allow each component to perform its specific request processing task. At some point along the pipeline, ASP.NET uses the configuration file to map the request to a particular handler. The main responsibility of the handler is to process the request and generate the appropriate markup text, which will then be sent back to the requesting browser.

Having two separate pipelines, that is, IIS and ASP.NET pipelines working on the same request, introduces the following problems:

- ❑ There's a fair amount of duplication. For example, both pipelines contain an authentication component, which means that the same request gets authenticated twice.
- ❑ Because the ASP.NET pipeline begins after the IIS pipeline maps the request to the `aspnet_isapi` extension module, the ASP.NET pipeline has no impact on the IIS pipeline steps prior to handler mapping.
- ❑ Because the rest of the IIS pipeline steps don't occur until the ASP.NET pipeline finishes, the ASP.NET pipeline has no impact on the IIS pipeline steps either.
- ❑ Because the ASP.NET pipeline kicks in only when the IIS pipeline maps the request to the `aspnet_isapi` extension module, and because this mapping is done only for requests to ASP.NET content, the ASP.NET pipeline components cannot be applied to requests to non-ASP.NET content such as `.jpg`, `.js`, `asp`, CGI, and the like. For example, you cannot use the ASP.NET authentication and authorization modules to protect the non-ASP.NET contents of your application.

IIS 7 has changed all that by removing the `aspnet_isapi` extension module and combining the ASP.NET and IIS pipelines into a single integrated request processing pipeline as shown in Figure 1-6.

This resolves all the previously mentioned problems as follows:

- ❑ The integrated pipeline does not contain any duplicate components. For example, the request is authenticated once.
- ❑ The ASP.NET modules are now first-class citizens in the integrated pipeline. They can come before, replace, or come after any native IIS 7 modules. This allows ASP.NET to intervene at any stage of the request processing pipeline.
- ❑ Because the integrated pipeline treats managed modules like native modules, you can apply your ASP.NET managed modules to non-ASP.NET content. For example, you can use the ASP.NET authentication and authorization modules to protect the non-ASP.NET contents of your application, such as `asp` pages.

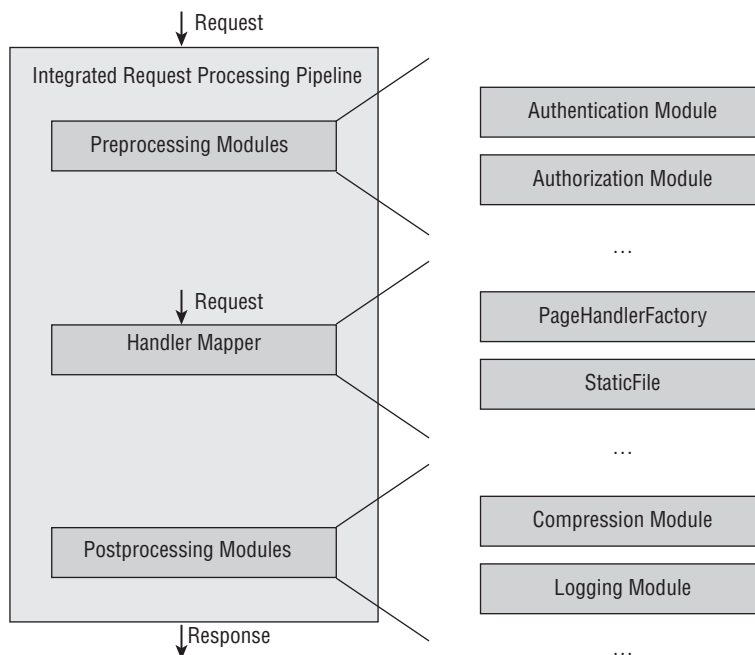


Figure 1-6

IIS 7 and ASP.NET Integrated Configuration Systems

In IIS 6.0, two separate configuration systems govern the IIS and ASP.NET pipelines. These configuration systems store their configuration settings in two different storage media, with two different schemas. IIS configuration settings are stored in the IIS 6.0 metabase, whereas ASP.NET configuration settings are stored in ASP.NET configuration files. Such separation of configuration systems makes the task of administering the Web server and its sites and applications much more complex and troublesome. For one thing, there's no way to delegate site- and application-specific IIS configuration settings to site and application administrators without compromising the integrity and security of the Web server, because all IIS configuration settings are centralized. This also takes away from the ASP.NET developers the opportunity to tailor the IIS configuration settings toward their own applications. Having two separate configuration systems for IIS and ASP.NET configuration settings also means that you have to learn two separate APIs to programmatically access and edit these configuration settings.

IIS 7 has changed all that. Having a single integrated pipeline made it possible for the IIS 7 team to introduce a single integrated configuration system for both IIS and ASP.NET settings. Because this integrated configuration system is an extension of the ASP.NET configuration system, the existing ASP.NET configuration files can easily merge into the new integrated configuration system with a little or no changes.

This integrated configuration system provides a lot of benefits to system administrators and developers alike. For one thing, both IIS and ASP.NET configuration settings are stored in storage media with the same schema. This is great news for ASP.NET developers because the new integrated schema is an extension of the ASP.NET configuration schema. Another obvious benefit of the integrated configuration system is that you can use the same set of APIs to programmatically access and set both IIS 7 and ASP.NET configuration settings.

One of the great new features of the IIS 7 and ASP.NET integrated configuration system is its declarative extensibility through a new integrated declarative schema extension markup language. Thanks to this integrated markup language, you can extend this integrated configuration system to add support for new configuration sections without writing a single line of imperative code such as C# or VB. This is a departure from the imperative extensibility model of the ASP.NET configuration system, which requires developers to write a fair amount of imperative code to extend the system.

IIS 7 and ASP.NET Integrated Administration

Having two separate configuration systems for ASP.NET and IIS in IIS 6.0 also means having two separate administration tools, GUIs, and APIs to administer and to manage them. Having a single integrated configuration system made it possible for the IIS 7 team to introduce brand new administration or management tools, GUIs, and APIs that make the task of server, site, and application administration a whole lot easier. This allows you to use the same integrated management tools, GUIs, and APIs to configure ASP.NET and IIS.

Two very important components of the IIS 7 and ASP.NET integrated administration are the integrated graphical management system and the integrated imperative management system. This book covers both of these systems and their extensibility models in detail. You will learn how to extend these two systems to add graphical and imperative management support for your own custom configuration sections.

Building a Customized Web Server

IIS 7 setup is completely modular, allowing you to custom-build your Web server from a list of more than 40 available feature modules. This ensures that your Web server contains only the feature modules you need, thereby decreasing the attack surface and footprint of your server. In this section, I walk you through the steps that you need to take to build your very own custom Web server on Windows Vista (including Windows Vista Home Premium, Windows Vista Professional, and Windows Vista Ultimate editions) and Windows Server 2008 operating systems.

In general, there are five different IIS 7 setup options:

- ☐ Windows Features dialog (Windows Vista only)
- ☐ Server Manager tool (Windows Server 2008 only)
- ☐ `pkgmgr.exe` command-line tool (both Windows Vista and Windows Server 2008)

- ❑ Unattended (both Windows Vista and Windows Server 2008)
- ❑ Upgrade (both Windows Vista and Windows Server 2008)

Before drilling down into the details of these five setup options, you need to understand the dependencies between the installable updates.

Update Dependencies

When you’re installing an update, you must also install the updates that it depends on. In general, there are two types of dependencies: interdependencies and parent-dependencies. The following table presents the update interdependencies:

Update	Depends On
IIS-WebServer	WAS-ProcessModel
IIS-ASP	IIS-ISAPIExtensions IIS-RequestFiltering
IIS-ASP.NET	IIS-DefaultDocument IIS-NetFxExtensibility WAS-NetFxEnvironment IIS-ISAPIExtensions IIS-ISAPIFilter IIS-RequestFiltering
IIS-NetFxExtensibility	WAS-NetFxEnvironment IIS-RequestFiltering
IIS-ManagementService	IIS-WebServer IIS-ManagementConsole WAS-NetFxEnvironment WAS-ConfigurationAPI
IIS-ManagementConsole	WAS-ConfigurationAPI
IIS-ManagementScriptingTools	WAS-ConfigurationAPI
IIS-LegacyScripts	IIS-Metabase IIS-WMICompatibility

Every update also depends on its parent update. For example, to install IIS-WebServer, you must also install its parent update, IIS-WebServerRole.

Windows Features Dialog

Follow these steps to use the Windows Features dialog to set up and custom-build your Web server on Windows Vista:

1. Launch the Control Panel.
2. Click the “Programs” option if Control Panel is displayed in its default view, or the “Programs and Features” option if Control Panel is displayed in Classic View.
3. Click “Turn on or off Windows features” to launch the Windows Features dialog shown in Figure 1-7. If you haven’t logged in as the built-in Administrator account, Vista will launch the User Account Control dialog. The content of this dialog depends on the privileges of your account. If your account has administrator privileges, the dialog will just ask you for confirmation. If your account does not have administrator privileges, the dialog will present you with the list of accounts with administrator privileges asking you to choose one and enter the required password. Keep in mind that you’ll get this dialog even if you have logged in as an account that has administrator privileges. This is one of the new security features.



Figure 1-7

4. Expand the Internet Information Services option to see the tree of update nodes discussed in the previous sections. You can install or uninstall each update by simply toggling it on or off and finally clicking the OK button. Notice that when you select an update, its parent update and the update that it depends on are automatically selected.

As you can see, building your own custom Web server with the Windows Features dialog is a piece of cake. You don't have to worry about the update dependencies; it's all taken care of behind the scenes. As you'll see in the following section, you don't have this luxury if you use the other two IIS 7 installation options.

Server Manager

In this section, I show you how to use the Server Manager tool to build your customized Web server on the Windows Server 2008 operating system. Before doing so, you need to familiarize yourself with three basic Windows Server 2008 terms known as *roles*, *role services*, and *features*.

Every server provides its clients with a set of services. These services are grouped into what are known as roles. Installing a server role means installing one or more role services that belong to the role. In other words, when you're installing a server role, you don't have to install all its associated role services.

Here is an example: There is a server role known as Web Server, which enables a server to exchange information over the Internet, an intranet, or an extranet. Another example of a server role is UDDI Services. This role enables a server to provide its clients with Universal Description, Discovery, and Integration (UDDI) services to exchange information about Web services over the Internet, an intranet, or an extranet.

A feature is a piece of software that does not belong to any particular role, but it provides services to one or more server roles and their associated role services. An example of a feature is the Windows Process Activation Service. This service enables the server in the Web Server role to process requests made through all kinds of communication protocols, such as TCP or HTTP.

A role, role service, or feature may depend on other roles, role services, and features. For example, the UDDI Services depend on the Web Server role for the actual exchange of information over the Internet, intranet, or extranet. When you attempt to install a role, role service, or feature that depends on other roles, role services, and features, the Server Manager prompts you to approve the installation of the roles, role services, and features on which the role, role service, or feature being installed depends.

Now back to the business at hand, which is building a customized Web server. Take one of the following steps to launch the Server Manager:

- ❑ Select Start ⇨ All Programs ⇨ Administrative Tools ⇨ Server Manager from Administrative Tools to launch the Server Manager tool shown in Figure 1-8.
- ❑ First launch the Control Panel, double-click the Administrative Tools icon in the Control Panel, and then double-click the Server Manager to launch the Server Manager tool shown in Figure 1-8.

As Figure 1-8 shows, the left pane contains a node named Server Manager, which in turn contains a child node named Roles. As just discussed, a server can be in one or more roles. As you can see from Figure 1-8, in a clean install of Windows Server 2008 the server is originally in no roles. The role that you're interested in is the Web Server role. Recall that this is the role that allows the server to share information on the Internet, an intranet, or an extranet. The first order of business is to launch a wizard named Add Roles to add this role to your server.

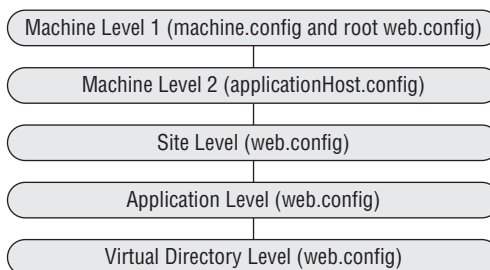


Figure 1-8

To launch the Add Roles Wizard, do one of the following:

- ☐ Click the Add Roles link button in Roles Summary panel.
- ☐ Right-click the Roles node in the Server Manager panel and select Add Roles.
- ☐ Click the Action menu and select Add Roles.

The first page of the Add Roles Wizard provides you with some preliminary instruction. Read the instructions and make sure your account meets the specified requirements as shown in Figure 1-9.

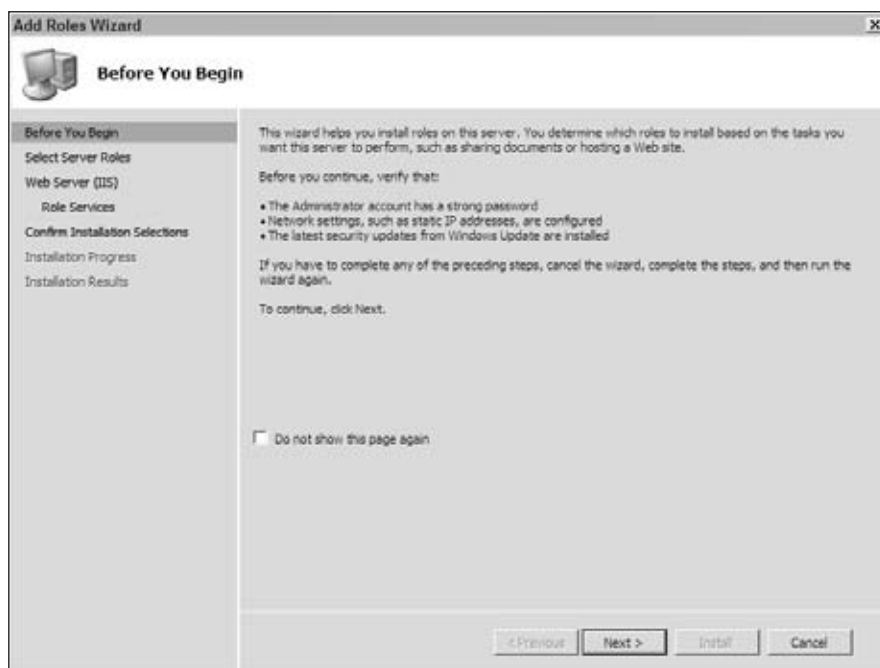


Figure 1-9

Chapter 1: IIS 7 and ASP.NET Integrated Architecture

Click the Next button to go to the page shown in Figure 1-10.

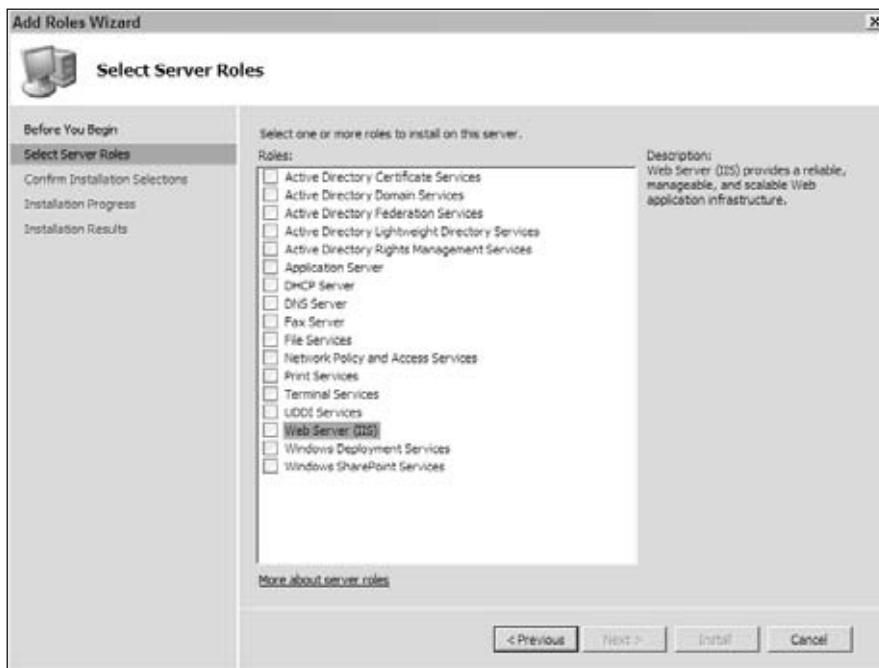


Figure 1-10

Check the Web Server (IIS) item shown in Figure 1-10. It should show you the popup shown in Figure 1-11 informing you that you need to install the Windows Process Activation Service. Click the Add Required Features button on this popup to install the Windows Process Activation Service.

Now click Next to go the next page, which provides some preliminary information. Click Next again to go to the page shown in Figure 1-12.

Notice that some package updates are already selected. These updates form the default installation of the Web server. Note that when you turn on an update that depends on other updates, the Server Manager tool pops up a message showing the updates on which the selected update depends and informing you that you need to install the dependent updates as well. For example, when you check the ASP.NET option, the Server Manager pops up the message shown in Figure 1-13.

After you're done with toggling on the desired updates, click the Next button in Figure 1-13 to move on to the confirmation page shown in Figure 1-14, which lists all the selected updates and their dependent updates. At this point these updates have not been installed yet.

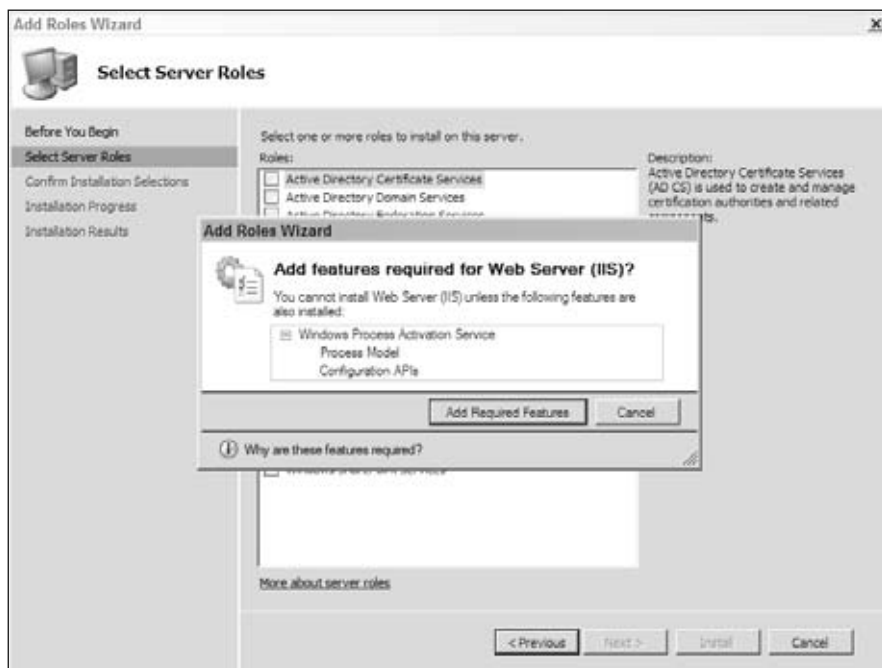


Figure 1-11



Figure 1-12



Figure 1-13

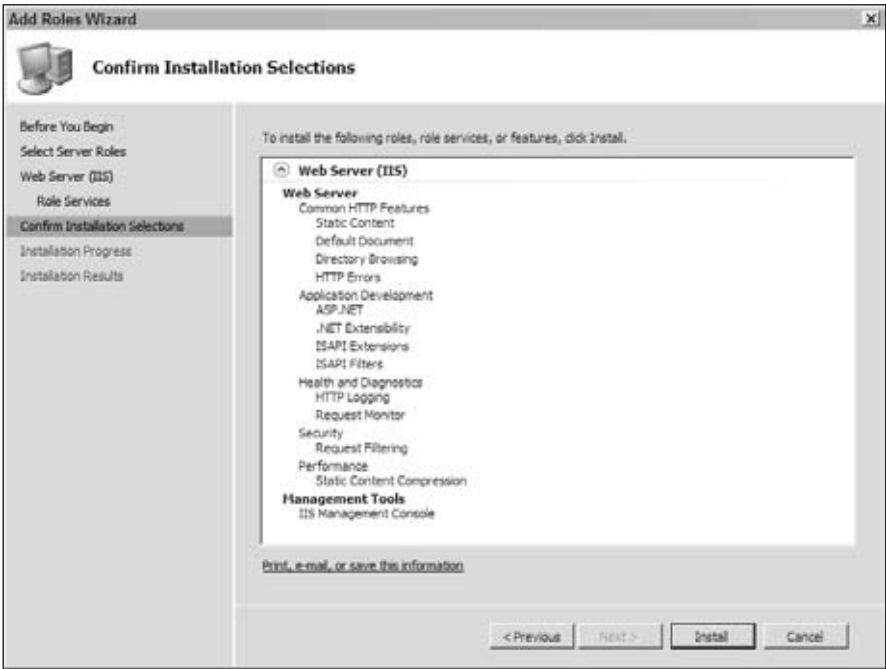


Figure 1-14

Click the Install button in Figure 1-14 to have Server Manager install the specified updates. This will take you to the progress page where you have to wait for a while for the updates to be installed. When the installation completes, the Add Roles Wizard automatically takes you to the page shown in Figure 1-15.

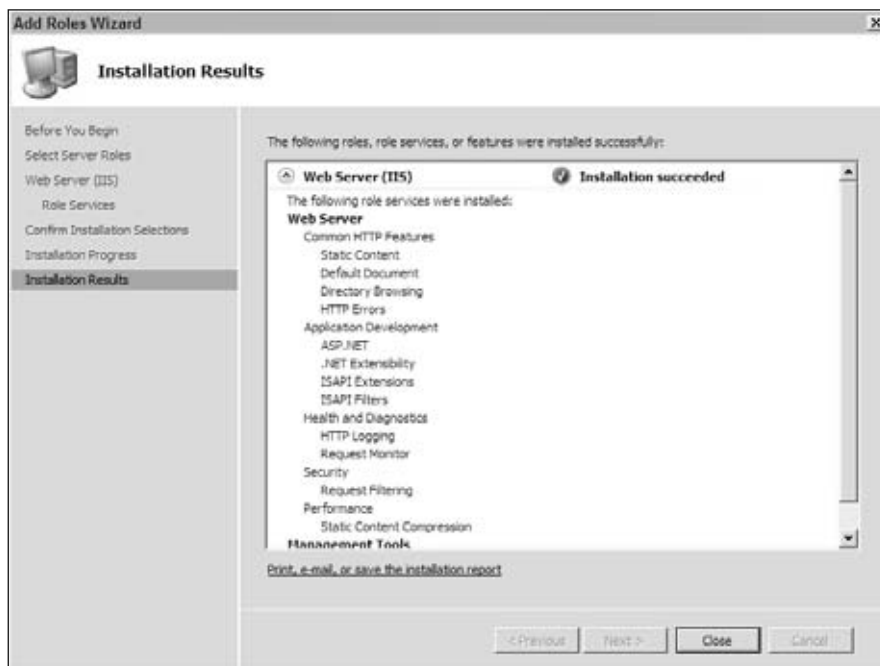


Figure 1-15

If you click the Close button in Figure 1-15, you'll be back to the Server Manager shown in Figure 1-16. Note that the Roles nodes on the left panel and the middle panel now contain a role named Web Server (IIS).

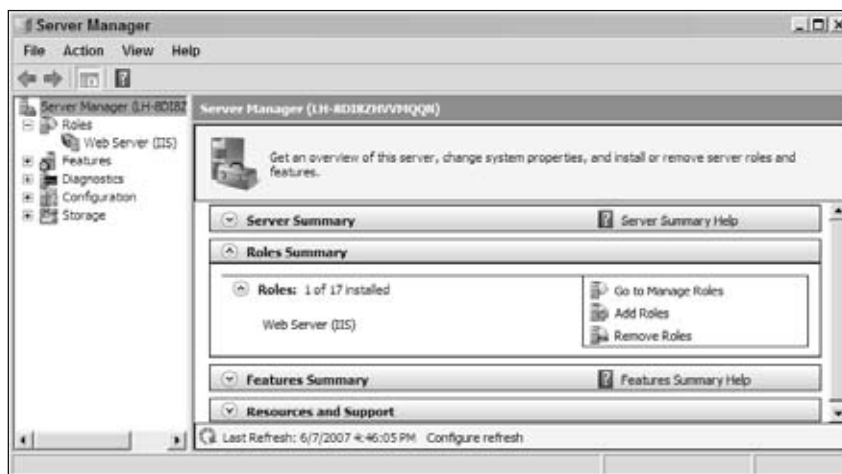


Figure 1-16

Command-Line Setup Option

Windows Vista and Windows Server 2008 come with a new command-line tool named `pkgmgr.exe` that you can use to custom install IIS 7. The following table describes the available options on this command-line tool:

Option	Description
<code>/iu:update1;update2...</code>	Run the tool with this option to install the specified updates. Notice that the update list contains a semicolon-separated list of update names discussed in the previous sections.
<code>/uu:update1;update2...</code>	Run the tool with this option to uninstall the specified updates. Notice that the update list contains a semicolon-separated list of update names discussed in the previous sections.
<code>/n:unattend.xml</code>	Run the tool with this option to install or uninstall the updates specified in the specified <code>unattend.xml</code> file. I cover this file in the following section.

When you use the `pkgmgr.exe` command-line tool to install specified updates, you must also explicitly specify and install the updates that your specified updates depend on. For example, if you decide to install the IIS-CommonHttpFeatures update, you must also install its parent update, that is, IIS-WebServer. To install the IIS-WebServer update you must also install its parent update, IIS-WebServerRole, and the update that it depends on, WAS-ProcessModel (see the update dependencies table). To install the WAS-ProcessModel update you must also install its parent update, WAS-WindowsActivationService update:

```
start /w /iu:IIS-WebServerRole;WAS-WindowsActivationService;WAS-ProcessModel;
IIS-WebServer;IIS-CommonHttpFeatures
```

Notice that if you don't specify the `start /w` option, the command-line tool will return immediately and process everything in the background, which means that you won't be able to see when the setup is completed.

Unattended Setup Option

As mentioned earlier, the `pkgmgr.exe` command-line tool comes with the `/n:unattend.xml` option. `unattend.xml` is the XML file that contains the updates to be installed or uninstalled. This XML file provides you with two benefits. First, you don't have to directly enter the names of the updates on the command line. Second, you can store this file somewhere for reuse in other Web server machines. This XML file must have the same schema as the XML file shown in Listing 1-1. This listing installs the IIS-CommandHttpFeatures update and the updates that it depends on as discussed in the previous section.

Listing 1-1: The unattend.xml File

```
<?xml version="1.0" ?>
<unattend xmlns="urn:schemas-microsoft-com:unattend"
  xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State">
  <servicing>
    <!--Install a selectable update in a package that is in the
    Windows Foundation namespace-->
    <package action="configure">
      <assemblyIdentity name="Microsoft-Windows-Foundation-Package"
        version="6.0.5308.6" language="neutral" processorArchitecture="x86"
        publicKeyToken="31bf3856ad364e35" versionScope="nonSxS" />

      <selection name="IIS-WebServerRole" state="true"/>
      <selection name="WAS-WindowsActivationService" state="true"/>
      <selection name="WAS-ProcessModel" state="true"/>
      <selection name="IIS-WebServer" state="true"/>
      <selection name="IIS-CommonHttpFeatures" state="true"/>
    </package>
  </servicing>
</unattend>
```

Notice that the `<servicing>` element contains one or more `<selection>` child elements, and each child element specifies a particular update. The `<selection>` child element features two attributes named `name` and `state`. The `name` attribute contains the update name to be installed or uninstalled. Set the `state` attribute to `true` to install or `false` to uninstall the specified update.

Upgrade

If you're upgrading from Windows XP to Windows Vista, or from Windows Server 2003 to Windows Server 2008, and if your old operating system has IIS installed, the Windows Vista or Windows Server 2008 setup automatically scans through the capabilities of the installed IIS and ensures that the new install of IIS 7 supports those features and capabilities. Unfortunately, due to the monolithic architecture of IIS 5.1 and IIS 6.0, this installation ends up installing almost all of the feature modules of IIS 7. I highly recommend that after the upgrade you use one of the previously discussed installation options to uninstall the updates that you do not need to decrease the attack surface and footprint of your Web server.

Summary

This chapter first covered the IIS 7 package updates and their constituent feature modules, and showed you how to custom-build your own Web server from the desired package updates to decrease the footprint of your Web server. The chapter then provided in-depth coverage of five different IIS7 setup options. The chapter also gave an overview of the main systems that make up the IIS7 and ASP.NET integrated infrastructure. As discussed, one of these systems is the IIS7 and ASP.NET integrated configuration system, which will be discussed thoroughly in the next chapter.

2

Using the Integrated Configuration System

This chapter discusses the new IIS 7 and ASP.NET integrated configuration system. You'll learn about the hierarchical structure of the configuration files that make up this integrated system, the hierarchical relationships among the files themselves, and the notion of the declarative versus imperative schema extension. The chapter then dives into the structure of the new IIS 7 machine-level configuration file named `applicationHost.config`, where you'll also learn how to override the configuration settings specified in different sections of this file in a particular site, application, or virtual directory.

Integrated Configuration System

IIS 7 comes with a new configuration system that has the following important characteristics:

- ❑ It has the same format, grammar, and syntax as the .NET Framework configuration system. This is great news for ASP.NET developers. They should immediately feel at home with this configuration system.
- ❑ It's heavily dependent on the file system for backup, restore, and security. This makes deployment easier because you can simply copy the configuration files from the development machine to the production machine. The file system security is based on file ACLs, which are very straightforward and easy to manage.

Chapter 2: Using the Integrated Configuration System

- ❑ It's hierarchical. A flat configuration file, such as the one used in IIS 6.0, introduces problems such as:
 - ❑ **Readability:** If you take a look at the `MetaBase.xml` file in IIS 6.0 you'll see that it consists of a long list of sections, which makes it extremely difficult to read and locate sections.
 - ❑ **Updatability:** Updating sections in `MetaBase.xml` is very error-prone due to the flatness of the document.
- ❑ It's distributed. Because the IIS 6.0 configuration system is centralized, every little configuration change requires direct involvement of the machine administrator. This causes many problems such as:
 - ❑ It doesn't give the opportunity to the site and application administrators and developers to perform site- and application-specific configuration tuning.
 - ❑ It puts too much on the machine administrator's plate.
- ❑ Its schema is declarative. As you'll see later, one of the problems with the .NET Framework configuration system is that you have to write a lot of code to extend the configuration schema. IIS 7 allows what is known as declarative schema extension, which does not require a single line of code.
- ❑ Configuration files are the master of the configuration state. This is very different from IIS 6.0 where the master is an in-memory configuration database that maintains the configuration state. IIS 7 has made things simple because it has removed this in-memory configuration database and relies completely on the content of the configuration files themselves. IIS 7 automatically picks up any changes made to the configuration files without any effort on the part of the administrator or developer. For example, administrators don't need to restart the server, site, or application. If there's a need for a restart, IIS 7 automatically takes care of it. This makes your life much easier. Make the configuration change in the configuration file, and IIS 7 picks it up right away. No more in-memory configuration database to worry about. Simple is good!

I discuss these and some other characteristics in more detail in the following sections.

Hierarchical Configuration Schema

One of the main problems with the IIS 6.0 `MetaBase.xml` configuration file is that it is flat. This flatness leads into an XML document that contains a very long list of sections. As discussed earlier, this makes it hard to read the document and locate and edit sections. Editing such a file is always error-prone.

An IIS 7 configuration file is hierarchical as follows. Following the .NET Framework configuration system, IIS 7 makes use of the notion of configuration sections and section groups. Configuration sections and section groups should be familiar notions to ASP.NET developers. Many ASP.NET features have a dedicated configuration section that allows page developers to configure the feature. For example, the page developer can use the `<sessionState>` configuration section to configure the ASP.NET session state infrastructure. As you'll see later, the configuration section is also the unit of extension. In other words, you have to add a configuration section to extend the existing configuration system.

To help you understand the notion of a hierarchical configuration file, let's examine a configuration section that most ASP.NET developers are already familiar with, that is, the `<compilation>` configuration section. This configuration section allows you to configure the ASP.NET dynamic compilation system. Listing 2-1 presents the portion of the `<compilation>` section of the root `web.config` file. I discuss the root `web.config` file in more detail later in this chapter.

Listing 2-1: Portion of the `<compilation>` Section of the Root `web.config` File

```
<configuration>
  <system.web>
    <compilation batch="true">
      <assemblies>
        <add assembly="mscorlib" />
        <add assembly="System, Version=2.0.0.0, Culture=neutral,
          PublicKeyToken=b77a5c561934e089" />
      </assemblies>
      <buildProviders>
        <add extension=".aspx" type="System.Web.Compilation.PageBuildProvider" />
        <add extension=".wsdl" type="System.Web.Compilation.WsdlBuildProvider" />
      </buildProviders>
    </compilation>
  </system.web>
</configuration>
```

The inspection of Listing 2-1 reveals the characteristics described in the following sections.

Section Groups

The `<compilation>` configuration section is enclosed within the opening and closing tags of the `<system.web>` element. The `<system.web>` element is an example of what is known as a *section group*. Whereas a configuration section such as `<compilation>` is used to configure a particular feature such as the compilation system, a section group such as `<system.web>` does not represent any particular feature. Instead it is used to organize and group configuration sections. For example, the `<system.web>` section group contains all the configuration sections that are used to configure ASP.NET features, but the section group itself is not used to configure any particular ASP.NET feature.

Here are some rules about configuration sections and section groups:

- ❑ A configuration section cannot belong to more than one section group.
- ❑ A configuration section does not have to belong to any group. This is because section groups are used for organizational and grouping purposes and have no impact on the feature being configured.
- ❑ A configuration section cannot contain another configuration section.
- ❑ A section group can contain other section groups, and can be contained by another section group. This is the secret of the hierarchical structure of configuration files.
- ❑ Because a section group does not configure any particular feature, it does not expose any attributes (see the `<system.web>` section group shown in Listing 2-1).

Configuration Sections

As discussed earlier, each configuration section is specifically designed to configure a particular feature. A configuration section provides you with one or more of the following means to set the configuration settings for the specified feature:

- ❑ Most configuration sections, such as `<compilation>`, expose XML attributes that you can set to specify the configuration settings. These XML attributes are known as *configuration properties*.

```
<compilation batch="true">
```

- ❑ Some configuration sections contain child XML elements that you can set to specify the configuration settings. These XML elements are known as *configuration elements*.
- ❑ Some configuration sections contain child XML elements that contain one or more `<add>`, `<remove>`, or `<clear>` elements, which you can set to specify the configuration settings. These child XML elements are known as *configuration collections*. Configuration collections do not expose any XML attributes. In this example, `<assemblies>` is the configuration collection because it contains the `<add>` elements that add new referenced assemblies.

```
<compilation batch="true">
  <assemblies>
    <add assembly="mscorlib" />
    <add assembly="System, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" />
  </assemblies>
</compilation>
```

Distributed Configuration System

So far I've been talking about the hierarchical nature of the internal structure of a given configuration file. Following the .NET Framework configuration system, the IIS 7 configuration system extends this hierarchical notion to the relationships between the configuration files themselves, as shown in Figure 2-1.

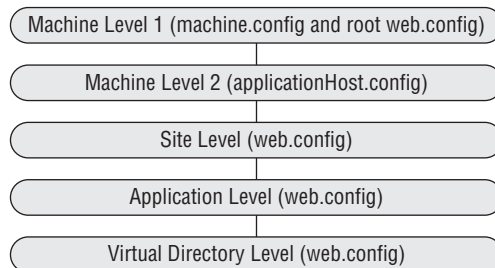


Figure 2-1

As Figure 2-1 shows, the IIS 7 and ASP.NET integrated or unified configuration system consists of five hierarchical levels as described in the following table:

Hierarchical Level	Description
Machine Level 1	This level consists of two configuration files named <code>machine.config</code> and the root <code>web.config</code>. The <code>machine.config</code> configuration file specifies the global .NET configuration settings except for a few settings that are specific to ASP.NET. The root <code>web.config</code> configuration file specifies the global ASP.NET configuration settings that apply to all ASP.NET applications running on the machine except for those settings specified within <code><location></code> tags. I discuss the <code>location</code> tags later in this chapter. Both of these machine-level configuration files reside in the <code>%WINDIR%\Microsoft.NET\Framework\v2.0.50727\CONFIG</code> directory.
Machine Level 2	This level consists of a single configuration file named <code>applicationHost.config</code>, which specifies the global IIS 7 configuration settings that apply to all sites, applications, and virtual directories running on the machine except for those IIS 7 configuration settings specified within <code><location></code> tags. The <code>applicationHost.config</code> file resides in the <code>%WINDIR%\system32\inetsrv\config</code> directory.
Site Level	This level consists of a configuration file named <code>web.config</code>, which specifies the IIS 7, .NET, and ASP.NET configuration settings that apply to all applications and virtual directories in the specified site. This file resides within the site root directory.
Application Level	This level consists of a configuration file named <code>web.config</code>, which specifies the IIS 7, .NET, and ASP.NET configuration settings that apply to the specified application and all its virtual directories. This file resides within the application root directory.
Virtual Directory Level	This level consists of a configuration file named <code>web.config</code>, which specifies the IIS 7, .NET, and ASP.NET configuration settings that apply to the specified virtual directory. This file resides within the root of the virtual directory.

Upon installation, the IIS 7 and ASP.NET unified configuration system contains only the machine-level configuration files, that is, `machine.config`, the root `web.config`, and `applicationHost.config`. However, the site-, application-, and virtual directory-level `web.config` files can be added as needed to tailor the IIS 7, .NET, and ASP.NET configuration settings toward a specific site, application, or virtual directory.

One consequence of the hierarchical relationship between the configuration files shown in Figure 1-17 is that the lower-level configuration files inherit configuration settings from the upper-level configuration files. The lower-level configuration files have the option of overriding the configuration settings they inherit from the upper-level configuration files to tailor them to their specific requirements.

Chapter 2: Using the Integrated Configuration System

Another important feature of the IIS 7 and ASP.NET unified configuration system is that an upper-level configuration file can lock down specified IIS 7, .NET, or ASP.NET configuration settings to prevent the lower-level configuration files from changing the values of these configuration settings. Upon installation, all IIS 7 configuration settings specified in `applicationHost.config` are locked down by default to ensure that only the machine administrator can change these settings. However, the machine administrator has the option of unlocking specified configuration settings to allow site and application administrators and developers to tailor these settings toward their specific needs. This is known as *delegation*, and is discussed in the next section.

The hierarchical/override/lockdown relationship between the configuration files in the IIS 7 and ASP.NET unified configuration system make up a distributed configuration system. This allows an application to xcopy its configuration file from the development to the test and production machine, where the application can start to work immediately without any further configuration.

You may be wondering why there are two sets of machine-level configuration files. Why not merge the `machine.config` and `root web.config` files into the `applicationHost.config` file so you could have a single configuration file at the machine level? Recall that the `machine.config` and `root web.config` files specify .NET- and ASP.NET-specific configuration settings, whereas the `applicationHost.config` file specifies the IIS 7-specific configuration settings. Because IIS 7 is an integral part of the Microsoft operating system, its release cycle follows the operating system. The .NET Framework, on the other hand, follows the release cycle of Visual Studio. As such it makes sense to keep the machine-level IIS 7 and .NET configuration settings in separate configuration files.

You may be also wondering why there are two-machine level .NET configuration files, `machine.config` and the `root web.config`. As you may have noticed, ASP.NET 1.x has a single machine-level configuration file, `machine.config`. Packing both ASP.NET and general .NET configuration settings in a single file leads into a very long file that is hard to read and edit. That's why .NET Framework 2.0 has moved the ASP.NET configuration settings to a different file known as the `root web.config`. As Figure 1-17 shows, the `machine.config` and `root web.config` files are at the same configuration system hierarchy level.

<location> Tags

As discussed earlier, you can add a `web.config` file to a site, application, or virtual directory to customize the IIS 7 and ASP.NET configuration settings at the site, application, or virtual directory level. There are scenarios where such configuration customization may not be possible by adding a `web.config` file, such as:

- ❑ The machine administrator wants to enforce certain configuration settings on a particular site, application, or virtual directory. Adding a new `web.config` file is not a viable solution because it allows the site or application administrator or the developer to change the specified configuration settings at will.
- ❑ The site administrator wants to enforce certain configuration settings on a particular application or virtual directory. Adding a new `web.config` file is not a viable solution because it allows the application administrator or developer to change the specified configuration settings at will.

- ❑ The application administrator wants to enforce certain configuration settings on a particular virtual directory. Adding a new `web.config` file is not a viable solution because it allows developers to change the specified configuration settings at will.
- ❑ Two or more virtual directories mapped to the same physical directory require different configuration settings. Adding a new `web.config` file is not a viable solution because both virtual directories will share the configuration settings specified in the `web.config` file.
- ❑ The machine, site, or application administrator wants to specify configuration settings that apply only to a particular file. Adding a new `web.config` file to the directory where the specified file is located is not a viable solution because all files in that directory and its subdirectories will share the same configuration settings specified in the `web.config` file.

This is where the `<location>` tags come into play. A `<location>` tag allows the configuration file at a higher level to enforce configuration settings on a lower level without adding `web.config` files. A `<location>` tag contains configuration settings for one or more configuration sections. The `<location>` tag features an XML attribute named `path` that can be set to one of the following values:

- ❑ `"."` or `""`: Specifies the configuration settings that apply to the level at which the `<location>` tag is added. For example, if you add the `<location>` tag to the `applicationHost.config` file, which is at the machine level, the configuration settings specified in the `<location>` tag will be global.
- ❑ `"SiteName"`: Specifies the configuration settings that apply only to the site with the specified name. The following listing is an excerpt from the `applicationHost.config` file. This `<location>` tag specifies the `Default.aspx`, `Default.htm`, `Default.asp`, `index.htm`, and `iisstart.asp` files as the default document for the "Default Web Site" site. If the URL of a request does not contain the name of the requested resource, it will default to the `Default.aspx` file.

```
<location path="Default Web Site">
  <system.webServer>
    <defaultDocument>
      <files>
        <clear />
        <add value="Default.aspx" />
        <add value="Default.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="iisstart.asp" />
      </files>
    </defaultDocument>
  </system.webServer>
</location>
```

- ❑ `"SiteName/AppName"`: Specifies the configuration settings that apply only to the application with the specified name that belongs to the site with the specified name. The following listing is an excerpt from an `applicationHost.config` file. This `<location>` tag specifies Windows Authentication as the authentication mechanism for the application named `MyApplication` that belongs to the "Default Web Site" site.

```
<location path="Default Web Site/MyApplication">
  <system.webServer>
```



```
<security>
  <authentication>
    <windowsAuthentication enabled="true" />
  </authentication>
</security>
</system.webServer>
</location>
```

- ❑ "SiteName/AppName/VirDirName": Specifies the configuration settings that apply only to the virtual directory with the specified name that belongs to the application with the specified name, which in turn belongs to the site with the specified name.
- ❑ "SiteName/AppName/VirDirName/PhysDirName": Specifies the configuration settings that apply only to the physical directory with the specified name that maps to the virtual directory with the specified name that belongs to the application with the specified name, which in turn belongs to the site with the specified name.
- ❑ "SiteName/AppName/VirDirName/PhyDirName/PhysDirName": Specifies the configuration settings that apply only to the physical directory with the specified name, which is a subdirectory of the physical directory with the specified name that maps to the virtual directory with the specified name that belongs to the application with the specified name, which in turn belongs to the site with the specified name.
- ❑ "SiteName/AppName/VirDirName/PhysDirName/FileName": Specifies the configuration settings that apply only to the file with the specified name, which is located in the physical directory with the specified name that maps to the virtual directory with the specified name that belongs to the application with the specified name, which in turn belongs to the site with the specified name.
- ❑ "SiteName/AppName/VirDirName/FileName": Specifies the configuration settings that apply only to the file with the specified name, which is located in the virtual directory with the specified name that belongs to the application with the specified name, which in turn belongs to the site with the specified name.

Keep the following four characteristics of `<location>` tags in mind:

- ❑ `<location>` tags can be used in configuration files at all levels.
- ❑ The value of the `path` attribute cannot reference a location above the current level. For example, you cannot specify a "site" location from within a `web.config` file that resides in a virtual directory.
- ❑ The same `<location>` tag can contain multiple configuration sections.
- ❑ Multiple `<location>` tags are allowed in the same configuration file in a given level provided that no two `<location>` tags have the same path values.

The `<location>` tag features an attribute named `overrideMode` with the possible values described in the following table:

Value	Description
Allow	Use this value to allow the lower-level configuration files to override the configuration settings of all the configuration sections contained in the <location> tag for a particular path.
Deny	Use this value to prevent the lower-level configuration files from overriding the configuration settings of the configuration sections contained in the <location> tag for a particular path.
Inherit	Use this value to have the contained configuration sections use their own default values for the <code>overrideMode</code> attribute. This is the default.

Include Files

As discussed earlier, moving from the flat IIS 6.0 configuration file structure to the hierarchical IIS 7 configuration file structure makes it easier to read and edit configuration files. However, it doesn't address the situations where you may have a few very long configuration sections that clutter your configuration file.

In these cases you can move a selected configuration section from your configuration file to a new configuration file and assign the physical path of the new configuration file to the `configSource` attribute of the selected section. For example, Listing 2-2 presents the very small portion of the following <site> element that represents the Default Web Site in the `applicationHost.config` file.

As you can see, a site could have numerous <application> child elements where each child element in turn could have numerous <virtualDirectory> child elements.

Listing 2-2: The Original Configuration File

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="Default Web Site" id="1" serverAutoStart="false">
        <application path="/" applicationPool="DefaultAppPool">
          <virtualDirectory path="/Example1" physicalPath="D:\IIS 7\Example1" />
          <virtualDirectory path="/junk2" physicalPath="D:\IIS 7\Example2" />
          . . .
        </application>
        <application path="/IISHelp" applicationPool="AppPool_Medium">
          <virtualDirectory path="/" physicalPath="C:\Windows\help\iishelp" />
        </application>
        . . .
      <bindings>
        <binding protocol="http" bindingInformation="127.0.0.1:80:" />
      </bindings>
      <traceFailedRequestsLogging enabled="true" />
      <logFile customLogPluginClsid="{FF160663-DE82-11CF-BC0A-00AA006111E0}" />
    </site>
    . . .
  </sites>
</system.applicationHost>
</configuration>
```

Chapter 2: Using the Integrated Configuration System

Listing 2-3 presents the same configuration file where the long `<site>` element has been replaced with a `<site>` element whose `configSource` attribute has been set to the physical path of the new configuration file, `DefaultWebSite.config`, that contains the original `<site>` element, as shown in Listing 2-4.

Listing 2-3: The Configuration File Where the `configSource` Attribute Is Used

```
<configuration>
  <system.applicationHost>
    <sites>
      <site configSource="DefaultWebSite.config" />
      . . .
    </sites>
  </system.applicationHost>
</configuration>
```

Listing 2-4: The New `DefaultWebSite.config` File

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="Default Web Site" id="1" serverAutoStart="false">
        <application path="/" applicationPool="DefaultAppPool">
          <virtualDirectory path="/Example1" physicalPath="D:\IIS 7\Example1" />
          <virtualDirectory path="/junk2" physicalPath="D:\IIS 7\Example2" />
          . . .
        </application>
        <application path="/IISHelp" applicationPool="AppPool_Medium">
          <virtualDirectory path="/" physicalPath="C:\Windows\help\iishelp" />
        </application>
        . . .
      <bindings>
        <binding protocol="http" bindingInformation="127.0.0.1:80:" />
      </bindings>
      <traceFailedRequestsLogging enabled="true" />
      <logFile customLogPluginClsid="{FF160663-DE82-11CF-BC0A-00AA006111E0}" />
    </site>
  </sites>
</system.applicationHost>
</configuration>
```

<configSections>

This is an optional section; when used, it is always the very first section of a configuration file. The `<configSections>` section is used to register sections that can be used in the configuration file and all the lower-level configuration files. Note that there's a difference between registering and implementing a section. `<configSections>` can only register already-implemented configuration sections. Later in Chapter 5, I show you how to implement a new configuration section. The following listing presents an excerpt from the `applicationHost.config` configuration file. This listing registers a section named `<applicationPools>` that belongs to the `<system.applicationHost>` section group.

```
<configuration>
  <configSections>
```

```
<sectionGroup name="system.applicationHost"
type="System.ApplicationHost.Configuration.SystemApplicationHostSectionGroup,
System.ApplicationHost, Version=7.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35">
  <section name="applicationPools" overrideModeDefault="Deny"
allowDefinition="MachineOnly" />
  . . .
</sectionGroup>
. . .
</configSections>
. . .
<system.applicationHost>
  <applicationPools>
    . . .
  </applicationPools>
  . . .
</system.applicationHost>
</configuration>
```

Note that the `<section>` element is used to register a configuration section and the `<sectionGroup>` element is used to specify the section group to which the section being registered belongs. Also notice that the `<section>` element features an XML attribute named `overrideModeDefault` that specifies the default value of the `overrideMode` attribute of the section being registered. The `overrideMode` attribute of a section specifies whether the lower-level configuration files can override the configuration settings specified in the section.

This allows a higher-level configuration file to lock certain configuration sections to prevent lower-level configuration files from overriding the configuration settings specified in those sections. On installation, all IIS 7 configuration sections are locked by default, which means that only the machine administrator can edit these configuration sections. This allows the machine administrator to unlock sections on a case-by-case basis to delegate the administration of the selected sections to the site or application administrators.

Notice that the `<section>` element exposes an attribute named `allowDefinition`, which can be used to specify the hierarchy level whose configuration files can use the registered section. For example, the `<section>` element in the previous listing sets the `allowDefinition` attribute to the `MachineOnly` value to specify that the `<applicationHost>` section can only be used in the machine-level configuration file, that is, the `applicationHost.config` file. In other words, you cannot use the `<applicationHost>` section in the site, application, or virtual directory configuration file.

There are two other important facts about the `<configSections>` section:

- ❑ As mentioned earlier, this section is optional. If you don't specify this section, you're limited to using the configuration sections registered in the higher-level configuration files.
- ❑ A lower-level configuration file cannot unregister or re-register a configuration section already registered in a higher-level configuration file. It can only register new configuration sections.

The new IIS 7 and ASP.NET integrated configuration system comes with a new machine-level configuration file named `applicationHost.config`. Which configuration settings are specified in this file depends on how you install and set up IIS 7, as discussed in the previous chapter. As such, the settings in your `applicationHost.config` file may be different from the settings I'm using in this chapter.

Chapter 2: Using the Integrated Configuration System

Understanding some parts of this file requires a good understanding of these three important architectural components of IIS 7: Protocol listeners, the World Wide Web Publishing Service (WWW Service), and the Windows Activation Service (WAS). As such, I'll begin this section with the coverage of these three components.

Protocol Listeners

Different applications may require their clients to use different protocols to communicate with them. Here are a few examples:

- ❑ The underlying protocol in Web applications is HTTP. These applications process HTTP requests and send HTTP responses back to the requesting browsers.
- ❑ The Windows Communications Foundation (WCF) applications support a variety of protocols including HTTP, NET.TCP, NET.PIPE, and NET.MSMQ.

A protocol listener is a component that is responsible for listening for incoming requests made through a specific type of protocol and passing them onto IIS 7 for processing. Each protocol has its own protocol listeners. IIS 7 comes with four protocol listeners: HTTP.SYS, NET.TCP, NET.PIPE, and NET.MSMQ. Introducing new protocols will require plugging new protocol listeners into IIS 7.

Notice that IIS 7 still uses HTTP.SYS for HTTP requests but with a new security enhancement, that is, support for SSL. HTTP.SYS in IIS 7 supports the same features that it does in IIS 6.0:

- ❑ HTTP.SYS is implemented as a kernel-mode device driver.
- ❑ HTTP.SYS directly delivers the incoming HTTP requests to the worker process responsible for processing the requests without any interprocess communication overhead. In versions of IIS prior to IIS 6.0, the HTTP request was first passed into a user-mode process named `inetinfo.exe`, which in turn passed the request to the worker process. This involved interprocess communication between IIS and the worker process.
- ❑ Each application pool has its own kernel-level request queue. When there's no worker process available for processing an HTTP request, HTTP.SYS queues the request in this kernel-level request queue. This allows the worker process to pick up the request directly from the queue, which again does not involve interprocess communication.
- ❑ HTTP.SYS caches the output response in a kernel-level cache to service subsequent requests, bypassing the application pool and worker process. This dramatically improves IIS performance.

Windows Process Activation Service

One of the main responsibilities of the Windows Process Activation Service (WAS) component is to read the configuration settings specified in the `applicationHost.config` file. Some of these settings are used to configure the protocol listeners. As discussed earlier, each protocol listener is specifically designed to handle a specific type of protocol. For example, HTTP.SYS is specifically designed to handle HTTP requests.

If WAS were to directly interact with the underlying protocol listener, it would be tied to the protocol that the listener is designed for, and would not be able to work with other protocols. Enter protocol listener adapters.

A protocol listener adapter isolates WAS from the associated protocol listener. Each protocol listener comes with its own adapter. For example, there is an adapter that knows how to adapt the HTTP.SYS listener to WAS. As a matter of fact, Windows Communications Foundation (WCF) comes with adapters for a variety of protocol listeners including HTTP.SYS, NET.TCP, NET.PIPE, and NET.MSMQ. As you'll see later in this book, thanks to these adapters, a WCF service hosted in IIS 7 can process requests made through a variety of communication protocols.

WAS passes the listener configuration settings that it reads from the `applicationHost.config` file to the associated protocol listener adapter. The adapter in turn uses these configuration settings to configure and set up its associated protocol listener for listening for the requests coming through a specific communication channel.

Besides listener configuration, WAS has two more important responsibilities. First, it must use the configuration settings from the `applicationHost.config` file to configure and set up application pools for processing requests. I discuss these configuration settings later in this chapter. Second, it must use the configuration settings from the `applicationHost.config` file to monitor, start, shut down, and manage the applications pools and their associated worker processes.

When a request arrives, the associated protocol listener picks it up. The protocol listener adapter then informs the WAS that a request for a specified application pool has arrived. The WAS checks whether a worker process has already been assigned to the application pool. If not, it spawns a new worker process and assigns the task of processing requests for the application pool to this worker process, which in turn picks up the request from its associated queue and processes it.

World Wide Web Publishing Service

The WWW Service has gone through lot of changes in the transition from IIS 6.0 to IIS 7. The main motivation for these changes was to add support for protocol listeners other than HTTP.SYS. These changes allow you to run Windows Communications Foundation (WCF) applications on IIS 7 to process requests made through variety of protocols such as HTTP, NET.TCP, NET.PIPE, and NET.MSMQ. These changes also leave the door open for supporting new communication protocols. Now, let's take a look at changes in the WWW Service.

The WWW Service in IIS 6.0 is responsible for all the following tasks:

- ❑ Configuring and setting up the HTTP.SYS HTTP protocol listener
- ❑ Monitoring performance and providing performance counters
- ❑ Configuring and setting up application pools and their associated worker processes
- ❑ Starting, monitoring, killing, and managing worker processes

Closer examination of the last two tasks and responsibilities reveals that these two tasks are not specific to the HTTP protocol. In other words, the same application pool or worker process may receive requests

Chapter 2: Using the Integrated Configuration System

based on any type of protocol. This means that the last two tasks must be performed regardless of what communication protocol is used.

IIS 7 has moved the last two tasks or responsibilities from the WWW Service to a new IIS 7 service known as WAS, which was discussed in the previous section. In other words, the WWW Service in IIS 7 is only responsible for configuring and setting up the HTTP.SYS protocol listener and providing performance counters. As such, the WWW Service in IIS 7 is in effect the listener adapter for the HTTP.SYS listener.

Moving the responsibility of those two tasks from the HTTP-specific WWW Service to the protocol-agnostic WAS component allows you to deploy Windows Communications Foundation (WCF) applications on an IIS 7 Web server where these applications can process requests made through variety of communication protocols.

The Structure of the `applicationHost.config` File

As discussed in the previous sections, the WAS reads the configuration settings specified in the `applicationHost.config` file and uses them to configure, set up, and manage application pools and their associated worker processes. I discuss the structure of this file in this section. `applicationHost.config` is located in the following directory on your machine:

```
%SystemRoot%\system32\inetsrv
```

You need administration privileges to view or edit the `applicationHost.config` file.

Like any other configuration file, `applicationHost.config` is an XML file with a document element named `<configuration>`. The `<configuration>` document element contains the following child elements:

- ❑ `<configSections>`: This section registers configuration sections used in the `applicationHost.config` file. The previous chapter discussed this section in detail.
- ❑ `<system.applicationHost>`: This section group contains configuration sections used by WAS. As such, this section group can be used only in the `applicationHost.config` configuration file, that is, at the machine level.
- ❑ `<system.webServer>`: This section group contains IIS 7-specific configuration sections and can be used in lower-level configuration files if permitted.
- ❑ Zero or more `<location>` tags. The previous chapter discussed `<location>` tags in detail.

`<system.applicationHost>`

The `<system.applicationHost>` section group allows you to manage IIS 7 application pools and Web sites. As such it contains the following important child elements:

- ❑ `<applicationPools>`
- ❑ `<sites>`

<applicationPools>

One of the important features of IIS 6.0 is application pools. An application pool is a set of applications that share one or more `w3wp.exe` worker processes. When the first request for an application pool arrives, IIS 6.0 spawns a new instance of the `w3wp.exe` worker process to process the requests for the specified application pool. A given worker process can process requests for one and only one application pool. In other words, application pools do not share the same copy of the worker process, which means that application pools are isolated by process boundaries. One of the main advantages of this process isolation is that if one application misbehaves in an application pool and brings down the worker process dedicated to that application pool, it will not bring down applications in other application pools because they're not running in the same worker process. This process isolation is one of the secrets behind the stability and reliability of IIS 6.0.

IIS 7 still uses application pools but it has also resolved the source of a big problem in IIS 6.0. One of the fundamental characteristics of the Common Language Runtime (CLR) is that you cannot load two versions of the CLR into the same operating system process, such as a `w3wp` worker process. As mentioned earlier, when the first request for an application pool arrives, IIS 6.0 spawns a new worker process to process requests for the application pool. This worker process loads the `aspnet_isapi.dll` ISAPI extension module and dispatches the request to it if the request is made for ASP.NET content. Each version of the ASP.NET Framework comes with its own version of `aspnet_isapi.dll`. `aspnet_isapi.dll` loads the version of the CLR that the ASP.NET Framework supports. For example, the `aspnet_isapi` ISAPI extension module that comes with the ASP.NET 1.1 Framework loads the CLR 1.1 into the worker process, whereas the `aspnet_isapi` extension module that comes with the ASP.NET Framework loads the CLR 2.0 into the worker process.

What this amounts to is that the first application in an application pool that receives the first ASP.NET request of the application pool gets to load the version of the CLR that the application needs. For example, if the first request of the application pool is for an ASP.NET 1.1 Web application, the CLR 1.1 gets loaded into the worker process, which means that all applications in the application pool now have to use the CLR 1.1 because the same worker process cannot contain two different versions of the CLR.

This is not a problem if you make sure that all the applications that you add to a given application pool use the same version of the CLR. This was a big source of confusion and errors in IIS 6.0. IIS 7 has fixed this problem by forcing you to set the CLR version at the application pool level. This means that if you attempt to add an application that needs the CLR 1.1 to an application pool that is configured to use the CLR 2.0, IIS 7 will not let you do that. Therefore if you're moving from IIS 6.0 to IIS 7, this is one of changes that you should expect.

Listing 2-5 presents an excerpt from the `applicationHost.config` file, showing the `<applicationPools>` section.

Listing 2-5: The <applicationPools> Section

```
<system.applicationHost>
  <applicationPools>

    <add name="DefaultAppPool" managedPipelineMode="Integrated">
      <processModel identityType="LocalSystem" />
    </add>
```

(Continued)

Listing 2-5: (continued)

```
<add name="AppPool_Medium" managedPipelineMode="ISAPI">
  <processModel identityType="SpecificUser" userName="IWAM_SK-A82E44308384"
    password="[enc:RsaProtectedConfigurationProvider:
      jAAAAAECAAADZgAAAKQAAAN2te8rHH1B5rQYFcJZ+2kihB2XmbqC/
      HSNkIfDpw5HVIMk8afZO8K47U78sgo4aBq1qWKwL27CUVhIsR+
      ZdfdTv5Lup1gFkimoah3PY6XLxx823BrKVtJziwpzodYz0nJSedVqV
      qcwiLkzdBexVrlbgI07Z3lNZvJKi+73vH1uula/ew8Ui6o+5Mbn0TLH7
      VMCC19gVrrvNhX2zkROOKkUO1Rvn5fxtw==:enc]" />
</add>

<add name="AppPool_Low" managedPipelineMode="ISAPI">
  <processModel identityType="LocalSystem" />
</add>

<add name="Classic .NET AppPool" managedPipelineMode="ISAPI" />

<applicationPoolDefaults>
  <processModel identityType="NetworkService" />
</applicationPoolDefaults>

</applicationPools>
</system.applicationHost>
```

As this excerpt shows, `<applicationPools>` consists of one or more `<add>` elements and a single `<applicationPoolDefaults>` element. Each `<add>` element adds a new application pool. The `<add>` element features a bunch of attributes and child elements. I discuss some of these attributes in this section and leave the discussion of some other attributes and the child elements to the following sections. One of these attributes is an attribute named `name` that specifies the name of the application pool. As you'll see later, this name will appear in the list of application pools displayed in the IIS 7 Manager dialog. Another attribute is a Boolean attribute named `autoStart`. If this attribute is set to `true`, the application pool is automatically started when it's created, or when IIS starts. Another attribute is the `managedRuntimeVersion`. I mentioned previously that you must specify the CLR version at application pool level. Therefore it shouldn't come as a surprise that the `<add>` element that adds the application pool exposes the `managedRuntimeVersion` attribute. The default is `v2.0`.

The managedPipelineMode Attribute

As discussed in the previous chapter, IIS 6.0 suffers from a fundamental architectural problem, that is, each ASP.NET request is handled by two different request processing pipelines: IIS and ASP.NET. IIS 7 addresses this problem by integrating these two pipelines into a single unified request processing pipeline. Such a fundamental pipeline change could cause problems for some ASP.NET applications that are moving from IIS 6.0 to IIS 7 because they may depend on some features of IIS 6.0 that require two separate IIS and ASP.NET pipelines.

To address the compatibility issues of these legacy ASP.NET applications, you can configure IIS 7 to run in ISAPI mode. In this mode, IIS 7 hands the request over to the `aspnet_isapi` extension module, where a separate ASP.NET pipeline is used to process the request. In other words, IIS 7 can run in two modes: ISAPI and integrated. When IIS 7 is running in ISAPI mode it operates pretty much like IIS 6.0.

It is highly recommended that you make the required changes in your legacy ASP.NET applications to make them run in IIS 7 integrated mode. You should use the ISAPI mode as the last resort.

Just like the CLR version discussed earlier, you have to set the pipeline mode at the application pool level. In other words, all applications running in the same application pool use the same pipeline mode. This allows you to add a new application pool to IIS 7, set its mode to ISAPI, and add your legacy ASP.NET applications to this application pool. Thanks to the process isolation of application pools, you can have multiple application pools running in different pipeline modes on the same Web server. When you make the required code changes in one of your legacy ASP.NET applications to make it work with the new integrated mode, you can simply move the application from the current application pool to the application pool that is running in integrated mode.

Now back to the `managedPipelineMode` attribute of the `<add>` element that adds a new application pool to IIS 7. As you may have already guessed, you can use this attribute to specify the pipeline mode in which you want the application pool to run. The possible values of this attribute are `ISAPI` and `Integrated`. The default is `Integrated`.

The `queueLength` Attribute

Every application pool has a dedicated request queue where the protocol listener queues the incoming requests. This queue plays several important roles. For example, it increases the reliability of the applications in the application pool. Suppose the worker process serving an application pool suddenly dies. Between the time the old process dies and the time when WAS spawns a new worker process, the queue keeps queuing up the incoming requests, which means that the end users may experience some delay, but their requests will not be rejected and will be eventually processed.

The `queueLength` attribute of the `<add>` element allows you to specify the maximum number of requests that can be queued in the queue before IIS 7 starts rejecting requests. The default is 1,000.

The `<processModel>` Child Element

The `<processModel>` element is the child element of the `<add>` element that adds the new application pool. The main responsibility of the `<processModel>` element is to configure the worker processes responsible for processing the requests for the application pool. One of these configuration settings involves specifying the identity of the worker processes. What is a process identity, anyway? Why does it matter?

Every user-mode process must run under a specific Windows account. This account constitutes the identity of the process. Windows uses this account to determine which resources the process can access. For example, the first time an ASP.NET page is accessed, the worker process needs to perform the following tasks:

- ❑ Read the associated `.aspx` and `.aspx.cs` files, which means that the worker process needs read access to these files.
- ❑ Generate the source code for the class that represents the ASP.NET page and save this source code into a file in a directory under the ASP.NET Temporary Files folder, which means that the worker process needs write access to this folder.
- ❑ Compile this source code into an assembly and save this assembly in a file in a directory under the ASP.NET Temporary Files folder. Again this requires the worker process to have write access to this folder.

Chapter 2: Using the Integrated Configuration System

Therefore, the worker process needs read, write, and read/write access to certain files and folders on the file system, which means that the Windows account under which the worker process runs must have these required permissions. The `<processModel>` element exposes an enumeration attribute named `identityType` with the possible values of `LocalSystem`, `LocalService`, `NetworkService`, and `SpecificUser`. The default is `NetworkService`, which means that by default the worker process runs under the built-in Network Service account. If this attribute is set to `LocalSystem` or `LocalService`, the worker process will run under the built-in Local System or Local Service account. Keep in mind that the Local System has a higher privileges than the Network Service and Local Service, which means that it introduces a serious security risk.

If you don't want to run the worker process under any of these accounts, you can set the following attributes to have the worker process run under a custom account:

- ❑ The `userName` attribute of the `<processModel>` must be set to the custom account name.
- ❑ The `password` attribute of the `<processModel>` must be set to the password of the Windows account.
- ❑ The `identityType` attribute must be set to `SpecificUser` to indicate that none of the built-in Windows accounts is being used.

Keep in mind that the custom account must provide the worker process with the minimum privileges that it needs to do its job.

The following table describes some other important attributes of the `<processModel>` element:

Attribute	Description
<code>idleTimeout</code>	Specifies the period of inactivity (in hh:mm:ss format) after which the worker process is automatically shut down. The default is 00:20:00.
<code>maxProcesses</code>	Specifies the maximum number of worker processes responsible for processing the requests for the application pool. The default is 1. A Web garden is an application pool whose <code>maxProcesses</code> attribute is set to a value greater than 1.
<code>shutdownTimeLimit</code>	WAS periodically recycles worker processes. When the time comes to recycle a worker process, WAS waits for the amount of time (in hh:mm:ss format) specified by the <code>shutdownTimeLimit</code> attribute before it terminates the worker process. This gives the process time to wrap up the requests it's currently processing. The default is 00:01:30.
<code>startupTimeLimit</code>	Specifies the amount of time (in hh:mm:ss format) that WAS waits for the worker process to start up. If the process doesn't start up within this time frame, WAS terminates the process. The default is 00:01:30.
<code>pingingEnabled</code>	The Boolean value that specifies whether WAS should periodically ping the worker process to monitor its health.

Attribute	Description
pingInterval	Specifies the frequency (in hh:mm:ss format) at which WAS pings the worker process. The default is 00:00:30.
pingResponseTime	Specifies how long (in hh:mm:ss format) WAS should wait for a response to a ping request. If the worker process does not respond within this time frame, WAS terminates the process and starts a new worker process. The default is 00:01:30.

The <recycling> Child Element

The <add> element contains a child element named <recycling>, which can be used to configure process recycling for the application pool. This child element features three attributes:

- ❑ **disallowOverlappingRotation**: As discussed earlier, a worker process can be shut down for a number of reasons. When this happens, a new worker process is created to replace the old one. If the `disallowOverlappingRotation` Boolean attribute is set to `false`, the new worker process is created while the old worker process is being shut down. If the worker process loads application code that does not allow simultaneous execution of multiple worker process instances, you must set this attribute to `true` to ensure that the new worker process is not created until the old worker process completely shuts down. By default, this attribute value is set to “false.”
- ❑ **disallowRotationOnConfigChange**: This Boolean attribute specifies whether WAS should rotate the worker processes in the application pool when the configuration changes. By default, this attribute value is set to “false.”
- ❑ **logEventOnRecycle**: This attribute tells IIS to log an event when the application pool is recycled. The value of this attribute is a bitwise-or’ed combination of the following enumeration values: `Time`, `Requests`, `Schedule`, `Memory`, `IsapiUnhealthy`, `OnDemand`, `ConfigChange`, and `PrivateMemory`. Each value indicates the reason why the application pool was recycled:
 - ❑ **Time**: The application pool was recycled because it was time to recycle it.
 - ❑ **Requests**: The application pool was recycled because it has already processed the maximum allowable number of requests.
 - ❑ **Schedule**: The application pool was recycled because it was scheduled to be recycled.
 - ❑ **Memory**: The application pool was recycled because it was consuming more memory than the maximum allowable memory (in megabytes).
 - ❑ **IsapiUnhealthy**: The application pool was recycled because the ISAPI extension module was misbehaving. This applies when IIS 7 is running in ISAPI mode.
 - ❑ **OnDemand**: The application pool was recycled because the administrator demanded it.
 - ❑ **ConfigChange**: The application pool was recycled because the configuration changed.
 - ❑ **PrivateMemory**: The application pool was recycled because its private memory consumptions exceeded the maximum allowable value.

Chapter 2: Using the Integrated Configuration System

The `<recycling>` element has a child element named `<periodicRestart>` that can be used to set the values that IIS uses to determine whether there's a good reason to recycle an application pool as just discussed. This child element has four attributes:

- ❑ `memory`: Specifies the maximum allowable memory consumption (in megabytes). As discussed, if the application pool consumes more memory than the value specified in this attribute, WAS will recycle the application pool.
- ❑ `privateMemory`: Specifies the maximum allowable private memory consumption (in megabytes).
- ❑ `requests`: Specifies the maximum number of requests the application pool can process.
- ❑ `time`: Specifies how often WAS should recycle the application pool.

The `<periodicRestart>` element contains a single child element named `<schedule>` that can be used to schedule the recycling of the application pool. This child element can contain one or more `<add>` elements. Each `<add>` element features an attribute that specifies a scheduled recycling time. This allows you to specify the exact times when the application pool should be recycled.

The `<cpu>` Child Element

The `<add>` element contains a child element named `<cpu>` that can be used to specify the CPU settings and actions for the application pool. This child element features the following five attributes:

- ❑ `limit`: Specifies the maximum consumption of CPU percentage (in 1/1000ths of a percentage) of the worker processes in the application pool within the time specified by the `resetInterval` attribute. If the CPU consumption exceeds this value, the WAS will recycle the application pool.
- ❑ `action`: Specifies the action that IIS must take when the CPU consumption exceeds the value specified by the `limit` attribute. The possible values are `NoAction` and `KillW3wp`. The `NoAction` value prevents IIS from taking any action other than logging a warning message. The `KillW3wp` value instructs IIS to recycle the application pool and its worker processes.
- ❑ `resetInterval`: Refer to the `limit` attribute for the description of this attribute.
- ❑ `smpAffinitized`: Turns the process affinity feature on or off.
- ❑ `smpProcessorAffinityMask`: A hexadecimal bitmask that determines which processors are eligible for running worker processes for this application pool. This attribute is applicable in Web garden scenarios. A Web garden is an application pool that has more than one associated worker process and runs on a multiprocessor machine.

`<applicationPoolDefaults>`

As discussed earlier, the `<applicationPools>` element contains one or more `<add>` child elements where each `<add>` element adds a new application pool to IIS 7. The attributes and the child elements of the `<add>` element specify the configuration settings for the new application pool. If the attributes and child elements of a given `<add>` element are not specified, the application pool will inherit the configuration settings specified in the `<applicationPoolDefaults>` element.

<sites>

Before diving into the details of the <sites> section, you need to understand the difference between these three concepts: *site*, *application*, and *virtual directory*. A site is a collection of one or more applications. Every site has a unique name and a unique id:

```
<sites>
  <site name="Default Web Site" id="1">
    . . .
  </site>
  . . .
</sites>
```

Every site has at least one application known as the *root application*, identified by the "/" virtual path:

```
<sites>
  <site name="Default Web Site" id="1">
    <application path="/">
      . . .
    </application>
    . . .
  </site>
  <site name="MySite2" id="2">
    <application path="/">
      . . .
    </application>
    . . .
  </site>
  . . .
</sites>
```

A site can have more than one *application*. Each application is uniquely identified by its virtual path. No two applications in the same site can have the same virtual path. Every application belongs to one and only one application pool:

```
<sites>
  <site name="Default Web Site" id="1">
    <application path="/" applicationPool="DefaultAppPool">
      . . .
    </application>
    . . .
  </site>
  . . .
</sites>
```

Applications belonging to the same site may have a parent/child relationship based on their virtual paths. For example, in the following listing, the application with virtual path "/" (the root application) is the parent of the application with the virtual path /MyApp1, which in turn is the parent of the application with the virtual path /MyApp1/MyApp2:

```
<sites>
  <site name="Default Web Site" id="1">
    <application path="/" applicationPool="DefaultAppPool">
```

Chapter 2: Using the Integrated Configuration System

```
    . . .
</application>
<application path="/MyApp1" applicationPool="MyPool1">
    . . .
</application>
<application path="/MyApp1/MyApp2" applicationPool="MyPool2">
    . . .
</application>
    . . .
</site>
    . . .
</sites>
```

In a practical sense, this parent/child relationship means that the child applications inherit configuration settings from their parent applications.

Every application must have at least one virtual directory with the "/" virtual path:

```
<sites>
  <site name="Default Web Site" id="1">
    <application path="/" applicationPool="DefaultAppPool">
      <virtualDirectory path="/" physicalPath="c:\inetpub\wwwroot\MyDir1" />
      . . .
    </application>
    <application path="/MyApp1" applicationPool="MyPool1">
      <virtualDirectory path="/" physicalPath="D:\MyDir1" />
      . . .
    </application>
    . . .
  </site>
  . . .
</sites>
```

Notice that every virtual directory has an optional `physicalPath` attribute that specifies the file system path that the virtual path maps to. Two different virtual directories of the same application can be mapped to the same physical path:

```
<sites>
  <site name="Default Web Site" id="1">
    <application path="/" applicationPool="DefaultAppPool">
      <virtualDirectory path="/" physicalPath="D:\MyDir1" />
      <virtualDirectory path="/MyVirDir1" physicalPath="D:\MyDir1" />
      . . .
    </application>
    . . .
  </site>
  . . .
</sites>
```

The virtual path of a virtual directory is relative to its containing application. The `<virtualDirectory>` element exposes two attributes named `userName` and `password` that you can use to limit access to the virtual directory to the specified credentials. Only users with the specified username and password will be allowed to access the directory.

As these discussions show, you have to specify a bunch of settings when you add a new site, application, or virtual directory. What if you add a site, application, or virtual directory, but you don't specify some of these settings? This is where the `<siteDefaults>`, `<applicationDefaults>`, and `<virtualDirectoryDefaults>` elements come into play, which respectively specify the default settings for sites, applications, and virtual directories.

`<system.webServer>`

The `<system.webServer>` section group contains all configuration sections that specify IIS Web server configuration settings. Contrary to the `<system.applicationHost>` section group and its sections, the `<system.webServer>` section group and its sections can be used in all lower-level configuration files if they're not locked in the higher-level configuration files. Note that for security reasons, upon installation these sections are locked by default; that is, only the machine administrator can change the IIS Web server configurations settings. However, the machine administrator may choose to unlock certain sections to allow the site and application administrators and developers to custom configure the Web server for their own sites and applications.

This section group contains these important sections: `<defaultDocument>`, `<directoryBrowse>`, `<globalModules>`, `<handlers>`, and `<security>`.

`<defaultDocument>`

The `<defaultDocument>` section specifies a list of files to be served if the request URL does not contain the name of the requested resource. This section features an attribute named `enabled` that specifies whether the default document functionality is enabled, and a single child element named `<files>`, which contains one or more `<add>` child elements. Each `<add>` element in this case specifies a document to be served by default. Here is an excerpt from `applicationHost.config`:

```
<configuration>
  <system.webServer>
    <defaultDocument enabled="true">
      <files>
        <add value="Default.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="index.html" />
        <add value="iisstart.htm" />
        <add value="default.aspx" />
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>
```

As you'd expect, all lower-level configuration files inherit these settings. If you want to remove one of these setting in a site, application, or virtual directory, add a `web.config` file to the site, application, or virtual directory and use the `<remove>` element to remove the specified document:

```
<configuration>
  <system.webServer>
    <defaultDocument enabled="true">
      <files>
```



```
<remove value="Default.htm" />
</files>
</defaultDocument>
</system.webServer>
</configuration>
```

If you want to remove all the default documents specified in the higher-level configuration files, use the `<clear>` element. Then use the `<add>` element to add a new default document.

<directoryBrowse>

The `<directoryBrowse>` section specifies whether the end user can see the contents of the current directory. The `<directoryBrowse>` section has two attributes named `enabled` and `showFlags`. The `enabled` Boolean attribute specifies whether the directory listing functionality is enabled. The `showFlags` attribute is a bitwise-or'ed combination of the following enumeration values: `None`, `Date`, `Time`, `Size`, `Extension`, and `LongDate`. By default, the directory listing functionality is disabled:

```
<directoryBrowse enabled="false" />
```

<globalModules>

As discussed in the previous chapter, IIS 7 has replaced the IIS 6.0 ISAPI extensibility API with two new sets of extensibility APIs: native and managed. The native API is an object-oriented C++ API that allows you to use native C++ code to implement custom feature modules that can be plugged into IIS 7 to extend the functionality of the core Web server. The managed API is an object-oriented API that allows you to use a .NET-compliant language such as C# or VB.NET to implement custom feature modules.

One of the main differences between native and managed modules is that you have to install your custom native module on IIS 7 before it can be used, whereas managed modules don't need installation. The installation basically adds the native module to the `<globalModules>` section. This section contains one or more `<add>` child elements, each of which installs a particular native module. Listing 2-6 is an excerpt from `applicationHost.config`, showing the `<globalModules>` section.

Listing 2-6: The Installed Native Modules

```
<configuration>
  <system.webServer>
    <globalModules>
      <add name="TracingModule"
        image="C:\Windows\system32\inetsrv\iisetw.dll" />
      <add name="HttpCacheModule"
        image="C:\Windows\system32\inetsrv\cachhttp.dll" />
      <add name="StaticCompressionModule"
        image="C:\Windows\system32\inetsrv\compstat.dll" />
      <add name="DefaultDocumentModule"
        image="C:\Windows\system32\inetsrv\defdoc.dll" />
      <add name="DirectoryListingModule"
        image="C:\Windows\system32\inetsrv\dirlist.dll" />
      <add name="HttpRedirectionModule"
        image="C:\Windows\system32\inetsrv\Redirect.dll" />
      <add name="StaticFileModule"
        image="C:\Windows\system32\inetsrv\static.dll" />
```

Listing 2-6: (continued)

```
<add name="AnonymousAuthenticationModule"
  image="C:\Windows\system32\inetsrv\authanon.dll" />
<add name="UrlAuthorizationModule"
  image="C:\Windows\system32\inetsrv\urlauthz.dll" />
<add name="BasicAuthenticationModule"
  image="C:\Windows\system32\inetsrv\authbas.dll" />
<add name="WindowsAuthenticationModule"
  image="C:\Windows\system32\inetsrv\authsspi.dll" />
<add name="DigestAuthenticationModule"
  image="C:\Windows\system32\inetsrv\authmd5.dll" />
<add name="IsapiModule" image="C:\Windows\system32\inetsrv\isapi.dll" />
<add name="IsapiFilterModule"
  image="C:\Windows\system32\inetsrv\filter.dll" />
<add name="ManagedEngine"
  image="C:\Windows\Microsoft.NET\Framework\v2.0.50727\webengine.dll"
  preCondition="integratedMode, runtimeVersionv2.0, bitness32" />
<add name="DynamicCompressionModule"
  image="C:\Windows\system32\inetsrv\compdyn.dll" />
</globalModules>
</system.webServer>
</configuration>
```

Notice that the `<add>` element has two important attributes named `name` and `image`, which respectively specify the module name and the physical path to the DLL that contains the module. Listing 2-6 clearly shows how modular the IIS 7 architecture is. Every feature is encapsulated in a module, allowing you to decide which feature or modules to install. For example, if you don't need support for Digest authentication, don't install the `DigestAuthenticationModule`.

Notice that Listing 2-6 installs a module named `ManagedEngine`:

```
<configuration>
  <system.webServer>
    <globalModules>
      . . .
      <add name="ManagedEngine"
        image="C:\Windows\Microsoft.NET\Framework\v2.0.50727\webengine.dll"
        preCondition="integratedMode, runtimeVersionv2.0, bitness32" />
      . . .
    </globalModules>
  </system.webServer>
</configuration>
```

The `ManagedEngine` module is the magic behind the IIS 7 and ASP.NET integrated pipeline. Every operating system process that needs to execute managed code requires a layer of code that uses the CLR hosting API to load the CLR into the process and allow the unmanaged and managed code to communicate. In IIS 6.0, the `aspnet_isapi.dll` ISAPI extension module used this layer of code to host the CLR in the `w3wp.exe` worker process. In IIS 7, the `webengine.dll` uses this layer of code to integrate managed modules into the IIS 7 request processing pipeline.

Chapter 2: Using the Integrated Configuration System

Also notice that Listing 2-6 installs two modules named `IsapiModule` and `IsapiFilterModule`:

```
<configuration>
  <system.webServer>
    <globalModules>
      . . .
      <add name="IsapiModule" image="C:\Windows\system32\inetsrv\isapi.dll" />
      <add name="IsapiFilterModule"
        image="C:\Windows\system32\inetsrv\filter.dll" />
      . . .
    </globalModules>
  </system.webServer>
</configuration>
```

These two modules are the magic behind the IIS 7 ISAPI mode. As discussed earlier, some legacy applications that are moving from IIS 6.0 to IIS 7 may have compatibility issues with the new IIS 7 integrated mode. These applications can configure IIS 7 to run in ISAPI mode, where IIS 7 acts pretty much like IIS 6.0, that is, it passes the request to the appropriate extension module for processing. The `IsapiModule` and `IsapiFilterModule` native modules allow IIS 7 to interact with ISAPI extension and filter modules such as `aspnet_isapi.dll` and `aspnet_filter.dll`. As you can see, the modular architecture of IIS 7 allows you to plug these two modules into the core Web server to make it work like IIS 6.0.

<handlers>

A *handler* is a component that is responsible for handling or processing requests for resources with particular file extensions. In earlier versions of IIS, upon installation, ASP.NET automatically registered the `aspnet_isapi.dll` ISAPI extension module with the IIS metabase as the handler for requests for resources with the ASP.NET-specific file extensions such as `.aspx`, `.asmx`, and `.ashx`. When a request arrives, the IIS handler mapping component examines the file extension of the requested resource and passes the request on to the `aspnet_isapi.dll` ISAPI extension module handler if the file extension is one of the ASP.NET-specific file extensions.

As discussed in the previous chapter, IIS 7 has replaced the metabase with a brand new configuration system, which is very similar to the .NET configuration system. Handler registration is now done in the `<handlers>` section of the configuration file. This rule applies to both native and managed handlers. As you'll see later, IIS 7 allows you to write handlers in a .NET-compliant language such as C#, where you can take full advantage of the rich .NET Framework environment and classes. Regardless of whether you write your handler in native or managed code, you must register it in the new `<handlers>` section. In other words, the `<handlers>` section replaces the `<httpHandlers>` section of the ASP.NET configuration system. As a matter of fact, when you're moving your existing ASP.NET applications from IIS 6.0 to IIS 7, you must move the contents of the `<httpHandlers>` section of your configuration files into the `<handlers>` section. If you do not, the IIS 7 integrated pipeline will not pick up your registered handlers.

Listing 2-7 presents an excerpt from `applicationHost.config`, showing the `<handlers>` section.

Listing 2-7: The <handlers> Section

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
```

Listing 2-7: (continued)

```
<handlers>
  <add name="ASPClassic" path="*.asp" verb="GET,HEAD,POST"
    modules="IsapiModule" resourceType="File"
    scriptProcessor="C:\Windows\system32\inetsrv\asp.dll" />

  <add name="PageHandlerFactory-ISAPI-2.0" path="*.aspx" verb="*"
    modules="IsapiModule"
    precondition="ISAPIMode, runtimeVersionv2.0, bitness32"
    scriptProcessor="C:\Windows\Microsoft.NET\Framework\
      v2.0.50727\aspnet_isapi.dll" />
  <add name="PageHandlerFactory-ISAPI-1.1" path="*.aspx" verb="*"
    modules="IsapiModule"
    scriptProcessor="C:\Windows\Microsoft.Net\Framework\
      v1.1.4322\aspnet_isapi.dll"
    precondition="ISAPIMode, runtimeVersionv1.1, bitness32" />
  <add name="PageHandlerFactory-Integrated" path="*.aspx" verb="*"
    type="System.Web.UI.PageHandlerFactory" precondition="integratedMode" />

  <add name="WebServiceHandlerFactory-ISAPI-2.0" path="*.asmx" verb="*"
    modules="IsapiModule"
    scriptProcessor="C:\Windows\Microsoft.NET\Framework\
      v2.0.50727\aspnet_isapi.dll"
    precondition="ISAPIMode, runtimeVersionv2.0, bitness32" />
  <add name="WebServiceHandlerFactory-ISAPI-1.1" path="*.asmx" verb="*"
    modules="IsapiModule"
    scriptProcessor="C:\Windows\Microsoft.Net\Framework\
      v1.1.4322\aspnet_isapi.dll"
    precondition="ISAPIMode, runtimeVersionv1.1, bitness32" />
  <add name="WebServiceHandlerFactory-Integrated" path="*.asmx" verb="*"
    type="System.Web.Services.Protocols.WebServiceHandlerFactory,
      System.Web.Services, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a"
    precondition="integratedMode" />

  <add name="svc-ISAPI-2.0" path="*.svc" verb="*" modules="IsapiModule"
    scriptProcessor="C:\Windows\Microsoft.NET\Framework\
      v2.0.50727\aspnet_isapi.dll"
    precondition="ISAPIMode, runtimeVersionv2.0, bitness32" />
  <add name="svc-Integrated" path="*.svc" verb="*"
    type="System.ServiceModel.Activation.HttpHandler, System.ServiceModel,
      Version=3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    precondition="integratedMode" />
</handlers>
</system.webServer>
</location>
</configuration>
```

Notice that `applicationHost.config` file uses a `<location>` tag with the path value of `"` to specify that all the handlers in this section are applicable to all applications running on the machine. Also notice that the `overrideMode` attribute of the `<location>` tag is set to `Allow` to allow the lower-level configuration files to remove or replace the handlers defined in `applicationHost.config` or add their own

Chapter 2: Using the Integrated Configuration System

handlers. For example, to register a custom handler for your application, you can add a `web.config` file to the root directory of your application and add the following section to it:

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandlerName" path="*.MyFileExtension" verb="*"
          type="MyNamespace.MyHandler" preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

Note that the `MyHandler` handler is added to the `<handlers>` subsection of the `<system.webServer>` as opposed to the `<httpHandlers>` subsection of the `<system.web>`. As I mentioned, the `<handlers>` section in IIS 7 replaces the `<httpHandlers>` section.

The `<handlers>` section contains one or more `<add>` child elements that are each used to register a particular handler. The `<add>` element exposes the following attributes:

- ❑ **name:** Set this attribute to the friendly name of your handler. You can use any string value as the friendly name as long as it's unique. The friendly name is normally used to reference the handler.
- ❑ **path:** Set this attribute to the comma-separated list of file extensions that your handler supports.
- ❑ **verb:** Set this attribute to the comma-separated list of HTTP verbs that your handler supports. The value of `*` indicates that the handler supports all HTTP verbs.
- ❑ **type:** Set this attribute to a string that contains a comma-separated list of up to five substrings. Only the first substring is required, and must contain the fully qualified name of your handler including its complete namespace containment hierarchy. The remaining substrings must specify the assembly that contains your handler. The `type` attribute is only applicable to managed handlers, that is, handlers written in managed code such as C# or Visual Basic.
- ❑ **scriptProcessor:** Set this attribute to the physical path to your handler's dynamic link library (DLL). This attribute is only applicable to native handlers. In other words, you have to use the `scriptProcessor` attribute instead of the `type` attribute if you're registering a native handler.
- ❑ **preCondition:** Set this attribute to `IntegratedMode` to indicate that your handler should be used only when IIS 7 is running in the integrated mode, or to `ISAPIMode` to indicate that your handler should be used only when IIS 7 is running in the ISAPI mode.

Next, I review the handlers registered in Listing 2-7 to help you get a better feel for the handlers and their relationship to the mode in which IIS 7 is running.

Classic ASP

The following listing shows the portion of Listing 2-7 that registers the `asp.dll` ISAPI extension module as the handler for processing requests for resources with the file extension `.asp`, that is, classic ASP pages. Because `asp.dll` is a native handler, the `scriptProcessor` attribute is used to specify the physical path to the `asp.dll` file.

```
<configuration>
  <location path="" overrideMode="Allow">
```

```
<system.webServer>
  <handlers>
    <add name="ASPCClassic" path="*.asp" verb="GET,HEAD,POST"
        modules="IsapiModule" resourceType="File"
        scriptProcessor="C:\Windows\system32\inetsrv\asp.dll" />
    . . .
  </handlers>
</system.webServer>
</location>
</configuration>
```

ASP.NET Pages

As the following excerpt from Listing 2-7 shows, the `applicationHost.config` file registers three handlers for processing requests for resources with the file extension `.aspx`, that is, ASP.NET pages.

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <handlers>
        . . .
        <add name="PageHandlerFactory-ISAPI-2.0" path="*.aspx" verb="*"
            modules="IsapiModule"
            preCondition="ISAPIMode, runtimeVersionv2.0, bitness32"
            scriptProcessor="C:\Windows\Microsoft.NET\Framework\
                           v2.0.50727\aspnet_isapi.dll" />

        <add name="PageHandlerFactory-ISAPI-1.1" path="*.aspx" verb="*"
            modules="IsapiModule"
            scriptProcessor="C:\Windows\Microsoft.Net\Framework\
                           v1.1.4322\aspnet_isapi.dll"
            preCondition="ISAPIMode, runtimeVersionv1.1, bitness32" />

        <add name="PageHandlerFactory-Integrated" path="*.aspx" verb="*"
            type="System.Web.UI.PageHandlerFactory" preCondition="integratedMode" />
        . . .
      </handlers>
    </system.webServer>
  </location>
</configuration>
```

The `preCondition` attribute of the `<add>` element that registers the first handler is set to `ISAPIMode` to inform IIS 7 that this handler must be called only when the corresponding application pool is running in ISAPI mode. The second handler is the ASP.NET 1.1 version of the first handler. This handler is called when the corresponding application pool is configured to use ASP.NET 1.1. Because both of these handlers are native handlers, the `scriptProcessor` attributes of their associated `<add>` elements are set to the physical path to the associated `aspnet_isapi.dll` file.

The third handler, on the other hand, is a managed handler. Notice that this handler is nothing but the `PageHandlerFactory` class. The `preCondition` attribute is set to `IntegratedMode` to tell IIS 7 that this handler should be invoked only when the corresponding application pool is running in integrated mode.

Windows Communications Foundation (WCF)

Here comes the interesting part. As the following excerpt from Listing 2-7 shows, the `applicationHost.config` file registers two handlers for processing requests for resources with the file extension `.svc`. These resources are known as Windows Communications Foundation (WCF) services. Thanks to the extensibility of the IIS 7 architecture, you can plug these handlers into the core Web server to enable your Web server to support WCF applications.

Notice that the first handler is the same native handler that handles the ASP.NET pages when IIS 7 is running in ISAPI mode. In other words, in ISAPI mode, IIS 7 treats both ASP.NET and WCF applications the same. The second handler, on the other hand, is a managed handler named `HttpHandler`.

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <handlers>
        . . .
        <add name="svc-ISAPI-2.0" path="*.svc" verb="*" modules="IsapiModule"
          scriptProcessor="C:\Windows\Microsoft.NET\Framework\
            v2.0.50727\aspnet_isapi.dll"
          precondition="ISAPIMode, runtimeVersionv2.0, bitness32" />

        <add name="svc-Integrated" path="*.svc" verb="*"
          type="System.ServiceModel.Activation.HttpHandler, System.ServiceModel,
            Version=3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
          precondition="integratedMode" />
      </handlers>
    </system.webServer>
  </location>
</configuration>
```

<modules>

The `<modules>` section is where both native and managed modules are registered. Whereas managed modules do not require installation, native modules have to be installed before they can be registered and added to the `<modules>` section. As discussed earlier, to install a native module, you have to add the module to the `<globalModules>` section.

The `<modules>` section contains one or more `<add>` child elements, each of which registers a particular native or managed module. The `<add>` element exposes the following three string attributes:

- ❑ **name:** This attribute specifies the friendly name of the module. If the module being registered is a native module, the value of the `name` attribute must match the value of the `name` attribute of the `<add>` element that adds the module to the `<globalModules>` section. If you're registering a managed custom module, you can choose any friendly name for the module as long as it's unique.
- ❑ **type:** The value of this attribute is a string that consists of a comma-separated list of up to five substrings. Only the first substring is required, and it specifies the fully qualified name of the module class including its complete namespace containment hierarchy. The remaining optional substrings specify the assembly that contains the module. The `type` attribute is only applicable to managed modules.
- ❑ **preCondition:** This attribute specifies whether the module being registered should be called when the corresponding application pool is running in integrated mode or ISAPI mode.

In the IIS 7 and ASP.NET unified configuration system, the `<modules>` subsection of the `<system.webServer>` section replaces the `<httpModules>` subsection of the `<system.web>` section. When you're moving your existing ASP.NET applications from IIS 6.0 to IIS 7 you must move the contents of the `<httpModules>` section to the `<modules>` section.

Listing 2-8 presents an excerpt from the `applicationHost.config` file that shows the contents of the `<modules>` section.

Listing 2-8: The `<modules>` Section of the `applicationHost.config` File

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <modules>
        <add name="HttpCacheModule" />
        <add name="StaticCompressionModule" />
        <add name="DefaultDocumentModule" />
        <add name="DirectoryListingModule" />
        <add name="HttpRedirectionModule" />
        <add name="StaticFileModule" />
        <add name="AnonymousAuthenticationModule" />
        <add name="UrlAuthorizationModule" />
        <add name="IsapiFilterModule" />
        <add name="BasicAuthenticationModule" />
        <add name="WindowsAuthenticationModule" />
        <add name="DigestAuthenticationModule" />
        <add name="IsapiModule" />
        . . .

        <add name="ServiceModel"
          type="System.ServiceModel.Activation.HttpModule, System.ServiceModel,
            Version=3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
          precondition="managedHandler" />
        <add name="OutputCache" type="System.Web.Caching.OutputCacheModule"
          precondition="managedHandler" />
        <add name="Session" type="System.Web.SessionState.SessionStateModule"
          precondition="managedHandler" />
        <add name="WindowsAuthentication"
          type="System.Web.Security.WindowsAuthenticationModule"
          precondition="managedHandler" />
        <add name="FormsAuthentication"
          type="System.Web.Security.FormsAuthenticationModule"
          precondition="managedHandler" />
        <add name="DefaultAuthentication"
          type="System.Web.Security.DefaultAuthenticationModule"
          precondition="managedHandler" />
        <add name="RoleManager" type="System.Web.Security.RoleManagerModule"
          precondition="managedHandler" />
        <add name="UrlAuthorization"
          type="System.Web.Security.UrlAuthorizationModule"
          precondition="managedHandler" />
        <add name="FileAuthorization"
```

(Continued)

Listing 2-8: (continued)

```
        type="System.Web.Security.FileAuthorizationModule"
        preCondition="managedHandler" />
    <add name="AnonymousIdentification"
        type="System.Web.Security.AnonymousIdentificationModule"
        preCondition="managedHandler" />
    <add name="Profile" type="System.Web.Profile.ProfileModule"
        preCondition="managedHandler" />
    <add name="UrlMappingsModule" type="System.Web.UrlMappingsModule"
        preCondition="managedHandler" />
    <add name="global.asax" type="System.Web.HttpApplication"
        preCondition="managedHandler" />
</modules>
</system.webServer>
</location>
</configuration>
```

As Listing 2-8 shows, the `<modules>` section of the `applicationHost.config` file registers two types of modules: native and managed. The `<add>` element that registers a native module needs to specify only the value of the `name` attribute, that is, the friendly name of the module. If you compare Listings 2-8 and 2-7, you'll notice that the value of the `name` attribute of each `<add>` element that registers a native module matches the value of the `name` attribute of the associated `<add>` element that installs the module. You have to follow the same rule when you're registering your own custom native modules.

Notice that the `preCondition` attribute of the `<add>` elements that register managed modules in Listing 2-8 is set to a value of `managedHandler`. This means that by default all registered managed modules will be applied only to those requests whose handlers are managed handlers, that is, requests for ASP.NET content.

Lower-level configuration files automatically inherit the modules added to the `<modules>` section of the `applicationHost.config` file. This means that these modules will be called for requests to any site, application, or virtual directory on the machine. If you don't want requests for a particular site, application, or virtual directory to be processed by one or more of these modules, you can add a `web.config` file to that site, application, or virtual directory and remove the specified modules from the `<modules>` section of your configuration file (alternatively you can use a `<location>` element in a higher-level configuration file to do this). Here is an example:

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="DefaultDocumentModule" />
      <remove name="WindowsAuthentication" />
    </modules>
  </system.webServer>
</configuration>
```

If a site, application, or virtual directory needs to replace a particular module registered in a higher-level configuration file with one of its own, it can add a new `web.config` file, remove the existing module,

and add the new module (alternatively you can do the same through a `<location>` element in a higher-level configuration file):

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="BasicAuthenticationModule" />
      <add name="MyBasicAuthenticationModule" />
    </modules>
  </system.webServer>
</configuration>
```

<security>

The `<security>` section is used to specify the Web server security configuration settings. This section contains these important child elements: `<access>`, `<authentication>`, and `<authorization>`.

<access>

The `<access>` element exposes these attributes:

- ❑ `sslFlags`: Use this attribute to configure the SSL. For example, the following listing configures 128-bit SSL security:

```
<configuration>
  <system.webServer>
    <security>
      <access sslFlags="ssl128" />
    </security>
  </system.webServer>
</configuration>
```

As this listing shows, IIS 7 has made SSL configuration a piece of cake. This is because most of the SSL configuration settings that IIS 6.0 stored in its metabase are now moved into the HTTP.SYS kernel-level device driver.

- ❑ `flags`: Use this attribute to specify the file access permissions for the current directory. The possible values are Read, Script, Source, and Write.

<authentication>

Recall that the `<globalModules>` section of the `applicationHost.config` file shown in Listing 2-6 installs the following native authentication modules:

```
<configuration>
  <system.webServer>
    <globalModules>
      . . .
      <add name="AnonymousAuthenticationModule"
        image="C:\Windows\system32\inetsrv\authanon.dll" />
      <add name="BasicAuthenticationModule"
        image="C:\Windows\system32\inetsrv\authbas.dll" />
      <add name="WindowsAuthenticationModule"
```

Chapter 2: Using the Integrated Configuration System

```
image="C:\Windows\system32\inetsrv\authsspi.dll" />
<add name="DigestAuthenticationModule"
image="C:\Windows\system32\inetsrv\authmd5.dll" />
. . .
</globalModules>
</system.webServer>
</configuration>
```

Also recall that the `<modules>` section of the `applicationHost.config` file registers these native authentication modules:

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <modules>
        <add name="AnonymousAuthenticationModule" />
        <add name="BasicAuthenticationModule" />
        <add name="WindowsAuthenticationModule" />
        <add name="DigestAuthenticationModule" />
        . . .
      </modules>
    </system.webServer>
  </location>
</configuration>
```

In other words, the `applicationHost.config` file installs and registers more than one native authentication module. Which of these authentication schemes should IIS 7 use? This is where the `<authentication>` subsection of the `<security>` section comes into play.

The `<authentication>` section is where you specify which native authentication module IIS 7 should use to authenticate requests. As the following discussion shows, this section contains one child element for each native authentication module, which exposes a Boolean attribute named `enabled` that can be set to enable or disable the associated native authentication module:

- ❑ `<anonymousAuthentication>`: By default, the `enabled` attribute of this child element is set to `true`, which means that by default the `AnonymousAuthenticationModule` is enabled for all applications running on the server. This child element features two attributes named `userName` and `password` that together specify the identity or Windows account that IIS will use when an anonymous user accesses an application. The default is a built-in account named `IUSR`, which has minimum rights and privileges. `IUSR` replaces the `IUSR_MachineName` account used in the earlier versions of IIS. The following setting tells IIS to use the built-in `IUSR` account:

```
<anonymousAuthentication enabled="true" userName="IUSR" defaultLogonDomain="" />
```

The IIS 7 anonymous authentication module supports a feature that allows you to tell IIS 7 to use the application pool or process identity. All you have to do is to set the `userName` and `password` attributes to empty strings and enable anonymous authentication as follows:

```
<anonymousAuthentication enabled="true" userName="" defaultLogonDomain="" />
```

Note that the process identity or account by default is an account named `Network Service`. However, you can change this identity in the `<processModel>` section of the application pool as discussed earlier.

- ❑ `<basicAuthentication>`: By default the `enabled` attribute of this child element is set to `false`, which means that by default the `BasicAuthenticationModule` native authentication module is not invoked for any of the sites and applications running on the server.
- ❑ `<digestAuthentication>`: By default the `enabled` attribute of this child element is set to `false`, which means that by default the `DigestAuthenticationModule` native authentication module is not invoked for any of the sites and applications running on the server.
- ❑ `<windowsAuthentication enabled>`: By default the `enabled` attribute of this child element is set to `true`, which means that by default the `WindowsAuthenticationModule` native authentication module is enabled for all the sites and applications running on the server.

If you want to enable, say, the `BasicAuthenticationModule` native authentication module for a particular site or application, add a `web.config` file to the site or application and add the following listing to this file:

```
<configuration>
  <system.webServer>
    <security>
      <authentication>
        <anonymousAuthentication enabled="false"/>
        <windowsAuthentication enabled="false"/>
        <basicAuthentication enabled="true" />
      </authentication>
    </security>
  </system.webServer>
</configuration>
```

You have to disable other native authentication modules first. Because only the `AnonymousAuthenticationModule` and `WindowsAuthenticationModule` are enabled by default, this code listing disables only these two modules.

So far, I've covered only native authentication modules. How about managed authentication modules? As Listing 2-8 shows, the `applicationHost.config` file registers the managed modules highlighted in the following code listing:

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <modules>
        <add name="WindowsAuthentication"
          type="System.Web.Security.WindowsAuthenticationModule"
          preCondition="managedHandler" />

        <add name="FormsAuthentication"
          type="System.Web.Security.FormsAuthenticationModule"
          preCondition="managedHandler" />

        <add name="DefaultAuthentication"
          type="System.Web.Security.DefaultAuthenticationModule"
          preCondition="managedHandler" />
      </modules>
    </system.webServer>
  </location>
</configuration>
```

Chapter 2: Using the Integrated Configuration System

The `applicationHost.config` file registers the `WindowsAuthentication`, `FormsAuthentication`, and `DefaultAuthentication` managed authentication modules. Notice that the `preCondition` attributes of all the managed authentication modules are set to `managedHandler`, which means that these modules are invoked only for ASP.NET requests.

If you want one or more of these managed authentication modules, say, `FormsAuthentication` to be invoked for non-ASP.NET requests, as well as ASP.NET requests to a particular site or application, add a `web.config` file to the site or application and add the following to this file:

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="FormsAuthentication" />
      <add name="FormsAuthentication"
        type="System.Web.Security.FormsAuthenticationModule" />
    </modules>
  </system.webServer>
</configuration>
```

Notice that the `<modules>` section first removes the `FormsAuthentication` module and then adds it back without the `preCondition` attribute. When this attribute is not specified for a managed module such as `FormsAuthentication`, IIS 7 invokes the module for both non-ASP.NET and ASP.NET requests.

<authorization>

As the highlighted portion of the following excerpt from Listing 2-6 shows, the `applicationHost.config` file installs a native module named `UrlAuthorizationModule`, which IIS 7 uses to authorize requests:

```
<configuration>
  <system.webServer>
    <globalModules>
      . . .
      <add name="UrlAuthorizationModule"
        image="C:\Windows\system32\inetsrv\urlauthz.dll" />
      . . .
    </globalModules>
  </system.webServer>
</configuration>
```

As the highlighted portion of the following excerpt from Listing 2-8 shows, the `applicationHost.config` file registers two URL authorization modules, that is, the `UrlAuthorizationModule` native authorization module and the `UrlAuthorization` managed authorization module:

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <modules>
        . . .
        <add name="UrlAuthorizationModule" />
        . . .
      </modules>
    </system.webServer>
  </location>
</configuration>
```

```
<add name="UrlAuthorization"
      type="System.Web.Security.UrlAuthorizationModule"
      preCondition="managedHandler" />
. . .
</modules>
</system.webServer>
</location>
</configuration>
```

Notice that the `preCondition` attribute of the `<add>` element that registers the `UrlAuthorization` managed module is set to a value of `managedHandler`, which means that IIS 7 will invoke this managed module only for requests for resources that are handled by managed handlers. If you want to enable a particular site or application to protect all resources with the `UrlAuthorization` managed module, add a `web.config` file to the site or application (if it doesn't already include this file), and add the following code listing to this file:

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="UrlAuthorization" />
      <add name="UrlAuthorization"
          type="System.Web.Security.UrlAuthorizationModule" />
    </modules>
  </system.webServer>
</configuration>
```

As the listing shows, you need to first remove the `UrlAuthorization` module and then add it again but this time without setting the `preCondition` attribute.

A URL authorization — be it managed or native — uses authorization rules specified in the configuration file to determine whether the current user is authorized to access the requested resource. The authorization rules for the `UrlAuthorization` managed module must be specified in the `<authorization>` subsection of the `<system.web>` section of the configuration file. For example, the following configuration file allows access to Shahram and denies access to anyone else:

```
<configuration>
  <system.web>
    <authorization>
      <allow users="Shahram"/>
      <deny users="*/>
    </authorization>
  </system.web>
</configuration>
```

The authorization rules for the `UrlAuthorizationModule` native module, on the other hand, must be specified in the `<authorization>` subsection of the `<security>` section of the `<system.webServer>` section group:

```
<configuration>
  <system.webServer>
    <security>
      <authorization>
```

```
<add accessType="allow" users="Shahram" />
<add accessType="deny" users="*" />
</authorization>
</security>
</system.webServer>
</configuration>
```

Summary

This chapter first discussed the new IIS 7 and ASP.NET integrated configuration system, where you learned about the hierarchical structure of the configuration files that make up this integrated system, the hierarchical relationships among the files themselves, and the notion of the declarative versus imperative schema extension. The chapter then walked you through important sections of the new IIS 7 configuration file, named `applicationHost.config`, and showed you how to override the configuration settings specified in different sections of this file in a particular site, application, or virtual directory. The next chapter will show you two different ways to interface with the new IIS7 and ASP.NET integrated configuration system.

3

Managing the Integrated Configuration System from IIS Manager and the Command Line

The previous two chapters provided in-depth coverage of the IIS7 and ASP.NET integrated configuration system. As discussed, you can use this system to manage both the Web server and ASP.NET sites and applications. This chapter shows you how to interact with this integrated system and how to extend its schema to add support for your own custom configuration sections.

Server Management

When it comes to interacting with the new configuration system, you have the following three options:

- ❑ Open and edit a configuration file such as `applicationHost.config` in your favorite text editor. This approach requires you to have a solid understanding of the XML structure of the `applicationHost.config` file as discussed in the previous chapter. This is a great option if you feel comfortable with manipulating XML elements and attributes.
- ❑ IIS7 exposes the XML contents of its configuration files via two convenient intermediary components that perform the required XML element/attribute manipulations under the hood on your behalf. This allows you to configure the server and ASP.NET with these

convenient intermediary components instead of direct manipulation of the XML elements and attributes. IIS7 comes with two intermediary components:

- ❑ Internet Information Services (IIS) Manager: Provides a rich, user-friendly GUI to manage the server and ASP.NET.
- ❑ `appcmd.exe`: Provides a convenient command-line tool to manage the server and ASP.NET.
- ❑ IIS7 exposes a managed API that you can use from your C# or Visual Basic code to programmatically manipulate the XML elements and attributes that make up the IIS7 configuration system. I cover this API in detail in the next chapter.

Internet Information Services (IIS) Manager

In this section I walk you through different features of the IIS Manager. There are two ways to launch the IIS Manager: GUI-based and command line. If you feel more comfortable with a GUI-based approach, follow these steps to launch the IIS7 Manager:

1. Launch the Control Panel.
2. Click System and Maintenance.
3. Click Administrative Tools.
4. Click the Internet Information Services (IIS) Manager.

If you feel more comfortable with command-line tools, use the following command line to launch the IIS Manager:

```
%windir%\system32\inetsrv\inetmgr.exe
```

You need administration privileges to launch the IIS Manager. If you don't log in with the built-in Administrator account, when you try to launch the IIS Manager Windows will launch a dialog. The content of this dialog depends on whether your account has administration privileges. If it does, the dialog will simply ask you to confirm the requested action. If it doesn't, the dialog will ask for the administrative credentials. This is a new Windows security feature. Figure 3-1 shows the IIS Manager.

As Figure 3-1 shows, the IIS Manager consists of three panes. The first pane, which is known as the Connections pane, contains a node that represents the Web server. This node has these two child nodes:

- ❑ Application Pools
- ❑ Sites. The label of this node is "Sites" on Windows Server 2008 and "Web Sites" on Windows Vista.

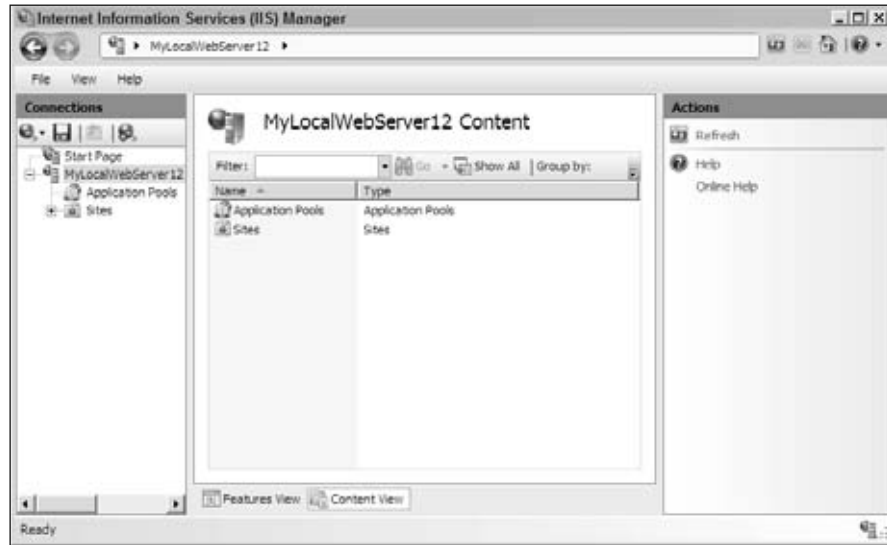


Figure 3-1

The second pane, which is known as the workspace pane, consists of these two tabs:

- ❑ **Features View:** If you select a node in the Connections pane, the Features View tab will allow you to edit the features associated with the selected node.
- ❑ **Content View:** If you select a node in the Connections pane, the Content View tab will display all the child nodes of the selected node.

The third pane, which is known as the Actions pane, contains a bunch of links where each link performs a particular task on the node selected in the first or second pane.

Application Pools

Now click the Application Pools node in the Connections pane to display the available application pools as shown in Figure 3-2.

Notice that the Actions pane contains a link named Add Application Pool. Click this link to launch the dialog shown in Figure 3-3. This dialog allows you to add a new application pool with a specified name. It also allows you to specify the .NET version that will be loaded into the application pool. As discussed in the previous chapter, all ASP.NET applications in the same application pool must use the same .NET version because two different .NET versions cannot be loaded into the same worker process.

The Managed pipeline mode drop-down list on this dialog contains two options, Integrated and Classic, as shown in Figure 3-3. This specifies whether the IIS should run in Integrated or Classic mode for this application pool. All applications in the same application pool use the same IIS mode.

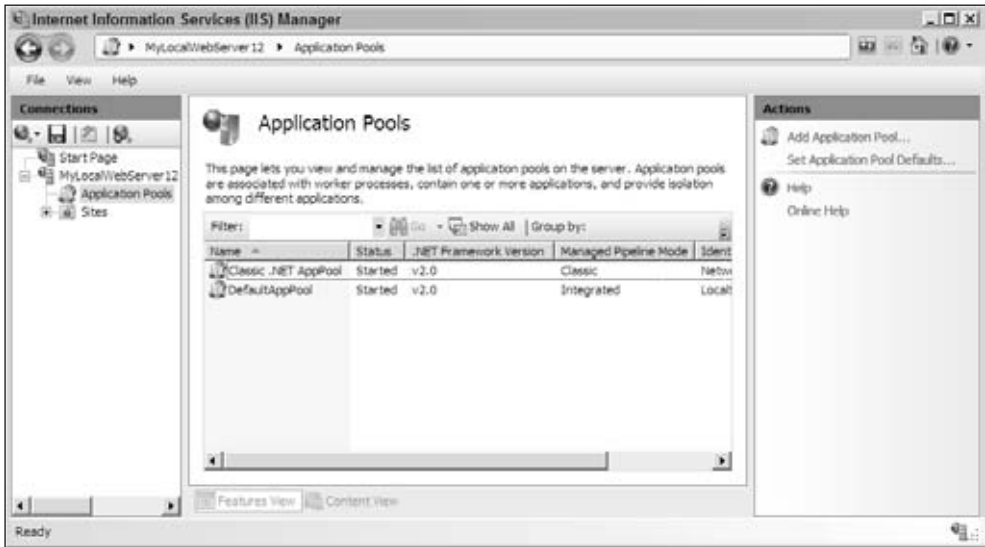


Figure 3-2



Figure 3-3

After making your selection, click OK to commit the changes. Now open the `applicationHost.config` file. You should see the boldfaced section shown in Listing 3-1.

Listing 3-1: The applicationHost.config File

```
<system.applicationHost>
  <applicationPools>
    . . .
    <add name="MyApplicationPool" />
    . . .
  </applicationPools>
</system.applicationHost>
```

Now click the newly created `MyApplicationPool` node in the middle pane. You should see new links on the Actions pane, which allow you to edit the properties of the application pool as shown in Figure 3-4.



Figure 3-4

Click the Advanced Settings link to launch the Advanced Settings dialog shown in Figure 3-5. Notice that all settings of the newly created application pool have default values. However, as Listing 3-1 shows, none of these values show up in the `applicationHost.config` file. Where are these values stored? As you'll see later, the new IIS7 configuration system maintains the schema of the `applicationHost.config` file in two files named `ASPNET_schema.xml` and `IIS_schema.xml`. These schema files also specify and store the default values for configuration sections, including the `<applicationPools>` section. Storing the default configuration settings in one location as opposed to adding them to every single `<add>` element that represents an application pool keeps the configuration files small and more readable.

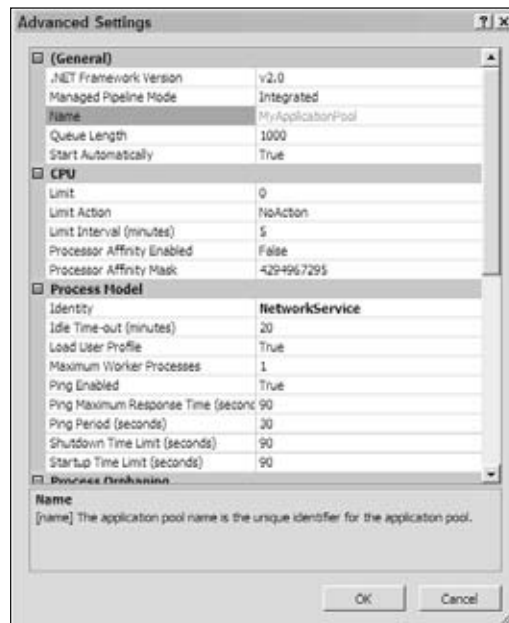


Figure 3-5

Chapter 3: Integrated Configuration from IIS Manager and Command Line

Now go to the General section of the Advanced Settings dialog, change the value of Start Automatically to false (see Figure 3-5), and click OK. Now if you open the `applicationHost.config` file, you should see the boldfaced portion shown in the following code listing:

```
<system.applicationHost>
  <applicationPools>
    . . .
    <add name="MyApplicationPool" autoStart="false" />
    . . .
  </applicationPools>
</system.applicationHost>
```

In other words, the `applicationHost.config` file records only the values that are different from the default.

Notice that the properties shown in Figure 3-5 map to the XML elements and attributes of the `<applicationPools>` section discussed in the previous chapter. When you click the OK button, the callback for this button performs the necessary XML manipulations under the hood to store the changes in the `applicationHost.config` XML file.

Web Sites

Now click the Sites node in the Connections pane of the IIS Manager. You should see a link titled Add Web Site in the Actions pane as shown in Figure 3-6. Click the link to launch the dialog shown in Figure 3-7.



Figure 3-6

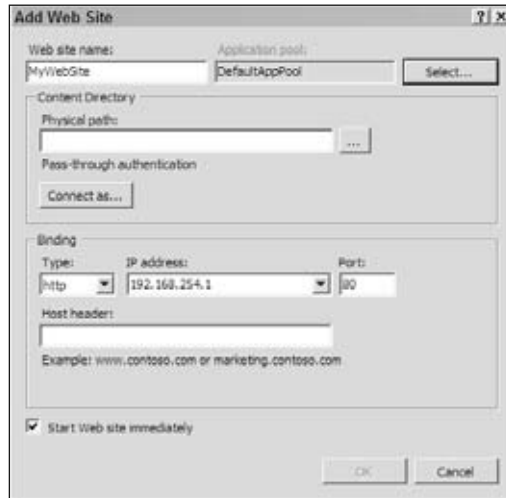


Figure 3-7

This dialog allows you to add a new Web site. Recall that a Web site is a collection of Web applications. Notice that the properties shown in this dialog map to the XML elements and attributes of the `<site>` element discussed in the previous chapter. Next, take these steps:

1. Enter a name in the Web site name text field for the new Web site, for example, MySite.
2. Use the Select button to choose the desired application pool.
3. Choose a physical path.
4. Specify a binding including a binding type, an IP address, and a port number.
5. Click the OK button to commit the changes.

Now open the `applicationHost.config` file again. You should see the boldfaced portion shown in Listing 3-2.

Listing 3-2: The `applicationHost.config` File

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="MySite" id="1727416169">
        <application path="/">
          <virtualDirectory path="/" physicalPath="D:\" />
        </application>
        <bindings>
          <binding protocol="http" bindingInformation="192.168.254.1:80:" />
        </bindings>
      </site>
    </sites>
  </system.applicationHost>
</configuration>
```

Chapter 3: Integrated Configuration from IIS Manager and Command Line

As Listing 3-2 shows, the dialog shown in Figure 3-7 sets the XML elements and attributes of the <site> element that represents the new site. Notice that the dialog automatically created an application with a virtual directory. As discussed in the previous chapter, every site must have at least one application with the virtual path “/” known as the root application that has at least one virtual directory with the virtual path “/” known as the root virtual directory. This dialog automatically takes care of that requirement behind the scenes.

Hierarchical Configuration

As discussed in the previous two chapters, the new IIS7 and ASP.NET integrated configuration system consists of a hierarchy of configuration files where lower-level configuration files inherit the configuration settings from higher-level configuration files. The lower-level configuration files can override only those inherited configuration settings that are not locked in the higher-level configuration files.

In this section, I show you how the IIS Manager takes the hierarchical nature of the IIS7 and ASP.NET integrated configuration system into account. Let’s begin with the ASP.NET configuration settings.

Launch the IIS Manager again, select the node that represents the local Web server in the Connections pane, and switch to the Features View tab in the workspace pane. The result should look like Figure 3-8.



Figure 3-8

Now double-click the Session State icon in the workspace pane. You should see what is shown in Figure 3-9.

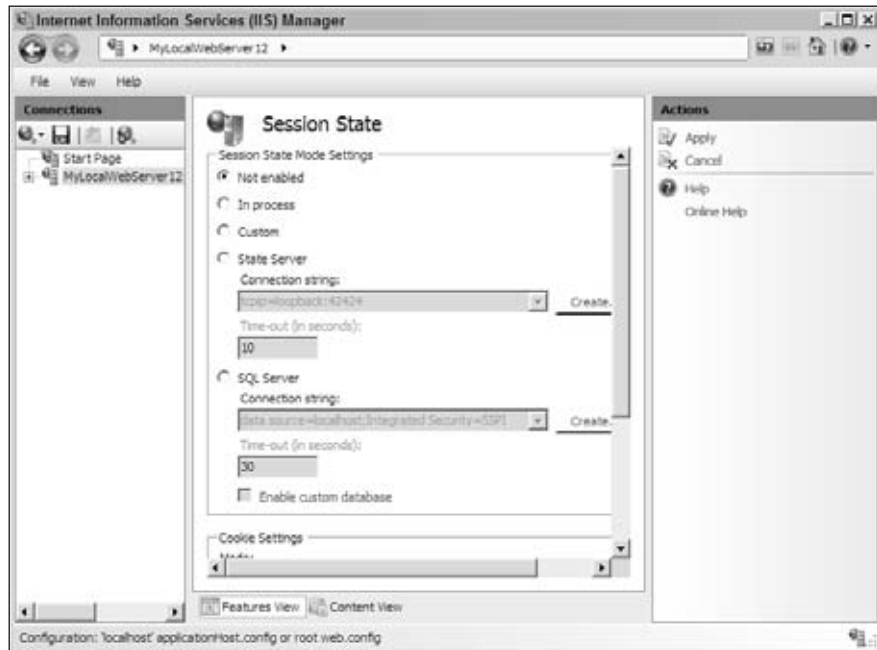


Figure 3-9

Note that the workspace now displays the GUI that allows you to change the session state configuration settings. Go to the Session State Mode Settings section, change the mode to Not enabled, and click the Apply link in the Tasks pane to commit the changes. Now open the root `web.config` file located in the following directory on your machine:

```
%SystemRoot%\Microsoft.NET\Framework\versionNumber\CONFIG\
```

You should see the boldfaced portion shown in the following listing:

```
<configuration>
  <system.web>
    <sessionState mode="off" />
  </system.web>
</configuration>
```

As this example shows, you can use the IIS Manager to specify the ASP.NET configuration settings as you do for the IIS settings. The tool is smart enough to know that the machine-level ASP.NET configuration settings should be saved into the machine-level `web.config` file (known as the root `web.config` file) instead of `applicationHost.config`.

The previous example changed the session state configuration settings at the machine level. Now let's change the session state configuration settings at the site level. Go back to the Connections pane, open the node that represents the local Web server, open the Sites node, and select the Default Web Site node. You should see the result shown in Figure 3-10.



Figure 3-10

Now double-click the Session State icon. The result should look like Figure 3-11. Change the Session State Mode settings to Not enabled and click the Apply link on the task panel to commit the changes. Now open the web.config file in the following directory on your machine:

```
%SystemDrive%\inetpub\wwwroot
```

You should see the boldfaced portion of the following code listing:

```
<configuration>
  <system.web>
    <sessionState mode="off" />
  </system.web>
</configuration>
```

As this example shows, the IIS Manager stores the site-level ASP.NET configuration settings to the site-level configuration file. If you repeat the same steps for application-level ASP.NET configuration settings, you'll see that the IIS Manager stores these configuration settings into the ASP.NET application-level configuration file.

So far I've shown you that the IIS Manager handles the hierarchical nature of the ASP.NET configuration settings. Next, I show you that the IIS Manager also takes the hierarchical nature of the IIS7 configuration settings into account.

Launch the IIS Manager, click the node that represents the local Web server in the Connections pane, switch to the Features View tab in the workspace, and select the Area option from the Group by combo box to group the items in the workspace by area. You should see the result shown in Figure 3-12.

Chapter 3: Integrated Configuration from IIS Manager and Command Line

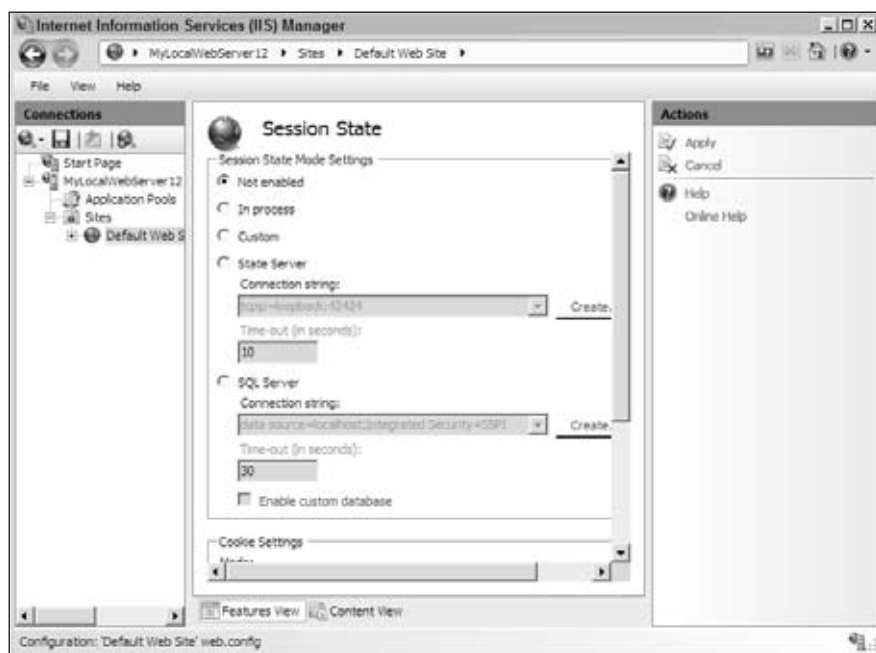


Figure 3-11



Figure 3-12

Chapter 3: Integrated Configuration from IIS Manager and Command Line

Now double-click the Default Document. The result should look like Figure 3-13.

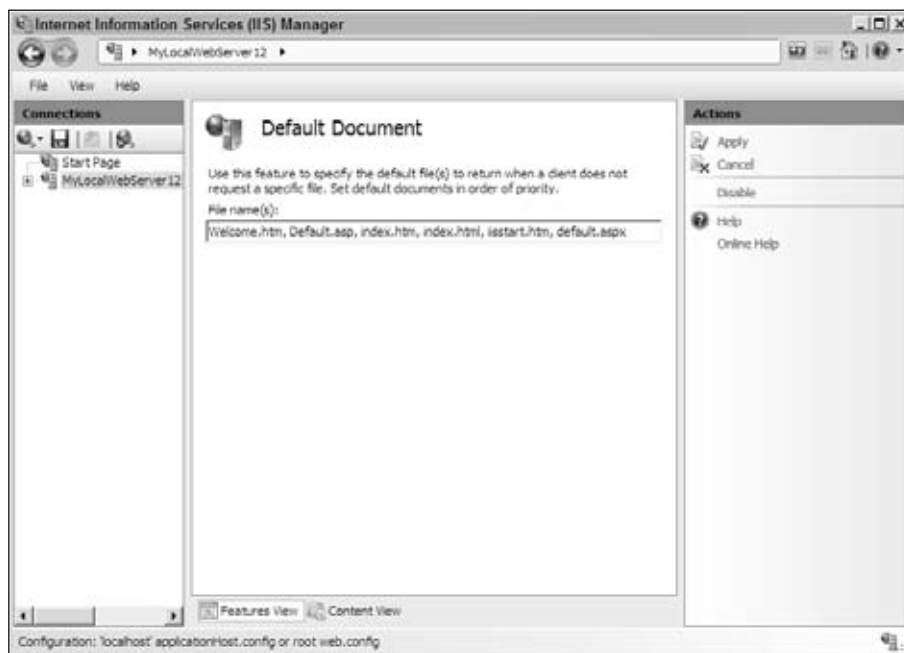


Figure 3-13

Notice that the workspace now contains a textbox that displays the list of default documents. Add a new default document named `Welcome.htm` to the list and click the `Apply` button in the task panel to commit the changes.

If you open the `applicationHost.config` file, you should see the boldfaced portion shown in Listing 3-3. Notice that the `<files>` element now contains a new `<add>` element whose value attribute is set to `"Welcome.htm"`.

Listing 3-3: The `applicationHost.config` File

```
<configuration>
  <system.webServer>
    <defaultDocument enabled="true">
      <files>
        <clear />
        <add value="Welcome.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="index.html" />
        <add value="iisstart.htm" />
        <add value="default.aspx" />
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>
```

```
</defaultDocument>
</system.webServer>
</configuration>
```

Now select the Default Web Site node from the Connections pane of the IIS Manager and double-click the Default Document icon to go to the page that displays the list of default documents. Note that the list contains the `Welcome.html` default document that you added before. This makes sense because the Default Web Site inherits all the default document settings from the machine-level `applicationHost.config` file. Now go ahead and remove the `Welcome.html` file from the list, add a new default document named `Start.html`, and click the Apply button to commit the changes. If you open the `web.config` file located in the Default Web Site's root directory, you should see the result shown in Listing 3-4.

Listing 3-4: The Root `web.config` File

```
<configuration>
  <system.webServer>
    <defaultDocument>
      <files>
        <clear />
        <add value="Start.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="index.html" />
        <add value="iisstart.htm" />
        <add value="default.aspx" />
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>
```

Delegation

As Listing 3-4 shows, a site- or application-level `web.config` file now can contain both ASP.NET and IIS configuration sections. This is one of the great new features of IIS7, which provides the following two benefits among many others:

- ❑ It allows you to configure IIS7 to meet your application-specific requirements.
- ❑ Because these IIS7 custom configuration settings are all stored in the `web.config` file of your application, which is located in the same directory with the rest of your application, you can xcopy this configuration file together with the rest of your application to the test machine, and from there to the production machine. This will allow your testers and clients to start testing and using your applications right away, without going through the tedious task of reconfiguring their Web servers to match the configuration you had on your Web server when you were developing the application.

You may be wondering whether it is a good idea to allow site and application administrators or developers to mess with the Web server settings, from a security perspective. The IIS7 and ASP.NET integrated configuration system has taken this security issue into account. Because of the hierarchical nature of the configuration files, changes made to a configuration file at a certain level of the hierarchy apply

Chapter 3: Integrated Configuration from IIS Manager and Command Line

only to that level and the levels below it. For example, if you make some configuration changes in the `web.config` located in the root directory of a site, it will only affect the applications and virtual directories in that site. Or if you make changes in the `web.config` located in the root directory of an application, it will only affect the virtual directories in that application.

In addition, most IIS configuration sections are locked by default at installation, which means that by default only the machine administrator can change these locked IIS configuration sections. However, the machine administrator can remove the lock from selected IIS configuration sections to allow selected sites, applications, or virtual directories to change these configuration sections. This is known as delegation. Let's take a look at an example.

Recall from the previous example (see Listing 3-4) that the Default Web Site site administrator was allowed to reconfigure the IIS7 default documents for all the applications and virtual directories of the Default Web Site. This was possible because by default there is no lock on the IIS7 default documents feature. To see this, take the following steps:

1. Launch the IIS Manager.
2. Click the node that represents the local Web server in the Connections pane.
3. Switch to the Features View tab in the workspace.
4. Select the Area option from the Group by combo box of the workspace.

The result should look like Figure 3-14.



Figure 3-14

Chapter 3: Integrated Configuration from IIS Manager and Command Line

Now double-click the Feature Delegation icon in the Management section of the workspace to go to the Feature Delegation page shown in Figure 3-15. As the name implies, the Feature Delegation page allows the machine administrator to delegate the administration of the selected IIS features to site and application administrators. Select the Delegation option from the Group by combo box and go to the Read/Write section of this page as shown in Figure 3-15. As the title implies, this section contains IIS7 features that can be read and written from the lower-level configuration files. Note that this section contains the Default Document feature.

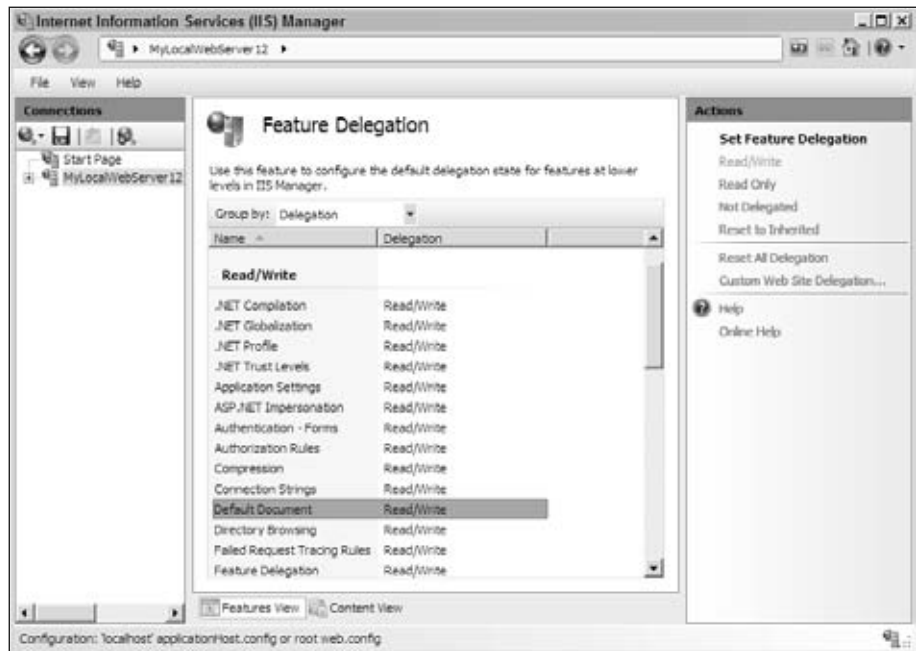


Figure 3-15

Select the Default Document from the Feature Delegation page as shown in Figure 3-15. Notice that the task pane contains a section titled Set Feature Delegation. This section contains six links named:

- ☐ Read/Write
- ☐ Read Only
- ☐ Not Delegated (on Windows Server 2008) or Remove Delegation (on Windows Vista)
- ☐ Reset to Inherited
- ☐ Reset All Delegation
- ☐ Custom Web Site Delegation (this link exists only on Windows Server 2008)

Note that the Read/Write link is grayed out, which means that the lower-level configuration files have the permission to change the IIS7 Default Document feature. Now click the Read Only link and open the `applicationHost.config` file. You should see the boldfaced portion shown in Listing 3-5.

Listing 3-5: The applicationHost.config File

```
<configuration>
  . . .
  <location path="" overrideMode="Deny">
    <system.webServer>
      <defaultDocument enabled="true">
        <files>
          <clear />
          <add value="Welcome.htm" />
          <add value="Default.asp" />
          <add value="index.htm" />
          <add value="index.html" />
          <add value="iisstart.htm" />
          <add value="default.aspx" />
        </files>
      </defaultDocument>
    </system.webServer>
  </location>
  . . .
</configuration>
```

As Listing 3-5 shows, the IIS Manager has added a new `<location>` tag whose `overrideMode` attribute is set to `Deny` to signal that the machine administrator does not want any lower-level configuration file to change the IIS7 default document feature. This means that every site, application, and virtual directory running on the machine inherits these authorization rules and has to live by them.

Now try this:

1. Select the Default Web Site node from the Connections pane.
2. Switch to the Features View tab in the workspace.
3. Select the Area option from the Group by combo box.
4. Double-click the Default Document option from the IIS section of the workspace.

You should get the popup dialog shown in Figure 3-16 telling you that the Default Web Site site cannot change the IIS default documents.

Command-Line Tool

The previous section showed you how to use the IIS Manager to indirectly manipulate the XML elements and attributes that make up the IIS7 unified configuration system. The IIS Manager is a very good choice if you feel more comfortable with GUI-based approaches. However, this convenience comes with a price. As you'll see shortly, there are things that you can do from the command line that you can't do with a graphical tool.

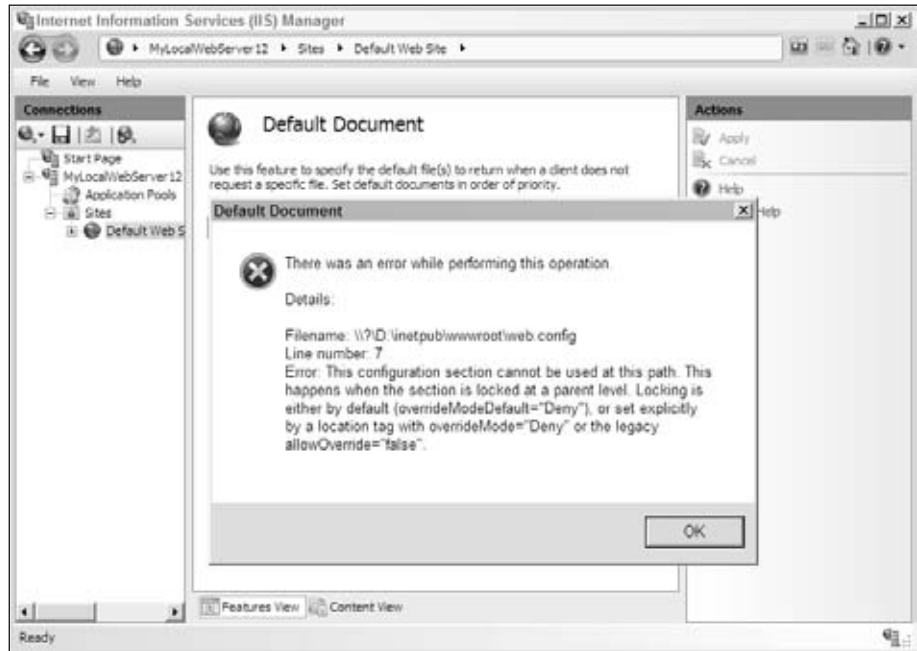


Figure 3-16

IIS7 comes with a brand new command-line tool named `appcmd.exe` (APPCMD) that allows you to indirectly manipulate these XML elements and attributes from the command line. You can find the `appcmd.exe` tool in the following directory on your machine:

```
%windir%\system32\inetsrv\
```

This command-line tool allows you to perform tasks such as the following, which are not possible in a graphical administration tool:

- ☐ Automating server management tasks
- ☐ Forming complex APPCMD commands by combining simpler APPCMD commands where the output of one APPCMD command is used as input to another APPCMD command
- ☐ Exporting the output of an APPCMD command to another program

The APPCMD tool represents each main XML element of the IIS7 configuration system with an object whose properties map to the attributes and child elements of the XML element that the object represents. To help you understand this notion of object, let's revisit the XML elements and attributes used in the `applicationHost.config` file as shown in Listing 3-6. Recall that this file contains the IIS7-specific configuration sections. Because we're only interested in the XML elements and attributes themselves, I've left out the attribute values in Listing 3-6.

Listing 3-6: The applicationHost.config File

```
<configuration>
  <applicationHost>

    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

        <processModel identityType="" userName="" password="" idleTimeout=""
          maxProcesses="" shutdownTimeLimit="" startupTimeLimit=""
          pingingEnabled="" pingInterval="" pingResponseTime="" />

        <recycling disallowOverlappingRotation=""
          disallowRotationOnConfigChange="" logEventOnRecycle="">
          <periodicRestart memory="" privateMemory="" requests="" time="" >
            <schedule>
              <add value="" />
            </schedule>
          </periodicRestart>
        </recycling>

        <cpu limit="" action="" resetInterval="" smpAffinitized=""
          smpProcessorAffinityMask="" />

      </add>
    </applicationPools>

    <sites>
      <site name="" id="" serverAutoStart="">
        <bindings>
          <binding protocol="" bindingInformation="" />
        </bindings>

        <application path="" applicationPool="" enabledProtocols="">
          <virtualDirectory path="" physicalPath="" userName="" password="" />
        </application>
      </site>
    </sites>

  </applicationHost>
  . . .
</configuration>
```

Notice that the `<applicationPools>` element consists of one or more `<add>` elements, each of which represents an application pool. The APPCMD tool has an object name APPPOOL, which represents an application pool. This means that the APPPOOL object maps to the `<add>` element.

As Listing 3-6 shows, the `<add>` element consists of these XML attributes: `name`, `queueLength`, `autoStart`, `managedRuntimeVersion`, and `managedPipelineMode`. The APPPOOL object exposes five properties with the same names as these attributes. The `<add>` element also contains child elements such as `<processModel>`, `<recycling>`, and `<cpu>`. The APPPOOL object exposes properties with the same name as these child elements, and each property exposes subproperties

Chapter 3: Integrated Configuration from IIS Manager and Command Line

with the same names as the attributes of these child elements. For example, you can use the expression `processModel.identityType` to refer to the `identityType` attribute of the `<processModel>` child element.

As shown in Listing 3-6, the `<sites>` element consists of one or more `<site>` child elements, each of which represents a Web site. The APPCMD tool has an object named `SITE` that maps to the `<site>` child element. Also notice that the `<site>` child element contains one or more `<application>` child elements that each represent a Web application. The APPCMD tool comes with an object named `APP` that represents the `<application>` element. This object exposes properties with the same name as the attributes of this element.

As Listing 3-6 shows, the `<application>` element also contains one or more `<virtualDirectory>` elements. The APPCMD tool's `VDIR` object represents the `<virtualDirectory>` element and exposes properties with the same names as the attributes of this element.

In addition to these objects, the APPCMD tool comes with objects that represent entities such as request (`REQUEST` object), worker process (`WP` object), and server module (`MODULE` object).

So far I've covered what I like to call the object model of the APPCMD tool. This tool allows you to perform certain operations on each object. Most objects support these four operations or commands: `LIST`, `ADD`, `DELETE`, and `SET`. These commands or operations are very similar to the four basic `SELECT`, `INSERT`, `DELETE`, and `UPDATE` database operations, respectively. As far as these operations go, you can think of an instance of the APPCMD object as a data record and each property of the object instance as a database field. Just as every data record has a database field (primary key) that uniquely identifies that record among other records, every APPCMD object instance has a property that uniquely identifies the instance among other instances as described in the following table:

Object	Identifying Property
APPPool	name: This property specifies the name of the application pool. Each application pool must have a unique name.
SITE	name and id: Each site must have a unique name and id.
APP	path: This property specifies the virtual path of the Web application.
VDIR	path: This property specifies the virtual path of the virtual directory relative to the Web application that contains the directory.

The general syntax of APPCMD is as follows:

```
APPCMD.EXE <COMMAND> <OBJECT> <ID> [ /parameter:value ]*
```

where:

- ❑ The `<COMMAND>` option specifies the command or operation to perform on the specified APPCMD object. Examples of such commands are `LIST`, `ADD`, `DELETE`, and `SET`.
- ❑ The `<OBJECT>` option specifies the APPCMD object on which the command is performed. Examples of such objects are `APPPool`, `SITE`, `APP`, and `VDIR`.

Chapter 3: Integrated Configuration from IIS Manager and Command Line

- ❑ The <ID> option specifies the identifier of the object instance on which the command is performed. Examples of such identifiers are the application pool name and the Web application virtual path.
- ❑ The [/parameter:value]* option specifies a list of /parameter:value options where each option specifies the name and value of a given property of the object on which the command is performed.

LIST

Just as the SELECT database operation selects or lists data records that meet the set of criteria specified in the WHERE clause, the LIST command lists object instances that meet a specified set of criteria. Here is an example:

```
APPCMD LIST APPPOOL
```

If you run this command you'll get something like Figure 3-17, which lists all the application pools running on the Web server.

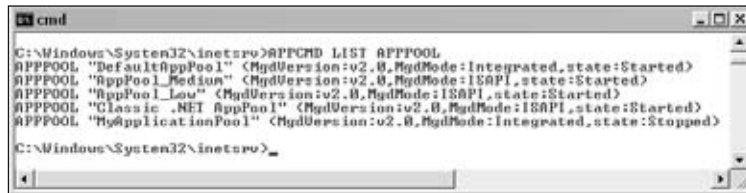


Figure 3-17

Notice that the information about each application pool is listed on a single line, which consists of three major parts. The first part is the object on which the command was performed, that is, APPPOOL. The second part is the identifier of the object instance, that is, the application pool name, for example, DefaultAppPool. The third part is a comma-separated list of items. Each item in the list consists of two parts separated by a colon (:), which respectively specifies the name and value of some of the properties of the object instance.

If you just want to see the information about a particular application pool, you need to specify its identifier as follows (see Figure 3-18):

```
APPCMD LIST APPPOOL "DefaultAppPool"
```

This is very similar to a SELECT database operation where you specify the primary key of the record in the WHERE clause.

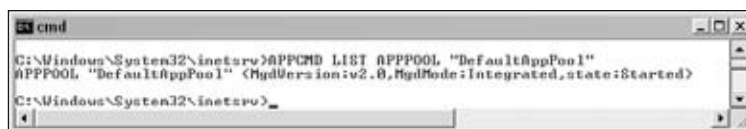


Figure 3-18

ADD

Just as the `INSERT` database operation inserts or adds a new data record, the `ADD` command adds a new object instance. When you're adding a new data record, you have to specify the values of non-nullable data fields. The same rule applies to the `ADD` command. When you're adding a new object instance, you must specify the value of the required object properties. If you don't know what the required properties are, use the following help command:

```
APPCMD ADD <OBJECT> /?
```

The `<OBJECT>` option could be any of the `APPCMD` objects such as `APPPPOOL`, `SITE`, and so on. Here is an example:

```
APPCMD ADD SITE /?
```

After you find out what the required properties of the object are, you can use an `ADD` command similar to the following to add a new instance of the object:

```
APPCMD ADD SITE /name:Site1 /id:4
```

This command adds a site named `Site1` with an `id` property value of 4.

DELETE

Just as the `DELETE` database operation deletes a data record with a specified primary key, the `DELETE` command deletes an object instance with the specified identifier. For example, the following command deletes the application pool with the specified name:

```
APPCMD DELETE APPPOOL "MyApplicationPool"
```

SET

Just as the `UPDATE` database operation updates a data record with a specified primary key, the `SET` command updates an object instance with the specified identifier. The following command changes the name of the specified site:

```
APPCMD SET SITE "Site1" /name:"Site2"
```

Summary

This chapter discussed three different ways to interact with the new IIS7 and ASP.NET integrated configuration system. The next chapter shows you how to access this configuration system from within your C# or Visual Basic code.

4

Managing the Integrated Configuration System with Managed Code

The previous chapters walked you through the XML structure of the IIS7 and ASP.NET integrated configuration system. You learned a great deal about the XML elements and attributes that make up this system. You also learned how to use the IIS7 Manager and `APPCMD` command-line tool to indirectly manipulate these XML elements and attributes.

There are times when you want to manipulate these XML elements and attributes from your C# or Visual Basic code. Obviously, GUI-based and command-line-based approaches such as the IIS7 Manager and the `APPCMD` tool won't help you with this.

The managed classes of the new `Microsoft.Web.Administration` namespace together form an API that allows you to treat the XML elements and attributes of the IIS7 and ASP.NET integrated configuration system as managed objects, which means you can use object-oriented managed code to manipulate them.

Class Diagrams

To help you understand the managed classes in the `Microsoft.Web.Administration` namespace, let's revisit the XML elements and attributes used in the `<applicationHost>` section group of the `applicationHost.config` file as shown in Listing 4-1. Recall that this file contains the IIS7-specific configuration sections. Because we're only interested in the XML elements and attributes themselves, I've left out the attribute values from Listing 4-1.

Listing 4-1: The applicationHost.config File

```
<configuration>
  <applicationHost>

    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

        <processModel identityType="" userName="" password="" idleTimeout=""
          maxProcesses="" shutdownTimeLimit="" startupTimeLimit=""
          pingingEnabled="" pingInterval="" pingResponseTime="" />

        <recycling disallowOverlappingRotation=""
          disallowRotationOnConfigChange="" logEventOnRecycle="">
          <periodicRestart memory="" privateMemory="" requests="" time="" >
            <schedule>
              <add value="" />
            </schedule>
          </periodicRestart>
        </recycling>

        <cpu limit="" action="" resetInterval="" smpAffinitized=""
          smpProcessorAffinityMask="" />

      </add>
      . . .
    </applicationPools>

    <sites>
      <site name="" id="" serverAutoStart="">
        <bindings>
          <binding protocol="" bindingInformation="" />
        </bindings>

        <application path="" applicationPool="" enabledProtocols="">
          <virtualDirectory path="" physicalPath="" userName="" password="" />
        </application>
      </site>
      . . .
    </sites>

  </applicationHost>
  . . .
</configuration>
```

The `Microsoft.Web.Administration` namespace represents each main XML element in the `applicationHost.config` file with a managed class whose properties map to the XML attributes and the XML child elements of the XML element as discussed in the following sections. Before diving into these mappings, take a look at the class diagram that represents the relationship between the classes of the `Microsoft.Web.Administration` namespace as shown in Figures 4-1 and 4-2. The IIS7 and ASP.NET integrated imperative management API's types fall into two main categories. The first category contains those types that represent the XML constructs making up the IIS7 and ASP.NET integrated configuration system. These are the types that you can use from within your C# or Visual Basic code to access and manipulate the XML constructs of the underlying configuration file programmatically. I discuss these

Chapter 4: Integrated Configuration from Managed Code

types in great detail in this chapter. The second category contains those types that provide detailed up-to-date runtime data that you can use for IIS7 troubleshooting. These types include `Request`, `RequestCollection`, `ApplicationDomain`, `ApplicationDomainCollection`, `WorkerProcess`, and `WorkerProcessCollection`. I cover these types in great detail in Chapter 12.

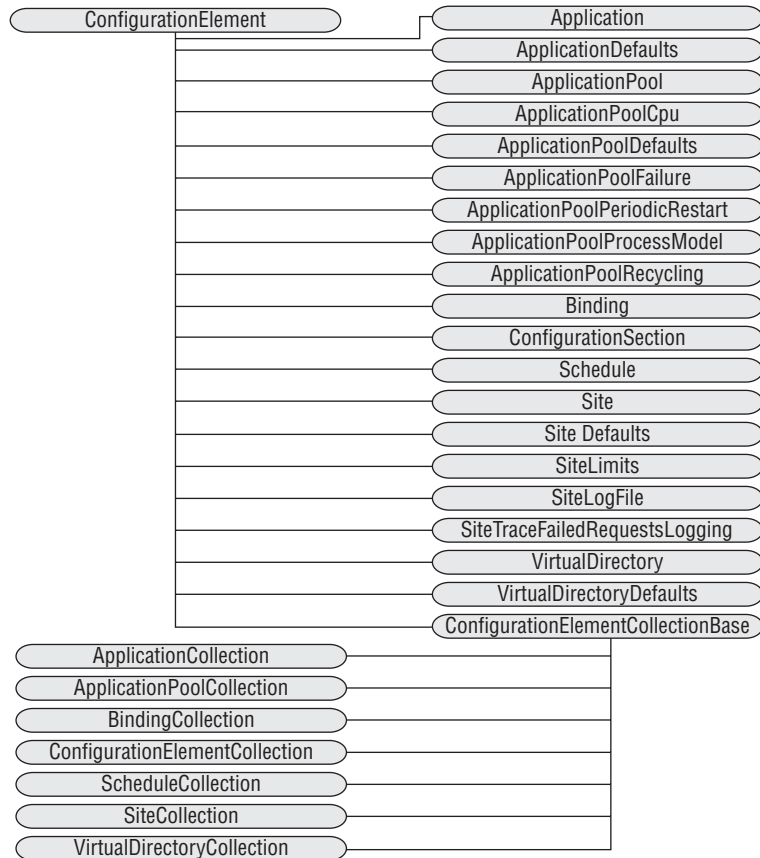


Figure 4-1

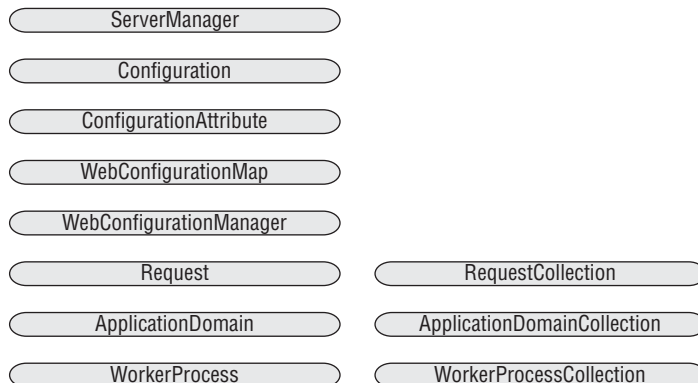


Figure 4-2

In the following sections I discuss some of the methods and properties of the managed classes of the IIS7 and ASP.NET imperative management API.

ConfigurationElement

The `ConfigurationElement` class represents an XML element, which makes it the base class for all the classes that represent the XML elements of the configuration files, such as the `applicationHost.config` file as shown in Figure 4-1. Listing 4-2 presents some of the members of the `ConfigurationElement` class.

Listing 4-2: The ConfigurationElement Class

```
public class ConfigurationElement
{
    // Methods
    public object GetAttributeValue(string attributeName);
    public void SetAttributeValue(string attributeName, object value);
    public ConfigurationElementCollection GetCollection(string collectionName);

    // Properties
    public object this[string attributeName] { get; set; }
    . . .
}
```

The following table describes these members:

Member	Description
<code>GetAttributeValue</code>	Gets the value of the XML attribute with the specified name.
<code>SetAttributeValue</code>	Sets the XML attribute with the specified name to the specified value.
<code>GetCollection</code>	Gets the <code>ConfigurationElementCollection</code> collection that represents a child collection element.

Notice that the `ConfigurationElement` class also exposes an indexer that returns the value of the XML attribute with the specified name. Every configuration section may contain one or more child collection elements. For example, the `<site>` element contains a child collection element named `<bindings>` as shown in Listing 4-1. The `GetCollection` method returns the `ConfigurationElementCollection` object that represents a given child collection element. I discuss collection elements in more detail later in this chapter.

ConfigurationElementCollectionBase<T>

As Listing 4-3 shows, the `ConfigurationElementCollectionBase<T>` class is a generic collection class that contains instances of the `ConfigurationElement` class. As such it is the base class for all classes that represent the XML elements in the configuration file that act as containers for other XML elements.

Listing 4-3: The ConfigurationElementCollectionBase Class

```
public abstract class ConfigurationElementCollectionBase<T> :
    ConfigurationElement, ICollection, IEnumerable<T>, IEnumerable
    where T : ConfigurationElement
{
    // Methods
    public T Add(T element);
    public T AddAt(int index, T element);
    public void Clear();
    public T CreateElement(string elementTagName);
    public IEnumerator<T> GetEnumerator();
    public int IndexOf(T element);
    public void Remove(T element);
    public void RemoveAt(int index);

    // Properties
    public bool AllowsAdd { get; }
    public bool AllowsClear { get; }
    public bool AllowsRemove { get; }
    public int Count { get; }
    public T this[int index] { get; }
}
```

Notice that the definition of the `ConfigurationElementCollectionBase<T>` class contains the following to ensure that only objects of type `ConfigurationElement` are added to the collection:

```
where T : ConfigurationElement
```

The following table describes the methods of the `ConfigurationElementCollectionBase<T>` class:

Method	Description
Add	Adds the specified <code>ConfigurationElement</code> object to the collection.
AddAt	Inserts the specified <code>ConfigurationElement</code> object into the collection at the specified location.
Clear	Clears the collection.
CreateElement	Creates a <code>ConfigurationElement</code> object with the specified tag name.
GetEnumerator	Returns an <code>IEnumerator<T></code> object that allows you to iterate through the <code>ConfigurationElement</code> objects in the collection. This also allows you to use this collection in a <code>foreach</code> loop.
IndexOf	Returns the index of the specified <code>ConfigurationElement</code> object.
Remove	Removes the specified <code>ConfigurationElement</code> object from the collection.
RemoveAt	Removes the <code>ConfigurationElement</code> object with the specified index from the collection.

Chapter 4: Integrated Configuration from Managed Code

The following table describes the properties of the `ConfigurationElementCollectionBase<T>` class:

Property	Description
<code>AllowsAdd</code>	Gets the Boolean value that specifies whether new <code>ConfigurationElement</code> objects can be added to the collection.
<code>AllowsClear</code>	Gets the Boolean value that specifies whether the collection can be cleared.
<code>AllowsRemove</code>	Gets the Boolean value that specifies whether <code>ConfigurationElement</code> objects can be removed from the collection.
<code>Count</code>	Gets the total number of <code>ConfigurationElement</code> objects in the collection.
<code>Item</code>	An indexer that gets the <code>ConfigurationElement</code> object at the specified location in the collection.

ApplicationPool

Let’s revisit the portion of Listing 4-1 highlighted in Listing 4-4. As this code shows, the `<applicationPools>` element contains one or more `<add>` child elements, each of which adds a particular application pool and specifies its configuration settings.

Listing 4-4: The Portion of the applicationPools Section of the applicationHost.config File

```
<configuration>
  <applicationHost>

    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">
        . . .
      </add>
      . . .
    </applicationPools>
    . . .
  </applicationHost>
  . . .
</configuration>
```

The IIS7 and ASP.NET integrated imperative management API represents each `<add>` child element with an instance of a managed class named `ApplicationPool`. The **boldfaced** portion of Listing 4-5 shows the properties of the `ApplicationPool` class that map to the attributes of the `<add>` child element highlighted in Listing 4-4. Notice that the `ApplicationPool` class derives from the `ConfigurationElement` base class, which means that it inherits all the properties and methods shown in Listing 4-2.

Listing 4-5: The Important Methods and Properties of the ApplicationPool Class

```

public sealed class ApplicationPool : ConfigurationElement
{
    // Methods
    public ObjectState Recycle();
    public ObjectState Start();
    public ObjectState Stop();

    // Properties
    public bool AutoStart { get; set; }
    public ManagedPipelineMode ManagedPipelineMode { get; set; }
    public string ManagedRuntimeVersion { get; set; }
    public string Name { get; set; }
    public long QueueLength { get; set; }

    public ApplicationPoolRecycling Recycling { get; }
    public ApplicationPoolCpu Cpu { get; }
    public ApplicationPoolProcessModel ProcessModel { get; }

    public WorkerProcessCollection WorkerProcesses { get; }
}

```

The ApplicationPool class also features the three methods described in the following table:

Method	Description
Recycle	Recycles the current application pool.
Start	Starts the current application pool.
Stop	Stops the current application pool.

As Listing 4-1 shows, the <add> element that defines an application pool contains three important child elements named <processModel>, <recycling>, and <cpu>, which were discussed thoroughly in Chapter 2. I discuss the programmatic or imperative representation of these child elements in the following sections.

ApplicationPoolProcessModel

As the highlighted portion of the following code listing shows (which repeats the associated portion from Listing 4-1), the <processModel> child element of a given <add> element specifies the process model for the application pool that the <add> element represents.

```

<configuration>
  <applicationHost>

    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

```

```
<processModel identityType="" userName="" password="" idleTimeout=""
    maxProcesses="" shutdownTimeLimit="" startupTimeLimit=""
    pingingEnabled="" pingInterval="" pingResponseTime="" />
    . . .
</add>
    . . .
</applicationPools>
    . . .
</applicationHost>
    . . .
</configuration>
```

The IIS7 and ASP.NET integrated imperative management API represents the `<processModel>` element with an instance of a managed class named `ApplicationPoolProcessModel` whose properties map to the XML attributes of the associated `<processModel>` element as shown in Listing 4-6.

Listing 4-6: The `ApplicationPoolProcessModel` Class

```
public sealed class ApplicationPoolProcessModel : ConfigurationElement
{
    // Properties
    public ProcessModelIdentityType IdentityType { get; set; }
    public TimeSpan IdleTimeout { get; set; }
    public long MaxProcesses { get; set; }
    public string Password { get; set; }
    public bool PingingEnabled { get; set; }
    public TimeSpan PingInterval { get; set; }
    public TimeSpan PingResponseTime { get; set; }
    public TimeSpan ShutdownTimeLimit { get; set; }
    public TimeSpan StartupTimeLimit { get; set; }
    public string UserName { get; set; }
}
```

Recall from Listing 4-5 that the `ApplicationPool` class features a property of type `ApplicationPoolProcessModel` named `ProcessModel`, which refers to the `ApplicationPoolProcessModel` object that represents the `<processModel>` child element.

ApplicationPoolRecycling

As the highlighted portion of Listing 4-7 (which repeats the portion of Listing 4-1) shows, the `<recycling>` child element of the `<add>` element is used to specify the recycling configuration settings for the application pool that the `<add>` element represents.

Listing 4-7: The `<recycling>` Element

```
<configuration>
  <applicationHost>
    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
```

Listing 4-7: (continued)

```
managedPipelineMode="">

<processModel . . . />

<recycling disallowOverlappingRotation=""
  disallowRotationOnConfigChange="" logEventOnRecycle="">
  <periodicRestart memory="" privateMemory="" requests="" time="" >
    <schedule>
      <add value="" />
    </schedule>
  </periodicRestart>
</recycling>
. . .
</add>
. . .
</applicationPools>
. . .
</applicationHost>
. . .
</configuration>
```

The IIS7 and ASP.NET integrated imperative management API comes with a managed class named `ApplicationPoolRecycling` that represents the `<recycling>` element. The boldfaced portion of Listing 4-8 presents the properties of the `ApplicationPoolRecycling` class that map to the XML attributes of the `<recycling>` element.

Listing 4-8: The `ApplicationPoolRecycling` Class

```
public sealed class ApplicationPoolRecycling : ConfigurationElement
{
    public bool DisallowOverlappingRotation { get; set; }
    public bool DisallowRotationOnConfigChange { get; set; }
    public RecyclingLogEventOnRecycle LogEventOnRecycle { get; set; }

    public ApplicationPoolPeriodicRestart PeriodicRestart { get; }
}
```

As Listing 4-5 shows, the `ApplicationPool` class features a property of type `ApplicationPoolRecycling` named `Recycling`, which refers to the `ApplicationPoolRecycling` object that represents the `<recycling>` child element.

ApplicationPoolPeriodicRestart

As the highlighted portion of Listing 4-9 (which repeats Listing 4-7) demonstrates, the `<recycling>` element contains a child element named `<periodicRestart>` that can be used to specify the conditions for which the application pool must be recycled.

Listing 4-9: The <recycling> Element

```
<configuration>
  <applicationHost>
    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

        <processModel . . . />

        <recycling disallowOverlappingRotation=""
          disallowRotationOnConfigChange="" logEventOnRecycle="">
          <periodicRestart memory="" privateMemory="" requests="" time="" >
            <schedule>
              <add value="" />
            </schedule>
          </periodicRestart>
        </recycling>
      . . .
    </add>
  . . .
</applicationPools>
. . .
</applicationHost>
. . .
</configuration>
```

It shouldn't come as a surprise that the IIS7 and ASP.NET integrated imperative management API comes with a class named `ApplicationPoolPeriodicRestart` that provides programmatic access to the `<periodicRestart>` element. The boldfaced portion of Listing 4-10 contains the properties of the `ApplicationPoolPeriodicRestart` class that map to the XML attributes of the `<periodicRestart>` element.

Listing 4-10: The `ApplicationPoolPeriodicRestart` Class

```
public sealed class ApplicationPoolPeriodicRestart : ConfigurationElement
{
    // Properties
    public long Memory { get; set; }
    public long PrivateMemory { get; set; }
    public long Requests { get; set; }
    public TimeSpan Time { get; set; }

    public ScheduleCollection Schedule { get; }
}
```

As Listing 4-8 demonstrates, the `ApplicationPoolRecycling` class features a property of type `ApplicationPoolPeriodicRestart` named `PeriodicRestart`, which refers to the `ApplicationPoolPeriodicRestart` object that represents the `<periodicRestart>` child element.

ScheduleCollection

The highlighted portion of Listing 4-11 shows that the `<periodicRestart>` child element of the `<recycling>` element is used to schedule the times at which the application pool must be recycled.

Listing 4-11: The `<recycling>` Element

```
<configuration>
  <applicationHost>
    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

        <processModel . . . />

        <recycling disallowOverlappingRotation=""
          disallowRotationOnConfigChange="" logEventOnRecycle="">
          <periodicRestart memory="" privateMemory="" requests="" time="" >
            <schedule>
              <add value="" />
            </schedule>
          </periodicRestart>
        </recycling>
        . . .
      </add>
      . . .
    </applicationPools>
    . . .
  </applicationHost>
  . . .
</configuration>
```

The `ScheduleCollection` class of the IIS7 and ASP.NET integrated imperative management API is the programmatic representation of the `<schedule>` element. As such it exposes a single method named `Add` that adds a new schedule time to the collection (see Listing 4-12).

Listing 4-12: The `ScheduleCollection` Class

```
public sealed class ScheduleCollection :
    ConfigurationElementCollectionBase<Schedule>
{
    public Schedule Add(TimeSpan scheduleTime);
}
```

ApplicationPoolCpu

Listing 4-13 repeats the `<applicationPools>` portion of Listing 4-1. As the highlighted portion of this code listing shows, the `<cpu>` child element of the `<add>` element is used to specify CPU configuration settings for the application pool that the `<add>` element represents.

Listing 4-13: The <applicationPools> Portion of the applicationHost.config File

```
<configuration>
  <applicationHost>

    <applicationPools>
      <add name="" queueLength="" autoStart="" managedRuntimeVersion=""
        managedPipelineMode="">

        <processModel . . . />

        <recycling . . . >
          . . .
        </recycling>

        <cpu limit="" action="" resetInterval="" smpAffinitized=""
          smpProcessorAffinityMask="" />

      </add>
      . . .
    </applicationPools>
    . . .
  </applicationHost>
  . . .
</configuration>
```

The `ApplicationPoolCpu` class of the IIS7 and ASP.NET integrated imperative management API allows you to access the `<cpu>` element from within your C# or Visual Basic code. As you'd expect, the properties of this class map to the XML attributes of the `<cpu>` element (see Listing 4-14).

Listing 4-14: The ApplicationPoolCpu Class

```
public sealed class ApplicationPoolCpu : ConfigurationElement
{
    // Properties
    public ProcessorAction Action { get; set; }
    public long Limit { get; set; }
    public TimeSpan ResetInterval { get; set; }
    public bool SmpAffinitized { get; set; }
    public long SmpProcessorAffinityMask { get; set; }
}
```

As Listing 4-5 shows, the `ApplicationPool` class contains a property of type `ApplicationPoolCpu` named `Cpu`, which refers to the `ApplicationPoolCpu` object that represents the `<cpu>` child element.

ApplicationPoolCollection

The `ApplicationPoolCollection` class represents the `<applicationPools>` XML element of the `applicationHost.config` file. This class acts as a container for the application pools defined in the `<applicationPools>` section. Listing 4-15 presents the members of the `ApplicationPoolCollection`

class. Notice that this class exposes an indexer that allows you to use the name of an application pool to return the `ApplicationPool`.

Listing 4-15 : The `ApplicationPoolCollection` Class

```
public sealed class ApplicationPoolCollection :
    ConfigurationElementCollectionBase<ApplicationPool>
{
    // Methods
    public ApplicationPool Add(string name);

    // Properties
    public ApplicationPool this[string key] { get; }
}
```

Site

As the highlighted portion of Listing 4-16 shows, the `<site>` element is used to add a new Web site with the specified name and id.

Listing 4-16: The `<site>` Element and Its Attribute

```
<configuration>
  <applicationHost>
    . . .
    <sites>
      <site name="" id="" serverAutoStart="">
        . . .
      </site>
    </sites>
  </applicationHost>
  . . .
</configuration>
```

The `Site` class of the IIS7 and ASP.NET integrated imperative management API provides you with programmatic access to the `<site>` element. As such, it exposes properties that map to the attributes of this element as illustrated in the boldfaced portion of Listing 4-17.

Listing 4-17: The `Site` Class

```
public sealed class Site : ConfigurationElement
{
    // Methods
    public Configuration GetWebConfiguration();
    public ObjectState Start();
    public ObjectState Stop();

    // Properties
```

(continued)

Listing 4-17: *(continued)*

```
public long Id { get; set; }
public string Name { get; set; }
public bool ServerAutoStart { get; set; }

public ApplicationCollection Applications { get; }
public BindingCollection Bindings { get; }
}
```

The following table describes the methods of the `Site` class:

Method	Description
<code>GetWebApplication</code>	Gets the <code>Configuration</code> object that represents the <code>web.config</code> file in the root directory of the Web site.
<code>Start</code>	Starts the Web site.
<code>Stop</code>	Stops the Web site.

Notice that the `GetWebApplication` method loads the contents of the root `web.config` file of the Web site into an instance of a class named `Configuration`. This class exposes an important method named `GetSection` that returns the `ConfigurationSection` object that represents the configuration section with the specified section path:

```
public ConfigurationSection GetSection(string sectionPath);
```

Binding

As the highlighted portion of Listing 4-18 shows, the `<bindings>` child element of the `<site>` element is used to specify the bindings that the Web site supports. Each binding is represented by a `<binding>` element that features two attributes named `protocol` and `bindingInformation`. The `protocol` attribute specifies the transport communication protocol, such as HTTP, that the client should use to communicate with the Web site. The `bindingInformation` attribute specifies three colon-separated pieces of information: the IP address of the Web site, the port number at which the Web site is listening for incoming requests, and optional host header. Here is an example:

```
<binding protocol="http" bindingInformation="*:80:" />
```

This binding does not specify the host header.

Listing 4-18: The `applicationHost.config` File

```
<configuration>
  <applicationHost>
    . . .
  </sites>
```

Listing 4-18: *(continued)*

```
<site name="" id="" serverAutoStart="">
  <bindings>
    <binding protocol="" bindingInformation="" />
  </bindings>
  . . .
</site>
. . .
</sites>

</applicationHost>
. . .
</configuration>
```

The Binding class of the IIS7 and ASP.NET integrated imperative management API represents the <binding> element (see Listing 4-19). This class exposes two properties that map to the XML attributes of the <binding> element.

Listing 4-19: The Binding Class

```
public class Binding : ConfigurationElement
{
    . . .
    public string BindingInformation { get; set; }
    public string Protocol { get; set; }
}
```

BindingCollection

As Listing 4-18 shows, the <site> element exposes a child element named <bindings>. The BindingCollection class of the IIS7 and ASP.NET integrated imperative management API represents this child element, which means that its instances act as containers for the Binding objects that represent <binding> elements. Listing 4-20 presents the BindingCollection class.

Listing 4-20: The BindingCollection Class

```
public sealed class BindingCollection : ConfigurationElementCollectionBase<Binding>
{
    // Methods
    public Binding Add(string bindingInformation, string bindingProtocol);
    public void Remove(Binding element);
    public void RemoveAt(int index);
}
```

Chapter 4: Integrated Configuration from Managed Code

The following table describes the methods of the `BindingCollection` class:

Method	Description
Add	Creates a <code>Binding</code> object with the specified protocol and binding information and adds it to the collection.
Remove	Removes the specified <code>Binding</code> object from the collection.
RemoveAt	Removes the <code>Binding</code> object with the specified index from the collection.

As Listing 4-17 shows, the `Site` class exposes a property of type `BindingCollection` named `Bindings` that represents the `<bindings>` child element. The `Bindings` collection contains the `Binding` objects that represent the `<binding>` child elements of the `<bindings>` element:

```
public sealed class Site : ConfigurationElement
{
    . . .
    public ApplicationCollection Applications { get; }
    public BindingCollection Bindings { get; }
}
```

Application

As the highlighted portion of Listing 4-21 reveals, the `<site>` element contains one or more `<application>` child elements, each of which adds a new Web application.

Listing 4-21: The `<application>` Element

```
<configuration>
  <applicationHost>
    . . .
    <sites>
      <site name="" id="" serverAutoStart="">
        <bindings>
          <binding protocol="" bindingInformation="" />
        </bindings>
        <application path="" applicationPool="" enabledProtocols="">
          <virtualDirectory path="" physicalPath="" userName="" password="" />
        </application>
        . . .
      </site>
      . . .
    </sites>

  </applicationHost>
  . . .
</configuration>
```

The `Application` class of the IIS7 and ASP.NET integrated imperative management API allows you to programmatically access the `<application>` element. The boldfaced portion of Listing 4-22 presents the properties of the `Application` class that map to the `path`, `applicationPool`, and `enableProtocols` XML attributes of the `<application>` element.

Listing 4-22: The Application Class

```
public sealed class Application : ConfigurationElement
{
    // Methods
    public Configuration GetWebConfiguration();

    // Properties
    public string ApplicationPoolName { get; set; }
    public string EnabledProtocols { get; set; }
    public string Path { get; set; }

    public VirtualDirectoryCollection VirtualDirectories { get; }
}
```

Note that the `Application` class features a method named `GetWebConfiguration` that loads the root `web.config` file of the application into a `Configuration` object.

ApplicationCollection

As discussed earlier, the `<site>` element contains one or more child elements named `<application>`, which specify the Web applications that the Web site contains (see Listing 4-21). The IIS7 and ASP.NET integrated imperative management API comes with a class named `ApplicationCollection` that acts as a container for the `Application` objects that represent `<applications>` elements as presented in Listing 4-23.

Listing 4-23: The ApplicationCollection Class

```
public sealed class ApplicationCollection :
    ConfigurationElementCollectionBase<Application>
{
    // Methods
    public Application Add(string path, string physicalPath);

    // Properties
    public Application this[string path] { get; }
}
```

The `ApplicationCollection` class, like any other collection class in the IIS7 and ASP.NET integrated imperative management API, inherits from the `ConfigurationElementCollectionBase` generic class. The `Add` method of this class creates an `Application` object with the specified virtual and physical path, and adds the object to the collection. Note that the `ApplicationCollection` features an indexer that returns the `Application` with the specified virtual path.

Chapter 4: Integrated Configuration from Managed Code

As Listing 4-17 shows, the `Site` class exposes a property of type `ApplicationCollection` named `Applications`. This property references the `ApplicationCollection` collection containing the `Application` objects that represent the `<application>` child elements of the `<site>` element that the `Site` class represents:

```
public sealed class Site : ConfigurationElement
{
    . . .
    public ApplicationCollection Applications { get; }
    public BindingCollection Bindings { get; }
}
```

VirtualDirectory

As the highlighted portion of Listing 4-24 demonstrates, the `<application>` element contains one or more `<virtualDirectory>` child elements. Each child element specifies the configuration settings for a particular virtual directory.

Listing 4-24: The `<application>` Element

```
<configuration>
  <applicationHost>
    . . .
    <sites>
      <site name="" id="" serverAutoStart="">
        <bindings>
          <binding protocol="" bindingInformation="" />
        </bindings>
        <application path="" applicationPool="" enabledProtocols="">
          <virtualDirectory path="" physicalPath="" userName="" password="" />
          . . .
        </application>
        . . .
      </site>
      . . .
    </sites>

  </applicationHost>
  . . .
</configuration>
```

The `VirtualDirectory` class of the IIS7 and ASP.NET integrated imperative management API provides programmatic access to the contents of the `<virtualDirectory>` element. This class exposes properties that map to the XML attributes of this element (see Listing 4-25).

Listing 4-25: The VirtualDirectory Class

```
public sealed class VirtualDirectory : ConfigurationElement
{
    // Properties
    public string Password { get; set; }
    public string Path { get; set; }
    public string PhysicalPath { get; set; }
    public string UserName { get; set; }
}
```

VirtualDirectoryCollection

The `VirtualDirectoryCollection` class, like any other collection class in this API, inherits the `ConfigurationElementCollectionBase` class and adds a method named `Add` and an indexer. The `Add` method creates a `VirtualDirectory` object with the specified virtual and physical paths and adds the object to the collection. The indexer allows you to use the virtual path as an index into the collection to access the associated `VirtualDirectory` object. Listing 4-26 presents the `VirtualDirectoryCollection` class.

Listing 4-26: The VirtualDirectoryCollection Class

```
public sealed class VirtualDirectoryCollection :
    ConfigurationElementCollectionBase<VirtualDirectory>
{
    // Methods
    public VirtualDirectory Add(string path, string physicalPath);

    // Properties
    public VirtualDirectory this[string path] { get; }
}
```

Recall from Listing 4-21 that the `Application` class exposes a collection property of type `VirtualDirectoryCollection` named `VirtualDirectories` that acts as a container for the `VirtualDirectory` objects:

```
public sealed class Application : ConfigurationElement
{
    . . .
    public VirtualDirectoryCollection VirtualDirectories { get; }
}
```

ConfigurationSection

The `ConfigurationSection` class allows you to programmatically access and modify a configuration section in a configuration file. As you'll see in Chapter 5, the configuration section is the unit of extension

Chapter 4: Integrated Configuration from Managed Code

in the IIS7 and ASP.NET integrated configuration system. Listing 4-27 presents the properties of the `ConfigurationSection` base class.

Listing 4-27: The `ConfigurationSection` Class

```
public class ConfigurationSection : ConfigurationElement
{
    // Properties
    public OverrideMode OverrideMode { get; set; }
    public string SectionPath { get; }
}
```

The possible values for the `OverrideMode` property are `Allow`, `Deny`, and `Inherit`.

ServerManager

The previous sections discussed which types in the IIS7 and ASP.NET integrated imperative management API represent which XML elements of the IIS7 and ASP.NET integrated configuration system. As you saw, the instances of these types provide programmatic access to the contents of their associated XML elements, which means that you can use them within your C# or Visual Basic code to read from and write to the configuration system. This presupposes that someone loads the content of each XML element into the instance of the appropriate type so you can use the instance to programmatically access the element, but who? Enter the `ServerManager` class.

As Listing 4-28 shows, the methods and properties of the `ServerManager` class allow you to access the objects that provide programmatic access to the contents of the configuration files.

Listing 4-28: The `ServerManager` Class

```
public sealed class ServerManager : IDisposable
{
    // Methods
    public void CommitChanges();
    public Configuration GetAdministrationConfiguration();
    public Configuration GetApplicationHostConfiguration();
    public Configuration GetWebConfiguration(string siteName);
    public Configuration GetWebConfiguration(string siteName, string virtualPath);

    // Properties
    public ApplicationPoolCollection ApplicationPools { get; }
    public SiteCollection Sites { get; }
    public WorkerProcessCollection WorkerProcesses { get; }
}
```

The following table describes the methods of the `ServerManager` class:

Method	Description
<code>GetAdministrationConfiguration</code>	Loads the <code>administration.config</code> file into a <code>Configuration</code> object and returns the object.
<code>GetApplicationHostConfiguration</code>	Loads the <code>applicationHost.config</code> file into a <code>Configuration</code> object and returns the object. You can use this object from within your C# or Visual Basic code to programmatically access the content of the file.
<code>GetWebConfiguration</code>	Loads the configuration file of a specified Web site with the specified virtual path into a <code>Configuration</code> object.
<code>CommitChanges</code>	Commits the changes made in the <code>Configuration</code> object to the underlying configuration file.

`ServerManager` also exposes the following three collection properties:

- ❑ `ApplicationPools`: Gets the `ApplicationPoolCollection` container that contains the `ApplicationPool` objects that represent the application pools running on the Web server.
- ❑ `Sites`: Gets the `SiteCollection` container that contains the `Site` objects that represent the Web sites running on the Web server.
- ❑ `WorkerProcesses`: Gets the `WorkerProcessCollection` container that contains the `WorkerProcess` objects that represent the worker processes running on the Web server.

Putting It All Together

So far you've learned a great deal about the managed classes of the IIS7 and ASP.NET integrated imperative management API. Now it's time to put all that knowledge into practice. In this section you learn how to use these classes to interact with the IIS7 and ASP.NET integrated configuration system programmatically.

In the following sections, I discuss the following four recipes for interacting with the IIS7 and ASP.NET integrated configuration system and examples that use these recipes:

- ❑ Recipe for loading a specified configuration file
- ❑ Recipe for accessing the specified attribute of a specified configuration section in the `administration.config`, `applicationHost.config`, `site web.config`, `application web.config`, or virtual directory `web.config` file
- ❑ Recipe for adding or removing an element from the specified collection element of a specified configuration section in the `administration.config`, `applicationHost.config`, `site web.config`, `application web.config`, or virtual directory `web.config` file
- ❑ Recipe for accessing the configuration sections in the `<system.applicationHost>` section group of the `applicationHost.config` file

Recipe for Loading a Specified Configuration File

Follow these steps to load a specified configuration file into a `Configuration` object:

1. Import the `Microsoft.Web.Administration.dll` assembly from the following directory on your machine:

```
%WINDIR%\System32\InetSrv
```

2. Import the `Microsoft.Web.Administration` namespace:

```
using Microsoft.Web.Administration;
```

3. Instantiate an instance of the `ServerManager` class:

```
ServerManager mgr = new ServerManager();
```

4. Call the appropriate `GetXXXConfiguration` (where `XXX` is the placeholder for `Administration`, `ApplicationHost`, or `Web`) method to load the desired configuration file into a `Configuration` object:

- ❑ Call the `GetAdministrationConfiguration` method if you need to access the `administration.config` file:

```
Configuration config = mgr.GetAdministrationConfiguration();
```

- ❑ Call the `GetApplicationHostConfiguration` method if you need to access the `applicationHost.config` file:

```
Configuration config = mgr.GetApplicationHostConfiguration();
```

- ❑ Call the `GetWebConfiguration` method if you need to access a site, application, or virtual directory `web.config` file:

```
Configuration config = mgr.GetWebConfiguration();
```

Recipe for Accessing the Specified Attribute of a Specified Configuration Section

Follow these steps to get and set the attribute values of a specified configuration section in the `administration.config`, `applicationHost.config`, `site web.config`, `application web.config`, or virtual directory `web.config` file:

1. Use the steps under “Recipe for Loading a Specified Configuration File” to load the configuration file into a `Configuration` object.
2. Call the `GetSection` method of the `Configuration` object to access the `ConfigurationSection` object that represents the configuration section with the specified location path (such as `system.web/compilation`):

```
ConfigurationSection section = config.GetSection("locationPath")
```

3. Use the `GetAttributeValue` method of the `ConfigurationSection` object to access the value of a specified attribute:

```
Object attrVal = section.GetAttributeValue("AttrName");
```

4. Use the `SetAttributeValue` method of the `ConfigurationSection` object to set the value of a specified attribute:

```
Section.SetAttributeValue("AttrName", attrValue);
```

5. Call the `CommitChanges` method on the `ServerManager` object to commit the changes. Recall that the `ServerManager` object loads the configuration file into a `Configuration` object, which is an in-memory entity. In other words, all the changes that you make using the IIS7 and ASP.NET integrated imperative management API are made to the in-memory representation of the underlying configuration file, not the file itself. Calling the `CommitChanges` method commits the changes to the file itself.

```
mgr.CommitChanges();
```

Here is an example. Launch Visual Studio, add a new console application, add a reference to the `Microsoft.Web.Administration.dll` assembly, and add the code shown in Listing 4-29 to the `Program.cs` file (when you add a console application, VS automatically adds this file to your project).

Listing 4-29: The `Program.cs` File

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration config = mgr.GetApplicationHostConfiguration();
        ConfigurationSection section =
            config.GetSection("system.webServer/defaultDocument");
        bool enabled = (bool)section.GetAttributeValue("enabled");
        section.SetAttributeValue("enabled", !enabled);
        mgr.CommitChanges();
    }
}
```

Run the program and open the `applicationHost.config` file. The result should look like the following:

```
<configuration>
  <system.webServer>
    . . .
    <defaultDocument enabled="false">
      . . .
    </defaultDocument>
```

Recipe for Adding or Removing an Element from the Specified Collection Element of a Specified Configuration Section

Follow these steps to add or remove an element from the specified collection element of a specified configuration section in the `administration.config`, `applicationHost.config`, `site.web.config`, `application.web.config`, or virtual directory `web.config` file:

1. Use the steps under “Recipe for Loading a Specified Configuration File” to load the configuration file into a `Configuration` object.
2. Call the `GetSection` method of the `Configuration` object to access the `ConfigurationSection` object that represents the specified configuration section.
3. Call the `GetCollection` method on the `ConfigurationSection` object to access the `ConfigurationElementCollection` object that represents the specified collection element:

```
ConfigurationElementCollection col =  
    section.GetCollection("CollectionElementName");
```

4. Call the `AllowsAdd` or `AllowsRemove` property of the `ConfigurationElementCollection` object to ensure that the object allows adding or removing elements.
5. Follow these steps to add an element to the `ConfigurationElementCollection` collection:
 - a. Call the `CreateElement` method of the `ConfigurationElementCollection` object to create a new `ConfigurationElement` object with the specified name.
 - b. Call the `SetAttributeValue` method on the `ConfigurationElement` object as many times as necessary to specify the attributes of the element.
 - c. Use the `Add` method of the `ConfigurationElementCollection` object to add the `ConfigurationElement` object to the collection.
6. Follow these steps to remove a specified element from the `ConfigurationElementCollection` collection:
 - a. Access the `ConfigurationElement` object with the specified name.
 - b. Call the `Remove` method on the `ConfigurationElementCollection` object to remove the `ConfigurationElement` object from the collection.
7. Call the `CommitChanges` method on the `ServerManager` object to commit the changes to the configuration file that the `Configuration` object represents.

Here is an example for adding an element to a collection element. Add a new console application in Visual Studio, add a reference to the `Microsoft.Web.Administration.dll` assembly, and add the code shown in Listing 4-30 to the `Program.cs` file.

Listing 4-30: Adding a New Element

```
using System;  
using Microsoft.Web.Administration;
```

Listing 4-30: (continued)

```
class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration config = mgr.GetApplicationHostConfiguration();
        ConfigurationSection section =
            config.GetSection("system.webServer/defaultDocument");
        ConfigurationElementCollection col = section.GetCollection("files");
        if (col.AllowsAdd)
        {
            ConfigurationElement addElement = col.CreateElement("add");
            addElement.SetAttributeValue("value", "Home.aspx");
            col.Add(addElement);
            mgr.CommitChanges();
        }
    }
}
```

Run the program and open the `applicationHost.config` file. As the boldfaced portion of the following code listing shows, the program adds a new `<add>` element with the `value` attribute value of `Home.aspx` to the `<files>` collection element of the `<defaultDocument>` configuration section:

```
<configuration>
  <system.webServer>
    <defaultDocument enabled="false">
      <files>
        . . .
        <add value="Home.aspx" />
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>
```

Now let's take a look at an example that removes an element from a collection element. Add a new console application in Visual Studio as usual, add a reference to the `Microsoft.Web.Administration.dll` assembly, and add the code presented in Listing 4-31 to the `Program.cs` file.

Listing 4-31: Removing an Element

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration config = mgr.GetApplicationHostConfiguration();
```

(continued)

Listing 4-31: *(continued)*

```
ConfigurationSection section =
    config.GetSection("system.webServer/defaultDocument");
ConfigurationElementCollection col = section.GetCollection("files");
if (col.AllowsRemove)
{
    ConfigurationElement addElement1 = null;
    foreach (ConfigurationElement addElement2 in col)
    {
        if (addElement2.GetAttributeValue("value").Equals("Home.aspx"))
        {
            addElement1 = addElement2;
            break;
        }
    }

    if (addElement1 != null)
    {
        col.Remove(addElement1);
        mgr.CommitChanges();
    }
}
}
```

Run the program and open the `applicationHost.config` file as usual. The `<add>` element with the value attribute value of `Home.aspx` should not be there anymore.

Recipe for Accessing the Configuration Sections in the <system.applicationHost> Section Group

As discussed earlier, the configuration sections in the `<system.applicationHost>` section group can only be used in the `applicationHost.config` file. In this section I use bunch of examples to show you how to use the managed classes of the IIS7 and ASP.NET integrated imperative management API to programmatically manage the configuration sections in `<system.applicationHost>`.

Adding an Application Pool

In this first example, you add a new application pool named `MyAppPool` to the Web server. Add a console application in Visual Studio as usual, add a reference to the `Microsoft.Web.Administration.dll` assembly, and add the following code to the `Program.cs` file:

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
```

```
{
    ServerManager mgr = new ServerManager();
    ApplicationPool myAppPool = mgr.ApplicationPools.Add("MyAppPool");
    myAppPool.AutoStart = true;
    myAppPool.Cpu.Action = ProcessorAction.KillW3wp;
    myAppPool.ManagedPipelineMode = ManagedPipelineMode.Integrated;
    myAppPool.ManagedRuntimeVersion = "V2.0";
    myAppPool.ProcessModel.IdentityType = ProcessModelIdentityType.NetworkService;
    myAppPool.ProcessModel.IdleTimeout = TimeSpan.FromMinutes(2);
    myAppPool.ProcessModel.MaxProcesses = 1;
    mgr.CommitChanges();
}
}
```

Run the program and open the `applicationHost.config` file. It should include the `<add>` element shown in boldface in the following code listing:

```
<configuration>
  <system.applicationHost>
    <applicationPools>
      <add name="DefaultAppPool" />
      <add name="Classic .NET AppPool" managedPipelineMode="Classic" />
      <add name="MyAppPool" autoStart="true" managedRuntimeVersion="V2.0"
        managedPipelineMode="Integrated">
        <processModel identityType="NetworkService" idleTimeout="00:02:00"
          maxProcesses="1" />
        <cpu action="KillW3wp" />
      </add>
      . . .
    </applicationPools>
  </system.applicationHost>
</configuration>
```

Adding a Web Site

The next example adds a new Web site named `MyWebSite` to the Web server:

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        mgr.Sites.Add("MyWebSite", "http", "198.1.1.0:80:",
            @"d:\MyWebSiteRootWebAppRootVirDir");
        mgr.CommitChanges();
    }
}
```


Chapter 4: Integrated Configuration from Managed Code

You should see the boldfaced `<site>` element shown in the following listing in the `applicationHost.config` file:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="MyWebSite" id="2">
        <application path="/">
          <virtualDirectory path="/"
            physicalPath="d:\MyWebSiteRootWebAppRootVirDir" />
        </application>
      <bindings>
        <binding protocol="http" bindingInformation="198.1.1.0:80:" />
      </bindings>
    </site>
  </sites>
</system.applicationHost>
</configuration>
```

Notice that the `Add` method of the `Sites` property of the `ServerManager` object adds a new Web site that contains a root Web application with the virtual path of `"/`", which in turn contains a root virtual directory with a virtual path of `"/`" and a physical path of `d:\MyWebSiteRootWebAppRootVirDir`. The `Add` method also adds a binding for the Web site. This binding specifies that clients must use HTTP as the transport protocol to access the `MyWebSite` Web site. It also specifies that clients must access this Web site at the IP address of `198.1.1.0` and port number of `80`.

Adding a Binding

The next example shows how to add a new binding to the `MyWebSite` Web site:

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Site myWebSite = mgr.Sites["MyWebSite"];
        myWebSite.Bindings.Add("192.168.254.1:80:", "http");
        mgr.CommitChanges();
    }
}
```

You should see the boldfaced portion of the following listing in the `applicationHost.config` file:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="MyWebSite" id="2">
        <application path="/"
          <virtualDirectory path="/"
```

```
        physicalPath="d:\MyWebSiteRootWebAppRootVirDir" />
    </application>
    <bindings>
        <binding protocol="http" bindingInformation="198.1.1.0:80:" />
        <binding protocol="http" bindingInformation="192.168.254.1:80:" />
    </bindings>
</site>
</sites>
</system.applicationHost>
</configuration>
```

The same Web site can be accessed via different transport protocols, IP addresses, and port numbers. This is a very important feature of the IIS7 architecture. As you'll see later in this book, this allows you to host the Windows Communications Foundation services in IIS7 allowing clients to communicate with the same service via different transport protocols such as HTTP, NET.TCP, NET.PIPE, and .NET.MSMQ.

Adding a Web Application

The next example shows you how to add a new Web application to the `MyWebSite` Web site:

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Site myWebSite = mgr.Sites["MyWebSite"];
        Application myApp = myWebSite.Applications.Add("/MyWebApp",
                                                    @"d:\MyWebSiteDir\MyWebAppRootVirDir");
        myApp.ApplicationPoolName = "MyAppPool";
        mgr.CommitChanges();
    }
}
```

You should see the boldfaced portion shown in the following code listing in the `applicationHost.config` file:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="MyWebSite" id="2">
        <application path="/">
          <virtualDirectory path="/" physicalPath="d:\MyWebSiteDir" />
        </application>
        <bapplication path="/MyWebApp" applicationPool="MyAppPool">
          <virtualDirectory path="/"
            physicalPath="d:\MyWebSiteDir\MyWebAppRootVirDir" />
        </bapplication>
      </sites>
      <bindings>
        <binding protocol="http" bindingInformation="198.1.1.0:80:" />
```

```
</bindings>
</site>
</sites>
</system.applicationHost>
</configuration>
```

The Add method added a new Web application with virtual path of /MyWebApp that belongs to the MyAppPool application pool. Note that the Add method has automatically added a root virtual directory to the newly created application.

Adding a Virtual Directory

Now add a new virtual directory to the MyWebApp Web application:

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Site myWebSite = mgr.Sites["MyWebSite"];
        Application myWebApp = myWebSite.Applications["/MyWebApp"];
        myWebApp.VirtualDirectories.Add("/MyVirDir",
                                         @"d:\MyWebSiteDir\MyWebAppRootVirDir");
        mgr.CommitChanges();
    }
}
```

After running this program, the applicationHost.config file should look like the following:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="MyWebSite" id="2">
        <application path="/">
          <virtualDirectory path="/" physicalPath="d:\MyWebSiteDir" />
        </application>
        <application path="/MyWebApp" applicationPool="MyAppPool">
          <virtualDirectory path="/"
            physicalPath="d:\MyWebSiteDir\MyWebAppRootVirDir" />
          <virtualDirectory path="/MyVirDir"
            physicalPath="d:\MyWebSiteDir\MyWebAppRootVirDir" />
        </application>
        <bindings>
          <binding protocol="http" bindingInformation="198.1.1.0:80:" />
        </bindings>
      </site>
    </sites>
  </system.applicationHost>
</configuration>
```

This example also reveals a very interesting fact about virtual directories: You can have two or more different virtual directories pointing to the same physical path.

Summary

In this chapter you learned a great deal about the IIS7 and ASP.NET integrated imperative management API and how to use it to access and/or modify the contents of the configuration files. The next chapter shows you how to extend the schema of the IIS7 and ASP.NET integrated configuration system to add support for your own custom configuration sections.

Extending the Integrated Configuration System and Imperative Management API

The previous chapters provided in-depth coverage of the following four important components of the IIS7 and ASP.NET integrated infrastructure:

- ❑ IIS7 and ASP.NET integrated configuration system
- ❑ IIS7 and ASP.NET integrated graphical management system (IIS7 Manager)
- ❑ IIS7 and ASP.NET integrated imperative management system
- ❑ IIS7 and ASP.NET integrated request processing pipeline

As discussed, the modular architecture of the IIS7 and ASP.NET integrated request processing pipeline allows you to plug your own custom feature modules into the integrated pipeline to add support for new custom processing capabilities. Adding a new custom feature module also requires you to:

- ❑ Extend the IIS7 and ASP.NET integrated configuration system to add support for a new configuration section to allow the clients of your custom feature module to configure the module from configuration files
- ❑ Extend the IIS7 and ASP.NET integrated graphical management system (IIS7 Manager) to add support for new graphical components to allow the clients of your custom feature module to configure the module from the IIS7 Manager
- ❑ Extend the IIS7 and ASP.NET integrated imperative management system to add support for new managed classes that allow the clients of your custom feature module to configure the module from their C# or Visual Basic code in a strongly-typed fashion.

Chapter 5: Extending the Integrated Configuration System

Therefore, extending the IIS7 and ASP.NET integrated infrastructure requires you to have a solid understanding of the following four extensibility models:

- ❑ IIS7 and ASP.NET integrated configuration extensibility model
- ❑ IIS7 and ASP.NET integrated graphical management extensibility model
- ❑ IIS7 and ASP.NET integrated imperative management extensibility model
- ❑ IIS7 and ASP.NET integrated request processing pipeline extensibility model

An extensibility model provides you with the necessary infrastructure and tools to add support for new extensions.

IIS7 and ASP.NET Integrated Configuration Extensibility Model

One of the great things about the new IIS7 and ASP.NET integrated configuration system is that all the configuration information is stored in XML files known as configuration files. The unit of extensibility in the new unified configuration system is known as a configuration section. A configuration section normally contains configuration settings for a particular feature module at a particular level of the configuration hierarchy.

For example, the `<defaultDocument>` configuration section of the `applicationHost.config` file allows you to configure the `DefaultDocumentModule` feature module for all the sites, applications, and virtual directories running on the Web server. The `<defaultDocument>` configuration section of the `web.config` file of a given Web site, on the other hand, allows you to configure the `DefaultDocumentModule` feature module only for the applications and virtual directories running in that Web site.

As discussed in the previous chapters, the extensibility model of the IIS7 and ASP.NET integrated request processing pipeline allows you to plug your own custom feature modules into the pipeline to extend its request processing capabilities. These custom feature modules need to be configured just like the standard feature modules. This means that every time you extend the integrated pipeline to plug in a new custom feature module, you have to also extend the integrated configuration system to plug in a new custom configuration section for your custom feature module to allow page developers to configure your module from configuration files.

Because configuration files are XML documents, extending the integrated configuration system means extending the XML schema of the system. The IIS7 and ASP.NET integrated infrastructure comes with an XML markup language that allows you to extend the schema of the integrated configuration system in a declarative fashion.

This declarative schema extension capability is in contrast to the traditional imperative or programmatic schema extension capability that ASP.NET developers have come to know. The traditional schema extensibility model requires you to write procedural code to extend the schema. The IIS7 and ASP.NET integrated declarative schema extension model allows you to extend the schema without writing a single line of procedural code. This is welcoming news for ASP.NET developers, allowing them to extend the schema with minimal time and effort.

IIS7 and ASP.NET Integrated Declarative Schema Extension Markup Language

The IIS7 and ASP.NET integrated declarative schema extension markup language, like any other markup language, consists of XML elements and attributes. These XML elements and attributes allow you to define the XML elements and attributes that make up a configuration section.

Every configuration section consists of one or more XML elements and their XML attributes. These XML elements fall in the following categories:

- ❑ **Containing or outermost XML element:** This XML element contains the rest of the XML elements that make up the configuration section. Obviously the name of the containing XML element varies from one configuration section to another.
- ❑ **Collection XML elements:** Each Collection XML element represents a collection of items. For example, as you'll see later, the `<defaultDocuments>` configuration section contains a Collection XML element named `<files>` that represents a collection of default documents.
- ❑ The name of a Collection XML element may vary from one configuration section to another. For example, as you'll see later, the Collection XML element of the `<defaultDocuments>` configuration section is an element named `<files>`, whereas the Collection XML element of the `<requestFiltering>` configuration section is an element named `<fileExtensions>`.
- ❑ The same configuration section may have more than one Collection XML element. For example, as you'll see later, the `<requestFiltering>` configuration section contains two Collection XML elements named `<fileExtensions>` and `<hiddenSegments>`.
- ❑ Every Collection XML element contains the following child elements:
 - ❑ **Add child elements:** Each Add child element of a Collection XML element adds a new item to the collection of items that the Collection XML element represents. For example, the `<files>` Collection XML element of the `<defaultDocuments>` configuration section contains one or more Add child elements named `<add>` where each `<add>` child element adds a default document to the collection of documents that the `<files>` Collection XML element represents.
 - ❑ The name of an Add child element may vary from one type of Collection XML element to another. For example, as you'll see later, the Add child element of the `<files>` Collection XML element of the `<defaultDocuments>` configuration section is an element named `<mimeType>`, whereas the Add child element of the `<fileExtensions>` Collection XML element of the `<requestFiltering>` configuration section is an element named `<add>`.
 - ❑ **Remove child elements:** Each Remove child element removes an existing item from the collection of items that the Collection XML element represents. For example, the `<files>` Collection XML element of the `<defaultDocuments>` configuration section may contain one or more Remove child elements named `<remove>` where each `<remove>` child element removes a default document from the collection of documents that the `<files>` Collection XML element represents.

Chapter 5: Extending the Integrated Configuration System

- ❑ The name of the Remove child element may vary from one type of Collection XML element to another. In other words, the Remove child element does not have to be named `<remove>`. It can be named `<delete>` or anything else you wish.
- ❑ **Clear child element:** The Clear child element clears the collection of items that the Collection XML element represents. Again, the name of the Clear child element may vary from one type of Collection XML element to another. In other words, the Clear child element does not have to be named `<clear>`. It can be named anything you wish.
- ❑ **Non-collection XML elements:** These are XML elements that do not represent collections of items. The name of a non-collection XML element may vary from one configuration section to another.

Let's look at a few examples from the `applicationHost.config` file. Listing 5-1 shows the first example.

Listing 5-1: The `<defaultDocument>` Configuration Section

```
<configuration>
  <system.webServer>
    <defaultDocument enabled="false">
      <files>
        <add value="Default.htm" />
        <add value="Default.asp" />
        <add value="index.htm" />
        <add value="index.html" />
        <add value="iisstart.htm" />
        <add value="default.aspx" />
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>
```

Now I'll identify the previously mentioned three types of XML elements making up the `<defaultDocument>` configuration section:

- ❑ **Containing XML element:** As Listing 5-1 clearly shows, the `<defaultDocument>` XML element contains the rest of the XML elements and attributes that make up this configuration section.
- ❑ **Collection XML element:** The `<files>` XML element represents a collection of files or documents. Each `<add>` element adds a new document to this collection of documents.
- ❑ **Non-collection XML elements:** The `<documentDefault>` configuration section does not contain any non-collection XML elements.

As this example shows, a configuration section is not required to contain all three types of XML elements. As a matter of fact, you can have a configuration section that contains only the Containing XML element.

The next example is also an excerpt from the `applicationHost.config` file, as shown in Listing 5-2.

Listing 5-2: The <staticContent> Configuration Section

```
<configuration>
  <system.webServer>
    <staticContent isDocFooterFileName="false" enableDocFooter="false">
      <mimeTypeMap fileExtension=".323" mimeType="text/h323" />
      <mimeTypeMap fileExtension=".aaf" mimeType="application/octet-stream" />
      <mimeTypeMap fileExtension=".aca" mimeType="application/octet-stream" />
      <mimeTypeMap fileExtension=".accdb" mimeType="application/msaccess" />
      <mimeTypeMap fileExtension=".accde" mimeType="application/msaccess" />
      <mimeTypeMap fileExtension=".accdt" mimeType="application/msaccess" />
      <mimeTypeMap fileExtension=".afm" mimeType="application/octet-stream" />
      <clientCache cacheControlMode=="NoControl" />
    </staticContent>
  </system.webServer>
</configuration>
```

Now I'll identify the three types of XML elements that make up the <staticContent> configuration section:

- ❑ **Containing XML element:** You guessed it — the <staticContent> element is the containing XML element in this case.
- ❑ **Collection XML elements:** The <staticContent> XML element itself also acts as the Collection XML element, which represents a collection of MIME mappings. The <mimeTypeMap> XML element in this case acts as the Add child element because it adds a new MIME mapping to the collection. As you can see, the Add child element does not have to be named <add>. It can be named anything you wish as long as it adds an item to the collection that the associated Collection XML element represents.
- ❑ **Non-collection elements:** The <clientCache> XML element is a non-collection XML element in this case because it does not represent a collection of items.

The IIS7 and ASP.NET integrated declarative schema extension markup language contains the XML elements and attributes that you need to describe the XML elements and attributes that make up your configuration section, as discussed in the following sections.

<sectionSchema>

Use the <sectionSchema> element to define the containing XML element of your configuration section. This element exposes an attribute named `name` that you must set to the fully qualified name of the containing XML element, including its entire section group hierarchy. For example, consider the <basicAuthentication> configuration section presented in the following listing:

```
<configuration>
  <system.webServer>
    <security>
      <authentication>
        <basicAuthentication ... />
      </authentication>
    </security>
  </system.webServer>
</configuration>
```

Chapter 5: Extending the Integrated Configuration System

Here is how this configuration section is defined:

```
<sectionSchema name="system.webServer/security/authentication/basicAuthentication">
    ...
</sectionSchema>
```

As you can see, the name attribute of the `<sectionSchema>` element has been set to a value that contains the entire section group hierarchy that the `<basicAuthentication>` configuration section belongs to.

<attribute>

You use the `<attribute>` element to define the XML attributes of the elements that make up your configuration section. The `<attribute>` element exposes the following XML attributes:

- ❑ `name`: Specifies the name of the XML attribute being defined.
- ❑ `type`: Specifies the data type of the XML attribute being defined.
- ❑ `defaultValue`: Specifies the default value of the XML attribute being defined.

Have another look at the `<basicAuthentication>` configuration section:

```
<configuration>
  <system.webServer>
    <security>
      <authentication>
        <basicAuthentication enabled="false" logonMethod="ClearText" />
      </authentication>
    </security>
  </system.webServer>
</configuration>
```

Here is how the `enabled` and `logonMethod` attributes of the `<basicAuthentication>` configuration section are defined:

```
<sectionSchema name="system.webServer/security/authentication/basicAuthentication">
  <attribute name="enabled" type="bool" defaultValue="false" />

  <attribute name="logonMethod" type="enum" defaultValue="ClearText">
    <enum name="Interactive" value="0" />
    <enum name="Batch" value="1" />
    <enum name="Network" value="2" />
    <enum name="ClearText" value="3" />
  </attribute>
</sectionSchema>
```

Note that if the attribute being defined is an enumeration, the `<attribute>` element contains one `<enum>` child element for each enumeration value. The `<enum>` element exposes two attributes: `name`, which contains the name of the enumeration, and `value`, which contains the integer number that the enumeration references.

<element>

Use the <element> element without the <collection> element to define a non-collection XML element of a configuration section. I discuss the <collection> element shortly. The <element> element exposes an attribute named `name` that must be set to the name of the XML element being defined. For example, consider the following <staticContent> configuration section:

```
<configuration>
  <system.webServer>
    <staticContent isDocFooterFileName="false" enableDocFooter="false">
      <mimeMap fileExtension=".323" mimeType="text/h323" />
      <mimeMap fileExtension=".aaf" mimeType="application/octet-stream" />
      <clientCache cacheControlMode="NoControl" />
    </staticContent>
  </system.webServer>
</configuration>
```

The <clientCache> non-collection XML element (shown in bold in the previous code) is defined as follows:

```
<sectionSchema name="staticContent">
  <element name="clientCache">
    <attribute name="cacheControlMode" type="enum" defaultValue="NoControl">
      <enum name="NoControl" value="0" />
      <enum name="DisableCache" value="1" />
      <enum name="UseMaxAge" value="2" />
      <enum name="UseExpires" value="3" />
    </attribute>
  </element>
</sectionSchema>
```

<collection>

Use the <element> and <collection> elements together to define a Collection element of a configuration section. The <collection> element exposes the following XML attributes:

- ❑ `addElement`: Specifies the name of the Add child element of the <collection> element. Recall that the Add child element adds a new item to the collection of items.
- ❑ `removeElement`: Specifies the name of the Remove child element of the <collection> element. Recall that the Remove child element removes an item from the collection of items.
- ❑ `clearElement`: Specifies the name of the Clear child element of the <collection> element. Recall that the Clear child element clears the collection of items.

The <collection> element also contains one or more <attribute> elements, each of which defines a particular attribute of the Add child element.

For example, consider the <requestFiltering> configuration section:

```
<configuration>
  <system.webServer>
    <security>
```

```
<requestFiltering>
  <fileExtensions allowUnlisted="true">
    <add fileExtension=".asax" allowed="false" />
    <add fileExtension=".ascx" allowed="false" />
    <add fileExtension=".master" allowed="false" />
    <add fileExtension=".skin" allowed="false" />
    <add fileExtension=".browser" allowed="false" />
    <add fileExtension=".sitemap" allowed="false" />
    ...
  </fileExtensions>
  <hiddenSegments>
    <add segment="web.config" />
    <add segment="bin" />
    <add segment="App_code" />
    <add segment="App_GlobalResources" />
    <add segment="App_LocalResources" />
    ...
  </hiddenSegments>
</requestFiltering>
</security>
</system.webServer>
</configuration>
```

Note that the `<requestFiltering>` configuration section contains two Collection elements named `<fileExtensions>` and `<hiddenSegments>`. The following code listing shows the definition of the `<fileExtensions>` Collection element:

```
<sectionSchema name="system.webServer/security/requestFiltering">
  <element name="fileExtensions">
    <attribute name="allowUnlisted" type="bool" defaultValue="true" />
    <collection addElement="add" clearElement="clear" removeElement="remove" >

      <attribute name="fileExtension" type="string" required="true"
        isUniqueKey="true" validationType="nonEmptyString" />

      <attribute name="allowed" type="bool" required="true" defaultValue="true" />
    </collection>
  </element>
</sectionSchema>
```

Follow these steps to define a Collection element of your configuration section:

1. Use the `<element>` element and set its name attribute to the name of the Collection element being defined:

```
<element name="fileExtensions">
```

2. Use the `<attribute>` element to define the attributes of the Collection element itself:

```
<attribute name="allowUnlisted" type="bool" defaultValue="true" />
```

3. Use the `<collection>` element and set its `addElement`, `clearElement`, and `removeElement` attributes to the names of the Add, Remove, and Clear child elements:

```
<collection addElement="add" clearElement="clear" removeElement="remove" >
```

4. Use the `<attribute>` element inside the `<collection>` element to define the attributes of the Add child elements:

```
<collection addElement="add" clearElement="clear" removeElement="remove" >
  <attribute name="fileExtension" type="string" required="true"
    isUniqueKey="true" validationType="nonEmptyString" />

  <attribute name="allowed" type="bool" required="true" defaultValue="true" />
</collection>
```

As the boldfaced portion of this code snippet shows, at least one of the `<attribute>` elements must have an `isUniqueKey` attribute assigned with a value of `true`, to indicate that the attribute serves as a key for retrieving items from the corresponding collection. For example, the preceding code snippet has set the `isUniqueKey` attribute of the `<attribute>` element that defines the `fileExtension` attribute to `true`. This specifies the `fileExtension` attribute of the file extension item that the `<add>` element adds to the collection of the file extensions as the identifier of the file extension item. Having an identifier attribute allows the clients of your custom configuration section to use the Remove child element to remove an item with the specified identifier from the collection.

This seems to suggest something like the following, which looks like a silly thing to do. Why would you want to use the `<add>` element to add a new file extension and then use the `<remove>` element to remove it as shown in the boldfaced portions?

```
<configuration>
  <system.webServer>
    <security>
      <requestFiltering>
        <fileExtensions allowUnlisted="true">
          <add fileExtension=".asax" allowed="false" />
          <add fileExtension=".ascx" allowed="false" />
          <add fileExtension=".master" allowed="false" />
          <remove fileExtension=".asax" />
          . . .
        </fileExtensions>
        . . .
      </requestFiltering>
    </security>
  </system.webServer>
</configuration>
```

You're right. It doesn't make any sense to use both `<add>` and `<remove>` in the same configuration file. However, it makes lot of sense to use `<add>` in one configuration file to add a file extension to the collection of file extensions, and use `<remove>` in a lower-level configuration file to remove the file extension from the collection for that particular level and its sublevels. In other words, removing an item removes the item from the collection only for a particular hierarchy level (and its sublevels), and has no effect on the higher levels.

Chapter 5: Extending the Integrated Configuration System

Now take a look at the schema for the `<staticContent>` configuration section presented in Listing 5-2:

```
<sectionSchema name="system.webServer/staticContent">
  <collection addElement="mimeMap" clearElement="clear" removeElement="remove">
    <attribute name="fileExtension" type="string" required="true"
      isUniqueKey="true" validationType="nonEmptyString" />
    <attribute name="mimeType" type="string" required="true"
      validationType="nonEmptyString" />
  </collection>
  . . .
</sectionSchema>
```

As mentioned before, in this case the `<staticContent>` element itself is the Collection element. That is why the `<collection>` element in this code listing is not contained in an `<element>` element. Another interesting point is that the `addElement` attribute of the `<collection>` element is set to `mimeMap`. This means that the child element of your Collection element does not have to be an `<add>` element. It could be any element that performs the add operation, such as the `<mimeType>` element, which adds a new MIME mapping to the `<staticContent>` collection.

Adding a Custom Configuration Section

The previous sections covered the XML elements and attributes that make up the new IIS7 and ASP.NET integrated declarative schema extension markup language. In this section you learn how to use this markup language to define the XML elements and attributes that make up a configuration section.

Follow these steps to extend the IIS7 and ASP.NET integrated configuration system to add support for your own custom configuration section:

1. Write down the configuration section, including all its XML elements and attributes.
2. Identify the following portions of the configuration section:
 - ☐ The containing XML element, and the names, data types, and default values of its attributes
 - ☐ The non-collection XML elements, and the names, data types, and default values of their attributes
 - ☐ The Collection XML elements, and the names, data types, and default values of their attributes
 - ☐ The child elements of each Collection element that perform the add, remove, and clear operations, and the names, data types, and default values of their attributes
3. Create a new XML file in the following directory on your machine:

```
%WINDIR%\system32\inetsrv\config\schema
```

By convention the name of this XML file consists of two parts separated by the underscore (`_`) character. The second part is always the word “schema,” and the first part is whatever name makes sense in the case of your custom configuration section. The first part is normally in capital letters. As a matter of fact, if you check out this schema directory, you’ll notice it contains two XML files named `IIS_schema.xml` and `ASPNET_schema.xml`, which contain the schemas for

the IIS and ASP.NET configuration sections. You should not add your own schema to either of these files. Instead you should add a new XML file to the schema directory.

4. Decide on the section group hierarchy where you want to add your configuration section.
5. Use the IIS7 and ASP.NET integrated declarative schema extension markup language discussed in the previous sections to implement the schema that defines the XML elements and attributes that make up your custom configuration section.
6. Register your custom configuration section with the `<configSections>` section of the `applicationHost.config` file.

Next, you use this recipe to extend the IIS7 and ASP.NET integrated configuration system to add support for a custom configuration section named `myConfigSection`. The first step requires you to write down a representative implementation of your configuration section, as shown in Listing 5-3.

Listing 5-3: The `<myConfigSection>` Configuration Section

```
<myConfigSection myConfigSectionBoolAttr="" myConfigSectionEnumAttr="">
  <myNonCollection myNonCollectionTimeSpanAttr="" />
  <myCollection myCollectionIntAttr="">
    <myAdd myCollectionItemBoolAttr="" myCollectionItemIdentifier="" />
    <myRemove myCollectionItemIdentifier="" />
    <myClear/>
  </myCollection>
</myConfigSection>
```

The second step requires you to identify different portions of the configuration section. The `<myConfigSection>` configuration section exposes a Boolean attribute named `myConfigSectionBoolAttr` and an enumeration attribute named `myConfigSectionEnumAttr` with the possible enumeration values of `myConfigSectionEnumVal1`, `myConfigSectionEnumVal2`, and `myConfigSectionEnumVal3`.

The `<myConfigSection>` configuration section contains a non-Collection element named `myNonCollection` that exposes a `TimeSpan` attribute named `myNonCollectionTimeSpanAttr`. This configuration section also contains a Collection element named `<myCollection>` that exposes an integer attribute named `myCollectionIntAttr`. This Collection element contains one or more `<myAdd>` child elements that expose a Boolean attribute named `myCollectionItemBoolAttr` and a string attribute named `myCollectionItemIdentifier`. The Collection element can contain one or more `<myRemove>` child elements.

Following the third step of the recipe, add an XML file named `MY_schema.xml` to the schema directory on your machine.

Next, you need to decide on the section group hierarchy to which you want to add the `<myConfigSection>` configuration section. In this case, add the configuration section to the `<system.webServer>` section group.

Next, you need to use the IIS7 and ASP.NET integrated declarative schema extension markup language to implement the schema for the `myConfigSection` configuration section and store this schema in the `MY_schema.xml` file.

Chapter 5: Extending the Integrated Configuration System

Listing 5-4 presents the content of the `MY_schema.xml` file.

Listing 5-4: The Content of the `MY_schema.xml` File

```
<configSchema>
  <sectionSchema name="system.webServer/myConfigSection">
    <attribute name="myConfigSectionBoolAttr" type="bool" defaultValue="false"/>
    <attribute name="myConfigSectionEnumAttr" type="enum"
      defaultValue="myConfigSectionEnumVal2">
      <enum name="myConfigSectionEnumVal1" value="1"/>
      <enum name="myConfigSectionEnumVal2" value="2"/>
      <enum name="myConfigSectionEnumVal3" value="3"/>
    </attribute>

    <element name="myNonCollection">
      <attribute name="myNonCollectionTimeSpanAttr" type="timeSpan"
        defaultValue="00:01:30"/>
    </element>

    <element name="myCollection">
      <attribute name="myCollectionIntAttr" type="int" defaultValue="5"/>
      <collection addElement="myAdd" removeElement="myRemove"
        clearElement="myClear">
        <attribute name="myCollectionItemBoolAttr" type="bool"
          defaultValue="true"/>
        <attribute name="myCollectionItemIdentifier" type="string"
          defaultValue="myId1" isUniqueKey="true" />
      </collection>
    </element>

  </sectionSchema>
</configSchema>
```

Listing 5-4 uses a `<sectionSchema>` element to define the Containing XML element of the `myConfigSection` configuration section. Note that the `name` attribute of the `<sectionSchema>` element is set to the fully qualified name of the configuration section, including its complete group hierarchy, that is, `system.webServer/myConfigSection`.

```
<sectionSchema name="system.webServer/myConfigSection">
```

The `<sectionSchema>` element contains two `<attribute>` child elements that define the `myConfigSectionBoolAttr` and `myConfigSectionEnumAttr` attributes of the `<myConfigSection>` containing element. Notice that the `name`, `type`, and `defaultValue` attributes of each `<attribute>` element respectively specify the name, type, and default value of the associated attribute.

```
<attribute name="myConfigSectionBoolAttr" type="bool" defaultValue="false"/>
<attribute name="myConfigSectionEnumAttr" type="enum"
  defaultValue="myConfigSectionEnumVal2">
  <enum name="myConfigSectionEnumVal1" value="1"/>
  <enum name="myConfigSectionEnumVal2" value="2"/>
  <enum name="myConfigSectionEnumVal3" value="3"/>
</attribute>
```

Also note that the `<attribute>` element that defines the `myConfigSectionEnumAttr` attribute of the `<myConfigSection>` Containing element contains three `<enum>` child elements. Each child element defines the name and value of a particular member of the enumeration type.

The `<sectionSchema>` element also contains an `<element>` child element with a name attribute value of `myNonCollection` that defines the `myNonCollection` non-collection element of the `<myConfigSection>` configuration section. The `<attribute>` child element of this `<element>` element defines the `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` non-collection element.

```
<element name="myNonCollection">
  <attribute name="myNonCollectionTimeSpanAttr" type="timeSpan"
    defaultValue="00:01:30"/>
</element>
```

Next, Listing 5-4 uses an `<element>` element with the name attribute value of `myCollection` to define the `<myCollection>` Collection element of the `<myConfigSection>` configuration section. As usual, the `<attribute>` child element of this `<element>` element defines the `myCollectionIntAttr` attribute of the `<myCollection>` Collection element:

```
<element name="myCollection">
  <attribute name="myCollectionIntAttr" type="int" defaultValue="5"/>
  <collection addElement="myAdd" removeElement="myRemove"
    clearElement="myClear">
    <attribute name="myCollectionItemBoolAttr" type="bool" defaultValue="true"/>
    <attribute name="myCollectionItemIdentifier" type="string"
      defaultValue="myId1"/>
  </collection>
</element>
```

This `<element>` element also contains a `<collection>` child element because it represents a Collection element. Notice that the `addElement`, `removeElement`, and `clearElement` attributes of this `<collection>` element define the `myAdd`, `myRemove`, and `myClear` child elements of the `<myCollection>` Collection element.

```
<element name="myCollection">
  <attribute name="myCollectionIntAttr" type="int" defaultValue="5"/>
  <collection addElement="myAdd" removeElement="myRemove"
    clearElement="myClear">
    <attribute name="myCollectionItemBoolAttr" type="bool" defaultValue="true"/>
    <attribute name="myCollectionItemIdentifier" type="string"
      defaultValue="myId1" isUniqueKey="true"/>
  </collection>
</element>
```

The `<collection>` element features two child `<attribute>` elements that define the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of the `<myAdd>` child element of the `<myCollection>` Collection element. Note that the `isUniqueKey` attribute of the `<attribute>` element that defines the `myCollectionItemIdentifier` attribute is set to `true` to specify this attribute as the key attribute. Recall that the `Clear` child element, which is the `myClear` element in this case, uses the key attribute to remove an item from the collection.

Chapter 5: Extending the Integrated Configuration System

Finally, you need to register the `<myConfigSection>` configuration section with the `<configSections>` of the `applicationHost.config` file as shown in Listing 5-5.

Listing 5-5: Registering the `<myConfigSection>` Configuration Section

```
<configSections>
  <sectionGroup name="system.webServer">
    <section name="myConfigSection" allowDefinition="Everywhere"
      overrideModeDefault="Allow" />
    . . .
  </sectionGroup>
  . . .
</configSections>
```

Because the `<myConfigSection>` configuration section belongs to the `<system.webServer>` section group, Listing 5-5 registers this configuration section in the `<sectionGroup>` element with the name attribute value of `system.webServer`. You must use a `<section>` element to register your custom configuration section. This element exposes the following three important attributes:

- ❑ **name:** You must set this attribute to the name of the containing element of your configuration section, such as `myConfigSection`.
- ❑ **allowDefinition:** Use this attribute to specify the configuration hierarchy level where your custom configuration section can be used. The possible values of this attribute are `MachineOnly`, `MachineToApplication`, and `Everywhere`. Listing 5-5 sets the `allowDefinition` attribute to `Everywhere` to allow the `<myConfigSection>` configuration section to be used in configuration files at all configuration hierarchy levels.
- ❑ **overrideModeDefault:** Use this attribute to specify whether the lower-level configuration files can override the configuration settings specified in the `applicationHost.config` file. The possible values are `Allow` and `Deny`. This attribute is the magic behind the new IIS7 administration delegation, allowing the machine administrator to decide whether to delegate the administration of a specified configuration section to a lower-level configuration file. Listing 5-5 sets the `overrideModeDefault` attribute to `Allow` to allow lower-level configuration files to reset the configuration settings specified in the `<myConfigSection>` configuration section of the `applicationHost.config` file.

Machine-Level Configuration File

Next, I implement a console application to show you that the steps you took in the previous section have indeed extended the IIS7 and ASP.NET integrated configuration system to add support for the new `<myConfigSection>` configuration section.

Launch Visual Studio and create a new C# console application. Add a reference to the `Microsoft.Web.Administration.dll`, located in the `%WINDIR%\System32\InetSrv` directory on your machine. Finally, edit the `Program.cs` file to import the `Microsoft.Web.Administration` namespace, and replace the contents of the `Program` class with the code shown in Listing 5-6.

Listing 5-6: The Content of the `Program.cs` File

```
using Microsoft.Web.Administration;
using System;
```

Listing 5-6: (continued)

```
class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
        ConfigurationSection myConfigSection =
            appHostConfig.GetSection("system.webServer/myConfigSection");
        myConfigSection.SetAttributeValue("myConfigSectionBoolAttr", true);
        myConfigSection.SetAttributeValue("myConfigSectionEnumAttr",
            "myConfigSectionEnumVal3");

        ConfigurationElement myNonCollection =
            myConfigSection.GetChildElement("myNonCollection");
        myNonCollection.SetAttributeValue("myNonCollectionTimeSpanAttr",
            TimeSpan.FromMinutes(2));

        ConfigurationElementCollection myCollection =
            myConfigSection.GetCollection("myCollection");
        myCollection.SetAttributeValue("myCollectionIntAttr", 100);
        ConfigurationElement myCollectionItem1 = myCollection.CreateElement("myAdd");
        myCollectionItem1.SetAttributeValue("myCollectionItemBoolAttr", true);
        myCollectionItem1.SetAttributeValue("myCollectionItemIdentifier", "myId1");
        myCollection.Add(myCollectionItem1);

        ConfigurationElement myCollectionItem2 = myCollection.CreateElement("myAdd");
        myCollectionItem2.SetAttributeValue("myCollectionItemBoolAttr", false);
        myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId2");
        myCollection.Add(myCollectionItem2);

        mgr.CommitChanges();
    }
}
```

Now run the console application and open the `applicationHost.config` file. The result should look like Listing 5-7.

Listing 5-7: The `applicationHost.config` File

```
<configuration>
  <system.webServer>
    . . .
    <myConfigSection myConfigSectionBoolAttr="true"
      myConfigSectionEnumAttr="myConfigSectionEnumVal3">
      <myNonCollection myNonCollectionTimeSpanAttr="00:02:00" />
      <myCollection myCollectionIntAttr="100">
        <myAdd myCollectionItemBoolAttr="true"
          myCollectionItemIdentifier="myId1" />
        <myAdd myCollectionItemBoolAttr="false"
          myCollectionItemIdentifier="myId2" />
      </myCollection>
    </myConfigSection>
  </system.webServer>
</configuration>
```

Chapter 5: Extending the Integrated Configuration System

As the boldfaced portion of Listing 5-7 shows, the IIS7 and ASP.NET integrated imperative management API has added a new `<myConfigSection>` configuration section to the `<system.webServer>` section group.

Now I'll review Listing 5-6 more closely. The main goal of this code listing is to use the IIS7 and ASP.NET integrated imperative management API to programmatically access and specify the configuration settings of the `<myConfigSection>` configuration section.

The first order of business is to create a `ServerManager` object:

```
ServerManager mgr = new ServerManager();
```

Next, you need to decide on the configuration hierarchy level you want to work with. In this example, you want to work with the server-level configuration hierarchy. That is why Listing 5-6 calls the `GetApplicationHostConfiguration` method on the `ServerManager` object to load the contents of the `applicationHost.config` server-level configuration file into a `Configuration` object:

```
Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
```

Next, you need to call the `GetSection` method on the `Configuration` object to access the `ConfigurationSection` object that allows you to programmatically access and manipulate the configuration settings of the `<myConfigSection>` configuration section:

```
ConfigurationSection myConfigSection =  
    appHostConfig.GetSection("system.webServer/myConfigSection");
```

Next, Listing 5-6 calls the `SetAttributeValue` method twice on the `ConfigurationSection` object to set the values of the `myConfigSectionBoolAttr` and `myConfigSectionEnumAttr` attributes of the `<myConfigSection>` element:

```
myConfigSection.SetAttributeValue("myConfigSectionBoolAttr", true);  
myConfigSection.SetAttributeValue("myConfigSectionEnumAttr",  
    "myConfigSectionEnumVal3");
```

Listing 5-6 then calls the `GetChildElement` method on the `ConfigurationSection` object to access the `ConfigurationElement` object that provides programmatic access to the `<myNonCollection>` non-collection element of the `<myConfigSection>` configuration section. The listing then uses the `SetAttributeValue` method to set the value of the `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` non-collection element:

```
ConfigurationElement myNonCollection =  
    myConfigSection.GetChildElement("myNonCollection");  
myNonCollection.SetAttributeValue("myNonCollectionTimeSpanAttr",  
    TimeSpan.FromMinutes(2));
```

Next Listing 5-6 uses the `GetCollection` method of the `ConfigurationSection` object to return the `ConfigurationElementCollection` object that represents the `<myCollection>` Collection element of the `<myConfigSection>` configuration section. It then calls the `SetAttributeValue` method to set the value of the `myCollectionAttr` attribute of the `<myCollection>` Collection element:

```
ConfigurationElementCollection myCollection =
```

```
myConfigSection.GetCollection("myCollection");  
myCollection.SetAttributeValue("myCollectionIntAttr", 100);
```

Listing 5-6 then calls the `CreateElement` method on the `ConfigurationElementCollection` object to create a new `<myAdd>` element and uses the `SetAttributeValue` method twice to set the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of this `<myAdd>` element. Note that Listing 5-6 uses the `Add` method of the `ConfigurationElementCollection` object to add the `ConfigurationElement` object to this collection. In other words, the `CreateElement` method creates the `ConfigurationElement` object, but it doesn't add the object to the collection:

```
ConfigurationElement myCollectionItem1 =  
    myCollection.CreateElement("myAdd");  
myCollectionItem1.SetAttributeValue("myCollectionItemBoolAttr", true);  
myCollectionItem1.SetAttributeValue("myCollectionItemIdentifier", "myId1");  
myCollection.Add(myCollectionItem1);
```

Listing 5-6 repeats the same steps to add another `<myAdd>` Add child element to the `<myCollection>` Collection element:

```
ConfigurationElement myCollectionItem2 =  
    myCollection.CreateElement("myAdd");  
myCollectionItem2.SetAttributeValue("myCollectionItemBoolAttr", false);  
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId2");  
myCollection.Add(myCollectionItem2);
```

Finally, Listing 5-6 calls the `CommitChanges` method on the `ServerManager` object to commit these changes to the underlying `applicationHost.config` file. This step is important because the previous changes were all made to the in-memory representation of the `applicationHost.config` file, not the file itself. In other words, first you make all the changes in memory, and then commit them in one shot.

```
mgr.CommitChanges();
```

Site-Level Configuration File

Recall that Listing 5-5 set the `allowDefinition` attribute of the `<section>` element that registers the `<myConfigSection>` configuration section with the `applicationHost.config` file to the `Everywhere` value to allow lower-level configuration files to include the `<myConfigSection>` configuration section. Let's see this in action. Launch Visual Studio and create a new C# console application. Add a reference to the `Microsoft.Web.Administration.dll`, located in the `%WINDIR%\System32\InetSrv` directory on your machine. Finally, edit the `Program.cs` file to import the `Microsoft.Web.Administration` namespace, and replace the contents of the `Program` class with the code shown in Listing 5-8.

Listing 5-8: A Console Application for a Site-Level Configuration

```
using Microsoft.Web.Administration;  
using System;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        ServerManager mgr = new ServerManager();
```

(Continued)

Listing 5-8: (continued)

```
Configuration siteConfig = mgr.GetWebConfiguration("Default Web Site");
ConfigurationSection myConfigSection =
    siteConfig.GetSection("system.webServer/myConfigSection");
myConfigSection.SetAttributeValue("myConfigSectionBoolAttr", false);
myConfigSection.SetAttributeValue("myConfigSectionEnumAttr",
    "myConfigSectionEnumVal3");

ConfigurationElement myNonCollection =
    myConfigSection.GetChildElement("myNonCollection");
myNonCollection.SetAttributeValue("myNonCollectionTimeSpanAttr",
    TimeSpan.FromMinutes(4));

ConfigurationElementCollection myCollection =
    myConfigSection.GetCollection("myCollection");
myCollection.SetAttributeValue("myCollectionIntAttr", 200);
ConfigurationElement myCollectionItem1 =
    myCollection.CreateElement("myAdd");
myCollectionItem1.SetAttributeValue("myCollectionItemBoolAttr", true);
myCollectionItem1.SetAttributeValue("myCollectionItemIdentifier", "myId3");
myCollection.Add(myCollectionItem1);

ConfigurationElement myCollectionItem2 =
    myCollection.CreateElement("myAdd");
myCollectionItem2.SetAttributeValue("myCollectionItemBoolAttr", false);
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId4");
myCollection.Add(myCollectionItem2);

mgr.CommitChanges();
}
}
```

Run the program and open the web.config file of the Default Web Site, which is located in the following directory on your machine:

```
%SystemDrive%\inetpub\wwwroot\web.config
```

The result should look like Listing 5-9.

Listing 5-9: The Site web.config File

```
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <system.webServer>
    <myConfigSection myConfigSectionBoolAttr="false"
      myConfigSectionEnumAttr="myConfigSectionEnumVal3">
      <myNonCollection myNonCollectionTimeSpanAttr="00:04:00" />
      <myCollection myCollectionIntAttr="200">
        <myAdd myCollectionItemBoolAttr="true"
          myCollectionItemIdentifier="myId3" />
        <myAdd myCollectionItemBoolAttr="false"
```

(Continued)

Listing 5-9: (continued)

```
        myCollectionItemIdentifier="myId4" />
    </myCollection>
</myConfigSection>
</system.webServer>
</configuration>
```

As the boldfaced portion of Listing 5-9 shows, the IIS7 and ASP.NET integrated imperative management API has added a new `<myConfigSection>` configuration section to the `web.config` file of the Default Web Site. Now compare Listings 5-6 and 5-8. As you can see, the main difference between these two code listings is the second code line. The second lines of these two code listings respectively call the `GetApplicationHostConfiguration` and `GetWebConfiguration` methods of the `ServerManager` object. In other words, you use the same exact code to interact with the IIS7-level `applicationHost.config` and ASP.NET Web site-level `web.config` configuration files. This wasn't possible in the earlier versions of IIS because IIS and ASP.NET were using two completely different configuration systems with two different schemas, which meant that you had to use different APIs to interact with these two configuration systems. Thanks to the IIS7 and ASP.NET integrated configuration system, you can use the same API to specify configuration settings in both the IIS and ASP.NET Web site levels.

Application-Level Configuration File

As the previous section showed, you can use the same code that you used to add the `<myConfigSection>` configuration section to the `applicationHost.config` file to add a new `<myConfigSection>` element to the site-level `web.config` file with only a single line of change, that is, replacing the call to the `GetApplicationHostConfiguration` method to the call to the `GetWebConfiguration` method. The same argument applies to the application-level `web.config` file. The only difference between a site-level and an application-level `web.config` file is that the call to the `GetWebConfiguration` method must pass the virtual path of the application as the second argument to this method.

To see this in action, add a new Web application with the virtual path of `/MyWebApp`. As discussed in the previous chapters, there are different ways to do this:

- ❑ Open the `applicationHost.config` file in your favorite editor and add the following code:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="Default Web Site" id="1">
        <application path="/MyWebApp">
          <virtualDirectory path="/"
            physicalPath="%SystemDrive%\inetpub\wwwroot\MyWebAppDir" />
        </application>
      </site>
    </sites>
  </system.applicationHost>
</configuration>
```

- ❑ You must add a virtual directory with the virtual path of `"/"` to your new application. Recall that this virtual directory is known as the root virtual directory of the application.
- ❑ Use the IIS Manager as discussed in Chapter 3.

Chapter 5: Extending the Integrated Configuration System

- ❑ Use the APPCMD command line tool as discussed in Chapter 3.
- ❑ Launch Visual Studio, select New Web Site from the File menu to launch the New Web Site dialog, select the HTTP option from the location combo box, and add a new Web site. VS automatically creates a Web application.
- ❑ Use the IIS7 and ASP.NET integrated imperative management API as discussed in Chapter 4.

Now launch Visual Studio, create a new console application as usual, add the code shown in Listing 5-8 to the `Program.cs` file, and replace the following lines of code:

```
Configuration siteConfig = mgr.GetWebConfiguration("Default Web Site");
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId3");
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId4");
```

with these lines:

```
Configuration siteConfig = mgr.GetWebConfiguration("Default Web Site", "/MyWebApp");
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId5");
myCollectionItem2.SetAttributeValue("myCollectionItemIdentifier", "myId6");
```

Run the console application and open the `web.config` file of the `MyWebApp` application. You should see the same boldfaced portion shown in Listing 5-9.

IIS7 and ASP.NET Integrated Imperative Management Extensibility Model

Take another look at Listings 5-6 and 5-8. Recall that these two code listings use the IIS7 and ASP.NET integrated imperative management API to add a new `<myConfigSection>` configuration section with the specified configuration settings to the `applicationHost.config` and `site.web.config` files. The main problem with these two code listings is that the code does not treat the `<myConfigSection>` configuration section and its content as strongly-typed objects:

- ❑ The configuration section is accessed through the general `ConfigurationSection` type and its attributes are set through the general `SetAttributeValue` method:

```
ConfigurationSection myConfigSection =
    siteConfig.GetSection("system.webServer/myConfigSection");
myConfigSection.SetAttributeValue("myConfigSectionBoolAttr", false);
myConfigSection.SetAttributeValue("myConfigSectionEnumAttr",
    "myConfigSectionEnumVal3");
```

- ❑ The non-collection content is accessed through the general `ConfigurationElement` type and its attribute is set through the general `SetAttributeValue` method:

```
ConfigurationElement myNonCollection =
    myConfigSection.GetChildElement("myNonCollection");
myNonCollection.SetAttributeValue("myNonCollectionTimeSpanAttr",
    TimeSpan.FromMinutes(4));
```

- ❑ The collection is accessed through the general `ConfigurationElementCollection` type and its attribute is set through the general `SetAttributeValue` method:

```
ConfigurationElementCollection myCollection =  
    myConfigSection.GetCollection("myCollection");  
myCollection.SetAttributeValue("myCollectionIntAttr", 100);
```

- ❑ The collection items are accessed through the general `ConfigurationElement` type and attributes are set through the general `SetAttributeValue` method:

```
ConfigurationElement myCollectionItem1 = myCollection.CreateElement("myAdd");  
myCollectionItem1.SetAttributeValue("myCollectionItemBoolAttr", true);  
myCollectionItem1.SetAttributeValue("myCollectionItemIdentifier", "myId1");
```

You may be wondering what is so great about exposing the content of your configuration section as strongly-typed objects and properties. Here are some of the benefits of strongly-typed entities:

- ❑ Visual Studio provides IntelliSense support for strongly-typed objects and properties, allowing you to catch problems as you're typing.
- ❑ Compilers provide type-checking support for strongly-typed objects and properties, allowing you to catch problems as you're compiling.
- ❑ Strongly-typed entities allow you to program in an object-oriented fashion, where you can take advantage of the well-known benefits of the object-oriented programming paradigm.

When you extend the IIS7 and ASP.NET integrated configuration system to add support for your own custom configuration section, you should also extend the IIS7 and ASP.NET integrated imperative management API to add support for the following classes:

- ❑ For each collection, design a class whose instances represent the collection items. This class must inherit the `ConfigurationElement` base class and expose the attributes of the collection item as strongly-typed properties.
- ❑ For each collection, design a class that represents the collection itself. This class must inherit the `ConfigurationElementCollectionBase` class and expose:
 - ❑ Strongly-typed properties to represent the attributes of the collection
 - ❑ Strongly-typed collection property that contains the collection items
- ❑ For each non-collection element, design a class to represent the element. This class must inherit the `ConfigurationElement` class and expose the attributes of the non-collection element as strongly-typed properties.
- ❑ Design a class that inherits from the `ConfigurationSection` class to represent the outermost element of your configuration section. This class must expose one strongly-typed collection property to represent each collection. The class must also expose the attributes of the outermost element as strongly-typed properties.

In the following sections I use this recipe to design the required classes for the `<myConfigSection>` configuration section.

Representing the Collection Item

In this section, I design a class named `MyCollectionItem` whose instances represent the collection items of the collection that the `<myCollection>` element represents. Listing 5-10 presents the `MyCollectionItem` class.

Listing 5-10: The `MyCollectionItem` Class

```
using Microsoft.Web.Administration;

namespace MyNamespace
{
    class MyCollectionItem: ConfigurationElement
    {
        public bool MyCollectionItemBoolProperty
        {
            get { return (bool)base["myCollectionItemBoolAttr"]; }
            set { base["myCollectionItemBoolAttr"] = value; }
        }

        public string MyCollectionItemIdentifier
        {
            get { return (string)base["myCollectionItemIdentifier"]; }
            set { base["myCollectionItemIdentifier"] = value; }
        }
    }
}
```

The `MyCollectionItem` class inherits from the `ConfigurationElement` base class and exposes the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of the `<myAdd>` as strongly-typed properties named `MyCollectionItemBoolProperty` and `MyCollectionItemIdentifier`, respectively.

Representing the Collection Element

Listing 5-11 presents a collection class named `MyCollection` that represents the `<myCollection>` Collection element.

Listing 5-11: The `MyCollection` Class

```
using Microsoft.Web.Administration;
using System;

namespace MyNamespace
{
    class MyCollection: ConfigurationElementCollectionBase<MyCollectionItem>
    {
        public MyCollectionItem Add(string myCollectionItemIdentifier,
                                    bool myCollectionItemBoolValue)
        {
            MyCollectionItem myCollectionItem = base.CreateElement();
            myCollectionItem["myCollectionItemIdentifier"] = myCollectionItemIdentifier;
        }
    }
}
```

```
        myCollectionItem["myCollectionItemBoolAttr"] = myCollectionItemBoolValue;
        base.Add(myCollectionItem);
        return myCollectionItem;
    }

    protected override MyCollectionItem CreateNewElement(string elementTagName)
    {
        return new MyCollectionItem();
    }

    public new MyCollectionItem this[string myCollectionItemIdentifier]
    {
        get
        {
            for (int i=0; i<base.Count; i++)
            {
                if (string.Equals(base[i].MyCollectionItemIdentifier,
                    myCollectionItemIdentifier, StringComparison.OrdinalIgnoreCase))
                    return base[i];
            }
            return null;
        }
    }

    public int MyCollectionIntProperty
    {
        get { return (int)base["myCollectionIntAttr"]; }
        set { base["myCollectionIntAttr"] = value; }
    }
}
}
```

The `MyCollection` class, like any other collection class, inherits the `ConfigurationElementCollectionBase` class and performs the following tasks:

- ❑ Exposes the `myCollectionIntAttr` attribute of the `<myCollection>` element as a strongly-typed property:

```
public int MyCollectionIntProperty
{
    get { return (int)base["myCollectionIntAttr"]; }
    set { base["myCollectionIntAttr"] = value; }
}
```

- ❑ Exposes an indexer that returns the `MyCollectionItem` object with the specified collection item identifier:

```
public new MyCollectionItem this[string myCollectionItemIdentifier]
{
    get
    {
        for (int i=0; i<base.Count; i++)
        {
            if (string.Equals(base[i].MyCollectionItemIdentifier,
```

```
        myCollectionItemIdentifier, StringComparison.OrdinalIgnoreCase))
        return base[i];
    }
    return null;
}
```

- ❑ Exposes an Add method that creates a `MyCollectionItem` object with the specified collection item identifier and adds the object to the collection:

```
public MyCollectionItem Add(string myCollectionItemIdentifier,
                           bool myCollectionItemBoolValue)
{
    MyCollectionItem myCollectionItem = base.CreateElement();
    myCollectionItem["myCollectionItemIdentifier"] = myCollectionItemIdentifier;
    myCollectionItem["myCollectionItemBoolAttr"] = myCollectionItemBoolValue;
    base.Add(myCollectionItem);
    return myCollectionItem;
}
```

- ❑ Overrides the `CreateNewElement` method of its base class to instantiate and return a `MyCollectionItem` object:

```
protected override MyCollectionItem CreateNewElement(string elementTagName)
{
    return new MyCollectionItem();
}
```

Representing the Non-collection Element

The `MyNonCollection` class represents the `<myNonCollection>` element as shown in Listing 5-12. This class, like any other class that represents a non-collection element, inherits the `ConfigurationElement` base class and exposes the attribute of its associated non-collection element, `myNonCollectionTimeSpanAttr`, as a strongly-typed property.

Listing 5-12: The `MyNonCollection` Class

```
using Microsoft.Web.Administration;
using System;

namespace MyNamespace
{
    class MyNonCollection : ConfigurationElement
    {
        public TimeSpan MyNonCollectionTimeSpanProperty
        {
            get { return (TimeSpan)base["myNonCollectionTimespanAttr"]; }
            set { base["myNonCollectionTimespanAttr"] = value; }
        }
    }
}
```

Representing the Outermost Element

The `MyConfigSection` class represents the outermost element of the configuration section, the `<myConfigSection>` element, as presented in Listing 5-13.

Listing 5-13: The `MyConfigSection` Class

```
using Microsoft.Web.Administration;

namespace MyNamespace
{
    class MyConfigSection: ConfigurationSection
    {
        public bool MyConfigSectionBoolProperty
        {
            get { return (bool)base["myConfigSectionBoolAttr"]; }
            set { base["myConfigSectionBoolAttr"] = value; }
        }

        public MyConfigSectionEnum MyConfigSectionEnumProperty
        {
            get { return (MyConfigSectionEnum)base["myConfigSectionEnumAttr"]; }
            set { base["myConfigSectionEnumAttr"] = value; }
        }

        public MyNonCollection MyNonCollection
        {
            get { return (MyNonCollection)base.GetChildElement("myNonCollection",
                                                                typeof(MyNonCollection)); }
            set { base["myNonCollection"] = value; }
        }

        private MyCollection myCollection;
        public MyCollection MyCollection
        {
            get
            {
                if (myCollection == null)
                {
                    myCollection = (MyCollection)base.GetCollection("myCollection",
                                                                    typeof(MyCollection));
                }
                return myCollection;
            }
        }
    }
}
```

The `MyConfigSection` class, like any other configuration section, inherits `ConfigurationSection` base class and performs the following tasks:

1. Exposes the attributes of the `<myConfigSection>` element as strongly-typed properties:

```
public bool MyConfigSectionBoolProperty
```

```
{
    get { return (bool)base["myConfigSectionBoolAttr"]; }
    set { base["myConfigSectionBoolAttr"] = value; }
}

public MyConfigSectionEnum MyConfigSectionEnumProperty
{
    get { return (MyConfigSectionEnum)base["myConfigSectionEnumAttr"]; }
    set { base["myConfigSectionEnumAttr"] = value; }
}
```

2. Exposes a property of type `MyNonCollection` named `MyNonCollection` that references the `MyNonCollection` object that represents the `<myNonCollection>` element of the configuration section:

```
public MyNonCollection MyNonCollection
{
    get { return (MyNonCollection)base.GetChildElement("myNonCollection",
                                                       typeof(MyNonCollection)); }
    set { base["myNonCollection"] = value; }
}
```

3. Exposes a property of type `MyCollection` named `MyCollection` that references the `MyCollection` object that represents the `<myCollection>` Collection element of the configuration section:

```
private MyCollection myCollection;
public MyCollection MyCollection
{
    get
    {
        if (myCollection == null)
        {
            myCollection = (MyCollection)base.GetCollection("myCollection",
                                                            typeof(MyCollection));
        }
        return myCollection;
    }
}
```

Listing 5-14 presents the definition of the `MyConfigSectionEnum` enumeration type.

Listing 5-14: The `MyConfigSectionEnum` Type

```
namespace MyNamespace
{
    public enum MyConfigSectionEnum
    {
        MyConfigSectionEnumVal1,
        MyConfigSectionEnumVal2,
        MyConfigSectionEnumVal3
    }
}
```

Putting It All Together

The previous sections extended the IIS7 and ASP.NET integrated imperative management API to add support for new managed classes that represent your configuration section and its constituent elements and attributes. Now it's time to put these classes to use. Launch Visual Studio, create a new console application, add a reference to the `Microsoft.Web.Administration.dll` assembly, import the `Microsoft.Web.Administration` namespace, and add the code shown in Listing 5-15 to the `Program.cs` file. Next, add five source files named `MyCollectionItem.cs`, `MyCollection.cs`, `MyNonCollection.cs`, `MyConfigSection.cs`, and `MyConfigSectionEnum.cs` to this console application and add the code shown in Listings 5-10 through 5-14 to these source files, respectively.

Listing 5-15: A Console Application That Uses the New Managed Classes

```
using Microsoft.Web.Administration;
using System;
using MyNamespace;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
        MyConfigSection myConfigSection =
            (MyConfigSection)appHostConfig.GetSection(
                "system.webServer/myConfigSection",
                typeof(MyConfigSection));
        myConfigSection.MyConfigSectionBoolProperty = true;
        myConfigSection.MyConfigSectionEnumProperty =
            MyConfigSectionEnum.MyConfigSectionEnumVal3;
        myConfigSection.MyNonCollection.MyNonCollectionTimeSpanProperty =
            TimeSpan.FromMinutes(2);
        myConfigSection.MyCollection.MyCollectionIntProperty = 50;
        myConfigSection.MyCollection.Add("myId5", true);
        myConfigSection.MyCollection.Add("myId6", false);
        mgr.CommitChanges();
    }
}
```

Run the program and open the `applicationHost.config` file in your favorite editor. The result should look like the following:

```
<configuration>
  <system.webServer>
    <myConfigSection myConfigSectionBoolAttr="true"
      myConfigSectionEnumAttr="myConfigSectionEnumVal3">
      <myNonCollection myNonCollectionTimeSpanAttr="00:02:00" />
      <myCollection myCollectionIntAttr="50">
        <myAdd myCollectionItemBoolAttr="true"
          myCollectionItemIdentifier="myId5" />
        <myAdd myCollectionItemBoolAttr="false"
          myCollectionItemIdentifier="myId6" />
      </myCollection>
    </myConfigSection>
  </system.webServer>
</configuration>
```


Chapter 5: Extending the Integrated Configuration System

```
</myConfigSection>
</system.webServer>
</configuration>
```

Now let's dissect Listing 5-14. The listing begins by creating a `ServerManager` object and calls its `GetApplicationHostConfiguration` method to load the `applicationHost.config` file into a `Configuration` object as usual:

```
ServerManager mgr = new ServerManager();
Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
```

Next, it calls the `GetSection` method to return the `MyConfigSection` object that provides programmatic access to the `<myConfigSection>` configuration section in strongly-typed manner. Note that the `Type` object representing the type of the `MyConfigSection` class is passed into the `GetSection` method. Under the hood, this method uses the .NET reflection and this type information to dynamically generate an instance of the `MyConfigSection` class.

```
MyConfigSection myConfigSection =
    (MyConfigSection)appHostConfig.GetSection(
        "system.webServer/myConfigSection",
        typeof(MyConfigSection));
```

Listing 5-14 then sets the values of the `MyConfigSectionBoolProperty` and `MyConfigSectionEnumProperty` properties of the `MyConfigSection` object. This is in contrast to Listing 5-8, where the generic `SetAttributeValue` method of the generic `ConfigurationSection` class is used to set these values.

```
myConfigSection.MyConfigSectionBoolProperty = true;
myConfigSection.MyConfigSectionEnumProperty =
    MyConfigSectionEnum.MyConfigSectionEnumVal3;
```

Next, Listing 5-14 accesses the `MyNonCollection` property of the `MyConfigSection` object and sets its `MyNonCollectionTimeSpanProperty` property in type-safe fashion:

```
myConfigSection.MyNonCollection.MyNonCollectionTimeSpanProperty =
    TimeSpan.FromMinutes(2);
```

Finally, it accesses the `MyCollection` property of the `MyConfigSection` object, sets its `MyCollectionProperty` property, and adds two new `MyCollectionItem` objects to the collection. These operations are all performed in a strongly-typed manner:

```
myConfigSection.MyCollection.MyCollectionIntProperty = 50;
myConfigSection.MyCollection.Add("myId5", true);
myConfigSection.MyCollection.Add("myId6", false);
mgr.CommitChanges();
```

Summary

The IIS7 and ASP.NET integrated infrastructure extensibility model consists of four different extensibility models that complement each other. This chapter covered two of these extensibility models, the IIS7 and ASP.NET integrated configuration system and imperative management extensibility models. The next chapters discuss the two remaining extensibility models, the IIS7 and ASP.NET integrated graphical management and request processing pipeline extensibility models.

6

Understanding the Integrated Graphical Management System

Chapter 3 provided in-depth coverage of the new IIS7 Manager and its rich capabilities, such as:

- ❑ You can use the IIS7 Manager to configure both the IIS7 Web server and ASP.NET Web applications. This is a departure from the previous versions of IIS, where you needed to use two separate management tools because the Web server and ASP.NET were using two completely different configuration systems.
- ❑ The IIS7 Manager contains the logic that takes the hierarchical nature of the IIS7 and ASP.NET integrated configuration system into account. Configuration changes made at a particular level of the configuration hierarchy are automatically saved into either the configuration file in that hierarchy level or a `<location>` element in a configuration file in a higher hierarchy level. For example, configuration changes made at site level are stored in the root `web.config` file of the site, which means that these changes will only affect the Web applications in that site.

Chapter 3 discussed these two and many other capabilities of the IIS7 Manager in detail. This and the next chapter discuss a very important aspect of the IIS7 Manager that wasn't covered in the previous chapters, that is, its extensible architecture. The extensibility of the IIS Manager is of paramount importance to ASP.NET developers and IIS7 administrators alike. If the IIS7 Manager is to be the management tool of choice for ASP.NET developers, it must allow developers to extend its graphical capabilities to add graphical support for new configuration sections. In other words, developers should be able to specify configuration settings of their own custom configuration sections from the IIS7 Manager tool.

Chapter 6: Understanding the Integrated Graphical Management System

Extending the IIS7 and ASP.NET integrated graphical management architecture requires you to have a solid understanding of this architecture and its main components. This is exactly what we're going to do in this chapter. This chapter sets the stage for the next chapter, where you'll use what you've learned in this chapter to extend this integrated architecture to add graphical support for a custom configuration section.

As you may recall from Chapter 3, interacting with the IIS7 Manager is much like interacting with a Web application. At a very basic level, a Web application is a collection of Web pages and a navigation system that allows end users to navigate through these Web pages. As such, the page is the module or unit of extensibility in a navigation/page-based application such as a Web application. That is, you extend the application by adding new pages. The IIS7 Manager's graphical architecture follows this page/navigation paradigm as discussed in the following sections.

Module Pages

IIS Manager includes a bunch of pages known as *module pages*. As the name implies, a module page is the unit of GUI extensibility. As Figure 6-1 shows, the IIS Manager interface consists of three columns. Module pages are shown one page at a time in the middle column. There are three main types of module pages as discussed in the following sections.

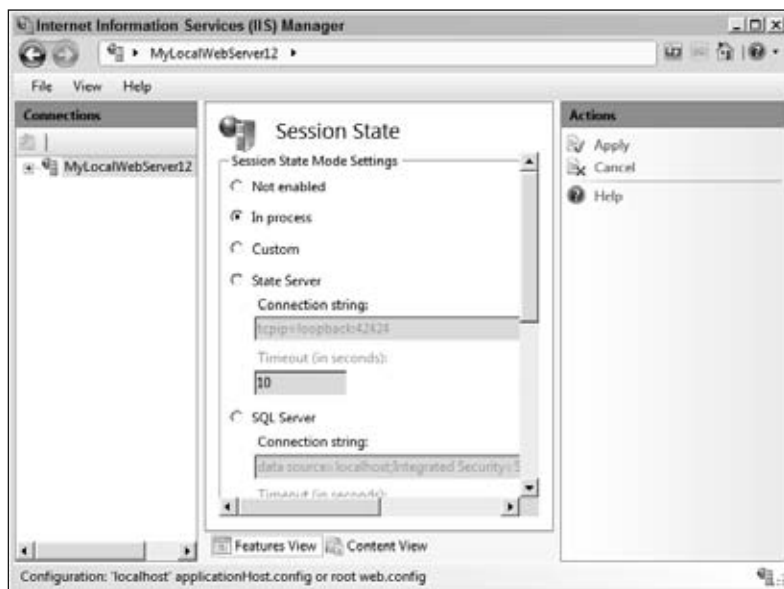


Figure 6-1

ModuleDialogPage

The first type of module page is known as a module dialog page. The middle column of Figure 6-1 shows an example of a module dialog page. As the name implies, a module dialog page acts like a traditional dialog box. Notice that the panel in the Actions pane contains the standard dialog box command buttons, such as Apply and Cancel. This panel is known as the *task panel* because it normally contains GUI elements, such as Apply, that perform tasks such as applying the changes. All module dialog pages inherit from the `ModuleDialogPage` abstract base class. For example, the module dialog page shown in Figure 6-1 is an instance of a control named `SessionStatePage` that inherits this base class.

ModuleListPage

The second type of module page is known as a *module list page*. Figure 6-2 presents an example of a module list page. As you can see, a module list page consists of a list of items and a combo box named *Group by* that allows the end user to group the items. All module list pages derive from the `ModuleListPage` abstract base class. For example, the module list page shown in Figure 6-2 is an instance of a control named `MimeTypesPage` that inherits the `ModuleListPage` class.

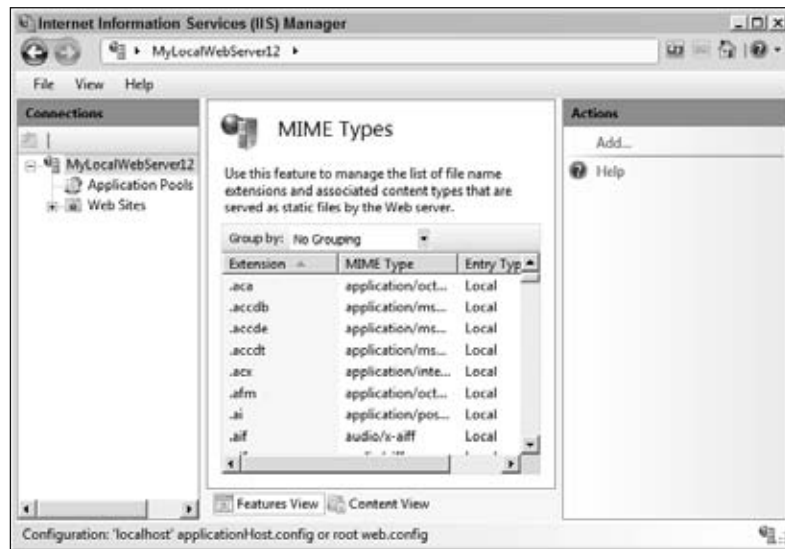


Figure 6-2

ModulePropertiesPage

The third type of module page is known as a *module properties page*. Figure 6-3 demonstrates an example of a module properties page. All module properties pages derive from the `ModulePropertiesPage` abstract base class. For example, the module properties page demonstrated in Figure 6-3 is an instance of

Chapter 6: Understanding the Integrated Graphical Management System

a control named `CompilationPage` that inherits the `ModulePropertiesPage` base class. Note that the task panel contains the typical dialog box command buttons, such as `Apply` and `Cancel`. This is because the `ModulePropertiesPage` base class inherits the `ModuleDialogPage` base class. In other words, every module properties page is also a module dialog page.

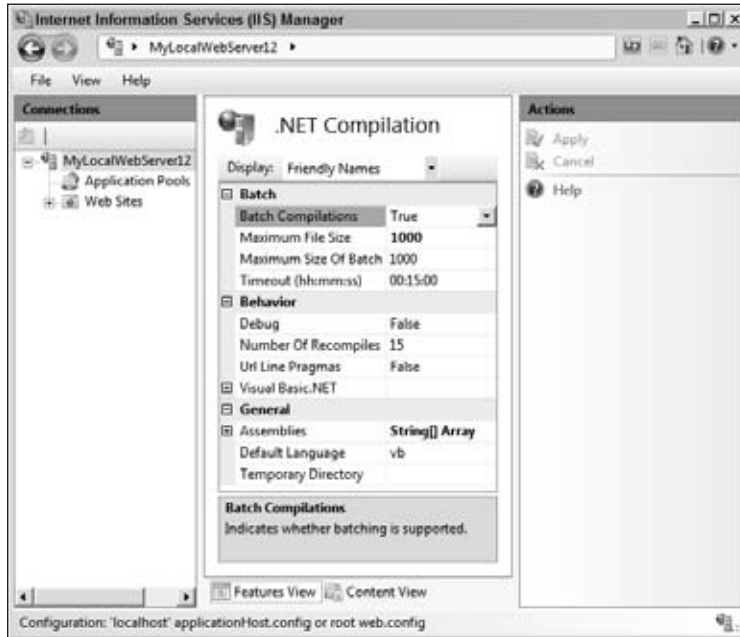


Figure 6-3

Writing a Custom Module Page

As you'll see in the next chapter, writing a custom module page involves, among several others, the following two important steps:

- ❑ Decide which type of the three types of module pages is the most appropriate user interface for displaying and editing the configuration settings of your custom configuration section.
- ❑ Implement a class that derives from the type that you chose in the previous step.

In other words, a custom module page is a class that inherits the `ModuleDialogPage`, `ModuleListPage`, or `ModulePropertiesPage` base class. You can also inherit your custom module page directly from the `ModulePage` base class. If you decide to go with this option, keep in mind that the `ModulePage` base class does not render any user interface and you're left with implementing most of the functionality that the `ModuleDialogPage`, `ModuleListPage`, and `ModulePropertiesPage` base classes already implement. Note that all these three base classes inherit the `ModulePage` class.

Tasks

As Figures 6-1 through 6-3 show, each module page is normally associated with a task panel that contains a bunch of buttons and links, which each correspond to a particular task. For example, the panel in the Actions pane of Figure 6-1 contains the Apply, Cancel, and Help buttons and the panel in the Actions pane of Figure 6-2 includes the Add and Help buttons.

The user clicks a button or link in the panel to perform the associated task. For example, the end user clicks the Apply button in Figure 6-1 to commit the changes to the underlying configuration file. In cases such as the Apply button, nothing much happens in terms of the user interface except for displaying a confirmation message. The following three sections discuss three other common scenarios that could occur when the end user clicks a button in the task panel associated with a given module page.

Page Navigation

The first scenario occurs when the event handler for the button or link uses the navigation service to navigate to a new page. For example, if the user clicks the View Application Pools link shown in the Actions pane of Figure 6-4, the IIS7 Manager will navigate to the module page shown in Figure 6-5.

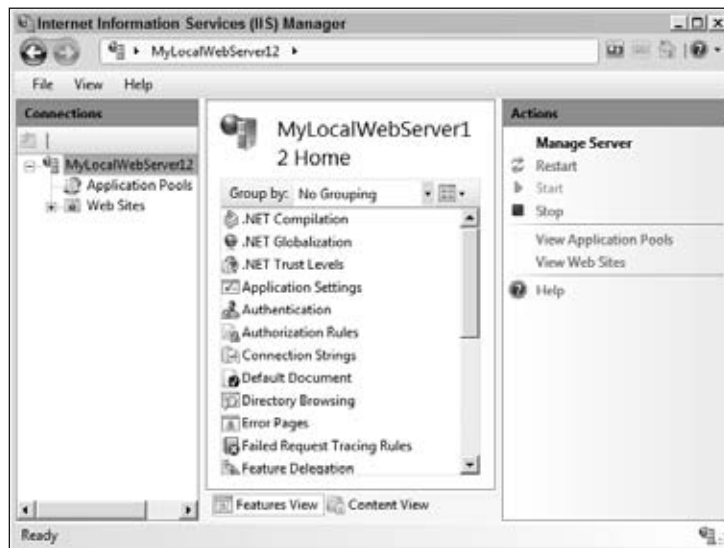


Figure 6-4

The event handler for the View Application Pools link uses the navigation service to navigate from the page shown in Figure 6-4 to the page shown in Figure 6-5. The navigation service is a class named `NavigationService` that implements an interface named `INavigationService`. As you'll see later, this interface exposes methods that you can call from your event handlers to navigate back to the previous page, forward to the next page, or to any arbitrary page. In other words, the navigation service of the IIS7 Manager acts like the navigation service of a Web application, allowing end users to navigate through the module pages that make up the IIS7 Manager's user interface.

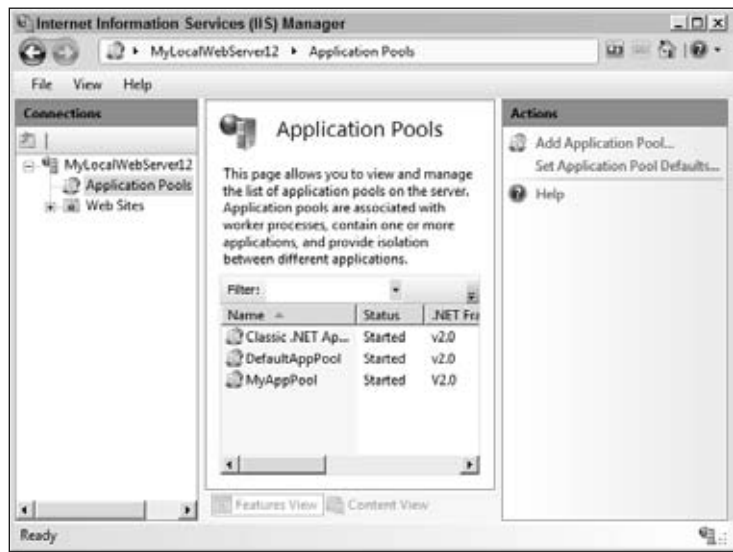


Figure 6-5

Task Forms

This scenario occurs when the event handler for a button or link in the task panel associated with a module page pops up a dialog to collect the user’s input in one shot. For example, when the user clicks the Add button shown in Figure 6-2, the event handler for this button pops up the dialog shown in Figure 6-6 to collect the input needed to add a new MIME type. This kind of dialog is known as a task form. Every task form inherits the `TaskForm` abstract base class. For example, the task form shown in Figure 6-6 is an instance of a control named `MimeTypesForm` that derives from the `TaskForm` base class.

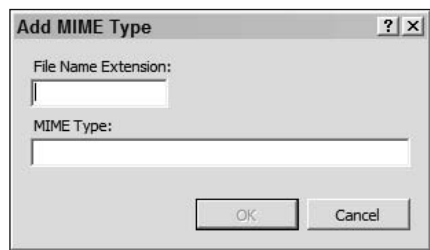


Figure 6-6

Wizard Forms

This scenario occurs when the event handler for a link in the task panel associated with a module page pops up a wizard form that takes the end user through a set of steps to collect the required inputs. For example, when the end user clicks the Add link shown in Figure 6-7, the event handler for this link pops

Chapter 6: Understanding the Integrated Graphical Management System

up the wizard form shown in Figure 6-8 to walk the user through a set of steps to collect the information needed to add a new failed request tracing rule. The main difference between a task form (see Figure 6-6) and a wizard form (see Figure 6-8) is that the task form collects the user inputs in one step, whereas the wizard form does it in several steps. All wizard forms derive from the `WizardForm` abstract base class. For example, the wizard form shown in Figure 6-8 is an instance of a control named `FailureTraceRequestWizardForm` that inherits the `WizardForm` class.



Figure 6-7

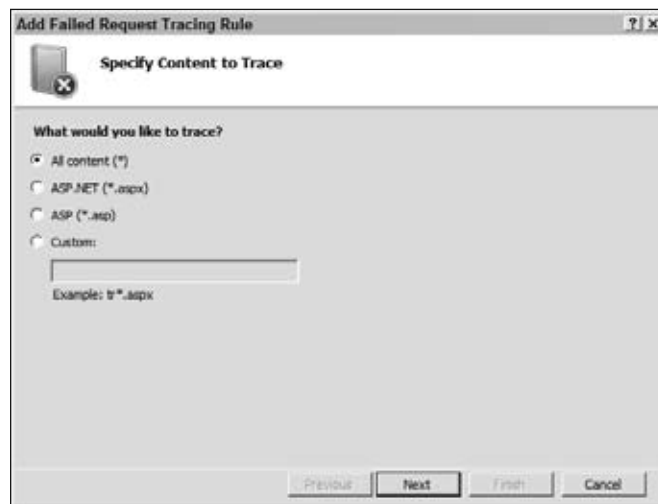


Figure 6-8

The IIS7 Manager Object Model

You need to have a solid understanding of the IIS7 Manager’s object model to extend the model to add graphical support for your own custom configuration sections. This section reviews some of the important classes in the IIS7 Manager’s object model.

Service

A service is an object that meets the following criteria:

- ❑ Its instantiation is completely hidden from its clients. Clients access the service as a live object.
- ❑ Its type implements a well-known interface. This interface acts as a contract between the service and its clients.
- ❑ There can be only a single instance of the service. No matter where in the application the clients access the service, it’s always the same object.
- ❑ It’s uniquely identified by the `Type` object that represents the interface that the service implements. This allows the clients of the service to use this `Type` object as the identifier to access the service.

Services play a central role in the IIS7 Manager graphical architecture. Most IIS7 Manager features are exposed as services. The following table presents some of these services and their associated interfaces:

Service	Interface
Win32ManagementHost	IManagementHost
NavigationService	INavigationService
ConnectionManager	IConnectionManager
ManagementUIService	IManagementUIService
PropertyEditingService	IPropertyEditingService

For example, the IIS7 Manager graphical architecture exposes its navigation system as a service that implements an interface named `INavigationService`.

IServiceProvider

A service provider is a class that implements the `IServiceProvider` interface as defined in Listing 6-1.

Listing 6-1: The IServiceProvider Interface

```
namespace System
{
    public interface IServiceProvider
```

Listing 6-1: (continued)

```
{  
    object GetService(Type serviceType);  
}  
}
```

The `IServiceProvider` interface exposes a single method named `GetService` that returns a service with a specified service type. The service type is the `Type` object that represents the interface that the service implements. For example, in the case of the navigation service, this `Type` object is `typeof(INavigationService)`.

The `IServiceProvider` interface allows the clients of a service to access the service without knowing its real type. For example, the clients of a navigation service call the `GetService` method, passing in the `typeof(INavigationService)` as its parameter to access the navigation service. The clients have no idea that the underlying navigation service is of type `NavigationService`. As far as the clients are concerned, the navigation service is of type `INavigationService`. Therefore, if the underlying `NavigationService` class is replaced with a new class, the client's code will work just fine as long as the new class implements the `INavigationService` interface.

The following table presents some of the classes in the IIS7 Manager graphical object model that implement the `IServiceProvider` interface; that is, they act as service providers:

Service Provider	Description
Connection	This class represents the connection between the IIS7 Manager and the back-end Web server.
Module	As you'll see in the next chapter, the subclasses of this class are used to register module pages with the IIS7 Manager.
WebMgrShellApplication	As you'll see later, this class is responsible for creating the form that contains the entire user interface of the IIS7 Manager.

Each of these three service providers exposes the `GetService` method to allow their clients to access their services in generic fashion.

IServiceContainer

A service container is a class that implements the `IServiceContainer` interface. Listing 6-2 presents the important methods of this interface.

Listing 6-2: The `IServiceContainer` Interface

```
public interface IServiceContainer : IServiceProvider  
{  
    void AddService(Type serviceType, ServiceCreatorCallback callback);  
    void AddService(Type serviceType, object serviceInstance);  
    void RemoveService(Type serviceType);  
}
```

Chapter 6: Understanding the Integrated Graphical Management System

Because the `IServiceContainer` interface extends the `IServiceProvider` interface, a service container is also a service provider. The `IServiceContainer` interface exposes two important methods: `AddService` (with two overloads) and `RemoveService`. The `AddService` method adds a service with the specified service type to an internal container. The `RemoveService` method, on the other hand, removes the service with the specified service type from the internal container.

Notice that the `AddService` method comes in two flavors. The following overload of this method takes the service instance itself and adds it to the internal container:

```
void AddService(Type serviceType, object serviceInstance);
```

The second overload of this method takes a `ServiceCreatorCallback` delegate instead of the service instance itself:

```
void AddService(Type serviceType, ServiceCreatorCallback callback);
```

The definition of this delegate is as follows:

```
public delegate object  
ServiceCreatorCallback(IServiceContainer container, Type serviceType);
```

When the client of a service calls the `GetService` method of the service container (recall that every service container is also a service provider), the service container invokes the registered `ServiceCreatorCallback` delegate and passes it the service type and a reference to the service container. It's the responsibility of the delegate to instantiate and to return the service. The service container then adds this newly instantiated service to its internal container.

Therefore, the first overload of the `AddService` method requires you to instantiate the service when you're registering it. The second overload, on the other hand, allows you to register the service and postpone its instantiation to the time when it's actually used for the first time.

The following table presents some of the classes that implement the `IServiceContainer` interface:

Service Container	Description
Connection	This class represents the connection between the IIS7 Manager and the back-end Web server.
ServiceContainer	This class stores the services in an internal hash table.

ManagementConfigurationPath

The `ManagementConfigurationPath` class represents a configuration path. Listing 6-3 presents the declaration of some the methods of this class.

Listing 6-3: The ManagementConfigurationPath Class

```
public sealed class ManagementConfigurationPath
{
    public string GetEffectiveConfigurationPath(ManagementScope scope);
    public bool IsEquivalentScope(ManagementScope scope);
    public string ApplicationPath { get; }
    public string SiteName { get; }
}
```

Listing 6-4 presents the definition of the ManagementScope enumeration.

Listing 6-4: The ManagementScope Enumeration

```
public enum ManagementScope
{
    Server,
    Site,
    Application
}
```

As this enumeration shows, the IIS7 Manager supports three configuration management scopes: server, site, and application. Each scope determines the configuration hierarchy level affected by the new configuration changes.

Connection

Every connection, be it server, site, or application, is represented by an instance of a class named *Connection*, which implements the *IServiceContainer* and *IServiceProvider* interfaces as shown in Listing 6-5. In other words, the *Connection* object acts as both the service container and service provider. The *Connection* class also exposes a method named *CreateProxy* and a property named *Modules*, which are discussed in the next chapter.

Listing 6-5: The Connection Class

```
public sealed class Connection : IServiceContainer, IServiceProvider, IDisposable
{
    public ModuleServiceProxy CreateProxy(Module module, Type proxyType);
    void IServiceContainer.AddService(Type serviceType,
                                     ServiceCreatorCallback callback);
    void IServiceContainer.AddService(Type serviceType, object serviceInstance);
    void IServiceContainer.RemoveService(Type serviceType);
    object IServiceProvider.GetService(Type serviceType);

    public ManagementConfigurationPath ConfigurationPath { get; }
    public IDictionary Modules { get; }
    public ManagementScope Scope { get; }
    public bool IsLocalConnection { get; }
    . . .
}
```

Chapter 6: Understanding the Integrated Graphical Management System

Note that the `Connection` class exposes a Boolean property named `IsLocalConnection` that specifies whether the back-end Web server is running locally on the same machine where the IIS7 Manager is running.

Navigation Item

The IIS7 Manager uses the navigation service to navigate through a set of what are known as navigation items. Each navigation item is an instance of a class named `NavigationItem` as presented in Listing 6-6.

Listing 6-6: The `NavigationItem` Class

```
public sealed class NavigationItem : IDisposable
{
    public ManagementConfigurationPath ConfigurationPath { get; }
    public Connection Connection { get; }
    public object NavigationData { get; }
    public IModulePage Page { get; }
    public Type PageType { get; }
}
```

The properties of this class are the following:

- ❑ `ConfigurationPath`: The `ManagementConfigurationPath` object that represents the current configuration path.
- ❑ `Connection`: The `Connection` object that represents the current connection to the back-end Web server.
- ❑ `NavigationData`: An optional object that contains the navigation item-specific navigation data.
- ❑ `Page`: The module page associated with the current navigation item. Recall that the whole purpose of navigation is to navigate to a specified module page.
- ❑ `PageType`: The `Type` object that represents the type of the page being displayed. I discuss this later.

So what is a navigation item? The `NavigationItem` represents a module page with the specified page type, connection, configuration path, and navigation data. In other words, in the IIS7 Manager jargon, navigation means moving from one navigation item to another.

As Listing 6-6 shows, the navigation item treats the module page as an object of type `IModulePage`. This is possible because the `ModulePage` base class, which is the base class for all module pages, implements the `IModulePage` interface. This interface acts as the contract between the module page associated with the navigation item and the navigation item itself. I discuss the `IModulePage` interface in detail later in this chapter.

Navigation Service

As mentioned, the IIS7 Manager uses the navigation service to navigate through navigation items. As discussed earlier, every IIS7 Manager service implements an interface. The navigation service is no exception. It implements an interface named `INavigationService` as defined in Listing 6-7.

Listing 6-7: The INavigationService Interface

```
public interface INavigationService
{
    event NavigationEventHandler NavigationPerformed;

    bool Navigate(Connection connection,
                  ManagementConfigurationPath configurationPath, Type pageType,
                  object navigationData);
    bool NavigateBack(int steps);
    bool NavigateForward();

    bool CanNavigateBack { get; }
    bool CanNavigateForward { get; }
    NavigationItem CurrentItem { get; }
    ReadOnlyCollection<NavigationItem> History { get; }
}
```

The `INavigationService` interface exposes the following three methods:

- ❑ **Navigate:** The IIS7 Manager uses this method to navigate to a navigation item with the specified connection, configuration path, page type, and navigation data.
- ❑ **NavigateBack:** The IIS7 Manager uses this method to navigate back to the navigation item with the specified index. Note that the navigation service stores the previously navigated navigation items in an internal collection. The `NavigateBack` method uses the specified index to locate the navigation item in the internal collection and then navigates to that item.
- ❑ **NavigateForward:** The IIS7 Manager uses this method to navigate to the next item.

The `INavigationService` interface also exposes the following four properties:

- ❑ **CanNavigateBack:** This Boolean property specifies whether a backward navigation is possible.
- ❑ **CanNavigateForward:** This Boolean property specifies whether a forward navigation is possible.
- ❑ **CurrentItem:** This property returns a reference to the current navigation item.
- ❑ **History:** This collection property contains references to all the previously navigated items.

The `INavigationService` interface supports a single event of type `NavigationEventHandler` named `NavigationPerformed`. This event is fired after the completion of navigation. The definition of the `NavigationEventHandler` delegate is:

```
public delegate void NavigationEventHandler(object sender, NavigationEventArgs e);
```

The `NavigationEventArgs` class is the event data class associated with this event. Listing 6-8 contains the definition of this class.

Listing 6-8: The NavigationEventArgs Class

```
public sealed class NavigationEventArgs : EventArgs
{
    public bool IsNew { get; }
    public NavigationItem NewItem { get; }
    public NavigationItem OldItem { get; }
}
```

This class features the following properties:

- ❑ **IsNew:** This Boolean property specifies whether the destination navigation item is accessed for the first time.
- ❑ **NewItem:** The new navigation item.
- ❑ **OldItem:** The old navigation item.

TaskItem

Recall from the previous discussions that the user interface of the IIS7 Manager contains three columns. The Actions pane includes GUI items such as buttons, links, text, and so on. For example, the Actions pane in Figure 6-1 includes the Apply, Cancel, and Help links. Each GUI item in the Actions pane is represented by an instance of a class named `TaskItem`. For example, the Apply, Cancel, and Help links in Figure 6-1 are each represented by a `TaskItem` object.

Listing 6-9 presents the definition of the `TaskItem` abstract base class.

Listing 6-9: The TaskItem Base Class

```
public abstract class TaskItem
{
    protected TaskItem(string text, string category);
    protected TaskItem(string text, string category, string description);

    public string Category { get; }
    public string Description { get; }
    public bool Enabled { get; set; }
    public string Text { get; }
}
```

The `TaskItem` base class features the following important properties:

- ❑ **Category:** This string property specifies the category where the task item will be displayed in the task panel. If you want to have several task items under the same category you must assign the same value to this property.
- ❑ **Description:** This string property specifies the short description that will be shown to the end users when they move the mouse over the task item. In other words, it's used in a tool tip.

Chapter 6: Understanding the Integrated Graphical Management System

- ❑ **Enabled:** This Boolean property specifies whether the task item is enabled.
- ❑ **Text:** This string property specifies the text that will be shown to the users as part of the task item. For example, the `Text` property of the task item that represents the Apply link in Figure 6-1 returns the string value “Apply”.

The `TaskItem` class has four subclasses: `MessageTaskItem`, `GroupTaskItem`, `MethodTaskItem`, and `TextTaskItem`. Each subclass represents a particular type of GUI item in the task panel as discussed in the following sections.

TextTaskItem

The `TextTaskItem` represents a simple text in the task panel. For example, the “Manage Server” text shown in the task panel in Figure 6-4 is represented by a `TextTaskItem` object. Listing 6-10 presents the definition of the `TextTaskItem` class.

Listing 6-10: The TextTaskItem Class

```
public sealed class TextTaskItem : TaskItem
{
    // Methods
    public TextTaskItem(string text, string category): this(text, category, false);
    public TextTaskItem(string text, string category, bool isHeading);
    public TextTaskItem(string text, string category, bool isHeading, Image image);

    // Properties
    public object Image { get; }
    public bool IsHeading { get; }
}
```

The `TextTaskItem` class presents the following properties:

- ❑ **IsHeading:** Specifies whether the text should be rendered as a heading. For example, the “Manage Server” text in Figure 6-4 is rendered as a heading. In other words, the `IsHeading` Boolean property of the `TextTaskItem` object that represents the “Manage Server” text returns `true`.
- ❑ **Image:** Specifies the `Image` object that will be rendered in addition to the text. For example, the following code snippet instructs the task panel to render the specified bitmap:

```
Image img = new Bitmap(@"D:\MyBitMap.bmp");
TextTaskItem item = new TextTaskItem("Related Features", "MyGroup", true, img);
```

MessageTaskItem

Each warning, informational, or error message is represented by a `MessageTaskItem` object. Let’s take a look at an example of a message task item. Launch the IIS7 Manager and navigate to the page shown in Figure 6-9. Change the value of the `Batch Compilation` parameter and click Apply. The result should look like Figure 6-10. If you compare the Actions panes in Figures 6-9 and 6-10, you’ll notice that Figure 6-10 includes a new panel named `Alerts` that displays this message “The changes have been successfully saved.” This is an example of an informational message. The other two types of

Chapter 6: Understanding the Integrated Graphical Management System

messages are warning and error messages. Notice the difference in location between the message task items on one hand and the text, method, and group task items on the other hand. The message task items are displayed in the Alert panel and the text, method, and group task items are displayed in the task panel. As Figure 6-10 shows, the Alert panel becomes visible only when there's a message task item to display.

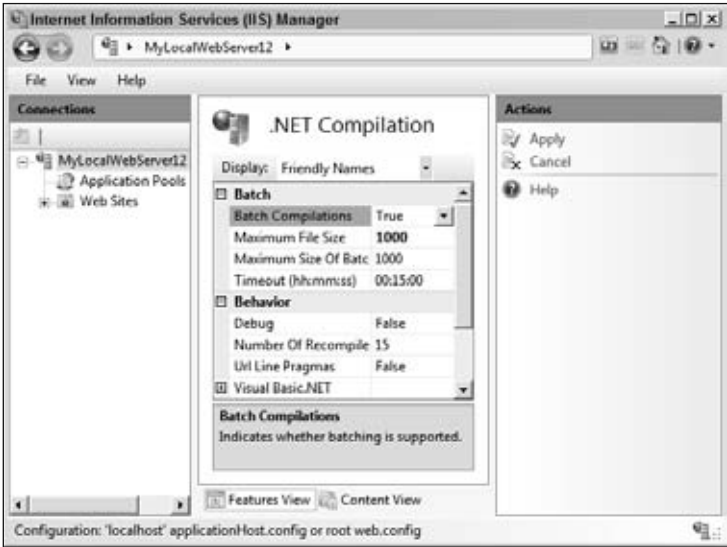


Figure 6-9

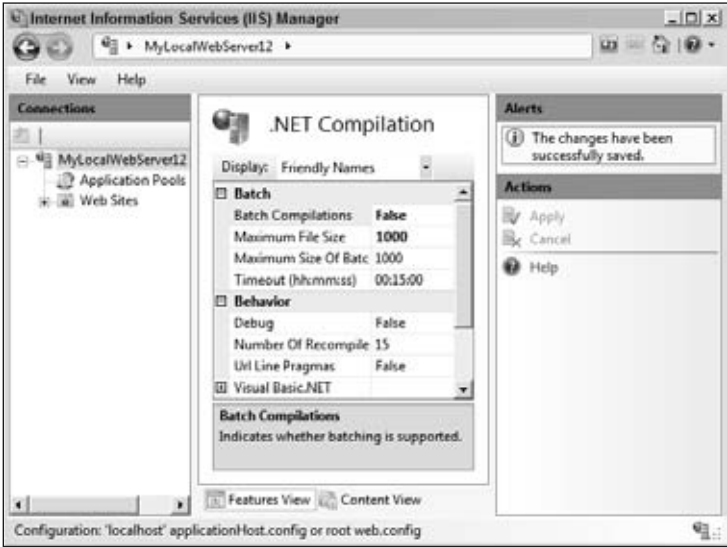


Figure 6-10

Chapter 6: Understanding the Integrated Graphical Management System

Listing 6-11 presents the definition of the `MessageTaskItem` class.

Listing 6-11: The `MessageTaskItem` Class

```
public sealed class MessageTaskItem : TaskItem
{
    public MessageTaskItem(MessageTaskItemType messageType, string text,
                           string category);
    public MessageTaskItem(MessageTaskItemType messageType, string text,
                           string category, string description);
    public MessageTaskItem(MessageTaskItemType messageType, string text,
                           string category, string description, string methodName,
                           object userData);

    public MessageTaskItemType MessageType { get; }
    public string MethodName { get; }
    public object UserData { get; }
}
```

The `MessageTaskItem` class features the following properties:

- ❑ **MessageType:** This `MessageTaskItemType` enumeration property specifies the type of message. As mentioned, there are three types of messages: informational, warning, and error.
- ❑ **MethodName:** This optional property allows you to specify a method as an event handler for the `OnClick` event of the message task item. This event is fired when the end user clicks the displayed message.
- ❑ **UserData:** The IIS7 Manager passes this optional property as an argument into the method specified by the `MethodName` property when it invokes this method.

Listing 6-12 contains the definition of the `MessageTaskItemType` enumeration.

Listing 6-12: The `MessageTaskItemType` Enumeration

```
public enum MessageTaskItemType
{
    Information,
    Warning,
    Error
}
```

MethodTaskItem

The `MethodTaskItem` represents a GUI item such as Apply that performs a predefined task when the user clicks it. Listing 6-13 presents the definition of the `MethodTaskItem` class.

Listing 6-13: The `MethodTaskItem` Class

```
public sealed class MethodTaskItem : TaskItem
{
    public MethodTaskItem(string methodName, string text, string category);
    public MethodTaskItem(string methodName, string text, string category,
```

(Continued)

Listing 6-13: (continued)

```
        string description);  
    public MethodTaskItem(string methodName, string text, string category,  
        string description, Image image);  
    public MethodTaskItem(string methodName, string text, string category,  
        string description, Image image, object userData);  
  
    public bool CausesNavigation { get; set; }  
    public Image Image { get; }  
    public string MethodName { get; }  
    public MethodTaskItemUsages Usage { get; set; }  
    public object UserData { get; }  
}
```

The `MethodTaskItem` class contains the following properties:

- ❑ **CausesNavigation:** This Boolean property specifies whether clicking the GUI item will cause navigation from the current module page to another page. For example, the View Application Pools link shown in Figure 6-4 causes the IIS7 Manager to navigate to the module page that displays the application pools. The Add link shown in Figure 6-2, on the other hand, doesn't cause the IIS7 Manager to navigate. Instead it launches the task form shown in Figure 6-6.
- ❑ **Image:** This property represents the image that will be rendered as part of rendering of the item. For example, in the case of Figure 6-5, the `Image` property of the task item that represents the Add Application Pool link represents the icon displayed next to this link.
- ❑ **MethodName:** This property contains the name of the method that will be called when the end user clicks the GUI item associated with the `MethodTaskItem`.
- ❑ **MethodTaskItemUsages:** This enumeration property specifies whether the associated GUI element will be part of a context menu, task list, or both. Listing 6-14 presents the definition of the `MethodTaskItemUsages` enumeration. Note that this enumeration is marked with the `Flags` attribute, which means that you can perform bitwise operations between the `ContextMenu` and `TaskList` values.
- ❑ **UserData:** As mentioned, when the user clicks the GUI element associated with the `MethodTaskItem` object, the method specified in the `MethodName` property is automatically invoked. If the optional `UserData` property is set, it will be automatically passed into this method as its argument.

Listing 6-14: The `MethodTaskItemUsages` Enumeration

```
[Flags]  
public enum MethodTaskItemUsages  
{  
    ContextMenu = 2,  
    TaskList = 1  
}
```

GroupTaskItem

The `GroupTaskItem` (shown in Listing 6-15) represents a group of `TaskItem` objects including `TextTaskItem`, `MessageTaskItem`, `MethodTaskItem`, and other `GroupTaskItem` objects. The `GroupTaskItem` renders a plus sign next to the name of the group if the group is expanded and a minus sign if the group is collapsed

Listing 6-15: The GroupTaskItem Class

```
public sealed class GroupTaskItem : TaskItem
{
    // Methods
    public GroupTaskItem(string memberName, string text, string category);
    public GroupTaskItem(string memberName, string text, string category,
        bool isHeading);

    // Properties
    public bool IsHeading { get; }
    public IList Items { get; }
    public string MemberName { get; }
}
```

The `GroupTaskItem` class presents the following properties:

- ❑ `IsHeading`: This Boolean property specifies whether the text should be rendered as a heading. This affects the look and feel of the text.
- ❑ `Items`: This `IList` collection contains the `TaskItem` objects in the group. You must use the `Add` method of this property to add new task items to the group. For example, the following code fragment adds text, method, and message task items to the group:

```
TextTaskItem text = new TextTaskItem("Related Features", "Related", true);
MethodTaskItem method = new MethodTaskItem("ViewCollectionItems",
    "View collection items", "MyGroup");
MessageTaskItem message = new MessageTaskItem(MessageTaskItemType.Warning,
    "This is a warning message", null);

GroupTaskItem group = new GroupTaskItem("MyMemberName", "MyText", "Actions", true);
group.Items.Add(text);
group.Items.Add(method);
group.Items.Add(message);
```

- ❑ `MemberName`: You'll see the role of this property in the next chapter in the context of an example.

TaskList

The `TaskList` class acts as a container for the `TaskItem` objects that represent the GUI items in the task panel. Listing 6-16 presents the definition of the `TaskList` abstract base class.

Listing 6-16: The TaskList Class

```
public abstract class TaskList
{
    public virtual object GetPropertyValue(string propertyName);
    public abstract ICollection GetTaskItems();
    public virtual object InvokeMethod(string methodName, object userData);
    public virtual void SetPropertyValue(string propertyName, object value);

    public virtual bool IsDirty { get; }
}
```

Chapter 6: Understanding the Integrated Graphical Management System

The `TaskList` class features an abstract method named `GetTaskItems` that returns an `ICollection` object that contains the `TaskItem` objects. Besides being a container, it also exposes three important utility methods named `GetPropertyValue`, `SetPropertyValue`, and `InvokeMethod`.

The best way to understand what these utility methods do is to look at their internal implementations. Listing 6-17 presents the internal implementation of the `GetPropertyValue` method.

Listing 6-17: The `GetPropertyValue` Method

```
public virtual object GetPropertyValue(string propertyName)
{
    PropertyDescriptor descriptor1 =
        TypeDescriptor.GetProperties(this)[propertyName];
    return descriptor1.GetValue(this);
}
```

This method calls the `GetProperties` method of the `TypeDescriptor` class to access the `PropertyDescriptorCollection` object that contains one `PropertyDescriptor` object for each property that the `TaskList` (or its subclasses) exposes. It then uses the property name as an index into this collection to return the `PropertyDescriptor` object that describes the associated property. Finally, it calls the `GetValue` method on this `PropertyDescriptor` object to access the value of the property. Therefore the `GetPropertyValue` method generically retrieves the value of a property of a task list with a specified name.

The internal implementation of the `SetPropertyValue` method is shown in Listing 6-18.

Listing 6-18: The `SetPropertyValue` Method

```
public virtual void SetPropertyValue(string propertyName, object value)
{
    TypeDescriptor.GetProperties(this)[propertyName].SetValue(this, value);
}
```

This method uses the same generic approach to set the value of the property of the task list with the specified name.

Finally, Listing 6-19 presents the internal implementation of the `InvokeMethod` method of the `TaskList` class.

Listing 6-19: The `InvokeMethod` Method

```
public virtual object InvokeMethod(string methodName, object userData)
{
    Type type1 = base.GetType();
    MethodInfo info1 = type1.GetMethod(methodName);
    if (info1 != null)
    {
        if (userData == null)
            return info1.Invoke(this, null);

        return info1.Invoke(this, new object[] { userData });
    }
}
```

Chapter 6: Understanding the Integrated Graphical Management System

The `InvokeMethod` method first calls the `GetType` method to access the `Type` object that represents the `TaskList` (or its subclass):

```
Type type1 = base.GetType();
```

Then, it calls the `GetMethod` method of the `Type` object to return the `MethodInfo` object that represents the method with the specified name:

```
MethodInfo info1 = type1.GetMethod(methodName);
```

Finally, it calls the `Invoke` method of the `MethodInfo` object to generically invoke the method with the specified name and specified parameter. In other words, the `InvokeMethod` utility method is used to invoke a method of a task list with the specified name and parameter.

As Listing 6-16 shows, the `TaskList` is an abstract class. In the next chapter you learn how to extend this class to implement your own custom task list subclass.

ModulePageInfo

The `ModulePageInfo` class acts as a bag of properties that contain information about a particular module page. As Listing 6-20 demonstrates, the `ModulePageInfo` class contains the following information about its associated module page:

- ❑ **AssociatedModule:** Refers to the `Module` object that registers the associated module page with the IIS7 Manager. I cover the `Module` class in the next chapter.
- ❑ **Description:** Contains a short description that describes what the module page does.
- ❑ **IsEnabled:** Specifies whether the module page is enabled.
- ❑ **LargeImage:** Refers to the image shown in the header of the module page. For example, in the case of Figure 6-7, this property references the icon next to the “Failed Request Tracing Rules” header text.
- ❑ **LongDescription:** Contains the long description about the module page. For example, in the case of Figure 6-7, this property contains the text shown below the “Failed Request Tracing Rules” header text.
- ❑ **PageType:** Refers to the `Type` object that represents the type of the module page. For example, in the case of Figure 6-3, this property returns the value `typeof(CompilationPage)` because the module page shown in this figure is an instance of a control named `CompilationPage`.
- ❑ **SmallImage:** References the icon that represents the module page.
- ❑ **Title:** Specifies the header text of the module page. For example, in the case of Figure 6-7, this property contains the text “Failed Request Tracing Rules”.

Listing 6-20: The `ModulePageInfo` Class

```
public sealed class ModulePageInfo
{
    public ModulePageInfo(Module associatedModule, Type pageType, string title);
    public ModulePageInfo(Module associatedModule, Type pageType, string title,
```

(Continued)

Listing 6-20: (continued)

```
        string description);  
public ModulePageInfo(Module associatedModule, Type pageType, string title,  
        string description, object smallImage, object largeImage);  
public ModulePageInfo(Module associatedModule, Type pageType, string title,  
        string description, object smallImage, object largeImage,  
        string longDescription);  
  
public Module AssociatedModule { get; }  
public string Description { get; }  
public bool IsEnabled { get; }  
public object LargeImage { get; }  
public string LongDescription { get; }  
public Type PageType { get; }  
public object SmallImage { get; }  
public string Title { get; }  
}
```

TaskListCollection

As discussed earlier, every module page is associated with a task panel, where the task items associated with the module page are displayed. These task items are contained in a task list. As discussed earlier, every module page directly or indirectly inherits the `ModulePage` abstract base class, which in turn implements an interface named `IModulePage`. This interface exposes a property of type `TaskListCollection` named `Tasks`. As you'll see in the next chapter, each subclass of the `ModulePage` class must override the `Tasks` property to add its own task list to this collection. In other words, the `Tasks` property of a module page contains the task lists of all of its ancestor module pages.

As Listing 6-21 shows, the `TaskListCollection` class exposes a method named `GetTaskListItems` that returns an `IEnumerable` collection of `KeyValuePair` objects.

Listing 6-21: The TaskListCollection Class

```
public sealed class TaskListCollection : CollectionBase  
{  
    internal IEnumerable<KeyValuePair<TaskList, ICollection>> GetTaskListItems();  
    . . .  
}
```

The `KeyValuePair` is a new generic type in .NET 2.0. Each `KeyValuePair` object in this collection represents a key/value pair. The key is an object of type `TaskList` and the value is an object of type `ICollection`. Therefore, you can use a `TaskList` object as the key to retrieve the associated `ICollection` object from the `IEnumerable` collection. This `ICollection` object contains the `TaskItem` objects associated with the `TaskList` object.

In other words, the `GetTaskListItems` method returns a collection of `KeyValuePair` objects where each `KeyValuePair` object contains the `TaskItem` objects associated with a particular ancestor module page. You may find this a little confusing, but don't worry about it because I cover this in more detail in the next chapter in the context of an example.

Putting It All Together

The previous sections discussed some of the important classes in the IIS7 Manager architecture. This section looks under the hood to help you understand how these seemingly unrelated classes work together. Such understanding will put you in a much better position to extend the IIS7 Manager interface to add support for your own module pages.

When the IIS7 Manager is launched, an instance of an internal class named `WebMgrShellApplication` is instantiated and a method named `Execute` is called on this instance. As mentioned before, this class implements the `IServiceProvider` interface, which means that this class is a service provider. Listing 6-22 presents the internal implementation of the `Execute` method.

Listing 6-22: The Execute Method

```
public void Execute()
{
    _theApplication = this;
    _serviceContainer = new ServiceContainer();

    _managementHost = new Win32ManagementHost(_serviceContainer,
                                              "WebManagementShell", "MyTitle");
    _serviceContainer.AddService(typeof(IManagementHost), _managementHost);

    _navigationService = new NavigationService(_serviceContainer);
    _serviceContainer.AddService(typeof(INavigationService), _navigationService);

    _connectionManager = new ConnectionManager(_serviceContainer, true);
    _serviceContainer.AddService(typeof(IConnectionManager), _connectionManager);

    _mainForm = new ShellMainForm(this, null);
    Application.Run(this._mainForm);
}
```

The `Execute` method performs four important tasks. First, it instantiates an instance of a class named `ServiceContainer`. This class implements the `IServiceContainer` interface, which means that this class is a service container. The `AddService` method of the `ServiceContainer` class uses the service type as the key and stores the specified service instance under this key in an internal hash table. The `RemoveService` method of the `ServiceContainer` class removes the service with the specified service type (the key) from the internal hash table.

One unique thing about the `ServiceContainer` class is that if you call its `GetService` method, passing in the `typeof(IServiceContainer)`, it returns a reference to itself. To see the significance of this, let's look at the internal implementation of the `GetService` method of the `WebMgrShellApplication` class. As Listing 6-23 demonstrates, this method simply delegates to the `GetService` method of the `ServiceContainer`. This means that you can call the `GetService` method of the `WebMgrShellApplication` class, passing in the `typeof(IServiceContainer)` to access the underlying service container so you can add new services to this container. You'll see an example of this shortly.

Listing 6-23: The GetService Method of the WebMgrShellApplication Class

```
protected virtual object GetService(Type serviceType)
{
    return this._serviceContainer.GetService(serviceType);
}
```

Now back to the `Execute` method shown in Listing 6-22. As this code listing shows, the `Execute` method creates three new services named `ManagementHost`, `NavigationService`, and `ConnectionManager`. As mentioned, every service implements a particular interface. These three services respectively implement the `IManagementHost`, `INavigationService`, and `IConnectionManager` interfaces.

Notice that the `AddService` method of the `ServiceContainer` uses the `Type` object that represents the interface that the service implements as the key under which the service is stored in the internal hash table:

```
_navigationService = new NavigationService(_serviceContainer);
_serviceContainer.AddService(typeof(INavigationService), _navigationService);
```

Besides creating and registering the previously mentioned services, the `Execute` method of the `WebMgrShellApplication` creates an instance of a form named `ShellMainForm` and calls the `Run` method of the `Application` object to run this form:

```
_mainForm = new ShellMainForm(this, null);
Application.Run(this._mainForm);
```

The `ShellMainForm` is the form that contains the entire user interface of the IIS7 Manager. The `ShellMainForm` class derives from another form named `BaseForm`. Listing 6-24 presents a portion of the `BaseForm` constructor.

Listing 6-24: The BaseForm Constructor

```
protected BaseForm(IServiceProvider serviceProvider)
{
    this._serviceProvider = serviceProvider;
}
```

The `BaseForm` implements a method named `GetService` that simply delegates to the `GetService` method of the `IServiceProvider` object passed into its constructor, as shown in Listing 6-25.

Listing 6-25: The GetService Method of the BaseForm Class

```
protected internal object GetService(Type serviceType)
{
    return this._serviceProvider.GetService(serviceType);
}
```

Now back to the `ShellMainForm` class. As mentioned, this class inherits the `BaseForm` class. Listing 6-26 presents a portion of the internal implementation of the `ShellMainForm` constructor.

Listing 6-26: The ShellMainForm Class

```
public ShellMainForm(IServiceProvider serviceProvider,
                    ShellComponents shellComponents) : base(serviceProvider)
{
    _uiService = new ManagementUIService(this, this);
    _propertyEditingService = new PropertyEditingService();
    _assemblyDownloadService = new AssemblyDownloadService();

    IServiceContainer container1 =
        (IServiceContainer)base.GetService(typeof(IServiceContainer));
    container1.AddService(typeof(IManagementUIService), _uiService);
    container1.AddService(typeof(IPropertyEditingService), _propertyEditingService);
    container1.AddService(typeof(IAssemblyDownloadService), _assemblyDownloadService);

    _managementFrame = new ManagementFrame(serviceProvider, this);
    base.SuspendLayout();
    _managementFrame.SuspendLayout();
    _managementFrame.Dock = DockStyle.Fill;
    base.Controls.Add(_managementFrame);
    _managementFrame.ResumeLayout(false);
    base.ResumeLayout();
}
```

Note that the `ShellMainForm` creates the following three new services: `ManagementUIService`, `PropertyEditingService`, and `AssemblyDownloadService`:

```
_uiService = new ManagementUIService(this, this);
_propertyEditingService = new PropertyEditingService();
_assemblyDownloadService = new AssemblyDownloadService();
```

The `ShellMainForm` calls the `GetService` method of its base class, passing in the `typeof(IServiceContainer)` to return a reference to the `ServiceContainer` service container discussed before. This is possible because the `GetService` method of the `ServiceContainer` returns a reference to itself when it is called with the `typeof(IServiceContainer)` argument, as discussed earlier.

```
IServiceContainer container1 =
    (IServiceContainer)base.GetService(typeof(IServiceContainer));
```

The `ShellMainForm` then calls the `AddService` method of the service container three times to add the `ManagementUIService`, `PropertyEditingService`, and `AssemblyDownloadService` services:

```
container1.AddService(typeof(IManagementUIService), _uiService);
container1.AddService(typeof(IPropertyEditingService), _propertyEditingService);
container1.AddService(typeof(IAssemblyDownloadService), _assemblyDownloadService);
```

Finally, the `ShellMainForm` instantiates an instance of a frame named `ManagementFrame`, and adds the instance to its `Controls` collection (see Listing 6-26). In other words, the `ShellMainForm` has a single child control, which means that the `ManagementFrame` contains the entire user interface of the `IIS7`

Chapter 6: Understanding the Integrated Graphical Management System

Manager. Notice that the `ShellMainForm` passes the service provider object into the constructor of the `ManagementFrame` class, which means that this class can now access all the services that the `ShellMainForm` and `WebMgrShellApplication` have added to the `ServiceContainer` object.

```
_managementFrame = new ManagementFrame(serviceProvider, this);
base.Controls.Add(_managementFrame);
```

The `ShellMainForm` class implements an interface named `IManagementFrameHost`, which exposes bunch of methods and properties. We're only interested in the `UpdateUI` method, shown in Listing 6-27.

Listing 6-27: The UpdateUI Method

```
void IManagementFrameHost.UpdateUI()
{
    this._managementFrame.UpdateUI();
}
```

The `UpdateUI` method of the `ShellMainForm` delegates to the `UpdateUI` method of the `ManagementFrame`. As the name implies, this method is called to update the user interface of the IIS7 Manager when something changes. You'll see the significance of this method later.

Next, I discuss the `ManagementFrame` class. Listing 6-28 presents a simplified version of the internal implementation of the `ManagementFrame` constructor.

Listing 6-28: The ManagementFrame Constructor

```
public ManagementFrame(IServiceProvider serviceProvider,
                       IManagementFrameHost owner)
{
    _serviceProvider = serviceProvider;
    _owner = owner;
    INavigationService service2 =
        (INavigationService)_serviceProvider.GetService(typeof(INavigationService));
    service2.NavigationPerformed +=
        new NavigationEventHandler(OnNavigationPerformed);

    CreateHeader();
    CreateMenuBar();
    CreateStatusBar();
    CreateMainArea();
}
```

The `ManagementFrame` first accesses the navigation service and registers the `OnNavigationPerformed` method as an event handler for the `NavigationPerformed` event of this service. The `ManagementFrame` then calls the `CreateHeader`, `CreateMenuBar`, `CreateStatusBar`, and `CreateMainArea` methods to create the header, menu bar, status bar, and main area of the IIS7 Manager interface as shown in Figure 6-11.

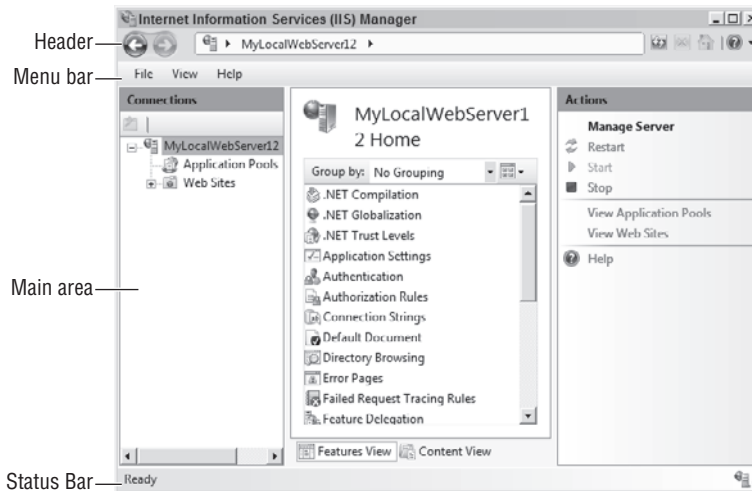


Figure 6-11

The main area is a standard `.NET System.Windows.Forms.Panel` control. As Figure 6-11 demonstrates, this panel (main area) consists of three columns. The left column is an internal control named `HierarchyPanel`. As the name implies, this control displays a hierarchy of items. The middle column is an internal control named `PageContainerPanel`, which inherits the standard `System.Windows.Forms.ContainerControl` control. The page container panel acts as a container for the module pages. In other words, the IIS7 Manager displays the modules pages in the page container panel.

```
private sealed class PageContainerPanel : ContainerControl
```

The right column is an internal control named `VerticalLayoutPanel`. As the name implies, the `VerticalLayoutPanel` lays out its child controls vertically. The `VerticalLayoutPanel` control acts as a container for the following two panels. The first panel displays the alerts and the second panel is a control named `TaskPanel` that contains the task items. For example, the `TaskPanel` control shown in Figure 6-11 contains the Restart, Start, Stop, View Application Pools, View Web Sites, and Help buttons.

Recall from Listing 6-28 that the `ManagementFrame` registers the `OnNavigationPerformed` method as an event handler for the `NavigationPerformed` event of the navigation service. The navigation service fires this event after navigating from one navigation item to another.

The navigation service is responsible only for navigation from the current navigation item to the new one. It is not responsible for actually rendering the module page associated with the new navigation item and the task items associated with this module page. Recall that each task item represents a GUI item in the task panel associated with the module page. The navigation service raises the `NavigationPerformed` event to allow the `OnNavigationPerformed` method of the `ManagementFrame` to render the associated module page and its task items.

Chapter 6: Understanding the Integrated Graphical Management System

Listing 6-29 contains the portion of the internal implementation of the `OnNavigationPerformed` event handler.

Listing 6-29: The `OnNavigationPerformed` Event Handler

```
private void OnNavigationPerformed(object sender, NavigationEventArgs e)
{
    NavigationItem item1 = e.NewItem;
    this.SetActiveConnection(item1.Connection);
    ModulePage modulePage = (ModulePage)item1.Page;
    ModulePage page1 = this._activePage;
    _activePage = modulePage;

    bool isInitialActivation = false;
    if (!_pageContainer.Controls.Contains(modulePage))
    {
        modulePage.Visible = false;
        _pageContainer.Controls.Add(modulePage);
        modulePage.Dock = DockStyle.Fill;
        isInitialActivation = true;
    }
    modulePage.OnActivated(isInitialActivation);
    ModulePageInfo info1 = modulePage.PageInfo;
    _pageContainer.Title = info1.Title;
    _pageContainer.PageDescription = info1.LongDescription;
    modulePage.Visible = true;
    modulePage.BringToFront();
    TaskListCollection collection1 = new TaskListCollection();
    foreach (TaskList list1 in ((IModulePage)_activePage).Tasks)
        collection1.Add(list1);

    IEnumerable<KeyValuePair<TaskList, ICollection>> enumerable1 =
                                                collection1.GetTaskListItems();
    _taskPanel.UpdateTasks(enumerable1);
}
```

The `OnNavigationPerformed` method first accesses the new navigation item:

```
NavigationItem item1 = e.NewItem;
```

Next, it accesses the `ModulePage` associated with the new navigation item and sets this module page as the active module page:

```
ModulePage modulePage = (ModulePage)item1.Page;
ModulePage page1 = _activePage;
_activePage = modulePage;
```

Then, it determines whether the `ModulePage` associated with the new navigation item is being accessed for the first time. If so, it adds the module page to the `Controls` collection of the page container panel.

```
bool isInitialActivation = false;
if (!_pageContainer.Controls.Contains(modulePage))
{
```

Chapter 6: Understanding the Integrated Graphical Management System

```
modulePage.Visible = false;
_pageContainer.Controls.Add(modulePage);
modulePage.Dock = DockStyle.Fill;
isInitialActivation = true;
}
```

Next, it accesses the `ModulePageInfo` object that contains the complete information about the module page and uses it to set the title and page description of the page container panel. The middle column of Figure 6-2 presents an example of this title and page description.

```
ModulePageInfo info1 = modulePage.PageInfo;
_pageContainer.Title = info1.Title;
_pageContainer.PageDescription = info1.LongDescription;
```

Then, it resets the visibility of the module page and brings the page to the front. Recall that the old module page is sitting in the front. As you can see, navigating from one module page to another does not remove the old page. It simply moves the new page to the front. In other words, the page container panel stacks up all the previously displayed modules.

```
modulePage.Visible = true;
modulePage.BringToFront();
```

So far, you've seen how the `OnNavigationPerformed` method displays the module page associated with the new navigation item. Next, you'll see how this method displays the task items associated with the module page. Recall that the task items are rendered in the task panel located in the Actions pane of the IIS7 Manager interface. As Listing 6-29 shows, the `OnNavigationPerformed` method first instantiates an instance of the `TaskListCollection` class:

```
TaskListCollection collection1 = new TaskListCollection();
```

Next, it iterates through the `TaskList` objects in the `Tasks` collection property of the module page and adds each enumerated `TaskList` object to the `TaskListCollection` object:

```
foreach (TaskList list1 in ((IModulePage)_activePage).Tasks)
    collection1.Add(list1);
```

Then, it calls the `GetTaskListItems` method of the `TaskListCollection` object to return an `IEnumerable` collection of `KeyValuePair`. As discussed before, each `KeyValuePair` contains the task items associated with a particular ancestor of the module page. In other words, the `IEnumerable` collection contains the task items that the module page and its ancestors have added.

```
IEnumerable<KeyValuePair<TaskList, ICollection>> enumerable1 =
    collection1.GetTaskListItems();
```

Finally, the `OnNavigationPerformed` method calls the `UpdateTasks` method of the task panel and passes the preceding `IEnumerable` object into it:

```
_taskPanel.UpdateTasks(enumerable1);
```


Chapter 6: Understanding the Integrated Graphical Management System

Under the hood, the `UpdateTasks` method iterates through the task items contained in the `IEnumerable` object and takes one of the following actions for each enumerated task item:

- ❑ If the task item is a `TextTaskItem`, it renders the specified text.
- ❑ If the task item is a `MethodTaskItem`, it renders a button and registers the specified method as the event handler for the `OnClick` event of the button.
- ❑ If the task item is a `MessageTaskItem`, it renders a warning, informational, or error message in the Alert panel.
- ❑ If the task item is a `GroupTaskItem`, it renders a tree view where each node is a task item. It also renders a plus sign next the root node if the tree view is collapsed and a minus sign if it's extended.

Summary

This chapter covered the main classes of the IIS7 Manager graphical architecture that you need to use to extend this architecture to add support for your own custom configuration sections. This chapter also looked under the hood where you saw how these classes work together and what roles they play in the IIS7 Manager. The next chapter shows you how to use what you've learned in this chapter to extend the IIS7 Manager interface.

7

Extending the Integrated Graphical Management System

The previous chapter provided in-depth coverage of the main classes of the IIS7 and ASP.NET 3.5 integrated graphical management architecture. This chapter shows you how to use what you've learned in the previous chapter to extend this architecture to add graphical management support for your own custom configuration sections.

You have to write two sets of managed code to extend the IIS7 and ASP.NET 3.5 integrated graphical management system: client-side managed code and server-side managed code. By client-side managed code I mean the managed code that you have to write to extend the user interface of the IIS7 Manager. By server-side managed code I mean the managed code that you have to write to enable the back-end Web server to communicate with the IIS7 Manager.

Client-Side Managed Code

The previous chapter presented examples of different types of module pages and task forms and briefly introduced the base classes from which these module pages and task forms derive. Figure 7-1 presents the class diagram for all the major classes that you will deal with when you're writing the client-side managed code. I discuss these classes in detail in the following sections.

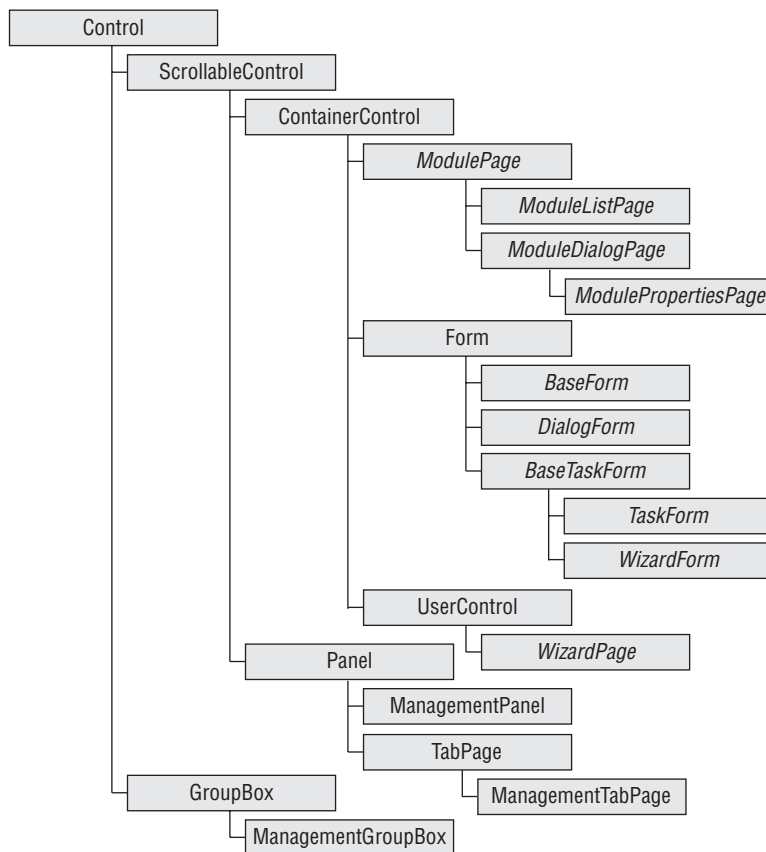


Figure 7-1

Extending the user interface of the IIS7 Manager involves the following tasks:

- ❑ Implement the appropriate module pages.
- ❑ Implement the necessary service proxies to facilitate the interaction between these module pages and the back-end Web server.
- ❑ Add support for new task items if necessary.
- ❑ Implement the task forms associated with these task items.
- ❑ Implement the necessary modules to register these module pages.

Keep this recipe in mind as you're reading through this chapter. To make our discussions more concrete, I'll use this recipe to extend the IIS7 Manager's user interface to add graphical management support for the `<myConfigSection>` configuration section discussed in Chapter 6. Launch Visual Studio and add a new Class Library project named `MyConfigSection`. Next, right-click the `MyConfigSection` project in the Solution Explorer and select the `Properties` option from the popup menu to launch the `Properties` dialog. Switch to the `Application` tab, enter `MyNamespace` in the "Default namespace" textbox, and save the changes.

Chapter 7: Extending the Integrated Graphical Management System

As mentioned earlier, extending the IIS7 and ASP.NET 3.5 integrated graphical management system requires two sets of managed code: client-side managed code and server-side managed code. Now add a directory named `GraphicalManagement` to the `MyConfigSection` project and add two subdirectories named `Client` and `Server` to the `GraphicalManagement` directory. As the names suggest, the `Client` and `Server` subdirectories will contain the managed classes that respectively make up the client-side and server-side managed code.

Recall that you extended the IIS7 and ASP.NET 3.5 integrated imperative management system in Chapter 5 to add support for the following imperative management classes:

- ❑ `MyCollectionItem` (see Listing 5-10)
- ❑ `MyCollection` (see Listing 5-11)
- ❑ `MyNonCollection` (see Listing 5-12)
- ❑ `MyConfigSection` (see Listing 5-13)
- ❑ `MyConfigSectionEnum` (see Listing 5-14)

As you'll see later in this chapter, the graphical management classes will use these imperative management classes. Add a new directory named `ImperativeManagement` to the `MyConfigSection` project. Add five source files named `MyCollectionItem.cs`, `MyCollection.cs`, `MyNonCollection.cs`, `MyConfigSection.cs`, and `MyConfigSectionEnum.cs` to the `ImperativeManagement` directory and add the code shown in Listings 5-10 through 5-14 to these source files, respectively. You also need to add a reference to the `Microsoft.Web.Administration.dll` assembly to the `MyConfigSection` project. This assembly is located in the following directory on your machine:

```
%windir%\System32\inetsvc
```

Now back to the implementation of the client-side managed code. Recall that the module page is the unit of graphical extensibility in the IIS7 Manager architecture. The configuration settings of a configuration section, such as `<myConfigSection>`, are normally exposed through one or more module pages that provide users with convenient graphical means to specify these settings. That means these module pages are tailored toward the specifics of their associated configuration sections.

If the IIS7 Manager architecture were to deal directly with the module pages associated with a particular configuration section, it would be tied to these specific module pages and consequently their associated configuration sections and would not be able to work with other configuration sections. That is why the IIS7 Manager extensibility model comes with an abstract base class named `ModulePage` that isolates the IIS7 Manager architecture from the specifics of the module pages being displayed. This base class defines the graphical API that all module pages must implement to allow the IIS7 Manager graphical architecture to interact with them in a generic fashion. Figure 7-2 shows the portion of the class diagram shown in Figure 7-1 that contains the class hierarchy of the `ModulePage` base class.

As this figure shows, the `ModulePage` base class acts as a scrollable container control for other controls on a module page. This base class does not provide any user interface and your module page class should not inherit directly from this base class. Instead you should derive from one of its subclasses. As Figure 7-2 shows, the `ModulePage` base class has two subclasses named `ModuleDialogPage` and `ModuleListPage`, and `ModuleDialogPage` also has a subclass named `ModulePropertiesPage`.

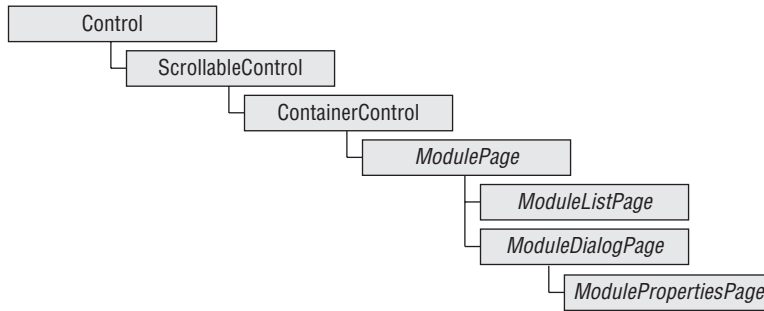


Figure 7-2

Your module page should inherit from the `ModuleDialogPage` if you want your module page to act like a traditional dialog box with typical command buttons such as `Apply` and `Cancel` (see Figure 6-1). Your module page should inherit from the `ModulePropertiesPage` if you want your module page to act like a traditional dialog box, but you also want to group the content of your module page to provide your users with a more user-friendly interface (see Figure 6-3). Your module page should inherit from the `ModuleListPage` if you need to present a set of items in a list with one or more columns (see Figure 6-2).

As an example, I'll design the appropriate module pages to allow the end users to specify the configuration settings of the `<myConfigSection>` configuration section directly from the IIS7 Manager. The first order of business is to decide which type of module pages you want to use. Keep in mind that you can use as many module pages as necessary to expose the configuration settings of a given configuration section. It does not have to be a single module page.

As a reminder, let's revisit the `<myConfigSection>` configuration section as shown in Listing 7-1.

Listing 7-1: The `<myConfigSection>` Configuration Section

```
<configuration>
  <system.webServer>
    <myConfigSection myConfigSectionBoolAttr="true"
      myConfigSectionEnumAttr="myConfigSectionEnumVal3">
      <myNonCollection myNonCollectionTimeSpanAttr="00:02:00" />
      <myCollection myCollectionIntAttr="50">
        <myAdd myCollectionItemBoolAttr="true"
          myCollectionItemIdentifier="myId1" />
        <myAdd myCollectionItemBoolAttr="false"
          myCollectionItemIdentifier="myId2" />
      </myCollection>
    </myConfigSection>
  </system.webServer>
</configuration>
```

Chapter 7: Extending the Integrated Graphical Management System

In this case I'll use two module pages. The first module page is a module dialog page named `MyConfigSectionPage`, which allows the end user to specify the values of the following attributes:

- ❑ The `myConfigSectionBoolAttr` and `myConfigSectionEnumAttr` attributes of the `<myConfigSection>` element
- ❑ The `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` element
- ❑ The `myCollectionIntAttr` attribute of the `<myCollection>` element

The second module page is a module list page named `MyCollectionPage`, which

- ❑ Displays the collection items
- ❑ Allows the user to add a new collection item and specify the values of its associated `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes
- ❑ Allows the user to delete a collection item
- ❑ Allows the user to edit the value of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of a collection item
- ❑ Allows the user to change the identifier of a collection item

Custom Module Pages and Task Forms in Action

Before diving into the details of the implementation of these two module pages, first let's see them in action. As Figure 7-3 shows, the Web server home page includes a new item named `MyConfigSection`.

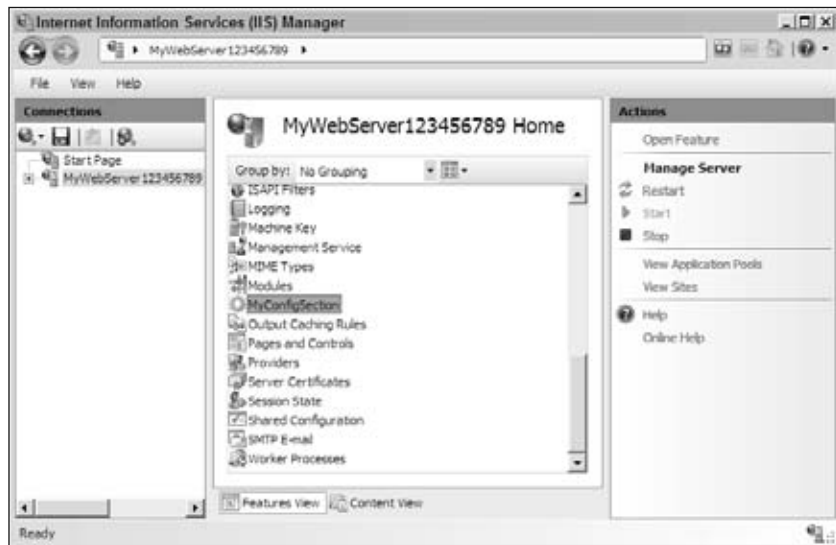


Figure 7-3

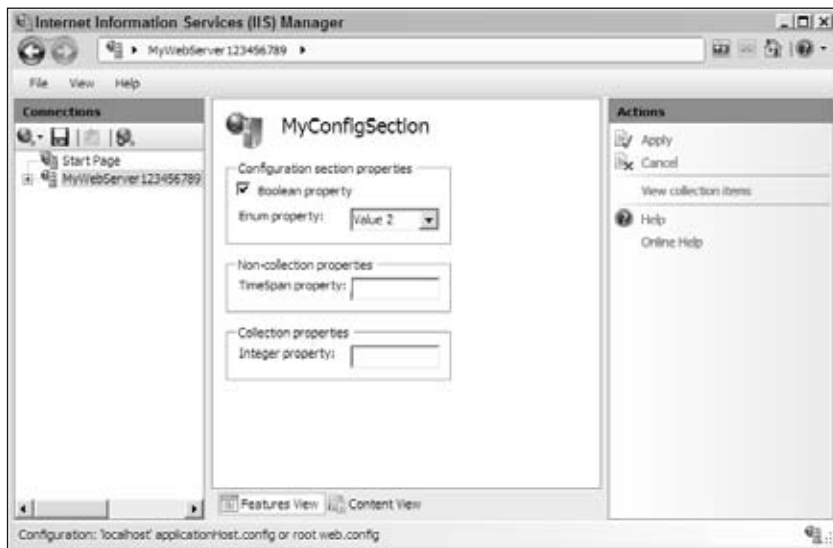


Figure 7-4

As you'll see later, this module page inherits from the `ModuleDialogPage` class. The page consists of three group boxes as follows:

- ❑ The first group box allows the user to specify the properties of the configuration section itself, that is, the attributes on the `<myConfigSection>` containing element.
- ❑ The second group box allows the user to specify the non-collection properties, that is, the attributes on the `<myNonCollection>` non-collection element.
- ❑ The third group box allows the user to specify the collection properties, that is, the attributes on the `<myCollection>` collection element.

Note that the Actions pane contains the Apply and Cancel buttons that you can find on a typical dialog box. The user makes the changes in the group boxes and clicks Apply to commit the changes to the underlying configuration file. Because the `MyConfigSectionPage` class derives from the `ModuleDialogPage` class, it automatically inherits the Apply and Cancel command buttons from the base class.

The command button labeled "View collection items" allows the user to navigate to a page named `MyCollectionPage` as shown in Figure 7-5.

This page inherits from the `ModuleListPage` base class and displays the list of available collection items, and the values of their identifier and Boolean properties. Notice that the Actions panel contains a link named "Add collection item." When the user clicks this link, the task form named

Chapter 7: Extending the Integrated Graphical Management System

MyCollectionItemTaskForm pops up, as shown in Figure 7-6. This task form allows the user to specify the properties of the collection item being added.

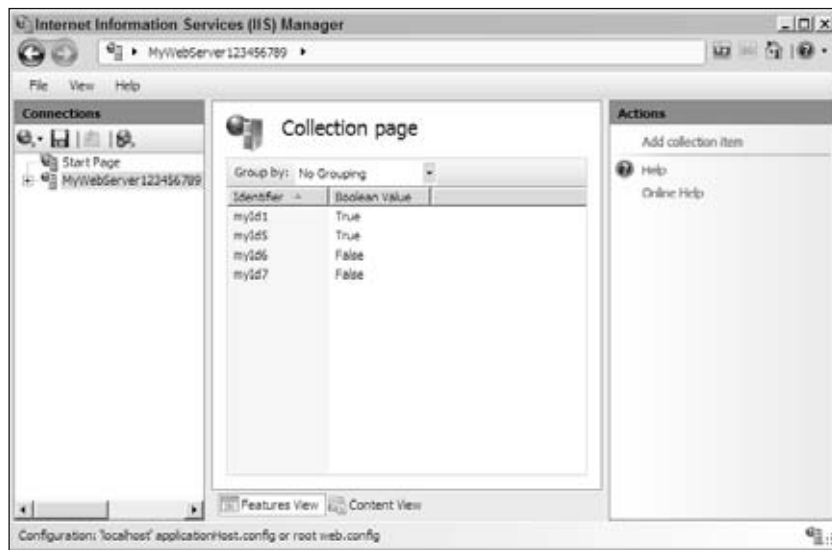


Figure 7-5

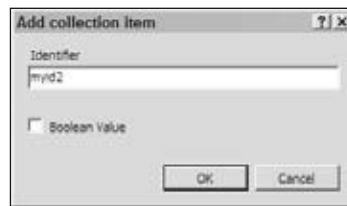


Figure 7-6

As Figure 7-7 shows, when the user selects an item from the list, the **Actions** panel displays three new links named “Update collection item,” “Change identifier,” and “Delete collection item.” The user clicks the “Update collection item” and “Delete collection item” links to respectively update and delete the selected item and the “Change identifier” link to change the identifier of the selected item.

When the user clicks the “Update collection item” link, the task form shown in Figure 7-8 is displayed. This task form allows the user to modify the selected item. The user can also launch this task form by double-clicking the selected item.



Figure 7-7

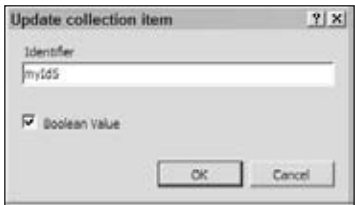


Figure 7-8

When the user selects an item from the list of displayed items and takes one of the following actions, the label that displays the identifier of the selected item becomes editable, as shown in Figure 7-9:

- ❑ The user clicks the selected item.
- ❑ The user clicks the “Change identifier” link.

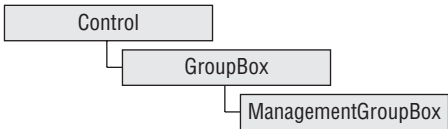


Figure 7-9

As you can see from Figure 7-10, the `MyCollectionPage` module list page includes a Group by combo box that contains a grouping criterion named Boolean Property. When the users select this grouping criterion from this combo box, they’ll see the result shown in Figure 7-11.

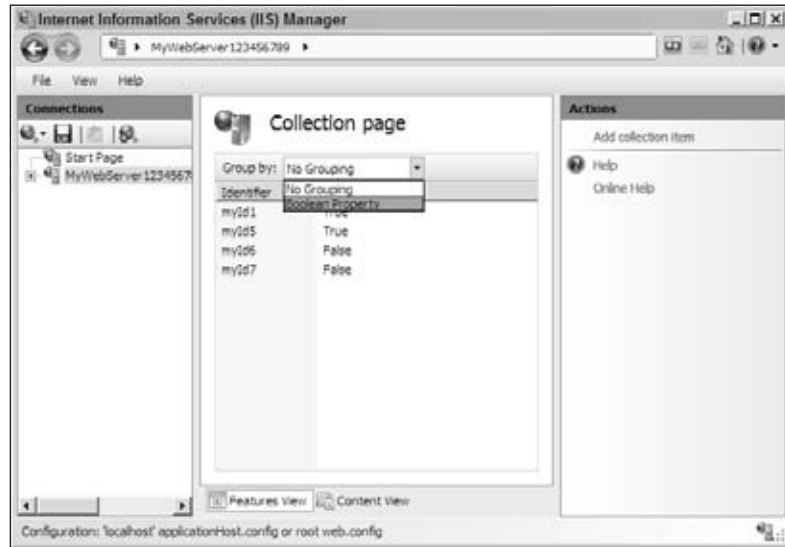


Figure 7-10

As you can see from Figure 7-11, the displayed items are grouped in two groups titled True and False. The True group contains collection items with a Boolean property value of `true`. The False group contains collection items with a Boolean property value of `false`.

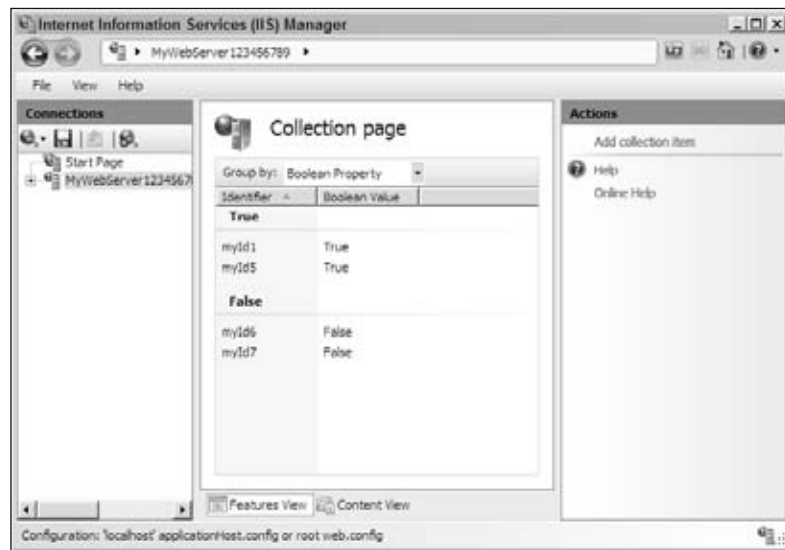


Figure 7-11

Proxies

As the previous discussion shows, you need to implement two module pages named `MyConfigSectionPage` and `MyCollectionPage` and a task form named `MyCollectionItemTaskForm`. What about the communications between these module pages and task form and the back-end Web server?

The IIS7 Manager is a desktop application and runs locally on the client machine. The back-end Web server, on the other hand, may or may not run locally on the client machine. In fact, IIS7 supports both local and remote administration. In the remote administration scenario, the IIS7 Manager and back-end Web server run in two different machines and communicate through the HTTPS protocol. This protocol provides two important benefits. First, it's firewall-friendly. Second, it's secure because it uses an SSL channel to transmit the messages between the IIS7 Manager and the back-end Web server.

Obviously, the `MyConfigSectionPage` and `MyCollectionPage` module pages and the `MyCollectionItemTaskForm` task form need to communicate with the back-end server. Let's study the details of these communications.

There are two communication scenarios involving the `MyConfigSectionPage` module page and the back-end server. In the first scenario, when the user double-clicks the `MyConfigSection` item in Figure 7-3 and navigates to the `MyConfigSectionPage` module page shown in Figure 7-4, this module page must ensure that its user interface reflects the current values of the associated configuration settings in the underlying configuration file as follows:

- ❑ The `CheckBox` and `ComboBox` controls shown in the top group box of Figure 7-4 must reflect the current values of the `myConfigSectionBoolAttr` and `myConfigSectionEnumAttr` attributes on the `<myConfigSection>` Containing element.
- ❑ The `TextBox` control shown in the middle group box of Figure 7-4 must display the current values of the `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` Non-collection element.
- ❑ The `TextBox` control shown in the bottom group box of Figure 7-4 must display the current value of the `myCollectionIntAttr` attribute of the `<myCollection>` element.

Therefore, the `MyConfigSectionPage` module page must retrieve the values of the `myConfigSectionBoolAttr`, `myConfigSectionEnumAttr`, `myNonCollectionTimeSpanAttr`, and `myCollectionIntAttr` attributes from the back-end Web server. The server-side code normally contains a class that exposes methods that the client-side code, such as the `MyConfigSectionPage` module page, must invoke to retrieve the current values of the required configuration settings such as the preceding attributes. As you'll see later, in our case the server-side class is a class named `MyConfigSectionModuleService` that exposes a method named `GetSettings` that returns the values of the attributes.

In the second communication scenario, the user makes changes in the `MyConfigSectionPage` module page's user interface as follows:

- ❑ The end user toggles the `CheckBox` control, or selects a new item from the `ComboBox` control shown in the top group box of Figure 7-4 to change the current values of the `myConfigSectionBoolAttr` and `myConfigSectionEnumAttr` attributes of the `<myConfigSection>` containing element.

- ❑ The user changes the value displayed in the `TextBox` control shown in the middle group box of Figure 7-4 to change the current value of the `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` non-collection element.
- ❑ The user changes the value displayed in the `TextBox` control shown in the bottom group box of Figure 7-4 to change the current value of the `myCollectionIntAttr` attribute of the `<myCollection>` element.

After making one or more of these changes, the user clicks the `Apply` button to commit the changes to the underlying configuration file, which means that the `MyConfigSectionPage` module now needs to invoke the appropriate method of the server-side class to update the current values of the `myConfigSectionBoolAttr`, `myConfigSectionEnumAttr`, `myNonCollectionTimeSpanAttr`, and `myCollectionIntAttr` attributes in the underlying configuration file. As you'll see later, the server-side `MyConfigSectionModuleService` class exposes a method named `UpdateSettings` that takes the new values and performs the necessary updates.

Next, I discuss the communication scenarios between the back-end `MyConfigSectionModuleService` class and the `MyCollectionPage` module page. When the end user clicks the "View collection items" link in the task panel of Figure 7-4 and navigates to the `MyCollectionPage` module page shown in Figure 7-5, this module page must invoke the appropriate method of the `MyConfigSectionModuleService` server-side class to retrieve the collection items and display them to the end user. In this case, the server-side `MyConfigSectionModuleService` class features a method named `GetCollectionItems` that returns the collection items.

In the second communication scenario, the user selects an item from the list shown in Figure 7-5 and clicks the `Delete` collection item link in the task panel. In this case, the `MyCollectionPage` module page must call the appropriate method of the server-side class to delete the associated collection item from the underlying configuration file. In our case, the server-side `MyConfigSectionModuleService` class exposes a method named `DeleteCollectionItem` that takes the value of the `myCollectionItemIdentifier` attribute as its argument and deletes the associated collection item from the configuration file.

In the third communication scenario, the user selects an item from the list of displayed items shown in Figure 7-5, either clicks the selected item, or clicks the "Change identifier" link to make the label that displays the identifier of the selected item editable, and finally changes the identifier of the selected item. In this case, the `MyCollectionPage` module list page must call the appropriate method of the server-side class to change the identifier of the selected item in the underlying configuration file. In our case, the server-side `MyConfigSectionModuleService` class exposes a method named `UpdateCollectionItemIdentifier` that performs the actual update in the configuration file.

Next, I discuss the communications between the back-end `MyConfigSectionModuleService` server-side class and the `MyCollectionItemTaskForm` task form. The first scenario occurs when the end user clicks the "Add collection item" link in the task panel in Figure 7-5 and launches the task form shown in Figure 7-6 to add a new collection item. After the user specifies the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of the new collection item and clicks OK on the `MyCollectionItemTaskForm` task form shown in Figure 7-6, this task form must call the appropriate method of the server-side class to add a new collection item with the specified attribute values to the underlying configuration file. In this case, the server-side `MyConfigSectionModuleService` class exposes a method named `AddCollectionItem` that takes these attribute values and adds the new item.

Chapter 7: Extending the Integrated Graphical Management System

The second communication scenario involving the `MyCollectionItemTaskForm` task form occurs when the end user double-clicks an item from the list shown in Figure 7-7 and launches this task form (see Figure 7-8) to update the current values of the `myCollectionItemIdentifier` and `myCollectionItemBoolAttr` attributes of the selected collection item. When the user is done with editing and clicks the OK button, the `MyCollectionItemTaskForm` task form must call the appropriate method of the `MyConfigSectionModuleService` server-side class to update the attribute values of the selected collection item in the configuration file. In this case, the server-side class exposes a method named `UpdateCollectionItem` that takes the new attribute values and updates the underlying attributes.

In summary, the `MyConfigSectionPage` and `MyCollectionPage` module pages and `MyCollectionItemTaskForm` task form need to communicate with the back-end Web server on a number of occasions as discussed. If the `MyConfigSectionPage` and `MyCollectionPage` module pages and `MyCollectionItemTaskForm` task form were to include the logic that talks to the back-end `MyConfigSectionModuleService` server-side class directly, they would have to worry about issues such as whether the Web server is running remotely, and if so how to use HTTPS to communicate with it. Imagine how complex your GUI code would be if it were to include the logic that handles issues like these two.

ModuleServiceProxy

The `MyConfigSectionPage` and `MyCollectionPage` module pages and `MyCollectionItemTaskForm` task form are not the only entities that need this communication logic to communicate with the back-end server. Other entities such as other module pages and task forms also need to use the same logic to communicate with the server. This is yet another reason why this communication logic should not be directly included into the entities that use this logic.

The IIS7 Manager architecture encapsulates the logic that deals with the details of the communications with the back-end server-side class into a standard client-side base class named `ModuleServiceProxy`. The methods and properties of this base class define the API that the `MyConfigSectionPage` and `MyCollectionPage` module pages and `MyCollectionItemTaskForm` task form can use to communicate indirectly with the back-end class and invoke its methods without getting involved in the dirty little communication details. All you have to do is to implement a client-side class known as a *proxy* that inherits the `ModuleServiceProxy` base class and exposes methods with the same signatures as the methods of the back-end server-side class. The implementation of these methods of this client-side proxy class must use the `Invoke` method of the `ModuleServiceProxy` base class to invoke the associated methods of the server-side class. This proxy class in our case is a class named `MyConfigSectionModuleServiceProxy` as shown in Listing 7-2.

Add a new source file named `MyConfigSectionModuleServiceProxy.cs` to the `GraphicalManagement/Client` subdirectory of the `MyConfigSection` project and add the code shown in Listing 7-2 to this source file. You also need to add a reference to the `Microsoft.Web.Management.dll` assembly to the `MyConfigSection` project because this assembly contains the `ModuleServiceProxy` and `PropertyBag` types. This assembly is located in the following directory on your machine:

```
%windir%\System32\inetsrv
```

Listing 7-2: The MyConfigSectionModuleServiceProxy Class

```
using Microsoft.Web.Management.Client;
using Microsoft.Web.Management.Server;

namespace MyNamespace
{
    class MyConfigSectionModuleServiceProxy : ModuleServiceProxy
    {
        public PropertyBag GetCollectionItems()
        {
            return (PropertyBag)base.Invoke("GetCollectionItems", new object[0]);
        }

        public void UpdateCollectionItem(PropertyBag itemToUpdate)
        {
            base.Invoke("UpdateCollectionItem", new object[] { itemToUpdate });
        }

        public void DeleteCollectionItem(PropertyBag itemToDelete)
        {
            base.Invoke("DeleteCollectionItem", new object[] { itemToDelete });
        }

        public void AddCollectionItem(PropertyBag itemToAdd)
        {
            base.Invoke("AddCollectionItem", new object[] { itemToAdd });
        }

        public PropertyBag GetSettings()
        {
            return (PropertyBag)base.Invoke("GetSettings", new object[0]);
        }

        public void UpdateSettings(PropertyBag updatedSettings)
        {
            base.Invoke("UpdateSettings", new object[] { updatedSettings });
        }

        public bool UpdateCollectionItemIdentifier(string oldIdentifier,
                                                    string newIdentifier)
        {
            return (bool)base.Invoke("UpdateCollectionItemIdentifier",
                                     new object[] { oldIdentifier, newIdentifier });
        }
    }
}
```

Each method of your custom proxy class must meet the following two important requirements:

- ❑ It must have the same signature as the corresponding server-side method. Recall that the signature of a method includes its name, return type, and its parameter types. For example, as you'll see later, the server-side `MyConfigSectionModuleService` class exposes the methods shown in Listing 7-3. If you compare this code listing with Listing 7-2 you'll notice that the

Chapter 7: Extending the Integrated Graphical Management System

MyConfigSectionModuleServiceProxy proxy class exposes methods with the same signatures as the server-side MyConfigSectionModuleService class.

- ❑ It must call the `Invoke` method of the `ModuleServiceProxy` base class and pass the following two parameters into it:
 - ❑ The name of the corresponding server-side method.
 - ❑ An array of objects where each object contains the value of a particular parameter of the corresponding server-side method. The order of the objects in this array must be the same as the order of the parameters of the server-side method.

For example, the `UpdateSettings` method of the `MyConfigSectionModuleServiceProxy` proxy class (see Figure 7-2) calls the `Invoke` method of the `ModuleServiceProxy` base class and passes the following two parameters into it:

- ❑ The string value "UpdateSettings", which is the name of the corresponding server-side method (see Listing 7-3)
- ❑ An array that contains a single element of type `PropertyBag`, which is the value of the parameter of the `UpdateSettings` server-side method (see Listing 7-3)

```
public void UpdateSettings(PropertyBag updatedSettings)
{
    base.Invoke("UpdateSettings", new object[] { updatedSettings });
}
```

Listing 7-3: The MyConfigSectionModuleService Server-Side Class

```
using Microsoft.Web.Management.Server;
using System.Collections;
using System;

namespace MyNamespace.GraphicalManagement.Server
{
    class MyConfigSectionModuleService : ModuleService
    {
        [ModuleServiceMethod]
        public PropertyBag GetSettings();

        [ModuleServiceMethod]
        public void UpdateSettings(PropertyBag updatedSettings);

        [ModuleServiceMethod]
        public PropertyBag GetCollectionItems();

        [ModuleServiceMethod]
        public void AddCollectionItem(PropertyBag bag);

        [ModuleServiceMethod]
        public void DeleteCollectionItem(PropertyBag bag);

        [ModuleServiceMethod]
        public void UpdateCollectionItem(PropertyBag bag);

        [ModuleServiceMethod]
        public bool UpdateCollectionItemIdentifier(string oldIdentifier,
                                                    string newIdentifier);
    }
}
```

As Listings 7-2 and 7-3 show, the client-side `MyConfigSectionModuleServiceProxy` proxy class and the server-side `MyConfigSectionModuleService` class use a `PropertyBag` object to exchange data. For example, the `AddCollectionItem` method of the `MyConfigSectionModuleServiceProxy` proxy class sends the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of the new collection item through a `PropertyBag` object.

What's PropertyBag Anyway?

As you saw in the previous sections, the server-side class and the client-side proxy exchange configuration data. The proxy and its associated server-side class normally encapsulate this data in an instance of a class named `PropertyBag`, which is optimized to improve performance.

Listing 7-4 presents the members of the `PropertyBag` class. Note that the `PropertyBag` implements the `IDictionary` interface, which means that it's a collection of `DictionaryEntry` objects.

Listing 7-4: The PropertyBag Class

```
public sealed class PropertyBag : IDictionary, ICollection, IEnumerable
{
    // Methods
    public PropertyBag();
    public PropertyBag(bool trackState);
    public void Add(int key, object value);
    public PropertyBag Clone();
    public PropertyBag Clone(bool readOnly);
    public bool Contains(int key);
    public static PropertyBag CreatePropertyBagFromState(string state);
    public static PropertyBag CreatePropertyBagFromState(string state,
                                                         bool readOnly);

    public string GetState();
    public T GetValue<T>(int index);
    public T GetValue<T>(int index, T defaultValue);
    public bool IsModified();
    public bool IsModified(int key);
    public void Remove(object key);

    // Properties
    public int Count { get; }
    public bool IsTrackingState { get; }
    public object this[int index] { get; set; }
    public ICollection Keys { get; }
    public ICollection ModifiedKeys { get; }
}
```

To help you understand the significance of the `PropertyBag` class, let's take a look at the internal implementation of some of its members.

Constructors

Listing 7-5 presents the internal implementations of the constructors of this class.

Listing 7-5: Constructors of the PropertyBag Class

```
public PropertyBag() : this(false) { }

public PropertyBag(bool trackState)
{
    this.bag = new HybridDictionary();
    if (trackState)
        this.modifiedKeys = new HybridDictionary();
}

private IDictionary bag;
```

Note that the `PropertyBag` class exposes a default constructor and a constructor with a Boolean parameter. The default constructor calls the other constructor, passing in the value of `false` as its parameter. Notice that the constructor instantiates an instance of the `HybridDictionary` class and assigns the instance to a private `IDictionary` field named `bag`.

When you call the `Add` method of the `PropertyBag` to add a new `DictionaryEntry` object into it, the `Add` method under the hood adds this object into the `bag` `IDictionary` field. When you call the `Remove` method of the `PropertyBag` to remove a `DictionaryEntry` object, the `Remove` method under the hood removes this object from the `bag` `IDictionary` field.

The great thing about the `HybridDictionary` class is its performance. The `HybridDictionary` collection stores its items in an internal `ListDictionary` collection, which is optimized for small numbers of items. When the number of items exceeds this optimum number, the `HybridDictionary` moves its items from this internal `ListDictionary` to an internal `Hashtable`. If the number of items exceeds this optimum number to begin with, the `HybridDictionary` stores them in the `Hashtable` to begin with.

Indexer

As Listing 7-5 shows, if the Boolean value of `true` is passed into the `PropertyBag` constructor, it creates two instances of the `HybridDictionary` class instead of one. The second `HybridDictionary` instance is assigned to a private field of type `IDictionary` named `modifiedKeys`. To see the significance of this private field, take a look at the internal implementation of the indexer property of the `PropertyBag` class as presented in Listing 7-6.

Listing 7-6: The Indexer Property of the PropertyBag Class

```
public object this[int index]
{
    get
    {
        if (index < 0)
            throw new ArgumentOutOfRangeException("index");
        return this.bag[index];
    }
    set
    {
        if (index < 0)
            throw new ArgumentOutOfRangeException("index");
```

Listing 7-6: *(continued)*

```
        if (this.ReadOnly)
            throw new InvalidOperationException("Cannot Modify Readonly Collection");

        object oldItem = this.bag[index];
        this.bag[index] = value;
        if ((this.modifiedKeys != null) && !object.Equals(oldItem, value))
            this.modifiedKeys[index] = string.Empty;
    }
}
```

Note that the indexer of the `PropertyBag` class delegates to the indexer of the `bag` `HybridDictionary` field. Now let's study the setter of this indexer more closely. The setter checks whether the `PropertyBag` is marked as read-only. If so, it throws an exception. Next, it accesses the item with the specified index and stores it in a local variable named `oldItem` and replaces the item with the new item:

```
object oldItem = this.bag[index];
this.bag[index] = value;
```

Then, it checks whether the `modifiedKeys` `HybridDictionary` collection has been instantiated. Recall from Listing 7-5 that the `modifiedKeys` collection is instantiated only when the value of `true` is passed into the `PropertyBag` constructor. If the `modifiedKeys` collection has been instantiated and if the new item is different from the old item, the setter adds a new entry to the `modifiedKeys` collection:

```
if ((this.modifiedKeys != null) && !object.Equals(oldItem, value))
    this.modifiedKeys[index] = string.Empty;
```

Therefore, the presence of an entry in the `modifiedKeys` collection at a specified index indicates that the entry in the `bag` collection at the same index has been replaced.

In other words, the `bag` collection contains the items and the `modifiedKeys` collection specifies which items in the `bag` collection have been changed. This means that if you pass the value of `true` to the `PropertyBag` constructor when you're instantiating a `PropertyBag` object, the object will track its state changes.

ModifiedKeys

As Listing 7-7 shows, the `ModifiedKeys` property of the `PropertyBag` class returns the `Keys` collection property of the `modifiedKeys` collection. In other words, it returns an `ICollection` that contains those indexes in the `bag` `HybridDictionary` whose associated values have changed.

Listing 7-7: The ModifiedKeys Property

```
public ICollection ModifiedKeys
{
    get
    {
        if (!this.IsTrackingState)
            throw new InvalidOperationException(
                "The state changes are not being tracked.");
        return this.modifiedKeys.Keys;
    }
}
```

GetState

As Listing 7-4 shows, the `PropertyBag` class exposes a method named `GetState`. Listing 7-8 presents the internal implementation of this method. As the name suggests, the `GetState` method returns a string that contains the state or content of the `PropertyBag`. In other words, the `GetState` method serializes the `PropertyBag` into a string.

Listing 7-8: The `GetState` Method of the `PropertyBag` Class

```
public string GetState()
{
    ObjectStateFormatter formatter1 = new ObjectStateFormatter();
    return formatter1.Serialize(this);
}
```

As Listing 7-8 shows, the `GetState` method instantiates an `ObjectStateFormatter` and calls its `Serialize` method, passing in the reference to the `PropertyBag` object. The `Serialize` method serializes this object into a string.

CreatePropertyBagFromState

As Listing 7-4 shows, the `PropertyBag` class also exposes a method named `CreatePropertyBagFromState`. Listing 7-9 shows the internal implementation of this method.

Listing 7-9: The `CreatePropertyBagFromState` Method of the `PropertyBag` Class

```
public static PropertyBag CreatePropertyBagFromState(string state, bool readOnly)
{
    ObjectStateFormatter formatter = new ObjectStateFormatter();
    PropertyBag bag = (PropertyBag) formatter.Deserialize(state);
    if (readOnly)
        bag.SetReadOnly();
    return bag;
}
```

The `CreatePropertyBagFromState` method instantiates an `ObjectStateFormatter` and calls its `Deserialize` method, passing in the string that contains the serialized version of the `PropertyBag` object. The `Deserialize` method deserializes a `PropertyBag` object from this string. In other words, the `CreatePropertyBagFromState` method does the opposite of the `GetState` method.

You may be wondering what this serialization is for. As discussed earlier, the `MyConfigSectionPage` and `MyCollectionPage` module pages and `MyCollectionItemTaskForm` task form need to communicate with the back-end Web server on a number of occasions. In other words, the methods of the client-side `MyConfigSectionModuleServiceProxy` proxy class (see Listing 7-2) must exchange data with the methods of the server-side `MyConfigSectionModuleService` class. This data has to be serialized on the sender side (which could be the client or server) and deserialized on the receiver side.

Therefore, the performance and interactivity of the IIS7 Manager depends on the performance of this serialization/deserialization process, especially when the back-end Web server and the IIS7 Manager are not running on the same machine. One of the great things about the `PropertyBag` class is that its serialization/deserialization is optimized as long as it doesn't contain complex data types. Therefore, when

you're writing your own custom module pages and task forms you must ensure that you do not store complex data types in the `PropertyBag` object that the methods of your proxy class pass to the server.

MyConfigSectionPage

In this section, I present and discuss the implementation of the `MyConfigSectionPage` module dialog page. The `MyConfigSectionPage` module dialog page, like any other module dialog page, derives from the `ModuleDialogPage` abstract base class, which in turn derives from the `ModulePage` abstract base class.

The `ModuleDialogPage` base class defines the API that every module page must implement in order to act as a dialog box with typical command buttons such as `Apply` and `Cancel`. Listing 7-10 presents those members of the `ModuleDialogPage` base class that the `MyConfigSectionPage` class overrides.

Listing 7-10: The `ModuleDialogPage` Base Class

```
public abstract class ModuleDialogPage : ModulePage
{
    // Methods
    protected abstract bool ApplyChanges();
    protected abstract void CancelChanges();
    protected virtual void OnRefresh();

    // Properties
    protected abstract bool CanApplyChanges { get; }
    protected override TaskListCollection Tasks { get; }
}
```

The `ModuleDialogPage` base class features the following important overridable members:

- ❑ `ApplyChanges`: Your custom module dialog page's implementation of the `ApplyChanges` method must contain the code that you want to run when the end user clicks the `Apply` button.
- ❑ `CancelChanges`: Your custom module dialog page's implementation of the `CancelChanges` method must contain the code that you want to run when the end user clicks the `Cancel` button.
- ❑ `OnRefresh`: Your custom module dialog page's implementation of the `OnRefresh` method must contain the code that you want to run when your module dialog page is refreshed.
- ❑ `CanApplyChanges`: Your custom module dialog page's implementation of the `CanApplyChanges` property must return a Boolean value specifying whether the changes can be committed to the underlying configuration file.
- ❑ `Tasks`: Your custom module dialog page's implementation of the `Tasks` property must take the following four steps:
 - ❑ Instantiate an instance of the `TaskListCollection` class.
 - ❑ Populate this instance with the content of the `Tasks` property of its base class.
 - ❑ Create new `TextTaskItem`, `MessageTaskItem`, `MethodTaskItem`, and `GroupTaskItem` task items as necessary.
 - ❑ Add these task items to the `TaskListCollection` instance and return the instance.

Chapter 7: Extending the Integrated Graphical Management System

Because the `ModuleDialogPage` base class inherits from the `ModulePage` base class, you may need to override some of the methods and properties of the `ModulePage` base class as well. Listing 7-11 presents those members of the `ModulePage` base class that the `MyConfigSectionPage` module dialog page overrides:

- ❑ **OnActivated:** Your module page's implementation of the `OnActivated` method must connect to the back-end Web server, retrieve the required configuration settings, and populate the module page with these settings.
- ❑ **CanRefresh:** Override this property to specify whether your module page can refresh.
- ❑ **HasChanges:** Override this property to implement the logic that determines whether the end user has edited the configuration settings displayed in your module page.
- ❑ **ReadOnly:** Override this property to specify whether your module page is in read-only mode where the user cannot edit the configuration settings that your module page displays. As you'll see later in this chapter, the value of this property reflects the value of the `isLocked` Boolean attribute on the associated configuration section. When this Boolean attribute is set to `true` in a configuration file at a specified configuration hierarchy level, none of the lower-level configuration files are allowed to change the associated configuration settings. As you'll see later in this chapter, your custom module page must disable its GUI elements if the `isLocked` Boolean attribute is set to `true`.

Listing 7-11: The `ModulePage` Abstract Base Class

```
public abstract class ModulePage : ContainerControl, IModulePage, IDisposable
{
    protected virtual void OnActivated(bool initialActivation);

    protected virtual bool CanRefresh { get; }
    protected virtual bool HasChanges { get; }
    protected virtual bool ReadOnly { get; }
}
```

Listing 7-12 presents the declarations of the members of the `MyConfigSectionPage` module dialog page. Now add a new source file named `MyConfigSectionPage.cs` to the `GraphicalManagement/Client` directory of the `MyConfigSection` project and add the code shown in this code listing to this source file. Note that Listing 7-12 does not contain the implementation of the methods and properties of the `MyConfigSectionPage` class. I present and discuss the implementation of these methods and properties in the following sections. You also need to add references to the `System.Drawing.dll` and `System.Windows.Forms.dll` assemblies to the `MyConfigSection` project because the `MyConfigSectionPage` module dialog page uses some of the classes from these two assemblies.

Listing 7-12: The `MyConfigSectionPage` Module Dialog Page

```
using Microsoft.Web.Management.Client.Win32;
using Microsoft.Web.Management.Client;
using Microsoft.Web.Management.Server;
using System.ComponentModel;
using System.Collections;
using System.Windows.Forms;
using System.Drawing;
using System;
using MyNamespace.ImperativeManagement;
```

Listing 7-12: (continued)

```
namespace MyNamespace.GraphicalManagement.Client
{
    class MyConfigSectionPage: ModuleDialogPage
    {
        private ManagementGroupBox myConfigSectionPropertiesGroupBox;
        private CheckBox myConfigSectionBoolPropertyCheckBox;
        private Label myConfigSectionEnumPropertyLabel;
        private ComboBox myConfigSectionEnumPropertyComboBox;
        private ManagementGroupBox myNonCollectionGroupBox;
        private Label myNonCollectionTimeSpanPropertyLabel;
        private TextBox myNonCollectionTimeSpanPropertyTextBox;
        private ManagementGroupBox myCollectionGroupBox;
        private Label myCollectionIntPropertyLabel;
        private TextBox myCollectionIntPropertyTextBox;
        private bool hasChanges;
        private MyConfigSectionModuleServiceProxy serviceProxy;
        private PropertyBag bag;
        private MyConfigSectionInfo localInfo;
        private bool errorGetSettings;
        private PropertyBag clone;
        private PageTaskList taskList;
        private bool readOnly;

        public MyConfigSectionPage();
        private void InitializeComponent();
        private void OnMyConfigSectionBoolAttrCheckBoxCheckedChanged(object sender,
                                                                    EventArgs e);

        private void OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged(
                                                                    object sender, EventArgs e);

        private void OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged(object sender,
                                                                    EventArgs e);

        private void OnmyCollectionIntPropertyTextBoxTextChanged(object sender,
                                                                    EventArgs e);

        private void UpdateUIState();

        protected override bool HasChanges { get; }
        protected override bool CanApplyChanges { get; }
        protected override void OnActivated(bool initialActivation);
        private void GetSettings();
        private void OnWorkerGetSettings(object sender, DoWorkEventArgs e);
        private void OnWorkerGetSettingsCompleted(object sender,
                                                    RunWorkerCompletedEventArgs e);

        private void InitializeUI();
        private void ClearSettings();
        private sealed class MyConfigSectionEnumObject { }
        private bool ValidateUserInputs();
        protected override bool ApplyChanges();
    }
}
```

(Continued)

Listing 7-12: *(continued)*

```
private void GetValues();
protected override void CancelChanges();
private sealed class PageTaskList : TaskList { }
public void ViewCollectionItems();
protected override TaskListCollection Tasks { get; }
protected override void OnRefresh();
protected override bool CanRefresh { get; }
protected sealed override bool ReadOnly { get; }
private void SetUIReadOnly(bool readOnly);
}
}
```

Constructor

The following code listing presents the implementation of the `MyConfigSectionPage` constructor, which in turn invokes another method named `InitializeComponent`. Replace the declaration of the `MyConfigSectionPage` constructor in the `MyConfigSectionPage.cs` file with the code shown here:

```
public MyConfigSectionPage()
{
    InitializeComponent();
}
```

The implementation of the `InitializeComponent` method is shown in Listing 7-13. Replace the declaration of the `InitializeComponent` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-13: The InitializeComponent Method

```
private void InitializeComponent()
{
    myConfigSectionPropertiesGroupBox = new ManagementGroupBox();
    myConfigSectionBoolPropertyCheckBox = new CheckBox();
    myConfigSectionEnumPropertyLabel = new Label();
    myConfigSectionEnumPropertyComboBox = new ComboBox();

    myNonCollectionGroupBox = new ManagementGroupBox();
    myNonCollectionTimeSpanPropertyLabel = new Label();
    myNonCollectionTimeSpanPropertyTextBox = new TextBox();

    myCollectionGroupBox = new ManagementGroupBox();
    myCollectionIntPropertyLabel = new Label();
    myCollectionIntPropertyTextBox = new TextBox();

    myConfigSectionPropertiesGroupBox.SuspendLayout();
    myNonCollectionGroupBox.SuspendLayout();
    myCollectionGroupBox.SuspendLayout();
    base.SuspendLayout();
}
```

Listing 7-13: (continued)

```
myConfigSectionPropertiesGroupBox.Controls.Add(
    myConfigSectionBoolPropertyCheckBox);
myConfigSectionPropertiesGroupBox.Controls.Add(myConfigSectionEnumPropertyLabel);
myConfigSectionPropertiesGroupBox.Controls.Add(
    myConfigSectionEnumPropertyComboBox);

myConfigSectionPropertiesGroupBox.Location = new Point(0, 10);
myConfigSectionPropertiesGroupBox.Name = "myConfigSectionPropertiesGroupBox";
myConfigSectionPropertiesGroupBox.Width = 200;
myConfigSectionPropertiesGroupBox.Height = 75;
myConfigSectionPropertiesGroupBox.TabStop = false;
myConfigSectionPropertiesGroupBox.Text = "Configuration section properties";

myConfigSectionBoolPropertyCheckBox.Location = new Point(9, 0x12);
myConfigSectionBoolPropertyCheckBox.Name = "myConfigSectionBoolPropertyCheckBox";
myConfigSectionBoolPropertyCheckBox.AutoSize = true;
myConfigSectionBoolPropertyCheckBox.TabIndex = 0;
myConfigSectionBoolPropertyCheckBox.Text = "Boolean property";
myConfigSectionBoolPropertyCheckBox.CheckedChanged +=
    new EventHandler(OnMyConfigSectionBoolAttrCheckBoxCheckedChanged);

myConfigSectionEnumPropertyLabel.Location = new Point(9, 0x2a);
myConfigSectionEnumPropertyLabel.Name = "myConfigSectionEnumPropertyLabel";
myConfigSectionEnumPropertyLabel.AutoSize = true;
myConfigSectionEnumPropertyLabel.TabIndex = 1;
myConfigSectionEnumPropertyLabel.Text = "Enum property:";
myConfigSectionEnumPropertyLabel.TextAlign = ContentAlignment.MiddleLeft;

myConfigSectionEnumPropertyComboBox.FormattingEnabled = true;
myConfigSectionEnumPropertyComboBox.Location = new Point(110, 0x2a);
myConfigSectionEnumPropertyComboBox.Name = "myConfigSectionEnumPropertyComboBox";
myConfigSectionEnumPropertyComboBox.Width = 80;
myConfigSectionEnumPropertyComboBox.TabIndex = 2;
myConfigSectionEnumPropertyComboBox.SelectedIndexChanged +=
    new EventHandler(OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged);

myNonCollectionGroupBox.Controls.Add(myNonCollectionTimeSpanPropertyLabel);
myNonCollectionGroupBox.Controls.Add(myNonCollectionTimeSpanPropertyTextBox);
myNonCollectionGroupBox.Location = new Point(0, 95);
myNonCollectionGroupBox.Name = "MyNonCollectionGroupBox";
myNonCollectionGroupBox.Width = 200;
myNonCollectionGroupBox.Height = 50;
myNonCollectionGroupBox.TabIndex = 1;
myNonCollectionGroupBox.TabStop = false;
myNonCollectionGroupBox.Text = "Non-collection properties";

myNonCollectionTimeSpanPropertyLabel.Location = new Point(9, 0x12);
myNonCollectionTimeSpanPropertyLabel.Name =
    "myNonCollectionTimeSpanPropertyLabel";
myNonCollectionTimeSpanPropertyLabel.AutoSize = true;
myNonCollectionTimeSpanPropertyLabel.TabIndex = 0;
```

(Continued)

Listing 7-13: (continued)

```
myNonCollectionTimeSpanPropertyLabel.Text = "TimeSpan property:";
myNonCollectionTimeSpanPropertyLabel.TextAlign = ContentAlignment.MiddleLeft;

myNonCollectionTimeSpanPropertyTextBox.Location = new Point(110, 0x12);
myNonCollectionTimeSpanPropertyTextBox.Name =
    "myNonCollectionTimeSpanPropertyTextBox";
myNonCollectionTimeSpanPropertyTextBox.Width = 80;
myNonCollectionTimeSpanPropertyTextBox.TabIndex = 3;
myNonCollectionTimeSpanPropertyTextBox.TextChanged +=
    new EventHandler(OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged);

myCollectionGroupBox.Controls.Add(myCollectionIntPropertyLabel);
myCollectionGroupBox.Controls.Add(myCollectionIntPropertyTextBox);
myCollectionGroupBox.Location = new Point(0, 155);
myCollectionGroupBox.Name = "myCollectionGroupBox";
myCollectionGroupBox.Width = 200;
myCollectionGroupBox.Height = 50;
myCollectionGroupBox.TabIndex = 1;
myCollectionGroupBox.TabStop = false;
myCollectionGroupBox.Text = "Collection properties";

myCollectionIntPropertyLabel.Location = new Point(9, 0x12);
myCollectionIntPropertyLabel.Name = "myCollectionIntPropertyLabel";
myCollectionIntPropertyLabel.AutoSize = true;
myCollectionIntPropertyLabel.TabIndex = 0;
myCollectionIntPropertyLabel.Text = "Integer property:";
myCollectionIntPropertyLabel.TextAlign = ContentAlignment.MiddleLeft;

myCollectionIntPropertyTextBox.Location = new Point(110, 0x12);
myCollectionIntPropertyTextBox.Name = "myCollectionIntPropertyTextBox";
myCollectionIntPropertyTextBox.Width = 80;
myCollectionIntPropertyTextBox.TabIndex = 3;
myCollectionIntPropertyTextBox.TextChanged +=
    new EventHandler(OnmyCollectionIntPropertyTextBoxTextChanged);
AutoScroll = true;
base.AutoScaleMode = AutoScaleMode.Font;
base.AutoScaleDimensions = new.SizeF(6f, 13f);
base.ClientSize = new Size(0x1d8, 0x228);
base.Controls.Add(myConfigSectionPropertiesGroupBox);
base.Controls.Add(myNonCollectionGroupBox);
base.Controls.Add(myCollectionGroupBox);
myConfigSectionPropertiesGroupBox.ResumeLayout(false);
myNonCollectionGroupBox.PerformLayout();
myCollectionGroupBox.PerformLayout();
base.ResumeLayout(false);
}
```

The main responsibility of the `InitializeComponent` method is to create the GUI elements that make up the `MyConfigSectionPage`'s user interface. Let's go over the important parts of the implementation of this method. Keep Figure 7-4 in mind as I review these parts. As this figure shows, the user interface

Chapter 7: Extending the Integrated Graphical Management System

of the `MyConfigSectionPage` consists of three group boxes. The IIS7 Manager graphical architecture comes with a class named `ManagementGroupBox`, which renders a group box. Figure 7-12 presents the portion of Figure 7-1 that contains the `ManagementGroupBox` hierarchy. As this figure shows, the `ManagementGroupBox` control inherits the standard `System.Windows.Forms.GroupBox` control.

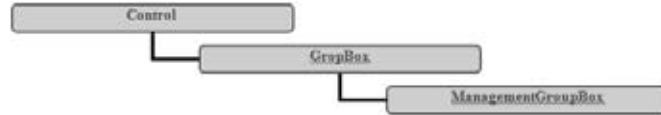


Figure 7-12

As Listing 7-13 shows, the `InitializeComponent` method creates three instances of the `ManagementGroupBox` control, one for each group box shown in Figure 7-4, starting with the top group box, which contains a `CheckBox`, a `Label`, and a `ComboBox` control (see Figure 7-4):

```
myConfigSectionPropertiesGroupBox = new ManagementGroupBox();
myConfigSectionBoolPropertyCheckBox = new CheckBox();
myConfigSectionEnumPropertyLabel = new Label();
myConfigSectionEnumPropertyComboBox = new ComboBox();
. . .
myConfigSectionPropertiesGroupBox.Controls.Add(
    myConfigSectionBoolPropertyCheckBox);
myConfigSectionPropertiesGroupBox.Controls.Add(myConfigSectionEnumPropertyLabel);
myConfigSectionPropertiesGroupBox.Controls.Add(
    myConfigSectionEnumPropertyComboBox);
```

`InitializeComponent` registers the `OnMyConfigSectionBoolAttrCheckBoxCheckedChanged` and `OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged` methods as the event handlers for the `CheckedChanged` event of the `CheckBox` control and the `SelectedIndexChanged` event of the `ComboBox` control, respectively:

```
myConfigSectionBoolPropertyCheckBox.CheckedChanged +=
    new EventHandler(OnMyConfigSectionBoolAttrCheckBoxCheckedChanged);

myConfigSectionEnumPropertyComboBox.SelectedIndexChanged +=
    new EventHandler(OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged);
```

Next, `InitializeComponent` instantiates the second instance of the `ManagementGroupBox` control to render the middle group box shown in Figure 7-4, which includes a `Label` and a `TextBox` control:

```
myNonCollectionGroupBox = new ManagementGroupBox();
myNonCollectionTimeSpanPropertyLabel = new Label();
myNonCollectionTimeSpanPropertyTextBox = new TextBox();
. . .
myNonCollectionGroupBox.Controls.Add(myNonCollectionTimeSpanPropertyLabel);
myNonCollectionGroupBox.Controls.Add(myNonCollectionTimeSpanPropertyTextBox);
```

Chapter 7: Extending the Integrated Graphical Management System

`InitializeComponent` registers the `OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged` method as the event handler for the `TextChanged` event of the `myNonCollectionTimeSpanPropertyTextBox` control:

```
myNonCollectionTimeSpanPropertyTextBox.TextChanged +=  
    new EventHandler(OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged);
```

Next, it instantiates the third instance of the `ManagementGroupBox` control to render the bottom group box shown in Figure 7-4, which contains a `Label` and a `TextBox` control:

```
myCollectionGroupBox = new ManagementGroupBox();  
myCollectionIntPropertyLabel = new Label();  
myCollectionIntPropertyTextBox = new TextBox();  
...  
myCollectionGroupBox.Controls.Add(myCollectionIntPropertyLabel);  
myCollectionGroupBox.Controls.Add(myCollectionIntPropertyTextBox);
```

`InitializeComponent` registers the `OnmyCollectionIntPropertyTextBoxTextChanged` method as the event handler for the `TextChanged` event of the `myCollectionIntPropertyTextBox` control:

```
myCollectionIntPropertyTextBox.TextChanged +=  
    new EventHandler(OnmyCollectionIntPropertyTextBoxTextChanged);
```

Finally, `InitializeComponent` adds the three `ManagementGroupBox` controls to the `Controls` collection of the `MyConfigSectionPage` control:

```
base.Controls.Add(myConfigSectionPropertiesGroupBox);  
base.Controls.Add(myNonCollectionGroupBox);  
base.Controls.Add(myCollectionGroupBox);
```

Event Handlers

Listing 7-14 presents the implementation of the `OnMyConfigSectionBoolAttrCheckBoxCheckedChanged`, `OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged`, `OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged`, and `OnmyCollectionIntPropertyTextBoxTextChanged` event handlers. Replace the declaration of these event handlers in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-14: The Event Handlers

```
private void OnMyConfigSectionBoolAttrCheckBoxCheckedChanged(object sender,  
                                                             EventArgs e)  
{  
    this.UpdateUIState();  
}  
  
private void OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged(  
                                                             object sender, EventArgs e)  
{  
    this.UpdateUIState();  
}
```

Listing 7-14: *(continued)*

```
private void OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged(object sender,
                                                                    EventArgs e)
{
    this.UpdateUIState();
}

private void OnmyCollectionIntPropertyTextBoxTextChanged(object sender,
                                                          EventArgs e)
{
    this.UpdateUIState();
}
```

As Listing 7-14 shows, all four event handlers call the `UpdateUIState` method. The implementation of this method is shown in Listing 7-15. Now go ahead and replace the declaration of the `UpdateUIState` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-15: The UpdateUIState Method

```
private void UpdateUIState()
{
    this.hasChanges = true;
    base.Update();
}
```

The `UpdateUIState` method simply sets the `hasChanged` Boolean field to `true` to signal that the user interface of the `MyConfigSectionPage` module has gone through changes.

HasChanges Property

As Listing 7-16 shows, the `MyConfigSectionPage` overrides the `HasChanges` property of the `ModulePage` abstract base class to return the value of the `hasChanges` Boolean field. Now replace the declaration of the `HasChanges` property in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-16: The HasChanges Property

```
protected override bool HasChanges
{
    get { return this.hasChanges; }
}
```

When the end user navigates to a module page, such as `MyConfigSectionPage`, the task items such as `Apply` and `Cancel` are grayed out. When the end user makes some changes in the GUI elements that make up the module page such as entering a value in a text field, selecting an item from a combo box, toggling a checkbox, and the like, the event handlers associated with these GUI elements such as `OnMyConfigSectionBoolAttrCheckBoxCheckedChanged`, `OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged`,

`OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged`, and `OnmyCollectionIntPropertyTextBoxTextChanged` are automatically called. As Listing 7-14 shows, these event handlers call the `UpdateUIState` method, which in turn calls the `Update` method of the base class, that is, the `ModuleDialogPage` as shown in Listing 7-15. The `Update` method in turn activates the `Apply` and `Cancel` buttons, allowing the end user to commit or cancel the changes.

CanApplyChanges Property

The `MyConfigSectionPage` also overrides the `CanApplyChanges` property of the `ModuleDialogPage` base class to return the value of the `hasChanges` Boolean field as shown in Listing 7-17. Now replace the declaration of the `CanApplyChanges` property in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

When the end user clicks the `Apply` button, the base class under the hood examines the value of this property to determine whether the `MyConfigSectionPage` module dialog page has indeed changes to store in the underlying configuration file.

Listing 7-17: The `CanApplyChanges` Property

```
protected override bool CanApplyChanges
{
    get {return this.hasChanges;}
}
```

OnActivated

The `MyConfigSectionPage` module dialog page overrides the `OnActivated` method of the `ModulePage` base class as shown in Listing 7-18. Now replace the declaration of the `OnActivated` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

This method is called when the module page is activated. For example, when the end user clicks the `MyConfigSection` item shown in Figure 7-3 and navigates to the `MyConfigSectionPage` module dialog page shown in Figure 7-4, the `OnActivated` method of the module page is automatically invoked.

Listing 7-18: The `OnActivated` Method

```
protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (MyConfigSectionModuleServiceProxy)base.CreateProxy(
                                                                    typeof(MyConfigSectionModuleServiceProxy));
        this.GetSettings();
    }
}
```

`OnActivated` checks whether this is the first time this module page is being activated. If so, this method must retrieve the required configuration settings from the back-end Web server and update the user

interface of the `MyConfigSectionPage` module dialog page accordingly. As discussed earlier, the communications between this module dialog page and the back-end Web server must go through the `MyConfigSectionModuleServiceProxy` proxy class. Therefore, the first order of business is to instantiate the proxy class:

```
this.serviceProxy = (MyConfigSectionModuleServiceProxy)base.CreateProxy(  
    typeof(MyConfigSectionModuleServiceProxy));
```

Next, `OnActivated` calls the `GetSettings` method, which actually uses the proxy class to retrieve the configuration settings from the back-end Web server.

GetSettings

Listing 7-19 presents the implementation of the `GetSettings` method of the `MyConfigSectionPage` module page. Next replace the declaration of the `GetSettings` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-19: The `GetSettings` Method

```
private void GetSettings()  
{  
    this.SetUIReadOnly(true);  
    base.StartAsyncTask("Getting settings...",  
        new DoWorkEventHandler(this.OnWorkerGetSettings),  
        new RunWorkerCompletedEventHandler(this.OnWorkerGetSettingsCompleted));  
    this.hasChanges = false;  
}
```

The `GetSettings` method first invokes another method named `SetUIReadOnly`, passing in `true` as its argument to change the state of the `MyConfigSectionPage` module dialog page to read-only. As mentioned earlier, a read-only module page does not allow the end user to change the associated configuration settings. Because the `GetSettings` method is about to retrieve configuration settings from the underlying configuration file, it makes sense not to allow the end user to interact with the user interface of the module dialog page while the configuration settings are being downloaded from the server. The following code listing presents the implementation of the `SetUIReadOnly` method. Now replace the declaration of the `SetUIReadOnly` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing:

```
private void SetUIReadOnly(bool readOnly)  
{  
    this.myConfigSectionPropertiesGroupBox.Enabled = !readOnly;  
    this.myConfigSectionBoolPropertyCheckBox.Enabled = !readOnly;  
    this.myConfigSectionEnumPropertyLabel.Enabled = !readOnly;  
    this.myConfigSectionEnumPropertyComboBox.Enabled = !readOnly;  
    this.myNonCollectionGroupBox.Enabled = !readOnly;  
    this.myNonCollectionTimeSpanPropertyLabel.Enabled = !readOnly;  
    this.myNonCollectionTimeSpanPropertyTextBox.ReadOnly = readOnly;  
    this.myCollectionGroupBox.Enabled = !readOnly;  
    this.myCollectionIntPropertyLabel.Enabled = !readOnly;  
    this.myCollectionIntPropertyTextBox.ReadOnly = readOnly;  
}
```

Chapter 7: Extending the Integrated Graphical Management System

Now back to the implementation of the `GetSettings` method shown in Listing 7-19. The `ModulePage` base class exposes a method named `StartAsyncTask` that allows you to communicate with the back-end Web server asynchronously to improve the performance of the application. The `StartAsyncTask` method takes three important arguments. The first argument is just a text message. The second argument is an instance of a delegate named `DoWorkEventHandler` defined as follows:

```
public delegate void DoWorkEventHandler(object sender, DoWorkEventArgs e);
```

The event data class associated with this handler is a class named `DoWorkEventArgs` as shown in Listing 7-20.

Every event handler delegate, such as `DoWorkEventHandler`, is associated with a class known as an event data class. When the event associated with a delegate is fired, an instance of this class is instantiated and populated with the required event data and passed into the delegate. The `DoWorkEventArgs` class is the event data class associated with the `DoWorkEventHandler` delegate. As you can tell from the properties of this event data class, the event data in this case includes two objects named `Argument` and `Result`.

Listing 7-20: The `DoWorkEventArgs` Event Data Class

```
public class DoWorkEventArgs : CancelEventArgs
{
    public DoWorkEventArgs(object argument);

    public object Argument { get; }
    public object Result { get; set; }
}
```

Notice that the `DoWorkEventArgs` event data class inherits the `CancelEventArgs` event data class, which is the base class for all cancelable events. Listing 7-21 shows the definition of this base class. Notice that this class exposes a single Boolean read/write property named `Cancel` that you can use to cancel the data retrieval process.

Listing 7-21: The `CancelEventArgs` Class

```
public class CancelEventArgs : EventArgs
{
    public CancelEventArgs();
    public CancelEventArgs(bool cancel);

    public bool Cancel { get; set; }
}
```

As Listing 7-19 shows, `GetSettings` wraps the `OnWorkerGetSettings` method in a `DoWorkEventHandler` delegate, which means that this method will be called to instantiate the data retrieval process.

Chapter 7: Extending the Integrated Graphical Management System

Also note that `GetSettings` wraps the `OnWorkerGetSettingsCompleted` method in a `RunWorkerCompletedEventHandler` delegate and passes the delegate as the third argument into the `StartAsyncTask` method. Here is the definition of this delegate:

```
public delegate void RunWorkerCompletedEventHandler(object sender,
                                                    RunWorkerCompletedEventArgs e);
```

Listing 7-22 presents the definition of the `RunWorkerCompletedEventArgs` class, which is the event data class associated with the `RunWorkerCompletedEventHandler` delegate.

Listing 7-22: The `RunWorkerCompletedEventArgs` Event Data Class

```
public class RunWorkerCompletedEventArgs : AsyncCompletedEventArgs
{
    public RunWorkerCompletedEventArgs(object result, Exception error,
                                        bool cancelled);

    public object Result { get; }
    public object UserState { get; }
}
```

The `StartAsyncTask` method automatically calls the `OnWorkerGetSettingsCompleted` method after the data is downloaded from the Web server.

OnWorkerGetSettings

Listing 7-23 contains the code for the `OnWorkerGetSettings` method. Recall that the `StartAsyncTask` method calls this method to start the data retrieval process. Now replace the declaration of the `OnWorkerGetSettings` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-23: The `OnWorkerGetSettings` Method

```
private void OnWorkerGetSettings(object sender, DoWorkEventArgs e)
{
    e.Result = this.serviceProxy.GetSettings();
}
```

This method simply calls the `GetSettings` method of the `MyConfigSectionModuleServiceProxy` proxy class to retrieve the required configuration settings from the Web server (see Listing 7-2). As discussed earlier, this proxy in turn invokes the `GetSettings` method of the `MyConfigSectionModuleService` server-side class (see Listing 7-3) to retrieve the configuration settings.

OnWorkerGetSettingsCompleted

Listing 7-24 presents the implementation of the `OnWorkerGetSettingsCompleted` method. Recall that the `StartAsyncTask` method automatically calls this method after the configuration settings are retrieved from the Web server. Now replace the declaration of the `OnWorkerGetSettings` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-24: The OnWorkerGetSettingsCompleted Method

```
private void OnWorkerGetSettingsCompleted(object sender,
                                         RunWorkerCompletedEventArgs e)
{
    try
    {
        this.bag = (PropertyBag)e.Result;
        this.localInfo = new MyConfigSectionInfo(this.bag);
        this.readOnly = localInfo.ReadOnly;
        this.errorGetSettings = false;
    }
    catch (Exception exception1)
    {
        base.StopProgress();
        base.DisplayErrorMessage(exception1.Message, "DoWorkerGetSettingsCompleted");
        this.errorGetSettings = true;
        this.SetUIReadOnly(true);
    }
    finally
    {
        if (this.bag != null)
            this.InitializeUI();

        if (this.ReadOnly)
            this.SetUIReadOnly(true);

        this.hasChanges = false;
    }
}
```

The `StartAsyncTask` method passes an instance of the `RunWorkerCompletedEventArgs` event data class into `OnWorkerGetSettingsCompleted`. As Listing 7-22 shows, this instance exposes a property named `Result` that contains the retrieved data. As you'll see later in this chapter, the `GetSettings` method of the `MyConfigSectionModuleService` server-side class packs the configuration settings in a `PropertyBag` collection and returns this collection to the client. In other words, in this case, the real type of the `Result` object is `PropertyBag`:

```
this.bag = (PropertyBag)e.Result;
```

`OnWorkerGetSettingsCompleted` then creates a `MyConfigSectionInfo` instance passing in the `PropertyBag`, assigns the instance to a private field named `localInfo`, sets the `readOnly` private field to the value of the `ReadOnly` property of the `MyConfigSectionInfo` instance, and sets the `errorGetSettings` field to `false` to signal that everything went fine with no errors. Notice that if there's an error, this field is set to `true`. Later you'll see what happens when this field is set to `true`.

```
this.localInfo = new MyConfigSectionInfo(this.bag);
this.readOnly = localInfo.ReadOnly;
this.errorGetSettings = false;
```

Chapter 7: Extending the Integrated Graphical Management System

`OnWorkerGetSettingsCompleted` then calls the `InitializeUI` method to initialize the user interface of the `MyConfigSectionPage` module dialog page with the retrieved configuration settings.

```
if (this.bag != null)
    this.InitializeUI();
```

Finally, it calls the `SetUIReadOnly` method passing in `true` if the `MyConfigSectionPage` module dialog page is in read-only mode where the end user is not allowed to modify the associated configuration settings.

```
if (this.ReadOnly)
    this.SetUIReadOnly(true);
```

As you can see from the following code listing, the `ReadOnly` property of the `MyConfigSectionPage` module dialog page simply returns the value of the `readOnly` field. Now replace the declaration of the `ReadOnly` property in the `MyConfigSectionPage.cs` file with the code shown in this code listing:

```
protected sealed override bool ReadOnly
{
    get { return this.readOnly; }
}
```

MyConfigSectionInfo

Recall from Listing 7-24 that the `OnWorkerGetSettingsCompleted` method instantiates an instance of the `MyConfigSectionInfo` type passing in the `PropertyBag` object that contains the configuration settings downloaded from the server and assigns this instance to a private field named `localInfo`. Listing 7-25 presents the implementation of the `MyConfigSectionInfo` class. Add a new source file name `MyConfigSectionInfo.cs` to the `GraphicalManagement/Client` directory of the `MyConfigSection` project and add the code shown in this code listing to this source file.

Listing 7-25: The `MyConfigSectionInfo` Class

```
using Microsoft.Web.Management.Server;
using System;
using MyNamespace.ImperativeManagement;

namespace MyNamespace.GraphicalManagement.Client
{
    public sealed class MyConfigSectionInfo
    {
        private PropertyBag bag;

        public MyConfigSectionInfo(PropertyBag bag)
        {
            this.bag = bag.Clone();
        }

        public bool MyConfigSectionBoolProperty
        {
            get { return (bool)this.bag[0]; }
        }
    }
}
```

(Continued)

Listing 7-25: (continued)

```
public MyConfigSectionEnum MyConfigSectionEnumProperty
{
    get { return (MyConfigSectionEnum)this.bag[1]; }
}

public TimeSpan MyNonCollectionTimeSpanProperty
{
    get { return (TimeSpan)this.bag[2]; }
}

public int MyCollectionIntProperty
{
    get { return (int)this.bag[3]; }
}

public bool ReadOnly
{
    get
    {
        object obj = this.bag[4];
        if (obj != null)
            return (bool)obj;

        return false;
    }
}
}
```

First, let me explain why you need the `MyConfigSectionInfo` class in the first place. As Listing 7-24 shows, the `MyConfigSectionPage` module dialog page receives a `PropertyBag` object from the server-side class that contains the retrieved configuration settings. As you can see from Listing 7-4, the `PropertyBag` class is an `IDictionary` class, which means that you have to use a type-unsafe approach such as indexing into the `IDictionary` object to access the retrieved configuration settings. The `MyConfigSectionInfo` class allows you to expose the content of the `PropertyBag` collection as strongly-typed properties, which will provide among many others the following benefits:

- ❑ Visual Studio provides IntelliSense support for strongly-typed properties, which means that you can catch problems as you're typing.
- ❑ Compilers provide type-checking support for strongly-typed properties, which means that you can catch problems as you're compiling.
- ❑ It allows you to use object-oriented techniques to program against these configuration settings, which means that you can take advantage of the well-known benefits of the object-oriented programming.

You won't get any of these benefits if you directly program against the `PropertyBag` object itself. You may be wondering why you should bother with the `PropertyBag` object to begin with. In other words, why doesn't the `GetSettings` method of the `MyConfigSectionModuleService` server-side class return a `MyConfigSectionInfo` to begin with? The answer lies in the fact that any information passed

Chapter 7: Extending the Integrated Graphical Management System

between the client and server must be serialized in one end and deserialized in the other end. As Listings 7-8 and 7-9 show, the `PropertyBag` uses an `ObjectStateFormatter` to serialize and deserialize itself. The `ObjectStateFormatter` class has been optimized for serializing and deserializing a `PropertyBag` object, as long as the object does not contain complex types.

Next, I discuss the implementation of the `MyConfigSectionInfo` class shown in Listing 7-25. The `PropertyBag` collection passed into the constructor of this class contains the following items downloaded from the server:

- ❑ The first item in this collection contains the value of the `myConfigSectionBoolAttr` attribute on the `<myConfigSection>` containing element. As Listing 7-25 shows, the `MyConfigSectionInfo` class exposes this value as a strong-typed property named `MyConfigSectionBoolProperty`:

```
public bool MyConfigSectionBoolProperty
{
    get { return (bool)this.bag[0]; }
}
```

- ❑ The second item in this collection contains the value of the `myConfigSectionEnumAttr` attribute on the `<myConfigSection>` containing element. As Listing 7-25 shows, the `MyConfigSectionInfo` class exposes this value as a strong-typed property named `MyConfigSectionEnumProperty`:

```
public MyConfigSectionEnum MyConfigSectionEnumProperty
{
    get { return (MyConfigSectionEnum)this.bag[1]; }
}
```

- ❑ The third item in this collection contains the value of the `myNonCollectionTimeSpanAttr` attribute on the `<myNonCollection>` non-collection element. As Listing 7-25 shows, the `MyConfigSectionInfo` class exposes this value as a strong-typed property named `MyNonCollectionTimeSpanProperty`:

```
public TimeSpan MyNonCollectionTimeSpanProperty
{
    get { return (TimeSpan)this.bag[2]; }
}
```

- ❑ The fourth item in this collection contains the value of the `myCollectionIntAttr` attribute on the `<myCollection>` *Collection* XML element. As Listing 7-25 shows, the `MyConfigSectionInfo` class exposes this value as a strongly-typed property named `MyCollectionIntProperty`:

```
public int MyCollectionIntProperty
{
    get { return (int)this.bag[3]; }
}
```

- ❑ The fifth item in this collection contains the value of the `isLocked` attribute on the `<myConfigSection>` containing element. As Listing 7-25 shows, the `MyConfigSectionInfo` class exposes this value as a strong-typed property named `ReadOnly`:

```
public bool ReadOnly
```

```
{
    get
    {
        object obj = this.bag[4];
        if (obj != null)
            return (bool)obj;

        return false;
    }
}
```

InitializeUI

Recall from Listing 7-24 that the `OnWorkerGetSettingsCompleted` method calls the `InitializeUI` method to initialize the user interface of the `MyConfigSectionPage` module page with the configuration settings retrieved from the server. Listing 7-26 contains the implementation of the `InitializeUI` method. Now replace the declaration of the `InitializeUI` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-26: The InitializeUI Method

```
private void InitializeUI()
{
    if (localInfo == null)
        return;

    this.SetUIReadOnly(false);
    ClearSettings();
    myConfigSectionEnumPropertyComboBox.Items.AddRange(new object[] {
        new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal1),
        new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal2),
        new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal3)
    });

    MyConfigSectionEnum enumVal = localInfo.MyConfigSectionEnumProperty;
    int num1 = 0;
    foreach (MyConfigSectionEnumObject obj1 in
        myConfigSectionEnumPropertyComboBox.Items)
    {
        if (obj1.EnumVal == enumVal)
        {
            myConfigSectionEnumPropertyComboBox.SelectedIndex = num1;
            break;
        }
        num1++;
    }
    myConfigSectionBoolPropertyCheckBox.Checked =
        localInfo.MyConfigSectionBoolProperty;
}
```

`InitializeUI` first calls the `SetUIReadOnly` method, passing in `false` as its argument to enable end users to modify the associated configuration settings:

```
this.SetUIReadOnly(false);
```

Next, it calls the `ClearSettings` method:

```
ClearSettings();
```

As Listing 7-27 shows, the `ClearSettings` method clears the `myConfigSectionEnumPropertyComboBox` combo box in the top group box shown in Figure 7-4 and the `myNonCollectionTimeSpanPropertyTextBox` text field in the middle group box of the same figure. Now replace the declaration of the `ClearSettings` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-27: The `ClearSettings` Method

```
private void ClearSettings()
{
    this.myConfigSectionEnumPropertyComboBox.Items.Clear();
    this.myConfigSectionEnumPropertyComboBox.SelectedIndex = -1;
    this.myNonCollectionTimeSpanPropertyTextBox.Clear();
}
```

After calling the `ClearSettings` method, `InitializeUI` initializes the `myConfigSectionEnumPropertyComboBox` combo box. Recall that this combo box displays the possible values of the `myConfigSectionEnumAttr` attribute of the `<myConfigSection>` Containing element.

```
myConfigSectionEnumPropertyComboBox.Items.AddRange(new object[] {
    new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal1),
    new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal2),
    new MyConfigSectionEnumObject(MyConfigSectionEnum.MyConfigSectionEnumVal3) });
```

Note that this code fragment represents each enumeration value with an instance of a class named `MyConfigSectionEnumObject`. This class is simply a wrapper around the value to allow the combo box to display the value in this user interface. Listing 7-28 presents the implementation of this class. Now replace the declaration of the `MyConfigSectionEnumObject` nested class in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-28: The `MyConfigSectionEnumObject` Class

```
private sealed class MyConfigSectionEnumObject
{
    public MyConfigSectionEnumObject(MyConfigSectionEnum enumVal)
    {
        this.enumVal = enumVal;
        switch (enumVal)
        {
            case MyConfigSectionEnum.MyConfigSectionEnumVal1:
                this.enumValText = "Value 1";
                break;

            case MyConfigSectionEnum.MyConfigSectionEnumVal2:
                this.enumValText = "Value 2";
                break;

            case MyConfigSectionEnum.MyConfigSectionEnumVal3:
```

(Continued)

Listing 7-28: (continued)

```
        this.enumValText = "Value 3";
        break;
    }
}

public override string ToString()
{
    return this.enumValText;
}

public MyConfigSectionEnum EnumVal
{
    get { return this.enumVal; }
}

private MyConfigSectionEnum enumVal;
private string enumValText;
}
```

Now back to the `InitializeUI` method. Next, `InitializeUI` sets the selected item of the `myConfigSectionEnumPropertyComboBox` combo box to display the current value of the `MyConfigSectionEnumProperty` property of the `localInfo` field. Recall from Listings 7-24 and 7-25 that the `localInfo` field is an instance of the `MyConfigSectionInfo` class and exposes the configuration settings retrieved from the server as strongly-typed properties. As Listing 7-25 shows, the `MyConfigSectionEnumProperty` property of the `localInfo` field contains the current value of the `myConfigSectionEnumAttr` attribute of the `<myConfigSection>` Containing element:

```
MyConfigSectionEnum enumVal = localInfo.MyConfigSectionEnumProperty;
int num1 = 0;
foreach (MyConfigSectionEnumObject obj1 in
    myConfigSectionEnumPropertyComboBox.Items)
{
    if (obj1.EnumVal == enumVal)
    {
        myConfigSectionEnumPropertyComboBox.SelectedIndex = num1;
        break;
    }
    num1++;
}
```

Finally, `InitializeUI` ensures that the `myConfigSectionBoolPropertyCheckBox` checkbox in the top group box of Figure 7-4 reflects the current value of the `MyConfigSectionBoolProperty` property of the `localInfo` field. As Listing 7-25 shows, this property contains the current value of the `myConfigSectionBoolAttr` attribute of the `<myConfigSection>` Containing element.

```
myConfigSectionBoolPropertyCheckBox.Checked =
    localInfo.MyConfigSectionBoolProperty;
```

ApplyChanges

Listing 7-29 contains the code for the `ApplyChanges` method of the `MyConfigSectionPage` module dialog page. Now replace the declaration of the `ApplyChanges` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-29: The `ApplyChanges` Method

```
protected override bool ApplyChanges()
{
    bool flag = false;
    if (!this.ReadOnly && this.ValidateUserInputs())
    {
        try
        {
            Cursor.Current = Cursors.WaitCursor;
            GetValues();
            this.serviceProxy.UpdateSettings(this.clone);
            this.bag = this.clone;
            this.localInfo = new MyConfigSectionInfo(this.bag);
            flag = true;
            this.hasChanges = false;
        }

        catch (Exception exception)
        {
            base.DisplayErrorMessage(exception.Message, "ApplyChanges");
        }

        finally
        {
            Cursor.Current = Cursors.Default;
            base.Update();
        }
    }

    return flag;
}
```

`ApplyChanges` first checks whether both of the following conditions are met:

- ❑ The `MyConfigSectionPage` module dialog page is *not* in read-only mode. In other words, the end user is allowed to modify the associated configuration settings. Recall that the value of the `ReadOnly` property of the `MyConfigSectionPage` module dialog page reflects the value of the `isLocked` attribute on the `<myConfigSection>` containing element in the underlying configuration file.
- ❑ The user inputs are valid. Your custom module page should expose a method such as `ValidateUserInputs` that contains the logic that validates the user inputs. To keep these discussions focused, the implementation of the `ValidateUserInputs` method always

Chapter 7: Extending the Integrated Graphical Management System

returns true. Now replace the declaration of the `ValidateUserInputs` method in the `MyConfigSectionPage.cs` file with the code shown in the following code listing:

```
private bool ValidateUserInputs()
{
    return true;
}
```

If both of these conditions are met, the `ApplyChanges` method calls the `GetValues` method to extract the new values of the associated configuration settings from the user interface of the `MyConfigSectionPage` module page:

```
GetValues();
```

Recall from Listing 7-18 that when the `MyConfigSectionPage` module page is accessed for the first time, the `OnActivated` method instantiates an instance of the `MyConfigSectionModuleServiceProxy` proxy class and assigns it to a private field named `serviceProxy`. The following code listing repeats Listing 7-18:

```
protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (MyConfigSectionModuleServiceProxy)base.CreateProxy(
                                                                    typeof(MyConfigSectionModuleServiceProxy));
        this.GetSettings();
    }
}
```

As Listing 7-29 shows, the `ApplyChanges` method calls the `UpdateSettings` method of this proxy object, passing in the clone `PropertyBag` collection to update the specified configuration settings in the underlying configuration file. As you'll see shortly, the `GetValues` method retrieves the user inputs from the user interface of the `MyConfigSectionPage` module dialog page and stores them in the clone collection, which is a `PropertyBag`.

```
this.serviceProxy.UpdateSettings(this.clone);
```

GetValues

Listing 7-30 contains the code for the `GetValues` method. Replace the declaration of the `GetValues` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-30: The `GetValues` Method

```
private void GetValues()
{
    this.clone = this.bag.Clone();
    this.clone[0] = (bool)myConfigSectionBoolPropertyCheckBox.Checked;
    object selectedItem = myConfigSectionEnumPropertyComboBox.SelectedItem;
    MyConfigSectionEnum enumVal = ((MyConfigSectionEnumObject)selectedItem).EnumVal;
```

Listing 7-30: (continued)

```
this.clone[1] = (int)enumVal;  
this.clone[2] = TimeSpan.Parse(myNonCollectionTimeSpanPropertyTextBox.Text);  
this.clone[3] = int.Parse(myCollectionIntPropertyTextBox.Text);  
}
```

The main responsibility of the `GetValues` method is to retrieve the user inputs from the user interface of the `MyConfigSectionPage` module dialog page and store them in the `clone` `PropertyBag` collection. Recall from Listing 7-29 that the `ApplyChanges` method passes this `clone` `PropertyBag` collection into the `UpdateSettings` method of the proxy object.

As Listing 7-30 shows, `GetValues` begins by cloning the `bag` `PropertyBag` collection and storing this cloned collection in the `clone` field. Next, it stores the value of the `Checked` Boolean property of the `myConfigSectionBoolPropertyCheckBox` checkbox as the first item in the `clone` `PropertyBag` collection. Recall that this checkbox represents the value of the `myConfigSectionBoolAttr` attribute of the `<myConfigSection>` `Containing` element:

```
this.clone[0] = (bool)myConfigSectionBoolPropertyCheckBox.Checked;
```

Next, `GetValues` stores the selected value of the `myConfigSectionEnumPropertyComboBox` combo box as the second item in the `clone` `PropertyBag` collection. Recall that this combo box displays the value of the `myConfigSectionEnumAttr` attribute of the `<myConfigSection>` containing element:

```
object selectedItem = myConfigSectionEnumPropertyComboBox.SelectedItem;  
MyConfigSectionEnum enumVal = ((MyConfigSectionEnumObject) selectedItem).EnumVal;  
this.clone[1] = (int)enumVal;
```

`GetValues` then retrieves the value that the user has entered into the `myNonCollectionTimeSpanPropertyTextBox` text field and stores it as the third item in the `clone` `PropertyBag` collection. This text field contains the value of the `myNonCollectionTimeSpanAttr` attribute of the `<myNonCollection>` non-collection element:

```
this.clone[2] = TimeSpan.Parse(myNonCollectionTimeSpanPropertyTextBox.Text);
```

Finally, `GetValues` stores the value of the `myCollectionIntPropertyTextBox` text field as the fourth item in the `clone` `PropertyBag` collection. This text field displays the value of the `myCollectionIntAttr` attribute of the `<myCollection>` collection element:

```
this.clone[3] = int.Parse(myCollectionIntPropertyTextBox.Text);
```

CancelChanges

The `MyConfigSectionPage` module dialog page overrides the `CancelChanges` method of the `ModuleDialogPage` base class to add the code that you want to run when the user clicks the `Cancel` button. Listing 7-31 presents the implementation of this method. Now replace the declaration of the `CancelChanges` method in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-31: The CancelChanges Method

```
protected override void CancelChanges()
{
    this.InitializeUI();
    this.hasChanges = false;
    base.Update();
}
```

`CancelChanges` first calls the `InitializeUI` method to reset the values that the user interface of the `MyConfigSectionPage` module page displays to the current values of the associated attributes in the underlying configuration file. Next, it calls the `Update` method of the base class to update the user interface.

Adding Support for New Task Items

Because the base class of the `MyConfigSectionPage` module dialog page automatically adds the `Apply` and `Cancel` buttons and registers the `ApplyChanges` and `CancelChanges` methods as event handlers for the `Click` events of these two buttons, all you had to do was to override these two event handlers to add the code that you want to run when the user clicks the `Apply` and `Cancel` buttons.

However, if you need to add a new button to the task panel associated with the `MyConfigSectionPage` module dialog page, there is no method like `ApplyChanges` or `CancelChanges` that you could override. You have to write extra code to add the new button, register an event handler, and implement the event handler.

In this section, you add a new link named “View collection items” to the task panel to allow the users to navigate to the `MyCollectionPage` module list page where they can view the collection items, update, delete, and add new items to the collection, and change the identifiers of the displayed items. Figure 7-4 shows this new link.

Follow these steps to add a new link to the task panel:

- ☐ Implement a class that inherits a base class named `TaskList`. This class must be a private class nested within your module page.
- ☐ Override the `Tasks` property of your module page.
- ☐ Implement the event handler associated with the new link.

In the following section I use this recipe to add the “View collection items” link.

TaskList

Recall from Listing 6-16 that the `TaskList` abstract class exposes an abstract method named `GetTaskItems` as shown in the highlighted portion of the following code listing:

```
public abstract class TaskList
{
    public virtual object GetPropertyValue(string propertyName);
    public abstract ICollection GetTaskItems();
    public virtual object InvokeMethod(string methodName, object userData);
}
```

```
public virtual void SetPropertyValue(string propertyName, object value);

public virtual bool IsDirty { get; }
}
```

You must implement a nested private class that derives from the `TaskList` abstract base class, and implements its `GetTaskItems` abstract method. Your implementation of this method must return an `ICollection` of `TaskItem` objects, each of which represents a new button, link, text, and so on that you want to add to the task panel.

Listing 7-32 presents the implementation of a class named `PageTaskList` that extends the `TaskList` base class. Keep in mind that this nested private class is normally named `PageTaskList`. Now replace the declaration of the `PageTaskList` nested private class in the `MyConfigSectionPage.cs` file with the code shown Listing 7-32.

Listing 7-32: The `PageTaskList` Class

```
private sealed class PageTaskList : TaskList
{
    public PageTaskList(MyConfigSectionPage owner)
    {
        this.owner = owner;
    }

    public override ICollection GetTaskItems()
    {
        ArrayList list1 = new ArrayList();
        if (this.owner.errorGetSettings)
        {
            MessageTaskItem message = new MessageTaskItem(MessageTaskItemType.Error,
                                                            "Error in getting settings",
                                                            null);

            list1.Add(message);
        }

        MethodTaskItem method = new MethodTaskItem("ViewCollectionItems",
                                                    "View collection items", "View");
        list1.Add(method);

        foreach (TaskItem item2 in list1)
        {
            if (!(item2 is MessageTaskItem) && !(item2 is TextTaskItem))
                item2.Enabled = !this.owner.InProgress;
        }
        return (TaskItem[])list1.ToArray(typeof(TaskItem));
    }

    public void ViewCollectionItems()
    {
        this.owner.ViewCollectionItems();
    }

    private MyConfigSectionPage owner;
}
```

Chapter 7: Extending the Integrated Graphical Management System

To implement a custom task list, first add a private field of the same type as your module page to the custom task list:

```
private MyConfigSectionPage owner;
```

Next, implement a constructor for your custom task list that takes an instance of the module page that uses it as its parameter and stores it in the private field you just created:

```
public PageTaskList(MyConfigSectionPage owner)
{
    this.owner = owner;
}
```

Override the `GetTaskItems` method of the `TaskList` base class. As Listing 7-32 shows, the `PageTaskList` class's implementation of the `GetTaskItems` method instantiates a local `ArrayList`:

```
ArrayList list1 = new ArrayList();
```

The `GetTaskItems` method then checks whether the module page associated with this task list had problems retrieving the configuration settings from the back-end Web server. If so, it instantiates a `MessageTaskItem` task item and adds it to the `ArrayList`. This message task item will display the specified error message to the end user. As discussed in Chapter 6, the message task item displays its message in the Alerts panel on the top of the task panel (see Figure 6-10).

```
if (this.owner.errorGetSettings)
{
    MessageTaskItem message = new MessageTaskItem(MessageTaskItemType.Error,
                                                    "Error in getting settings", null);
    list1.Add(message);
}
```

Recall from Listing 7-24 that the `OnWorkerGetSettingsCompleted` method sets the `errorGetSettings` field to true when there's an error in getting the configuration settings as shown in highlighted portion of the following code listing:

```
private void OnWorkerGetSettingsCompleted(object sender,
                                          RunWorkerCompletedEventArgs e)
{
    try
    {
        this.bag = (PropertyBag)e.Result;
        this.localInfo = new MyConfigSectionInfo(this.bag);
        this.readOnly = localInfo.ReadOnly;
        this.errorGetSettings = false;
    }

    catch (Exception exception1)
    {
        base.StopProgress();
        base.DisplayErrorMessage(exception1.Message, "DoWorkerGetSettingsCompleted");
        this.errorGetSettings = true;
        this.SetUIReadOnly(true);
    }
}
```

```
    }

    finally
    {
        if (this.bag != null)
            this.InitializeUI();

        if (this.ReadOnly)
            this.SetUIReadOnly(true);

        this.hasChanges = false;
    }
}
```

Now back to the implementation of the `GetTaskItems` method. Next, this method instantiates an instance of the `MethodTaskItem` class to represent the “View collection items” link in the task panel and adds the instance to the `ArrayList`:

```
MethodTaskItem method = new MethodTaskItem("ViewCollectionItems",
                                           "View collection items", "View");
list1.Add(method);
```

Note that the `MethodTaskItem` instance registers the `ViewCollectionItems` method as the event handler for the `Click` event of the “View collection items” link. Next, the `GetTaskItems` method iterates through the task items in the `ArrayList` (that is, the message and method task items) and disables the “View collection items” link if the module page is still busy.

```
foreach (TaskItem item2 in list1)
{
    if (!(item2 is MessageTaskItem) && !(item2 is TextTaskItem))
        item2.Enabled = !this.owner.InProgress;
}
```

Finally, the `GetTaskItems` method dumps the content of the `ArrayList` into an array and returns the array to its caller. In other words, the caller of this method receives an array that contains all the task items that the task list has added. As you’ll see later, the caller of this method is the associated module page, which is the `MyConfigSectionPage` module dialog page in our case.

```
return (TaskItem[])list1.ToArray(typeof(TaskItem));
```

As mentioned, the `PageTaskList` instantiates a `MethodTaskItem` task item that registers the `ViewCollectionItems` method as the event handler for the “View collection items” link. Listing 7-33 presents the implementation of this event handler.

Listing 7-33: The `ViewCollectionItems` Method

```
public void ViewCollectionItems()
{
    this.owner.ViewCollectionItems();
}
```

Chapter 7: Extending the Integrated Graphical Management System

The `ViewCollectionItems` method simply delegates to the `ViewCollectionItems` method of the `MyConfigSectionPage` module dialog page, which is discussed in the next section.

ViewCollectionItems

Listing 7-34 contains the code for the `ViewCollectionItems` method of the `MyConfigSectionPage` module dialog page. Now replace the declaration of the `ViewCollectionItems` method in the `MyConfigSectionPage.cs` file with the code shown Listing 7-34.

Listing 7-34: The ViewCollectionItems Method

```
public void ViewCollectionItems()
{
    Type type1 = typeof(MyCollectionPage);
    base.Navigate(type1);
}
```

`ViewCollectionItems` calls the `Navigate` method of its base class, passing in the `Type` object that represents the `MyCollectionPage` module dialog page. The `Navigate` method under the hood calls the `Navigate` method of the navigation service to navigate to the `MyCollectionPage` module dialog page.

Tasks

The `MyConfigSectionPage` module page overrides the `Tasks` property of its base class, as shown in Listing 7-35. Now replace the declaration of the `Tasks` property in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-35: The Tasks Property

```
protected override TaskListCollection Tasks
{
    get
    {
        if (this.taskList == null)
            this.taskList = new PageTaskList(this);

        TaskListCollection col = base.Tasks;
        col.Add(this.taskList);
        return col;
    }
}
```

`Tasks` first instantiates the `PageTaskList` class, if it hasn't already been instantiated:

```
if (this.taskList == null)
    this.taskList = new PageTaskList(this);
```

Then, it adds the `PageTaskList` instance to the `Tasks` collection of its base class:

```
TaskListCollection col = base.Tasks;
col.Add(this.taskList);
return col;
```

It's very important that you add your custom task list to the `Tasks` collection property of the base class as opposed to creating a new `TaskListCollection` collection. Otherwise, you would lose the task items created by the task list of the base class. In our case, the task list of `ModuleDialogPage` base class creates the method task items that represent the `Apply` and `Cancel` buttons. If you were to create a new `TaskListCollection` collection, add your task list to this new collection, and return the new collection ignoring the task list of the base class, the task panel would not include the `Apply` and `Cancel` buttons, which means that the code that you've added to the `ApplyChanges` and `CancelChanges` methods would not run.

Refreshing

Recall from Chapter 6 that an instance of a frame named `ManagementFrame` contains the entire user interface of the IIS7 Manager. Listing 7-36 repeats Listing 6-28, which contains a simplified version of the internal implementation of the `ManagementFrame` constructor.

Listing 7-36: The `ManagementFrame` Constructor

```
public ManagementFrame(IServiceProvider serviceProvider,
                       IManagementFrameHost owner)
{
    _serviceProvider = serviceProvider;
    _owner = owner;
    INavigationService service2 =
        (INavigationService)_serviceProvider.GetService(typeof(INavigationService));
    service2.NavigationPerformed +=
        new NavigationEventHandler(OnNavigationPerformed);

    CreateHeader();
    CreateMenuBar();
    CreateStatusBar();
    CreateMainArea();
}
```

As the highlighted portion of Listing 7-36 shows, the constructor of the `ManagementFrame` calls four methods named `CreateHeader`, `CreateMenuBar`, `CreateStatusBar`, and `CreateMainArea` to create the header, menu bar, status bar, and main area of the IIS7 Manager interface as shown in Figure 6-11 in the previous chapter.

As you'll see shortly, the `CreateHeader` method instantiates an instance of a control named `PageHeader` to represent the header of the IIS7 Manager user interface. Listing 7-37 presents the `PageHeader` class and its members.

Listing 7-37: The `PageHeader` Class

```
internal sealed class PageHeader : ToolStrip
{
    // Fields
    private bool active;
    private ToolStripButton backButton;
    private BreadCrumbBar breadcrumbBar;
    private ToolStripButton forwardButton;
    private ToolStripSplitButton helpButton;
```

(Continued)

Listing 7-37: (continued)

```
private ToolStripButton homeButton;
private ToolStripSeparator navigationSeparator;
private ToolStripButton refreshButton;
private IServiceProvider serviceProvider;
private ToolStripButton stopButton;

// Methods
public PageHeader(IServiceProvider serviceProvider);
private void InitializeButtons();
protected override void OnGotFocus(EventArgs e);
private void Reset();

// Properties
public bool Active { get; set; }
public ToolStripButton BackButton { get; }
public BreadcrumbBar Breadcrumb { get; }
public ToolStripButton ForwardButton { get; }
public ToolStripSplitButton HelpButton { get; }
public ToolStripButton HomeButton { get; }
public ToolStripButton RefreshButton { get; }
public ToolStripButton StopButton { get; }
}
```

As you can see from Listing 7-38, the `PageHeader` control extends the standard `ToolStrip` control of the `System.Windows.Forms` namespace to add support for the following GUI elements:

- ❑ *Back Button:* This button, the first from the left in the header of the IIS7 Manager, allows the end user to navigate backward.
- ❑ *Forward Button:* This button, located to the right of the back button, allows the end user to navigate forward.
- ❑ *Breadcrumbs Bar:* As the name suggests, the breadcrumbs bar, located to the right of the forward button, allows users to keep track of their navigation through the module pages.
- ❑ *Refresh Button:* This button is located to the right of the breadcrumbs bar. The user clicks this button to refresh the module page shown in middle pane (also known as the workspace) of the IIS7 Manager's user interface, which is the `MyConfigSectionPage` module dialog page in our case.
- ❑ *Stop Button:* This button, located to the right of the refresh button, allows the end user to stop the current operation.
- ❑ *Home Button:* As the name suggests, this button, located to the right of the stop button, allows the end user to navigate to the home page.
- ❑ *Help Button:* This button is located to the right of the home button.

Next, I walk you through the internal implementation of the `CreateHeader` method shown in Listing 7-38.

Listing 7-38: The CreateHeader Method

```
private void CreateHeader()
{
    this.pageHeader = new PageHeader(this.serviceProvider);
    this.pageHeader.Dock = DockStyle.Top;
    this.pageHeader.TabStop = true;
    this.pageHeader.BackButton.Click += new EventHandler(this.OnCommandBarBack);
    this.pageHeader.ForwardButton.Click +=
        new EventHandler(this.OnCommandBarForward);
    this.pageHeader.HomeButton.Click += new EventHandler(this.OnCommandBarHome);
    this.pageHeader.StopButton.Click += new EventHandler(this.OnCommandBarStop);
    this.pageHeader.RefreshButton.Click +=
        new EventHandler(this.OnCommandBarRefresh);
    this.pageHeader.HelpButton.DropDownItems.AddRange(
        (ToolStripItem[]) helpButtonItems.ToArray(typeof(ToolStripItem)));
    this.pageHeader.HelpButton.ButtonClick +=
        new EventHandler(this.OnCommandBarHelp);
    base.Controls.Add(this.pageHeader);
}
```

The main responsibility of the `CreateHeader` method is to instantiate and initialize the `PageHeader` control that represents the header of the IIS7 Manager. As such, this method begins by creating an instance of this control:

```
this.pageHeader = new PageHeader(this.serviceProvider);
```

Next, `CreateHeader` registers methods named `OnCommandBarBack`, `OnCommandBarForward`, `OnCommandBarHome`, `OnCommandBarStop`, `OnCommandBarRefresh`, and `OnCommandBarHelp` as event handlers for the `Click` events of the back, forward, home, stop, refresh, and help buttons, respectively.

We're only interested in the refresh button and consequently in the `OnCommandBarRefresh` method that the `CreateHeader` method registers for the `Click` event of this button. Listing 7-39 presents a portion of the internal implementation of the `OnCommandBarRefresh` method of the `ManagementFrame` control.

Listing 7-39: The OnCommandBarRefresh Method of ManagementFrame

```
private void OnCommandBarRefresh(object sender, EventArgs e)
{
    if ((this.activePage != null) && !this.activePage.InProgress)
    {
        IModulePage page = this.activePage;
        if (page.CanRefresh)
            page.Refresh();
    }

    . . .
}
```

The `ManagementFrame` control maintains the reference to the active module page in a private field named `activePage`. Recall from Chapter 6 that there can be only one active module page at a time. The

Chapter 7: Extending the Integrated Graphical Management System

active module page is the module page currently displayed in the middle pane (also known as the workspace) of the IIS7 Manager. As you can see from Listing 7-39, the `OnCommandBarRefresh` method first invokes the `CanRefresh` property on the active module page to determine whether the module page is refreshable. If so, it invokes the `Refresh` method on the active module page to have this module page refresh itself.

The `ModulePage` class, which is the base class for all module pages, implements an interface named `IModulePage`, which exposes a Boolean property named `CanRefresh`. As Listing 7-40 shows, the `ModulePage` base class's implementation of the `CanRefresh` property of the `IModulePage` interface returns `false`.

Listing 7-40: The `CanRefresh` Property of the `ModulePage` Base Class

```
public abstract class ModulePage : ContainerControl, IModulePage, IDisposable
{
    bool IModulePage.CanRefresh
    {
        get { return this.CanRefresh; }
    }

    protected virtual bool CanRefresh
    {
        get { return false; }
    }
}
```

Because every module page, including your own custom module page, directly or indirectly derives from the `ModulePage` base class, it automatically inherits the `ModulePage` base class's implementation of the `CanRefresh` property. This means that your custom module page is not refreshable by default. To put it differently, by default the end user cannot refresh your custom module page by clicking the `Refresh` button in the header of the IIS7 Manager user interface.

The `MyConfigSectionPage` module dialog page overrides the `CanRefresh` property that it inherits from the `ModulePage` base class to return `true` to allow the end user to use the `Refresh` button to refresh the module page as shown in Listing 7-41. Now replace the declarations of the `CanRefresh` property in the `MyConfigSectionPage.cs` file with the code shown in this code listing.

Listing 7-41: The `CanRefresh` Property of `MyConfigSectionPage` Module Dialog Page

```
protected override bool CanRefresh
{
    get { return true; }
}
```

As discussed earlier, the `OnCommandBarRefresh` method first invokes the `CanRefresh` property on the active module page to check whether the module page is refreshable. If it is, it invokes the `Refresh` method on the active module page to have the module page refresh itself.

Listing 7-42 presents the `ModulePage` base class's implementation of the `Refresh` method of `IModulePage` interface.

Listing 7-42: The Refresh Method of ModulePage Base Class

```
public abstract class ModulePage : ContainerControl, IModulePage, IDisposable
{
    void IModulePage.Refresh()
    {
        this.Refresh();
    }

    protected virtual void Refresh()
    {
        throw new NotImplementedException();
    }
}
```

As you can see, the `ModulePage` base class's implementation of the `Refresh` method simply raises a `NotImplementedException` exception. The `ModuleDialogPage` base class, which is the base class for all module dialog pages, including your `MyConfigSectionPage` module dialog page, overrides the `Refresh` method that it inherits from the `ModulePage` base class. Listing 7-43 presents the internal implementation of the `Refresh` method of the `ModuleDialogPage` base class.

Listing 7-43: The Refresh Method of the ModuleDialogPage Base Class

```
protected sealed override void Refresh()
{
    if (this.HasChanges && !this.ReadOnly)
    {
        string msg =
            "The changes you have made will be lost. Do you want to save changes?";
        switch (base.ShowMessage(msg,
                                MessageBoxButtons.YesNoCancel,
                                MessageBoxIcon.Exclamation,
                                MessageBoxDefaultButton.Button3))
        {
            case DialogResult.Cancel:
                return;

            case DialogResult.Yes:
                if (this.CanApplyChanges)
                {
                    if (!this.ApplyChanges())
                        return;
                }

                else
                {
                    base.ShowMessage("Changes cannot be applied.");
                    return;
                }
                break;
        }
    }
}
```

(Continued)

Listing 7-43: *(continued)*

```
if (this.dialogTaskList != null)
    ((DialogTaskList) this.dialogTaskList).AppliedChanges = false;

this.showDirtyPageAlert = false;
this.OnRefresh();
}
```

As the name suggests, the main responsibility of the `Refresh` method is to download fresh data from the back-end server and to refresh the user interface of the module page with this data. Downloading data involves two classes: the server-side class, which is the `MyConfigSectionModuleService` class in this case, and the client-side class, which is the `MyConfigSectionModuleServiceProxy` class in this case. As should be clear by now, different types of module pages use different types of server-side and client-side classes. In other words, the logic that downloads data from the back-end server varies from one type of module dialog page to another. The logic that updates the user interface of the module dialog page with fresh data is also module-dialog-page-specific because the user interface of different types of module dialog pages consists of different types of GUI elements.

Because the `ModuleDialogPage` base class is the base class for all types of module dialog pages including our `MyConfigSectionPage` module dialog page and because all types of module dialog pages inherit the `ModuleDialogPage` base class's implementation of the `Refresh` method, this implementation must not contain module-dialog-page-specific code such as the logic that downloads fresh data from the back-end server and the logic that updates the user interface of the module dialog page with the fresh data. That is why the `ModuleDialogPage` base class's implementation of the `Refresh` method delegates the responsibility of downloading fresh data from the back-end server and updating the user interface of the module dialog page to another method named `OnRefresh`.

As Listing 7-44 shows, the `ModuleDialogPage` base class's implementation of the `OnRefresh` method simply raises a `NotImplementedException` exception. It is the responsibility of the subclasses of the `ModuleDialogPage` base class, such as `MyConfigSectionPage`, to implement the `OnRefresh` method to incorporate the logic that downloads the fresh data from the back-end server and the logic that updates the user interface of the module dialog page with the fresh data.

Listing 7-44: The OnRefresh Method

```
protected virtual void OnRefresh()
{
    throw new NotImplementedException();
}
```

Even though the `ModuleDialogPage` base class does not implement the `OnRefresh` method, it does guarantee the automatic invoking of this method every time the end user clicks the `Refresh` button in the header of the IIS7 Manager user interface. The `ModuleDialogPage` base class provides this guarantee by automatically invoking the `OnRefresh` method from the `Refresh` method as shown in Listing 7-43. Recall that the `Refresh` method is automatically invoked every time the end user clicks the `Refresh` button in the header of the IIS7 Manager user interface.

Therefore, your custom module dialog page does not have to worry about invoking the `OnRefresh` method. It is done automatically behind the scenes. Your custom module dialog page's sole responsibility

Chapter 7: Extending the Integrated Graphical Management System

is to implement this method to incorporate the logic that downloads fresh data from the server and the logic that updates its user interface with this fresh data. I present the `MyConfigSectionPage` module dialog page's implementation of the `OnRefresh` method shortly. However, before diving into this implementation, let's study the `ModuleDialogPage` base class's implementation of the `Refresh` method as shown in Listing 7-43.

As you can see, the `Refresh` method first checks whether both of the following conditions are met:

- ❑ The `HasChanges` property of the module dialog page returns `true`. This property returns `true` when the module dialog page has changes to commit to the underlying configuration file.
- ❑ The `ReadOnly` property of the module dialog page returns `false` to indicate that the module dialog page is *not* in read-only mode. Recall that when a module page is in read-only mode, the end users cannot update the configuration settings that the module page displays.

If both of these conditions are met, the `Refresh` method knows that the end user has changed some of the values displayed in the module dialog page's user interface. Therefore, the `Refresh` method needs to do several things before an attempt is made to download fresh data from the server and to update the user interface with this fresh data. First, it pops up a dialog with Yes, No, and Cancel buttons that displays the following message:

```
string msg =  
    "The changes you have made will be lost. Do you want to save changes?";
```

To see this dialog in action, launch the IIS7 Manager. Click the Default Web Site node from the Connections pane and double-click the Session State option from the workspace to navigate to the Session State page shown in Figure 7-13.



Figure 7-13

Chapter 7: Extending the Integrated Graphical Management System

Now select the “In process” radio button. Now if you click the Refresh button in the header of the IIS7 Manager’s user interface, the dialog box shown in Figure 7-14 will pop up.

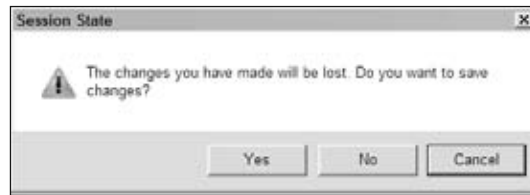


Figure 7-14

As you can see, this dialog box contains three buttons: Yes, No, and Cancel.

Now back to the `ModuleDialogPage` base class’s implementation of the `Refresh` method shown in Listing 7-43. If the end user clicks the Cancel button, the `Refresh` method simply returns, bypassing the call into the `OnRefresh` method, which is the method that downloads fresh data and updates the user interface of the module dialog page with this fresh data.

```
case DialogResult.Cancel:
    return;
```

If the end user clicks the Yes button, the `Refresh` method first invokes the `CanApplyChanges` method on the module dialog page to determine whether the module dialog page allows changes to be committed to the underlying configuration file. Recall that the `MyConfigSectionPage` module dialog page’s implementation of the `CanApplyChanges` property simply returns the value of the `hasChanged` Boolean value. If the module dialog page does allow changes to be committed, the `Refresh` method calls the `ApplyChanges` method on the module dialog page to commit the changes to the underlying configuration file. If the `ApplyChanges` method returns `false`, indicating that something went wrong and the changes were not committed, the `Refresh` method simply returns, bypassing the call into the `OnRefresh` method.

```
case DialogResult.Yes:
    if (this.CanApplyChanges)
    {
        if (!this.ApplyChanges())
            return;
    }
```

Finally, the `Refresh` method invokes the `OnRefresh` method to download fresh data from the server and to update the module dialog page’s user interface with the fresh data:

```
this.OnRefresh();
```

The following code presents the `MyConfigSectionPage` module dialog page’s implementation of the `OnRefresh` method. Now replace the declaration of the `OnRefresh` method in the `MyConfigSectionPage.cs` file with the code shown here:

```
protected override void OnRefresh()
```

```
{  
    this.GetSettings();  
}
```

As you can see, the `OnRefresh` method invokes the `GetSettings` method discussed earlier in this chapter to download fresh data from the server and to update the user interface of the `MyConfigSectionPage` module dialog page with this fresh data.

MyCollectionPage

As discussed, when the user clicks the “View collection items” link shown in Figure 7-4, the `ViewCollectionItems` method of the `MyConfigSectionPage` module dialog page uses the navigation service to navigate to the `MyCollectionPage` module list page.

Listing 7-45 presents the `MyCollectionPage` class and the declaration of its members. Add a new source file named `MyCollectionPage.cs` to the `Client` subdirectory of the `GraphicalManagement` directory of the `MyConfigSection` project and add the code shown in this code listing to this source file.

Listing 7-45: The `MyCollectionPage` Class

```
using Microsoft.Web.Management.Client;  
using Microsoft.Web.Management.Client.Win32;  
using Microsoft.Web.Management.Server;  
using System.ComponentModel;  
using System.Collections;  
using System.Windows.Forms;  
using System;  
  
namespace MyNamespace.GraphicalManagement.Client  
{  
    public class MyCollectionPage : ModuleListPage, IModuleChildPage  
    {  
        private IModulePage parentPage;  
        private bool errorGetCollectionItems;  
        private ColumnHeader myCollectionItemBoolValueColumnHeader;  
        private ColumnHeader myCollectionItemIdentifierColumnHeader;  
        private MyConfigSectionModuleServiceProxy serviceProxy;  
        private PageTaskList taskList;  
        private PropertyBag bag;  
        private bool readOnly;  
        private ModuleListPageGrouping booleanPropertyGrouping;  
        private ListViewGroup trueGroup;  
        private ListViewGroup falseGroup;  
  
        protected override void InitializeListPage();  
        protected override void OnActivated(bool initialActivation);  
        private void GetCollectionItems();  
        private void OnWorkerGetCollectionItems(object sender, DoWorkEventArgs e);  
        private void OnWorkerGetCollectionItemsCompleted(object sender,  
                                                         RunWorkerCompletedEventArgs e);  
    }  
}
```

(Continued)

Listing 7-45: (continued)

```
private sealed class MyCollectionItemListViewItem : ListViewItem { }
private void AddItem(MyCollectionItemInfo itemInfo, bool isSelected);
private sealed class PageTaskList : TaskList { }
protected override TaskListCollection Tasks { get; }
private void AddCollectionItem();
private void DeleteCollectionItem();
private void UpdateCollectionItem();

IModulePage IModuleChildPage.ParentPage { get; set; }
private MyCollectionItemListViewItem SelectedCollectionItem { get; }

private void ReplaceItem(MyCollectionItemListViewItem item,
                        MyCollectionItemInfo itemInfo);

private void OnListViewSelectedIndexChanged(object sender, EventArgs e);
private void OnListViewDoubleClick(object sender, EventArgs e);
private void OnListViewKeyUp(object sender, KeyEventArgs e);
private void SetItemGroup(MyCollectionItemListViewItem item);
protected override void OnGroup(ModuleListPageGrouping grouping);
protected override ListViewGroup[] GetGroups(ModuleListPageGrouping grouping);
public override ModuleListPageGrouping[] Groupings { get; }
protected override void Refresh();
protected override bool CanRefresh { get; }
protected sealed override bool ReadOnly { get; }
protected override string ReadOnlyDescription { get; }
private void UpdateCollectionItemIdentifier();
private void OnListViewBeforeLabelEdit(object sender, LabelEditEventArgs e);
private void OnListViewAfterLabelEdit(object sender, LabelEditEventArgs e);
}
}
```

I present and discuss the implementations of the methods and properties of the `MyCollectionPage` module list page in the following sections.

The `MyCollectionPage` module list page, like any other module list page, inherits from the `ModuleListPage` base class, which in turn inherits from the `ModulePage`. Note that the `MyCollectionPage` module list page also implements the `IModuleChildPage` interface:

The following code listing presents the definition of the `IModuleChildPage` interface:

```
public interface IModuleChildPage
{
    IModulePage ParentPage { get; set; }
}
```

This interface exposes a single property named `ParentPage`, which references the parent module page. In our case, the parent module page of the `MyCollectionPage` module list page is the `MyConfigSectionPage` module dialog page. The following code listing contains the `MyCollectionPage`

Chapter 7: Extending the Integrated Graphical Management System

module page's implementation of the `ParentPage` property of this interface. Replace the declaration of the `IModuleChildPage.ParentPage` property in the `MyCollectionPage.cs` file with the following code:

```
IModulePage IModuleChildPage.ParentPage
{
    get { return parentPage; }
    set { parentPage = value; }
}
```

Your module page mustn't assign a value to the `ParentPage` property. The IIS7 Manager infrastructure does this automatically behind the scenes as shown in Listing 7-46, which contains a portion of the internal implementation of the `NavigationService` class.

Listing 7-46: The `NavigationService` Class

```
internal sealed class NavigationService : INavigationService, IDisposable
{
    bool INavigationService.Navigate(Connection connection,
                                     ManagementConfigurationPath configurationPath,
                                     Type pageType, object navigationData)
    {
        NavigationItem destinationNavigationItem =
            this.CreateNavigationItem(connection, configurationPath,
                                     pageType, navigationData);

        return this.NavigateToItem(destinationNavigationItem, true);
    }

    private NavigationItem CreateNavigationItem(Connection connection,
                                                ManagementConfigurationPath configurationPath,
                                                Type pageType, object navigationData)
    {
        NavigationItem destinationNavigationItem =
            new NavigationItem(connection, configurationPath, pageType, navigationData);

        if (typeof(IModuleChildPage).IsAssignableFrom(pageType))
        {
            IModuleChildPage childModulePage =
                destinationNavigationItem.Page as IModuleChildPage;
            if (childModulePage != null)
            {
                INavigationService navigationService = (INavigationService)
                    ((IServiceProvider) connection).GetService(typeof(INavigationService));
                NavigationItem currentNavigationItem = navigationService.CurrentItem;
                childModulePage.ParentPage = currentNavigationItem.Page;
            }
        }

        return destinationNavigationItem;
    }
    . . .
}
```

Listing 7-46 presents the `NavigationService` class's implementation of the `Navigate` method of the `INavigationService` interface. This method is automatically invoked every time the end user takes an

Chapter 7: Extending the Integrated Graphical Management System

action that triggers navigation from one module page to another. As discussed in Chapter 6, navigation involves two `NavigationItem` navigation items: the current navigation item and destination navigation item. In other words, the navigation service navigates from the current navigation item to the destination navigation item. For example, when the end user clicks the “View collection items” link in the task panel associated with the `MyConfigSectionPage` module dialog page, the navigation service navigates from the current navigation item, which represents the `MyConfigSectionPage` module dialog page, to the destination navigation item, which represents the `MyCollectionPage` module list page.

As you can see from Listing 7-46, the `Navigate` method begins by invoking another method named `CreateNavigationItem` to create the destination navigation item with the specified connection, configuration path, module page type, and navigation data as discussed in Chapter 6:

```
NavigationItem destinationNavigationItem =  
    this.CreateNavigationItem(connection, configurationPath,  
                             pageType, navigationData);
```

Finally, the `Navigate` method invokes the `NavigateToItem` method to navigate to the destination navigation item:

```
this.NavigateToItem(destinationNavigationItem, true);
```

Next, I walk you through the implementation of the `CreateNavigationItem` method. As Listing 7-47 shows, this method begins by instantiating the target navigation item with the specified connection, configuration path, module page type, and navigation data as you would expect:

```
NavigationItem destinationNavigationItem =  
    new NavigationItem(connection, configurationPath, pageType, navigationData);
```

Next, it invokes the `IsAssignableFrom` method on the `Type` object that represents the `IModuleChildPage` interface, passing in the `Type` object that represents the target module page to determine whether the target module page implements the `IModuleChildPage` interface. For example, in this case, when the end user clicks the “View collection items” link in the task panel associated with the `MyConfigSectionPage` module list page, a call into the `Navigate` method of the `NavigationService` is automatically triggered. This call in turns triggers the call into the `CreateNavigationItem` method, which in turn calls the `IsAssignableFrom` method to determine whether the target module page, which is the `MyCollectionPage` module list page, implements the `IModuleChildPage`. In our case, the `MyCollectionPage` module list page indeed implements this interface.

As Listing 7-46 shows, if the target module page implements the `IModuleChildPage` interface, the `CreateNavigationItem` method invokes the `Page` property on the newly instantiated navigation item to return a reference to the target module page, which is the `MyCollectionPage` module list page in this case.

```
IModuleChildPage childModulePage =  
    destinationNavigationItem.Page as IModuleChildPage;
```

Next, the `CreateNavigationItem` invokes the `GetService` method on the connection object that represents the current connection to the server, passing in the `Type` object that represents the `INavigationService` interface to return a reference to the `NavigationService` object:

```
INavigationService navigationService = (INavigationService)  
    ((IServiceProvider) connection).GetService(typeof(INavigationService));
```

Chapter 7: Extending the Integrated Graphical Management System

Next, the `CreateNavigationItem` method invokes the `CurrentItem` property on the navigation service to return a reference to the current navigation item, which is the navigation item that represents the `MyConfigSectionPage` module list page in this case:

```
NavigationItem currentNavigationItem = navigationService.CurrentItem;
```

Then, `CreateNavigationItem` calls the `Page` property on the current navigation item to return a reference to the module page associated with the current navigation item, which is the `MyConfigSectionPage` module list page in this case, and assigns this reference to the `ParentPage` property of the target module page, which is the `MyCollectionPage` module list page in our case:

```
childModulePage.ParentPage = currentNavigationItem.Page;
```

Therefore, when the user attempts to navigate from a module page to its child module page, the navigation service automatically assigns the reference to the parent module page, which is the `MyConfigSectionPage` module dialog page in this case, to the `ParentPage` property of the child module page, which is the `MyCollectionPage` module list page in this case. As such you mustn't directly set the value of the `ParentPage` property of your custom module page.

Listing 7-47 presents those members of the `ModuleListPage` base class that `MyCollectionPage` overrides.

Listing 7-47: The `ModuleListPage` Class

```
public abstract class ModuleListPage : ModulePage
{
    protected abstract void InitializeListPage();
    protected override void OnActivated(bool initialActivation);
    protected virtual ListViewGroup[] GetGroups(ModuleListPageGrouping grouping);
    protected virtual void OnGroup(ModuleListPageGrouping grouping);
    public virtual ModuleListPageGrouping[] Groupings { get; }
}
```

Note that the `ModuleListPage` base class features an abstract method named `InitializeListPage` that your custom module list page must implement if it inherits from this base class.

Listing 7-48 presents those members of the `ModulePage` base class that the `MyCollectionPage` overrides.

Listing 7-48: The `ModulePage` Class

```
public abstract class ModulePage : ContainerControl, IModulePage, IDisposable
{
    protected virtual void Refresh();
    protected virtual TaskListCollection Tasks { get; }
    protected virtual bool CanRefresh { get; }
    protected virtual bool ReadOnly { get; }
    protected virtual string ReadOnlyDescription { get; }
}
```

InitializeListPage

Every module page that inherits from the `ModuleListPage` base class must implement the `InitializeListPage` method because this method is marked as abstract. Listing 7-49 contains the `MyCollectionPage` module list page's implementation of the `InitializeListPage` method. Now replace the declaration of the `InitializeListPage` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-49: The InitializeListPage Method

```
protected override void InitializeListPage()
{
    myCollectionItemBoolValueColumnHeader = new ColumnHeader();
    myCollectionItemBoolValueColumnHeader.Text = "Boolean Value";
    myCollectionItemBoolValueColumnHeader.Width = 90;
    myCollectionItemIdentifierColumnHeader = new ColumnHeader();
    myCollectionItemIdentifierColumnHeader.Text = "Identifier";
    myCollectionItemIdentifierColumnHeader.Width = 90;
    base.ListView.Columns.Clear();
    base.ListView.Columns.AddRange(new ColumnHeader[] {
        myCollectionItemIdentifierColumnHeader,
        myCollectionItemBoolValueColumnHeader });

    base.ListView.MultiSelect = false;
    base.ListView.LabelEdit = true;
    base.ListView.AfterLabelEdit +=
        new LabelEditEventHandler(this.OnListViewAfterLabelEdit);
    base.ListView.BeforeLabelEdit +=
        new LabelEditEventHandler(this.OnListViewBeforeLabelEdit);
    base.ListView.SelectedIndexChanged +=
        new EventHandler(this.OnListViewSelectedIndexChanged);
    base.ListView.DoubleClick += new EventHandler(this.OnListViewDoubleClick);
    base.ListView.KeyUp += new KeyEventHandler(this.OnListViewKeyUp);
}
```

`InitializeListPage` creates two columns to display the values of the Boolean property and identifier of the collection items, and adds them to the list of columns. It then registers the `OnListViewAfterLabelEdit`, `OnListViewBeforeLabelEdit`, `OnListViewSelectedIndexChanged`, `OnListViewDoubleClick`, and `OnListViewKeyUp` methods as event handlers for the `AfterLabelEdit`, `BeforeLabelEdit`, `SelectedIndexChanged`, `DoubleClick`, and `KeyUp` events of the `ListView` list view that displays the collection items:

```
base.ListView.AfterLabelEdit +=
    new LabelEditEventHandler(this.OnListViewAfterLabelEdit);
base.ListView.BeforeLabelEdit +=
    new LabelEditEventHandler(this.OnListViewBeforeLabelEdit);
base.ListView.SelectedIndexChanged +=
    new EventHandler(this.OnListViewSelectedIndexChanged);
base.ListView.DoubleClick += new EventHandler(this.OnListViewDoubleClick);
base.ListView.KeyUp += new KeyEventHandler(this.OnListViewKeyUp);
```

OnActivated

The `OnActivated` method is invoked when the `MyCollectionPage` is accessed (see Listing 7-50). If the module page is being accessed for the first time, the method calls the `CreateProxy` method of the `ModulePage` base class to instantiate the `MyConfigSectionModuleServiceProxy` proxy class and stores this instance in a private field named `serviceProxy` for future reference. Note that `OnActivated` calls the `GetCollectionItems` method to download the collection items from the server. Replace the declaration of the `OnActivated` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-50.

Listing 7-50: The `OnActivated` Method

```
protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (MyConfigSectionModuleServiceProxy)base.CreateProxy(
                                                                    typeof(MyConfigSectionModuleServiceProxy));
        this.GetCollectionItems();
    }
}
```

GetCollectionItems

The `GetCollectionItems` method passes two delegates to the `StartAsyncTask` method of the base class (see Listing 7-51). The first delegate is a `DoWorkEventHandler` delegate that wraps the `OnWorkerGetCollectionItems` method. The second delegate is a `RunWorkerCompletedEventHandler` delegate that encapsulates the `OnWorkerGetCollectionItemsCompleted` method. These two delegates and the `StartAsyncTask` method were discussed in the previous chapter. Replace the declaration of the `GetCollectionItems` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-51.

Listing 7-51: The `GetCollectionItems` Method

```
private void GetCollectionItems()
{
    base.ListView.LabelEdit = false;
    base.StartAsyncTask("Getting collection items",
                        new DoWorkEventHandler(OnWorkerGetCollectionItems),
                        new RunWorkerCompletedEventHandler(OnWorkerGetCollectionItemsCompleted));
}
```

OnWorkerGetCollectionItems

The `OnWorkerGetCollectionItems` method simply calls the `GetCollectionItems` method of the proxy to download the collection items from the server (see Listing 7-52). As discussed in the previous chapter, the `GetCollectionItems` method of the proxy calls the `GetCollectionItems` method of the server-side class. Replace the declaration of the `OnWorkerGetCollectionItems` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-52.

Listing 7-52: The OnWorkerGetCollectionItems Method

```
private void OnWorkerGetCollectionItems(object sender, DoWorkEventArgs e)
{
    e.Result = serviceProxy.GetCollectionItems();
}
```

OnWorkerGetCollectionItemsCompleted

The `OnWorkerGetCollectionItemsCompleted` method is automatically called right after the collection items are retrieved from the server (see Listing 7-53). Now replace the declaration of the `OnWorkerGetCollectionItemsCompleted` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-53.

Listing 7-53: The OnWorkerGetCollectionItemsCompleted Method

```
private void OnWorkerGetCollectionItemsCompleted(object sender,
                                                RunWorkerCompletedEventArgs e)
{
    base.ListView.BeginUpdate();
    try
    {
        if (e.Result != null)
        {
            base.ListView.Items.Clear();
            this.bag = (PropertyBag)e.Result;
            this.readOnly = (bool)this.bag[1];
            ArrayList list1 = (ArrayList)this.bag[0];
            if (list1 != null)
            {
                for (int num1 = 0; num1 < list1.Count; num1++)
                {
                    MyCollectionItemInfo info1 =
                        new MyCollectionItemInfo((PropertyBag)list1[num1]);
                    this.AddItem(info1, false);
                }
            }
            this.errorGetCollectionItems = false;
        }
    }
    catch (Exception exception1)
    {
        base.StopProgress();
        base.DisplayErrorMessage(exception1.Message, "GetCollectionItemsCompleted");
        this.errorGetCollectionItems = true;
        return;
    }
    finally
    {
        base.ListView.LabelEdit = true;
        base.ListView.EndUpdate();
    }
}
```

Chapter 7: Extending the Integrated Graphical Management System

This method first clears the `Items` collection of the `ListView` list view that displays the collection items:

```
base.ListView.Items.Clear();
```

Then, it stores the `PropertyBag` collection that contains the retrieved collection items in the local bag `PropertyBag` for future reference:

```
this.bag = (PropertyBag)e.Result;
```

As you'll see later, the `GetCollectionItems` method of the server-side class adds the value of the `isLocked` attribute of the underlying configuration section as the second item to the `PropertyBag` collection. `OnWorkerGetCollectionItemsCompleted` assigns this item to the `readOnly` Boolean field:

```
this.readOnly = (bool)this.bag[1];
```

As a matter of fact, the `MyCollectionPage` module list page overrides the `ReadOnly` and `ReadOnlyDescription` properties of its base class, as shown in Listing 7-54. Now replace the declarations of the `ReadOnly` and `ReadOnlyDescription` properties in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-54: The `ReadOnly` and `ReadOnlyDescription` Properties

```
protected sealed override bool ReadOnly
{
    get{return this.readOnly;}
}

protected override string ReadOnlyDescription
{
    get{return "This feature is locked";}
}
```

Now back to the implementation of the `OnWorkerGetCollectionItemsCompleted` method. As you'll see later, the `GetCollectionItems` method of the server-side class creates one `PropertyBag` object for each collection item, stores the values of the Boolean property and identifier of the collection item in this `PropertyBag`, and adds the `PropertyBag` object into an `ArrayList`. It then creates another `PropertyBag` object, stores the `ArrayList` into it, and sends this `PropertyBag` to the client. As mentioned earlier, this `PropertyBag` collection also contains the `isLocked` attribute value of the configuration section. Therefore, the `OnWorkerGetCollectionItemsCompleted` method accesses the `ArrayList`:

```
ArrayList list1 = (ArrayList)this.bag[0];
```

It then iterates through the `PropertyBag` objects in this `ArrayList`, creates one `MyCollectionItemInfo` object for each enumerated `PropertyBag` object, and calls the `AddItem` method, passing in the `MyCollectionItemInfo` object:

```
if (list1 != null)
{
    for (int num1 = 0; num1 < list1.Count; num1++)
```



```
{
    MyCollectionItemInfo info1 = new
        MyCollectionItemInfo((PropertyBag)list1[num1]);
    this.AddItem(info1, false);
}
```

MyCollectionItemInfo

The `MyCollectionItemInfo` class exposes the contents of its associated `PropertyBag` object as strongly-typed properties (see Listing 7-55). The previous section discussed the benefits of exposing the contents of a `PropertyBag` object as strongly-typed properties. Add a new source file named `MyCollectionItemInfo.cs` to the `GraphicalManagement/Client` directory of the `MyConfigSection` project and add the code shown in Listing 7-55 to this source file.

Listing 7-55: The MyCollectionItemInfo Class

```
using Microsoft.Web.Management.Server;

namespace MyNamespace.GraphicalManagement.Client
{
    internal sealed class MyCollectionItemInfo
    {
        internal MyCollectionItemInfo(PropertyBag bag)
        {
            this.bag = bag;
        }

        public bool MyCollectionItemBoolProperty
        {
            get { return (bool)this.bag[1]; }
            set { this.bag[1] = value; }
        }

        public string MyCollectionItemIdentifier
        {
            get { return (string)this.bag[0]; }
            set { this.bag[0] = value; }
        }

        private PropertyBag bag;
    }
}
```

As you can see from Listing 7-55, the `MyCollectionItemInfo` class exposes the first item in the `PropertyBag` collection, which is nothing but the value of the `myCollectionItemIdentifier` attribute, as a strongly-typed property named `MyCollectionItemIdentifier`. It exposes the second item in the `PropertyBag` collection, which is nothing but the value of the `myCollectionItemBoolAttr` attribute, as a strongly-typed property named `MyCollectionItemBoolProperty`.

MyCollectionItemListViewItem

When you write a custom module page that inherits the `ModuleListPage` base class, you must also implement a class that inherits the `ListViewItem` class. This class must be private and nested within your custom module list page (see Listing 7-56). Each instance of this class will represent a particular item in the list of items that your custom module list page displays.

Following this pattern, Listing 7-56 implements a private nested class named `MyCollectionItemListViewItem` that inherits the `ListViewItem` class. Note that the constructor of this class passes the identifier of the associated item to the constructor of the `ListViewItem` base class. The base class uses this value to uniquely identify each item in the list of displayed items. Replace the declaration of the `MyCollectionItemListViewItem` nested class in the `MyCollectionPage.cs` file with the code shown in Listing 7-56.

Listing 7-56: The `MyCollectionItemListViewItem` Class

```
class MyCollectionPage : ModuleListPage
{
    . . .
    private sealed class MyCollectionItemListViewItem : ListViewItem
    {
        public MyCollectionItemListViewItem(MyCollectionItemInfo itemInfo)
            : base(itemInfo.MyCollectionItemIdentifier)
        {
            this.itemInfo = itemInfo;
            base.SubItems.Add(new ListViewItem.ListViewSubItem(this,
                itemInfo.MyCollectionItemBoolProperty.ToString()));
        }

        public MyCollectionItemInfo ItemInfo
        {
            get { return this.itemInfo; }
        }

        private MyCollectionItemInfo itemInfo;
    }
}
```

AddItem

Recall from Listing 7-53 that the `OnWorkerGetCollectionItemsCompleted` method iterates through the `ArrayList` of `PropertyBag` objects that it has received from the server, creates a `MyCollectionItemInfo` object for each enumerated `PropertyBag` object, and calls the `AddItem` method, passing in the `MyCollectionItemInfo` object. The main responsibility of the `AddItem` method is to create a `MyCollectionItemListViewItem` list view item to represent the associated `MyCollectionItemInfo` object and add this `MyCollectionItemListViewItem` list view item to the `Items` collection of the `ListView` list view that displays the collection items. This collection contains the list view items that represent the displayed items. Note that the `AddItem` method invokes the `SetItemGroup` method, passing in the new `MyCollectionItemListViewItem` list view item to set the item's group. I discuss grouping later in this chapter. Now replace the declaration of the `AddItem` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-57.

Listing 7-57: The AddItem Method

```
private void AddItem(MyCollectionItemInfo itemInfo, bool isSelected)
{
    MyCollectionItemListViewItem item1 = new MyCollectionItemListViewItem(itemInfo);
    base.ListView.Items.Add(item1);
    this.SetItemGroup(item1);

    if (isSelected)
    {
        item1.Selected = true;
        item1.Focused = true;
        base.ListView.EnsureVisible(base.ListView.Items.IndexOf(item1));
    }
}
```

Adding Support for New Task Items

Next, you would like to add four new links titled “Add collection item,” “Update collection item,” “Delete collection item,” and “Change identifier” to the task panel associated with the `MyCollectionPage` module page (see Figure 7-7). As discussed earlier, it takes three steps to add these new links:

- ❑ Implement a private nested class that inherits the `TaskList` base class. This nested class is normally named `PageTaskList`.
- ❑ Override the `Tasks` property of your module page.
- ❑ Implement an event handler associated with each new link.

Custom Task List

Listing 7-58 presents the implementation of the `PageTaskList` class. As you can see, `PageTaskList`, like any other task list, derives from the `TaskList` base class and overrides the `GetTaskItems` method that it inherits from this base class. Replace the declaration of the `PageTaskList` nested class in the `MyCollectionPage.cs` file with the code shown in Listing 7-58.

Listing 7-58: The PageTaskList Class

```
namespace MyNamespace.GraphicalManagement.Client
{
    public class MyCollectionPage : ModuleListPage, IModuleChildPage
    {
        . . .

        private sealed class PageTaskList : TaskList
        {
            public PageTaskList(MyCollectionPage owner)
            {
                this.owner = owner;
            }

            public void AddCollectionItem()
```

Listing 7-58: (continued)

```
{
    this.owner.AddCollectionItem();
}

public void DeleteCollectionItem()
{
    this.owner.DeleteCollectionItem();
}

public void UpdateCollectionItem()
{
    this.owner.UpdateCollectionItem();
}

public void UpdateCollectionItemIdentifier()
{
    this.owner.UpdateCollectionItemIdentifier();
}

public override ICollection GetTaskItems()
{
    ArrayList list1 = new ArrayList();
    if (!owner.ReadOnly && !owner.errorGetCollectionItems)
    {
        list1.Add(new MethodTaskItem("AddCollectionItem",
                                     "Add collection item", "Add"));
        if (owner.SelectedCollectionItem != null)
        {
            list1.Add(new MethodTaskItem("UpdateCollectionItem",
                                         "Update collection item", "Tasks"));
            list1.Add(new MethodTaskItem("UpdateCollectionItemIdentifier",
                                         "Change identifier", "Tasks"));
            list1.Add(new MethodTaskItem("DeleteCollectionItem",
                                         "Delete collection item", "Tasks"));
        }
    }
    if (owner.errorGetCollectionItems)
        list1.Add(new MessageTaskItem(MessageTaskItemType.Error, "Error",
                                       "Info", "Error"));

    foreach (TaskItem item2 in list1)
    {
        if (!(item2 is MessageTaskItem) && !(item2 is TextTaskItem))
            item2.Enabled = !owner.InProgress;
    }
    return (TaskItem[])list1.ToArray(typeof(TaskItem));
}

private MyCollectionPage owner;
}
```

Chapter 7: Extending the Integrated Graphical Management System

Now, let's walk through the implementation of the `GetTaskItems` method of our `PageTaskList` nested class. This method follows the same pattern as the `GetTaskItems` method of any other task list class. First, it creates an `ArrayList`:

```
ArrayList list1 = new ArrayList();
```

Next, it checks whether both of the following conditions are met:

- ❑ The `MyCollectionPage` module list page is editable, that is, the end user can add, remove, and update collection items and change their identifiers.
- ❑ The `MyCollectionPage` module list page did not have problems downloading the collection items from the server.

If both of these conditions are met, the method creates a `MethodTaskItem` task item to represent the "Add collection item" link. Note that it specifies the `AddCollectionItem` method as the event handler for the `Click` event of this link button. As Listing 7-58 shows, this method calls the `AddCollectionItem` method of the `MyCollectionPage` module page:

```
list1.Add(new MethodTaskItem("AddCollectionItem", "Add collection item", "Add"));
```

The `GetTaskItems` method then checks whether an item has been selected from the list of displayed items. If so, it creates three more `MethodTaskItem` task items to represent the "Update collection item," "Change identifier," and "Delete collection item" links. This means that these three links are rendered only when an item is selected from the list. This makes sense because the user must first select an item before updating or deleting it or changing its identifier. Notice that these three method task items register the `UpdateCollectionItem`, `UpdateCollectionItemIdentifier`, and `DeleteCollectionItem` methods as event handlers for the `Click` events of these three links. As Listing 7-58 shows, these three methods call the `UpdateCollectionItem`, `UpdateCollectionItemIdentifier`, and `DeleteCollectionItem` methods of the `MyCollectionPage` module page, respectively. Note that `GetTaskItems` method passes the same string "Tasks" as the third argument to the constructor of the `MethodTaskItem` class when it is instantiating these three task items to instruct the IIS7 Manager that all these three tasks items belong to the same category named "Tasks."

```
if (owner.SelectedCollectionItem != null)
{
    list1.Add(new MethodTaskItem("UpdateCollectionItem",
                                "Update collection item", "Tasks"));
    list.Add(new MethodTaskItem("UpdateCollectionItemIdentifier",
                                "Change identifier", "Tasks"));
    list1.Add(new MethodTaskItem("DeleteCollectionItem",
                                "Delete collection item", "Tasks"));
}
```

`GetTaskItems` then checks whether the `MyCollectionPage` module list page had trouble retrieving the collection items from the server. If so, it creates a `MessageTaskItem` task item to display an error message in the Alerts panel.

```
if (owner.errorGetCollectionItems)
    list1.Add(new MessageTaskItem(MessageTaskItemType.Error, "Error", "Info", "Error"));
```

Tasks

The `MyCollectionPage` module list page overrides the `Tasks` property where it follows the same pattern as the `Tasks` property of any other module page (see Listing 7-59). First, it instantiates the `PageTaskList` if it hasn't already been instantiated. Next, it adds this `PageTaskList` task list to the `Tasks` collection property of its base class. Replace the declaration of the `Tasks` property in the `MyCollectionPage.cs` file with the code shown in Listing 7-59.

Listing 7-59: The Tasks Property

```
protected override TaskListCollection Tasks
{
    get
    {
        if (this.taskList == null)
            this.taskList = new PageTaskList(this);

        TaskListCollection collection1 = base.Tasks;
        collection1.Add(this.taskList);
        return collection1;
    }
}
```

UpdateCollectionItemIdentifier

As discussed earlier, the `GetTaskItems` method of the `PageTaskList` nested type registers the `UpdateCollectionItemIdentifier` method of the `PageTaskList` nested type as the event handler for the `Click` event of the “Change identifier” link button. As Listing 7-58 shows, this method simply delegates to the `UpdateCollectionItemIdentifier` method of the `MyCollectionPage` module dialog page shown in Listing 7-60. This method in turn invokes the `BeginEdit` method on the `MyCollectionItemListViewItem` list view item that represents the selected item to start the label editing process, which allows the end user to edit the identifier of the selected item. Replace the declaration of the `UpdateCollectionItemIdentifier` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-60.

Listing 7-60: The UpdateCollectionItemIdentifier Method of MyCollectionPage

```
private void UpdateCollectionItemIdentifier()
{
    this.SelectedCollectionItem.BeginEdit();
}
```

AddCollectionItem

Listing 7-61 presents the implementation of the `AddCollectionItem` method of the `MyCollectionPage` module list page. Replace the declaration of the `AddCollectionItem` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

The `AddCollectionItem` method first instantiates and launches a `MyCollectionItemTaskForm` task form to allow the end user to specify the values of the Boolean property and identifier of the collection item being added (see Listing 7-61). As you'll see later, when the user clicks the OK button on the task form, the event handler for the button uses the proxy to add the new collection item to the underlying

Chapter 7: Extending the Integrated Graphical Management System

configuration file. When the task form finally returns, `AddCollectionItem` takes these steps to add the new collection item to the list of displayed items:

1. Creates a `PropertyBag`:

```
PropertyBag bag1 = new PropertyBag();
```

2. Populates the `PropertyBag` with the values of the Boolean property and identifier of the new collection item. As you'll see later, the `MyCollectionItemTaskForm` task form exposes these two values as strongly-typed `MyCollectionItemBoolProperty` and `MyCollectionItemIdentifier` properties.

```
bag1[0] = form1.MyCollectionItemIdentifier;  
bag1[1] = form1.MyCollectionItemBoolProperty;
```

3. Creates a `MyCollectionItemInfo` object, passing in the `PropertyBag`. Recall that the `MyCollectionItemInfo` object exposes the content of the `PropertyBag` as strongly-typed properties. Finally it calls the `AddItem` method to add the new collection item to the list of items displayed in the `MyCollectionPage` module list page.

```
this.AddItem(new MyCollectionItemInfo(bag1), true);
```

Listing 7-61: The `AddCollectionItem` Method of `MyCollectionPage` Module List Page

```
private void AddCollectionItem()  
{  
    using (MyCollectionItemTaskForm form1 =  
        new MyCollectionItemTaskForm(base.Module, this.serviceProxy))  
    {  
        if (base.ShowDialog(form1) == DialogResult.OK)  
        {  
            PropertyBag bag1 = new PropertyBag();  
            bag1[0] = form1.MyCollectionItemIdentifier;  
            bag1[1] = form1.MyCollectionItemBoolProperty;  
            this.AddItem(new MyCollectionItemInfo(bag1), true);  
        }  
    }  
}
```

DeleteCollectionItem

Listing 7-62 contains the code for the `DeleteCollectionItem` method of the `MyCollectionPage` module list page. Replace the declaration of the `DeleteCollectionItem` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-62: The `DeleteCollectionItem` Method of `MyCollectionPage` Module List Page

```
private void DeleteCollectionItem()  
{  
    MyCollectionItemListViewItem item1 = this.SelectedCollectionItem;  
    if (item1 != null)  
    {  
        DialogResult result1 = base.ShowMessage(  

```

Listing 7-62: (continued)

```
        "Do you really want to delete this item?", MessageBoxButtons.YesNoCancel,
        MessageBoxIcon.Question, MessageBoxDefaultButton.Button1, "Removed");
    if (result1 == DialogResult.Yes)
    {
        try
        {
            Cursor.Current = Cursors.WaitCursor;
            PropertyBag bag = new PropertyBag();
            bag[0] = item1.ItemInfo.MyCollectionItemIdentifier;
            this.serviceProxy.DeleteCollectionItem(bag);
            base.ListView.Items.Remove(item1);
        }
        catch (Exception exception1)
        {
            base.DisplayErrorMessage(exception1, null);
            return;
        }
        finally
        {
            Cursor.Current = Cursors.Default;
        }
    }
}
```

Next, I walk you through the implementation of the `DeleteCollectionItem` method. This method first accesses the `MyCollectionItemListViewItem` list view item that represents the selected item:

```
MyCollectionItemListViewItem item1 = this.SelectedCollectionItem;
```

Here is the implementation of the `SelectedCollectionItem` property. Replace the declaration of the `SelectedCollectionItem` property in the `MyCollectionPage.cs` file with the code shown in this code listing:

```
private MyCollectionItemListViewItem SelectedCollectionItem
{
    get
    {
        if (base.ListView.SelectedItems.Count != 0)
            return (MyCollectionItemListViewItem)base.ListView.SelectedItems[0];

        return null;
    }
}
```

Now back to the implementation of the `DeleteCollectionItem` method. Next, this method launches a message box to double-check whether the user indeed wants to delete the selected collection item:

```
DialogResult result1 = base.ShowMessage(
    "Do you really want to delete this item?", MessageBoxButtons.YesNoCancel,
    MessageBoxIcon.Question, MessageBoxDefaultButton.Button1, "Removed");
```


Chapter 7: Extending the Integrated Graphical Management System

If the user confirms the deletion, the `DeleteCollectionItem` method takes these steps:

1. Creates a `PropertyBag`:

```
PropertyBag bag = new PropertyBag();
```

2. Adds the identifier of the collection item being deleted to the `PropertyBag`:

```
bag[0] = item1.ItemInfo.MyCollectionItemIdentifier;
```

3. Calls the `DeleteCollectionItem` method of the proxy, passing in the `PropertyBag` to delete the collection item from the underlying configuration file:

```
this.serviceProxy.DeleteCollectionItem(bag);
```

4. Removes the deleted collection item from the list of items displayed in the `MyCollectionPage` module list page:

```
base.ListView.Items.Remove(item1);
```

UpdateCollectionItem

Listing 7-63 presents the implementation of the `UpdateCollectionItem` method of the `MyCollectionPage` module list page. Replace the declaration of the `UpdateCollectionItem` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-63: The UpdateCollectionItem Method of MyCollectionPage Module List Page

```
private void UpdateCollectionItem()
{
    MyCollectionItemListViewItem item1 = this.SelectedCollectionItem;
    if (item1 != null)
    {
        MyCollectionItemInfo info1 = item1.ItemInfo;
        using (MyCollectionItemTaskForm form1 =
            new MyCollectionItemTaskForm(base.Module, this.serviceProxy,
                                         item1.ItemInfo.MyCollectionItemIdentifier,
                                         item1.ItemInfo.MyCollectionItemBoolProperty))
        {
            if ((base.ShowDialog(form1) == DialogResult.OK) && form1.HasChanges)
            {
                info1.MyCollectionItemIdentifier = form1.MyCollectionItemIdentifier;
                info1.MyCollectionItemBoolProperty = form1.MyCollectionItemBoolProperty;
                this.ReplaceItem(item1, info1);
            }
        }
    }
}
```

`UpdateCollectionItem` first accesses the `MyCollectionItemListViewItem` list view item that represents the selected collection item. Recall that the users must first select the item that they want to update:

```
MyCollectionItemListViewItem item1 = this.SelectedCollectionItem;
```

Chapter 7: Extending the Integrated Graphical Management System

Then, it launches the `MyCollectionItemTaskForm` task form to allow the user to update the values of the Boolean property and identifier of the collection item being updated:

```
MyCollectionItemTaskForm form1 =  
    new MyCollectionItemTaskForm(base.Module, this.serviceProxy,  
                                item1.ItemInfo.MyCollectionItemIdentifier,  
                                item1.ItemInfo.MyCollectionItemBoolProperty)
```

As you'll see later, after the user updates the values and clicks the OK button on the task form, the event handler for this button calls the `UpdateCollectionItem` method of the proxy to update the values of the corresponding collection item in the underlying configuration file. If the user has indeed changed the current values, `UpdateCollectionItem` takes these steps to replace the item in the list of displayed items in the `MyCollectionPage` module page:

```
info1.MyCollectionItemIdentifier = form1.MyCollectionItemIdentifier;  
info1.MyCollectionItemBoolProperty = form1.MyCollectionItemBoolProperty;  
this.ReplaceItem(item1, info1);
```

Here is the implementation of the `ReplaceItem` method. Replace the declaration of the `ReplaceItem` method in the `MyCollectionPage.cs` file with the code shown in this code listing:

```
private void ReplaceItem(MyCollectionItemListViewItem item,  
                        MyCollectionItemInfo itemInfo)  
{  
    base.ListView.Items.Remove(item);  
    this.AddItem(itemInfo, true);  
}
```

As you can see, the `ReplaceItem` method first removes the specified `MyCollectionItemListViewItem` list view item from the `Items` collection of the `ListView` list view that displays the collection items, and then calls the `AddItem` method to create and to add a new `MyCollectionItemListViewItem` list view item to the `Items` collection.

OnListViewBeforeLabelEdit

Recall from Listing 7-49 that the `InitializeListPage` method of the `MyCollectionPage` module list page registers the `OnListViewBeforeLabelEdit` method as an event handler for the `BeforeLabelEdit` event of the `ListView` list view that displays the list of available items. This `ListView` list view fires this event right after the user clicks the "Change identifier" link button to edit the label that displays the identifier of the selected item and right before the label becomes editable.

```
base.ListView.BeforeLabelEdit +=  
    new LabelEditEventHandler(this.OnListViewBeforeLabelEdit);
```

Listing 7-64 presents the implementation of the `OnListViewBeforeLabelEdit` method. Replace the declaration of the `OnListViewBeforeLabelEdit` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-64: The OnListViewBeforeLabelEdit Method of MyCollectionPage

```
private void OnListViewBeforeLabelEdit(object sender, LabelEditEventArgs e)
{
    MyCollectionItemListViewItem item =
        (MyCollectionItemListViewItem)base.ListView.Items[e.Item];

    // Use application-specific validation logic here to determine whether
    // label editing is allowed. If not, set the CancelEdit property of the
    // LabelEditEventArgs passed into the method to true to cancel label editing
    // e.CancelEdit = true;
}
```

As you can see, `OnListViewBeforeLabelEdit` first accesses the `MyCollectionItemListViewItem` list view item that represents the selected item in the list of displayed items:

```
MyCollectionItemListViewItem item =
    (MyCollectionItemListViewItem)base.ListView.Items[e.Item];
```

Next, you can use the `MyCollectionItemListViewItem` list view item to extract whatever information is needed about the selected list view item and use the appropriate validation logic to determine whether the end user should be allowed to perform the requested label editing operation. In this case, this label editing operation basically changes the identifier of the selected collection item. If the validation fails, you can set the value of the `CancelEdit` property of the `LabelEditEventArgs` object passed into the `OnListViewBeforeLabelEdit` method to cancel the label editing operation. To keep our discussions focused, we let the operation go through without further validation.

OnListViewAfterLabelEdit

Recall from Listing 7-49 that the `InitializeListPage` method of the `MyCollectionPage` module list page also registers the `OnListViewAfterLabelEdit` method as an event handler for the `AfterLabelEdit` event of the `ListView` control that displays the list of available collection items:

```
base.ListView.AfterLabelEdit +=
    new LabelEditEventHandler(this.OnListViewAfterLabelEdit);
```

Listing 7-65 presents the implementation of the `OnListViewAfterLabelEdit` method. Replace the declaration of the `OnListViewAfterLabelEdit` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-65: The OnListViewAfterLabelEdit Method

```
private void OnListViewAfterLabelEdit(object sender, LabelEditEventArgs e)
{
    bool flag = true;
    if (e.Label != null)
    {
        MyCollectionItemListViewItem item =
            (MyCollectionItemListViewItem)base.ListView.Items[e.Item];
        string oldIdentifier = item.ItemInfo.MyCollectionItemIdentifier;
        string newIdentifier = e.Label.Trim();
    }
}
```

Listing 7-65: (continued)

```
if ((oldIdentifier != newIdentifier) && (newIdentifier.Length != 0))
{
    bool flag2 = false;

    try
    {
        Cursor.Current = Cursors.WaitCursor;
        flag2 = this.serviceProxy.UpdateCollectionItemIdentifier(oldIdentifier,
                                                                newIdentifier);
    }

    catch (Exception exception)
    {
        string errorMessage;
        string errorText;
        if (string.Equals(
            ModuleServiceProxy.GetErrorInformation(exception, null,
                                                    out errorText, out errorMessage),
            "A collection item with the specified identifier already exists!",
            StringComparison.OrdinalIgnoreCase))
        {
            if (base.ShowMessage(errorText, MessageBoxButtons.YesNo,
                                MessageBoxIcon.Question,
                                MessageBoxDefaultButton.Button1) ==
                DialogResult.Yes)
            {
                base.BeginInvoke(new MethodInvocation(this.Refresh));
            }

            else
            {
                base.DisplayErrorMessage(exception, null);
            }
        }

        finally
        {
            Cursor.Current = Cursors.Default;
        }

        if (flag2)
        {
            item.ItemInfo.MyCollectionItemIdentifier = newIdentifier;
            flag = false;
        }
    }
}

e.CancelEdit = flag;
base.Update();
}
```

Next, I walk through the implementation of `OnListViewAfterLabelEdit`. This method uses an internal Boolean flag named `flag` that will determine whether the label editing operation should be canceled.

Chapter 7: Extending the Integrated Graphical Management System

Notice that this method initializes this flag to `true`. In other words, this method assumes the label editing operation should be canceled.

```
bool flag = true;
```

Next, the method accesses the `MyCollectionItemListViewItem` list view item that represents the item in the list of displayed items whose identifier is being edited:

```
MyCollectionItemListViewItem item =  
    (MyCollectionItemListViewItem)base.ListView.Items[e.Item];
```

Next, the method stores the current identifier of this item in a local variable:

```
string oldIdentifier = item.ItemInfo.MyCollectionItemIdentifier;
```

Then, it retrieves the new identifier of this item from the editable label and stores it in a local variable:

```
string newIdentifier = e.Label.Trim();
```

If the new identifier is different from the old identifier, it invokes the `UpdateCollectionItemIdentifier` method on the `MyConfigSectionModuleServiceProxy` proxy to change the identifier of the item in the underlying configuration file:

```
flag2 = this.serviceProxy.UpdateCollectionItemIdentifier(oldIdentifier,  
                                                         newIdentifier);
```

If the `UpdateCollectionItemIdentifier` method succeeds in changing the identifier of the item in the underlying configuration file, `OnListViewAfterLabelEditing` updates the `MyCollectionPage` module list page's user interface. First, it assigns the new identifier to the `MyCollectionItemIdentifier` property of the `ItemInfo` property of the `MyCollectionItemListViewItem` list view item that represents the item whose identifier is being edited:

```
item.ItemInfo.MyCollectionItemIdentifier = newIdentifier;
```

Second, it sets the Boolean flag to `false` and assigns this flag to the `CancelEdit` property of the `LabelEditEventArgs` event data object to specify that the label editing operation should go through:

```
flag = false;  
e.CancelEdit = flag;
```

Third, it calls the `Update` method to update the user interface of the `MyCollectionPage` module list page:

```
base.Update();
```

If the `UpdateCollectionItemIdentifier` method fails in changing the identifier of the item in the underlying configuration file, `OnListViewAfterLabelEditing` invokes the `GetErrorInformation` static method on the `ModuleServiceProxy` base class to determine whether the error was due to the fact that the underlying configuration file already contains an item with the same identifier. If so, it pops up a message asking the user whether to refresh the `MyCollectionPage` module list page with the latest list of items because the chances are that someone else had already added an item with the same name to the underlying configuration file and consequently the `MyCollectionPage` module list page

needs refreshing. If the user says yes, the method calls the `Refresh` method to refresh the module list page. The user may choose not to refresh because the user may have a good reason to believe that nothing had changed in the underlying configuration file.

```
if (string.Equals(
    ModuleServiceProxy.GetErrorInformation(exception, null,
                                           out errorText, out errorMessage),
    "An item with the specified identifier already exists!",
    StringComparison.OrdinalIgnoreCase))
{
    if (base.ShowMessage(errorText, MessageBoxButtons.YesNo,
                        MessageBoxIcon.Question,
                        MessageBoxDefaultButton.Button1) ==
        DialogResult.Yes)
        base.BeginInvoke(new MethodInvocation(this.Refresh));
}
```

If the error is not due to the fact that the underlying configuration file contains an item with the same identifier, the method simply displays the error message to the end user:

```
else
    base.DisplayErrorMessage(exception, null);
```

OnListViewDoubleClick

Recall from Listing 7-49 that the `InitializeListPage` method of the `MyCollectionPage` module list page registers the `OnListViewDoubleClick` method as an event handler for the `DoubleClick` event of the `ListView` control that displays the list of available items. This `ListView` control fires this event when the user double-clicks a displayed item to edit the item:

```
base.ListView.DoubleClick += new EventHandler(this.OnListViewDoubleClick);
```

Listing 7-66 presents the implementation of the `OnListViewDoubleClick` method. Replace the declaration of the `OnListViewDoubleClick` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-66: The `OnListViewDoubleClick` Method

```
private void OnListViewDoubleClick(object sender, EventArgs e)
{
    if ((this.SelectedCollectionItem != null) && !this.ReadOnly)
        this.UpdateCollectionItem();
}
```

`OnListViewDoubleClick` calls the `UpdateCollectionItem` method if both of the following two conditions are met:

- ❑ The user has indeed selected an item from the list of displayed items. This condition is bound to be met because double-clicking an item also selects the item.
- ❑ The `MyCollectionPage` module list page is editable. Recall that the `ReadOnly` property of this module list page reflects the value of the `isLocked` attribute on the `<myConfigSection>` configuration section.

OnListViewKeyUp

Recall from Listing 7-49 that the `InitializeListPage` method registers the `OnListViewKeyUp` method as an event handler for the `KeyUp` event of the `ListView` control that displays the list of available items:

```
base.ListView.KeyUp += new KeyEventHandler(this.OnListViewKeyUp);
```

Listing 7-67 presents the code for the `OnListViewKeyUp` method. Replace the declaration of the `OnListViewKeyUp` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-67: The `OnListViewKeyUp` Method of `MyCollectionPage`

```
private void OnListViewKeyUp(object sender, KeyEventArgs e)
{
    if ((this.SelectedCollectionItem != null) && (e.KeyData == Keys.Delete))
        this.DeleteCollectionItem();
}
```

This method calls the `DeleteCollectionItem` method if both of the following two conditions are met:

- ☐ The user has indeed selected an item to delete.
- ☐ The user has clicked the Delete button.

OnListViewSelectedIndexChanged

Recall from Listing 7-49 that `InitializeListPage` registers the `OnListViewSelectedIndexChanged` method as an event handler for the `SelectedIndexChanged` event of the `ListView` control that displays the list of available items:

```
base.ListView.SelectedIndexChanged +=
    new EventHandler(this.OnListViewSelectedIndexChanged);
```

Listing 7-68 presents the code for the `OnListViewSelectedIndexChanged` method. As you can see, this method simply calls the `Update` method to update the user interface of the `MyCollectionPage` module list page. Replace the declaration of the `OnListViewSelectedIndexChanged` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-68: The `OnListViewSelectedIndexChanged` Method of `MyCollectionPage`

```
private void OnListViewSelectedIndexChanged(object sender, EventArgs e)
{
    base.Update();
}
```

Grouping

Every module list page supports a combo box named `Group by` that contains the list of available grouping criteria. When the end user selects a grouping criterion from this combo box, the module list page groups the displayed items by the selected grouping criterion. The `ModuleListPage` base class comes

Chapter 7: Extending the Integrated Graphical Management System

with several grouping-related members that every module list page must implement to support grouping. Next, I discuss these grouping-related members and walk through the `MyCollectionPage` module list page's implementation of these members.

The `ModuleListPage` base class exposes a read-only property of type `ModuleListPageGrouping[]` named `Groupings` as defined in Listing 7-69. The `ModuleListPage` base class's implementation of this property returns `null`.

Listing 7-69: The `Groupings` Property of `ModuleListPage`

```
public virtual ModuleListPageGrouping[] Groupings
{
    get { return null; }
}
```

Every module list page that needs to support grouping must override the `Grouping` property of the `ModuleListPage` base class to instantiate and to return an array of `ModuleListPageGrouping` objects where each object represents a grouping criterion. The `ModuleListPage` base class uses the objects in this array to populate the Group by combo box. Listing 7-70 presents the internal implementation of the `ModuleListPageGrouping` class. As you can see, the constructor of this class takes two string parameters where the first parameter specifies a name for the grouping criterion and the second parameter specifies the text that will appear in the Group by combo box.

Listing 7-70: The `ModuleListPageGrouping` Class

```
public sealed class ModuleListPageGrouping
{
    private string name;
    private string text;

    public ModuleListPageGrouping(string name, string text)
    {
        if (string.IsNullOrEmpty(name))
            throw new ArgumentNullException("name");

        if (string.IsNullOrEmpty(text))
            throw new ArgumentNullException("text");

        this.name = name;
        this.text = text;
    }

    public override bool Equals(object obj)
    {
        ModuleListPageGrouping grouping = obj as ModuleListPageGrouping;
        if ((grouping != null) && string.Equals(grouping.Name, this.Name))
            return string.Equals(grouping.Text, this.Text);

        return false;
    }

    public override int GetHashCode()
```

(Continued)

Listing 7-70: *(continued)*

```
{
    return (this.Name.GetHashCode() + this.Text.GetHashCode());
}

public override string ToString()
{
    return this.Text;
}

public string Name
{
    get { return this.name; }
}

public string Text
{
    get { return this.text; }
}
}
```

As you can see from Listing 7-71, the `MyCollectionPage` module list page's implementation of the `Groupings` property returns an array that contains a single `ModuleListPageGrouping` object because this module list page only supports grouping by Boolean property value. Also note that Listing 7-71 instantiates two `ListViewGroup` objects to represent the two groups in the Boolean property grouping criterion and stores these two objects in private fields for future reference. As the name suggests, the `ListViewGroup` class is used to represent a group. Note that the constructor of this class takes a single argument that specifies the name of the group. In this case, you have two groups in the Boolean property grouping criterion. The first group contains collection items with a Boolean property value of `true`. The second group contains collection items with a Boolean property value of `false`.

Now replace the declaration of the `Groupings` property in the `MyCollectionPage.cs` file with the code shown in Listing 7-71.

Listing 7-71: The Groupings Property

```
public override ModuleListPageGrouping[] Groupings
{
    get
    {
        if (this.booleanPropertyGrouping == null)
        {
            this.booleanPropertyGrouping =
                new ModuleListPageGrouping("BooleanProperty", "Boolean Property");
            this.trueGroup = new ListViewGroup("True");
            this.falseGroup = new ListViewGroup("False");
        }
        return new ModuleListPageGrouping[] { this.booleanPropertyGrouping };
    }
}
```

Chapter 7: Extending the Integrated Graphical Management System

The `ModuleListPage` base class exposes a method named `Group` as shown in Listing 7-72. When the end user selects a grouping criterion from the Group by combo box, this method is automatically invoked and the `ModuleListPageGrouping` representing the selected grouping criterion is automatically passed into the method.

Listing 7-72: The Group Method of ModuleListPage

```
protected void Group(ModuleListPageGrouping grouping)
{
    ListViewGroup[] groups = null;
    if (grouping != null)
        groups = this.GetGroups(grouping);

    this.listView.BeginUpdate();

    if ((grouping == null) || grouping.Equals(EmptyGrouping) ||
        (groups == null) || (groups.Length == 0))
        this.listView.ShowGroups = false;
    else
    {
        this.listView.ShowGroups = true;
        for (int i = this.listView.Groups.Count - 1; i >= 0; i--)
        {
            this.listView.Groups.RemoveAt(i);
        }
        this.listView.Groups.AddRange(groups);
        this.OnGroup(grouping);
    }

    this.listView.EndUpdate();

    this.selectedGrouping = grouping;
    this.pageHeader.UpdateGroupingCommands();
}
```

As Listing 7-72 shows, the `Group` method of the `ModuleListPage` base class invokes another method named `GetGroups`, passing in the `ModuleListPageGrouping` object that represents the selected grouping criterion. Listing 7-73 presents the `ModuleListPage` base class's implementation of the `GetGroups` method. As you can see, the base implementation of this method returns `null`.

Listing 7-73: The GetGroups Method

```
protected virtual ListViewGroup[] GetGroups(ModuleListPageGrouping grouping)
{
    return null;
}
```

Every module list page that needs to support grouping must override the `GetGroups` method to instantiate and return an array of `ListViewGroup` objects where each object represents a particular group in the grouping criterion represented by the `ModuleListPageGrouping` object.

Chapter 7: Extending the Integrated Graphical Management System

As Listing 7-74 shows, the `MyCollectionPage` module list page overrides the `GetGroups` method to return an array of `ListViewGroup` objects where each object represents a particular Boolean property group. Because there are only two Boolean property groups, that is, `true` and `false`, the array that the `GetGroups` method returns contains only two `ListViewGroup` objects. Replace the declaration of the `GetGroups` method in the `MyCollectionPage.cs` file with the code shown in Listing 7-74.

Listing 7-74: The Overridden `GetGroups` Method

```
protected override ListViewGroup[] GetGroups(ModuleListPageGrouping grouping)
{
    if (grouping == this.booleanPropertyGrouping)
        return new ListViewGroup[] { this.trueGroup, this.falseGroup };

    return null;
}
```

As Listing 7-72 illustrates, the `Group` method of the `ModuleListPage` base class also invokes the `OnGroup` method passing in the `ModuleListPageGrouping` object that represents the selected grouping criterion. As you can see from Listing 7-75, the `ModuleListPage` base class's implementation of the `OnGroup` method simply raises an exception.

Listing 7-75: The `OnGroup` Method

```
protected virtual void OnGroup(ModuleListPageGrouping grouping)
{
    throw new InvalidOperationException();
}
```

Every module list page that needs to support grouping must override the `OnGroup` method to run the logic that performs the actual grouping of the displayed items by the selected grouping criterion. Listing 7-76 presents the `MyCollectionPage` module list page's implementation of the `OnGroup` method. Now go ahead and replace the declaration of the `OnGroup` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-76: The Overridden `OnGroup` Method

```
protected override void OnGroup(ModuleListPageGrouping grouping)
{
    if (grouping == this.booleanPropertyGrouping)
    {
        foreach (MyCollectionItemListViewItem item in base.ListView.Items)
        {
            this.SetItemGroup(item);
        }
    }
}
```

The `OnGroup` method iterates through the `MyCollectionItemListViewItem` list view items in the `Items` collection of the `ListView` control (recall that this control displays the items) and invokes the `SetItemGroup` method to place each item in its appropriate group. Listing 7-77 presents the `MyCollectionPage` module list page's implementation of the `SetItemGroup` method. Replace the

declaration of the `SetItemGroup` method in the `MyCollectionPage.cs` file with the code shown in this code listing.

Listing 7-77: The `SetItemGroup` Method

```
private void SetItemGroup(MyCollectionItemListViewItem item)
{
    if (base.SelectedGrouping == this.booleanPropertyGrouping)
    {
        if (item.ItemInfo.MyCollectionItemBoolProperty)
            item.Group = this.trueGroup;
        else
            item.Group = this.falseGroup;
    }
}
```

`MyCollectionItemListViewItem` inherits a property of type `ListViewGroup` from its base class. This property specifies the group in which the list view item is displayed. As you can see from Listing 7-77, the `SetItemGroup` method sets this property to `trueGroup` if the value of the Boolean property of the collection item that the list view item represents is `true`. Otherwise it sets this property to `falseGroup`. Thanks to the grouping infrastructure, specifying the `Group` property of a `MyCollectionItemListViewItem` list view item is all it takes to have the associated collection item displayed in the specified group.

Refreshing

As you can see from Listing 7-78, the `MyCollectionPage` module list page overrides the `Refresh` method of its base class, and invokes a private method named `GetCollectionItems` to retrieve the list of available items from the underlying configuration file and display them to the end user. Recall that the `Refresh` method is automatically invoked every time the user clicks the `Refresh` button in the header of the IIS7 Manager.

Listing 7-78: The `Refresh` Method

```
protected override void Refresh()
{
    this.GetCollectionItems();
}
```

As Listing 7-79 shows, the `MyCollectionPage` module list page also overrides the `CanRefresh` Boolean property of its base class to specify that the module list page is refreshable. Recall that only the `Refresh` methods of those module pages whose `CanRefresh` property return `true` will be invoked when the end user clicks the `Refresh` button in the header of the IIS7 Manager.

Listing 7-79: The `CanRefresh` Property

```
protected override bool CanRefresh
{
    get{return true;}
}
```

MyCollectionItemTaskForm

Listing 7-80 presents the implementation of the `MyCollectionItemTaskForm` task form. Add a new source file named `MyCollectionItemTaskForm.cs` to the `GraphicalManagement/Client` directory and add the code shown in this code listing to this source file.

Listing 7-80: The `MyCollectionItemTaskForm` Task Form

```
using Microsoft.Web.Management.Client.Win32;
using Microsoft.Web.Management.Server;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;
using System;

namespace MyNamespace.GraphicalManagement.Client
{
    internal sealed class MyCollectionItemTaskForm : TaskForm
    {
        private MyConfigSectionModuleServiceProxy serviceProxy;
        private bool inModificationMode;
        private string originalMyCollectionItemIdentifier;
        private TextBox myCollectionItemIdentifierTextBox;
        private CheckBox myCollectionItemBoolPropertyCheckBox;
        private bool hasChanges;
        private ManagementPanel contentPanel;
        private Label myCollectionItemIdentifierLabel;
        private string myCollectionItemIdentifier;
        private bool myCollectionItemBoolProperty;
        private bool canAccept;

        public MyCollectionItemTaskForm(IServiceProvider serviceProvider,
                                         MyConfigSectionModuleServiceProxy proxy)
            : this(serviceProvider, proxy, string.Empty, false) { }

        public MyCollectionItemTaskForm(IServiceProvider serviceProvider,
                                         MyConfigSectionModuleServiceProxy proxy,
                                         string myCollectionItemIdentifier,
                                         bool myCollectionItemBoolProperty)
            : base(serviceProvider)
        {
            serviceProxy = proxy;
            InitializeComponent();
            inModificationMode = !string.IsNullOrEmpty(myCollectionItemIdentifier);
            if (inModificationMode)
            {
                originalMyCollectionItemIdentifier = myCollectionItemIdentifier;
                myCollectionItemIdentifierTextBox.Text = myCollectionItemIdentifier;
                myCollectionItemBoolPropertyCheckBox.Checked =
                    myCollectionItemBoolProperty;
                Text = "Update collection item";
            }
            else
            {

```

Listing 7-80: (continued)

```
        Text = "Add collection item";

        UpdateUIState();
        hasChanges = false;
    }

    private void InitializeComponent()
    {
        contentPanel = new ManagementPanel();
        myCollectionItemIdentifierLabel = new Label();
        myCollectionItemIdentifierTextBox = new TextBox();
        myCollectionItemBoolPropertyCheckBox = new CheckBox();
        contentPanel.SuspendLayout();
        base.SuspendLayout();
        contentPanel.Controls.Add(myCollectionItemIdentifierLabel);
        contentPanel.Controls.Add(myCollectionItemIdentifierTextBox);
        contentPanel.Controls.Add(myCollectionItemBoolPropertyCheckBox);

        contentPanel.Dock = DockStyle.Fill;
        contentPanel.Location = new Point(0, 0);
        contentPanel.Name = "contentPanel";
        contentPanel.Size = new Size(0x114, 110);
        contentPanel.TabIndex = 0;
        myCollectionItemIdentifierLabel.Location = new Point(0, 0);
        myCollectionItemIdentifierLabel.Name = "_nameLabel";
        myCollectionItemIdentifierLabel.AutoSize = true;
        myCollectionItemIdentifierLabel.TabIndex = 0;
        myCollectionItemIdentifierLabel.TextAlign = ContentAlignment.MiddleLeft;
        myCollectionItemIdentifierLabel.Text = "Identifier";
        myCollectionItemIdentifierTextBox.Anchor =
            AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
        myCollectionItemIdentifierTextBox.Location = new Point(0, 0x10);
        myCollectionItemIdentifierTextBox.Name = "_nameTextBox";
        myCollectionItemIdentifierTextBox.Size = new Size(0x114, 0x15);
        myCollectionItemIdentifierTextBox.TabIndex = 1;
        myCollectionItemIdentifierTextBox.TextChanged +=
            new EventHandler(OnMyCollectionItemIdentifierTextBoxTextChanged);

        myCollectionItemBoolPropertyCheckBox.Location = new Point(0, 0x3b);
        myCollectionItemBoolPropertyCheckBox.Name = "_valueTextBox";
        myCollectionItemBoolPropertyCheckBox.Text = "Boolean Value";
        myCollectionItemBoolPropertyCheckBox.Size = new Size(0x114, 0x15);
        myCollectionItemBoolPropertyCheckBox.Anchor =
            AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
        myCollectionItemBoolPropertyCheckBox.CheckedChanged +=
            new EventHandler(OnMyCollectionItemBoolPropertyCheckBoxChanged);
        myCollectionItemBoolPropertyCheckBox.TabIndex = 3;
        base.ClientSize = new Size(300, 150);
        base.AutoScaleMode = AutoScaleMode.Font;
        base.Name = "MyCollectionItemTaskForm";
        contentPanel.ResumeLayout(false);
    }
}
```

(Continued)

Listing 7-80: *(continued)*

```
        contentPanel.PerformLayout();
        base.SetContent(contentPanel);
        base.ResumeLayout(false);
    }

    protected override void OnAccept()
    {
        myCollectionItemIdentifier = myCollectionItemIdentifierTextBox.Text.Trim();
        myCollectionItemBoolProperty = myCollectionItemBoolPropertyCheckBox.Checked;
        base.StartAsyncTask(new DoWorkEventHandler(OnWorkerDoWork),
                            new RunWorkerCompletedEventHandler(OnWorkerCompleted));
        base.UpdateTaskForm();
    }

    private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
    {
        if (this.hasChanges)
        {
            PropertyBag bag = new PropertyBag();
            if (!this.inModificationMode)
            {
                bag[0] = this.myCollectionItemIdentifier;
                bag[1] = (bool)this.myCollectionItemBoolProperty;

                this.serviceProxy.AddCollectionItem(bag);
            }

            else
            {
                bag[0] = this.originalMyCollectionItemIdentifier;
                bag[1] = this.myCollectionItemIdentifier;
                bag[2] = (bool)this.myCollectionItemBoolProperty;

                this.serviceProxy.UpdateCollectionItem(bag);
            }
        }
    }

    private void OnWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
    {
        base.UpdateTaskForm();
        if (e.Error != null)
            this.DisplayErrorMessage(e.Error, null);

        else
        {
            base.DialogResult = DialogResult.OK;
            base.Close();
        }
    }
}
```

Listing 7-80: *(continued)*

```
private void OnMyCollectionItemIdentifierTextBoxTextChanged(object sender,
                                                             EventArgs e)
{
    this.UpdateUIState();
}

private void OnMyCollectionItemBoolPropertyCheckBoxChanged(object sender,
                                                            EventArgs e)
{
    this.UpdateUIState();
}

private void UpdateUIState()
{
    this.hasChanges = true;
    this.canAccept =
        !string.IsNullOrEmpty(this.myCollectionItemIdentifierTextBox.Text);
    base.UpdateTaskForm();
}

protected override bool CanAccept
{
    get
    {
        if (!base.BackgroundJobRunning)
            return this.canAccept;

        return false;
    }
}

public string MyCollectionItemIdentifier
{
    get { return this.myCollectionItemIdentifier; }
}

public bool MyCollectionItemBoolProperty
{
    get { return this.myCollectionItemBoolProperty; }
}

public bool HasChanges
{
    get { return this.hasChanges; }
}
}
```

The following sections present and discuss the implementation of the members of the `MyCollectionItemTaskForm` task form.

Constructors

Listing 7-81 contains the implementation of the constructors of `MyCollectionItemTaskForm`.

Listing 7-81: The Constructors of `MyCollectionItemTaskForm`

```
public MyCollectionItemTaskForm(IServiceProvider serviceProvider,
                               MyConfigSectionModuleServiceProxy proxy)
    : this(serviceProvider, proxy, string.Empty, false) { }

public MyCollectionItemTaskForm(IServiceProvider serviceProvider,
                               MyConfigSectionModuleServiceProxy proxy,
                               string myCollectionItemIdentifier,
                               bool myCollectionItemBoolProperty)
    : base(serviceProvider)
{
    serviceProxy = proxy;
    InitializeComponent();
    inModificationMode = !string.IsNullOrEmpty(myCollectionItemIdentifier);
    if (inModificationMode)
    {
        originalMyCollectionItemIdentifier = myCollectionItemIdentifier;
        myCollectionItemIdentifierTextBox.Text = myCollectionItemIdentifier;
        myCollectionItemBoolPropertyCheckBox.Checked =
            myCollectionItemBoolProperty;
        Text = "Update collection item";
    }
    else
        Text = "Add collection item";

    UpdateUIState();
    hasChanges = false;
}
```

The first constructor delegates to the second constructor, which calls the `InitializeComponent` method to create the user interface of the `MyCollectionItemTaskForm` task form. Because the same task form is used for both updating and adding collection items, the value of the `myCollectionItemIdentifier` parameter is used to determine whether the user is trying to update or add a collection item. If the user is updating a collection item, the constructor initializes the user interface of the task form with the current values of the Boolean property and the identifier of the collection item being updated:

```
myCollectionItemIdentifierTextBox.Text = myCollectionItemIdentifier;
myCollectionItemBoolPropertyCheckBox.Checked = myCollectionItemBoolProperty;
```

InitializeComponent

The main responsibility of the `InitializeComponent` method is to create the user interface of the `MyCollectionItemTaskForm` task form (see Listing 7-82).

Listing 7-82: The InitializeComponent Method

```
private void InitializeComponent()
{
    contentPanel = new ManagementPanel();
    myCollectionItemIdentifierLabel = new Label();
    myCollectionItemIdentifierTextBox = new TextBox();
    myCollectionItemBoolPropertyCheckBox = new CheckBox();
    contentPanel.SuspendLayout();
    base.SuspendLayout();
    contentPanel.Controls.Add(myCollectionItemIdentifierLabel);
    contentPanel.Controls.Add(myCollectionItemIdentifierTextBox);
    contentPanel.Controls.Add(myCollectionItemBoolPropertyCheckBox);

    contentPanel.Dock = DockStyle.Fill;
    contentPanel.Location = new Point(0, 0);
    contentPanel.Name = "contentPanel";
    contentPanel.Size = new Size(0x114, 110);
    contentPanel.TabIndex = 0;
    myCollectionItemIdentifierLabel.Location = new Point(0, 0);
    myCollectionItemIdentifierLabel.Name = "_nameLabel";
    myCollectionItemIdentifierLabel.AutoSize = true;
    myCollectionItemIdentifierLabel.TabIndex = 0;
    myCollectionItemIdentifierLabel.TextAlign = ContentAlignment.MiddleLeft;
    myCollectionItemIdentifierLabel.Text = "Identifier";
    myCollectionItemIdentifierTextBox.Anchor =
        AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
    myCollectionItemIdentifierTextBox.Location = new Point(0, 0x10);
    myCollectionItemIdentifierTextBox.Name = "_nameTextBox";
    myCollectionItemIdentifierTextBox.Size = new Size(0x114, 0x15);
    myCollectionItemIdentifierTextBox.TabIndex = 1;
    myCollectionItemIdentifierTextBox.TextChanged +=
        new EventHandler(OnMyCollectionItemIdentifierTextBoxTextChanged);

    myCollectionItemBoolPropertyCheckBox.Location = new Point(0, 0x3b);
    myCollectionItemBoolPropertyCheckBox.Name = "_valueTextBox";
    myCollectionItemBoolPropertyCheckBox.Text = "Boolean Value";
    myCollectionItemBoolPropertyCheckBox.Size = new Size(0x114, 0x15);
    myCollectionItemBoolPropertyCheckBox.Anchor =
        AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
    myCollectionItemBoolPropertyCheckBox.CheckedChanged +=
        new EventHandler(OnMyCollectionItemBoolPropertyCheckBoxChanged);
    myCollectionItemBoolPropertyCheckBox.TabIndex = 3;
    base.ClientSize = new Size(300, 150);
    base.AutoScaleMode = AutoScaleMode.Font;
    base.Name = "MyCollectionItemTaskForm";
    contentPanel.ResumeLayout(false);
    contentPanel.PerformLayout();
    base.SetContent(contentPanel);
    base.ResumeLayout(false);
}
```

Chapter 7: Extending the Integrated Graphical Management System

`InitializeComponent` uses a `ManagementPanel` control as the container for the entire user interface of the task form. Figure 7-15 is an excerpt from Figure 7-1 that contains only the `ManagementPanel` hierarchy. As this figure shows, the `ManagementPanel` is a scrollable panel control.

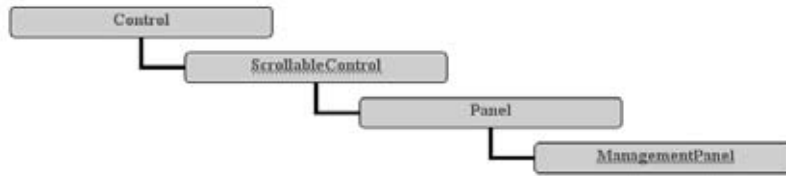


Figure 7-15

As Listing 7-82 shows, the `InitializeComponent` method first instantiates the `ManagementPanel` control and adds a label, textbox, and checkbox. The textbox and checkbox controls are used to display or specify the values of the identifier and Boolean property of the associated collection item:

```
contentPanel = new ManagementPanel();
myCollectionItemIdentifierLabel = new Label();
myCollectionItemIdentifierTextBox = new TextBox();
myCollectionItemBoolPropertyCheckBox = new CheckBox();
contentPanel.Controls.Add(myCollectionItemIdentifierLabel);
contentPanel.Controls.Add(myCollectionItemIdentifierTextBox);
contentPanel.Controls.Add(myCollectionItemBoolPropertyCheckBox);
```

Finally, `InitializeComponent` registers the `OnMyCollectionItemIdentifierTextBoxTextChanged` and `OnMyCollectionItemBoolPropertyCheckBoxChanged` methods as event handlers for the `TextChanged` event of the textbox and `CheckedChanged` event of the checkbox, respectively:

```
myCollectionItemIdentifierTextBox.TextChanged +=
    new EventHandler(OnMyCollectionItemIdentifierTextBoxTextChanged);

myCollectionItemBoolPropertyCheckBox.CheckedChanged +=
    new EventHandler(OnMyCollectionItemBoolPropertyCheckBoxChanged);
```

The following code listing presents the implementation of these two methods:

```
private void
OnMyCollectionItemIdentifierTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void
OnMyCollectionItemBoolPropertyCheckBoxChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}
```

As you can see, these two methods simply call the `UpdateUIState` method to update the user interface of the `MyCollectionItemTaskForm` task form.

OnAccept

Recall that `MyCollectionItemTaskForm` inherits from the `TaskForm` base class. This base class exposes an overridable method named `OnAccept` that its subclasses must override to add the code that they want to run when the user clicks the OK button of the task form. Listing 7-83 presents the `MyCollectionItemTaskForm` class's implementation of the `OnAccept` method.

Listing 7-83: The OnAccept Method

```
protected override void OnAccept()
{
    myCollectionItemIdentifier = myCollectionItemIdentifierTextBox.Text.Trim();
    myCollectionItemBoolProperty = myCollectionItemBoolPropertyCheckBox.Checked;
    base.StartAsyncTask(new DoWorkEventHandler (OnWorkerDoWork),
                       new RunWorkerCompletedEventHandler (OnWorkerCompleted));
    base.UpdateTaskForm();
}
```

`OnAccept` first retrieves the new values of the Boolean property and identifier of the associated collection item from the checkbox and textbox controls and stores them in the `myCollectionItemBoolProperty` and `myCollectionItemIdentifier` fields for future reference:

```
myCollectionItemIdentifier = myCollectionItemIdentifierTextBox.Text.Trim();
myCollectionItemBoolProperty = myCollectionItemBoolPropertyCheckBox.Checked;
```

It then calls the `StartAsyncTask` method, passing in `DoWorkEventHandler` and `RunWorkerCompletedEventHandler` delegates that respectively represent the `OnWorkerDoWork` and `OnWorkerCompleted` methods. These two delegates and the `StartAsyncTask` method were thoroughly discussed in the previous chapter.

OnWorkerDoWork

Listing 7-84 demonstrates the implementation of the `OnWorkerDoWork` method.

Listing 7-84: The OnWorkerDoWork Method

```
private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
{
    if (this.hasChanges)
    {
        PropertyBag bag = new PropertyBag();
        if (!this.inModificationMode)
        {
            bag[0] = this.myCollectionItemIdentifier;
            bag[1] = (bool)this.myCollectionItemBoolProperty;

            this.serviceProxy.AddCollectionItem(bag);
        }
    }
}
```

(Continued)

Listing 7-84: *(continued)*

```
    }

    else
    {
        bag[0] = this.originalMyCollectionItemIdentifier;
        bag[1] = this.myCollectionItemIdentifier;
        bag[2] = (bool)this.myCollectionItemBoolProperty;

        this.serviceProxy.UpdateCollectionItem(bag);
    }
}
```

This method first creates a `PropertyBag`, which will be used to transfer data to the back-end server:

```
PropertyBag bag = new PropertyBag();
```

It then checks whether the `MyCollectionItemTaskForm` task form is being used to add a new collection item. If so, it populates the `PropertyBag` with the values of the Boolean property and identifier of the collection item being added, and calls the `AddCollectionItem` method of the proxy, passing in the `PropertyBag` to add a new collection item with the specified values to the underlying configuration file:

```
bag[0] = this.myCollectionItemIdentifier;
bag[1] = (bool)this.myCollectionItemBoolProperty;
this.serviceProxy.AddCollectionItem(bag);
```

If the task form is being used to update an existing collection item, `OnWorkerDoWork` populates the `PropertyBag` with the values of the Boolean property, original identifier, and new identifier of the collection item being updated, and invokes the `UpdateCollectionItem` method of the proxy, passing in the `PropertyBag` to update the respective collection item in the configuration file. Notice that the `PropertyBag` passed into the `UpdateCollectionItem` must also contain the original identifier to allow the server-side `UpdateCollectionItem` method to identify the collection item being updated.

```
bag[0] = this.originalMyCollectionItemIdentifier;
bag[1] = this.myCollectionItemIdentifier;
bag[2] = (bool)this.myCollectionItemBoolProperty;
this.serviceProxy.UpdateCollectionItem(bag);
```

OnWorkerCompleted

Listing 7-85 contains the code for the `OnWorkerCompleted` method.

Listing 7-85: The `OnWorkerCompleted` Method

```
private void OnWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    base.UpdateTaskForm();
    if (e.Error != null)
        this.DisplayErrorMessage(e.Error, null);
}
```

Listing 7-85: *(continued)*

```
else
{
    base.DialogResult = DialogResult.OK;
    base.Close();
}
}
```

`OnWorkerCompleted` checks whether everything went fine. If an error has occurred, it displays the error message to the end user.

Module

The previous sections showed you how to implement the `MyConfigSectionPage` and `MyCollectionPage` module pages. Implementing your custom module page is just the first step. Next, you need to register your module page with the IIS7 Manager so it gets instantiated and called. The IIS7 Manager comes with a base class named `Module` that defines the API that you need to implement to register your custom module pages. Next, I discuss the `Module` base class.

Module

As Listing 7-86 shows, the `Module` base class exposes three important methods as follows:

- ❑ `Dispose`: Your custom module must override this method to perform final cleanup such as releasing the resources (if any) that your custom module is holding before it is disposed of.
- ❑ `GetService`: Your custom module should call this method to access a given service. You'll see an example of this in the next section.
- ❑ `Initialize`: You must override this method to register your custom module.

Listing 7-86: The Module Base Class

```
public abstract class Module : IServiceProvider, IDisposable
{
    protected virtual void Dispose()
    {
        this.serviceProvider = null;
        this.moduleInfo = null;
    }

    protected virtual object GetService(Type serviceType)
    {
        return this.serviceProvider.GetService(serviceType);
    }

    protected internal virtual void Initialize(IServiceProvider serviceProvider,
                                                ModuleInfo moduleInfo)
    {

```

(Continued)

Listing 7-86: *(continued)*

```
        this.serviceProvider = serviceProvider;
        this.moduleInfo = moduleInfo;
    }

    public ModuleInfo ModuleInfo
    {
        get { return this.moduleInfo; }
    }

    // Fields
    private ModuleInfo moduleInfo;
    private IServiceProvider serviceProvider;
}
```

MyConfigSectionModule

Listing 7-87 implements a custom module named `MyConfigSectionModule` that registers the `MyConfigSectionPage` and `MyCollectionPage` module pages with the IIS7 Manager. Add a new source file named `MyConfigSectionModule.cs` to the `GraphicalManagement/Client` directory and add the code shown in this code listing to this source file.

Listing 7-87: The MyConfigSectionModule Module

```
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System;

namespace MyNamespace.GraphicalManagement.Client
{
    class MyConfigSectionModule : Module
    {
        protected override void Initialize(IServiceProvider serviceProvider,
                                           ModuleInfo moduleInfo)
        {
            base.Initialize(serviceProvider, moduleInfo);
            IControlPanel panel1 = (IControlPanel)GetService(typeof(IControlPanel));
            ModulePageInfo info1 =
                new ModulePageInfo(this, typeof(MyConfigSectionPage),
                                   "MyConfigSection", "Displays MyConfigSection page");
            panel1.RegisterPage(info1);
            ModulePageInfo info2 =
                new ModulePageInfo(this, typeof(MyCollectionPage),
                                   "Collection page", "Collection page");
            panel1.RegisterPage(info2);
        }
    }
}
```

`MyConfigSectionModule` overrides the `Initialize` method of the `Module` base class. It first calls the `Initialize` method of the base class to allow the base class to do its own initialization as shown in Listing 7-87. Next, it calls the `GetService` method of the `Module` base class, passing in the `Type` object that represents the `IControlPanel` interface to access the control panel service. This service exposes a method named `RegisterPage` that you can use to register your module page.

The `RegisterPage` method takes the `ModulePageInfo` object that represents the module page being registered. The `ModulePageInfo` object encapsulates the complete information about the module page that it represents and exposes this information through its properties as thoroughly discussed in Chapter 6.

As Listing 7-87 shows, the `Initialize` method instantiates the `ModulePageInfo` object that represents the `MyConfigSectionPage` module page, and passes this object into the `RegisterPage` method of the control panel service to register the page:

```
ModulePageInfo info1 =  
    new ModulePageInfo(this, typeof(MyConfigSectionPage),  
                        "MyConfigSection", "Displays MyConfigSection page");  
panel1.RegisterPage(info1);
```

`Initialize` does the same thing to register the `MyCollectionPage` module page:

```
ModulePageInfo info2 =  
    new ModulePageInfo(this, typeof(MyCollectionPage),  
                        "Collection page", "Collection page");  
panel1.RegisterPage(info2);
```

Server-Side Managed Code

As mentioned earlier, extending the IIS7 Manager requires writing two sets of code: client- and server-side code. So far, we've only covered the client-side code. This section shows you how to implement the necessary server-side code to enable the back-end server to communicate with your module pages, task forms, wizard forms, and so on.

Take these steps to write the server-side code:

1. Implement a custom module service.
2. Implement a custom module provider to register your custom module and custom module service. Recall that a custom module is a class that inherits the `Module` base class and is used to register a custom module page as discussed in the previous section.
3. Compile your custom module provider into a strongly-named assembly and add the assembly to the Global Assembly Cache (GAC).
4. Register your custom module provider with the `administration.config` file located in the following directory on your machine:

```
%windir%\system32\inetsrv\config
```


Module Service

A custom module service is a class that inherits from the `ModuleService` base class and exposes methods that interact with the underlying configuration file to retrieve, add, delete, or update configuration settings. Listing 7-88 presents the members of a custom module service named `MyConfigSectionModuleService`, which like any other module service extends the `ModuleService` base class. Add a new source file named `MyConfigSectionModuleService.cs` to the `Server` subdirectory of the `GraphicalManagement` directory of the `MyConfigSection` project and add the code shown in Listing 7-88 to this source file.

Listing 7-88: The `MyConfigSectionModuleService` Server-Side Class

```
using Microsoft.Web.Management.Server;
using System.Collections;
using System;
using MyNamespace.ImperativeManagement;

namespace MyNamespace.GraphicalManagement.Server
{
    class MyConfigSectionModuleService : ModuleService
    {
        private MyConfigSection GetMyConfigSectionSection()
        {
            if (base.ManagementUnit.Configuration != null)
            {
                MyConfigSection section1 =
                    (MyConfigSection)base.ManagementUnit.Configuration.GetSection(
                                                                    "system.webServer/myConfigSection",
                                                                    typeof(MyConfigSection));

                if (section1 == null)
                    base.RaiseException("MyConfigSectionConfigurationError");

                return section1;
            }

            base.RaiseException("MyConfigSectionConfigurationError");
            return null;
        }

        [ModuleServiceMethod]
        public PropertyBag GetSettings()
        {
            PropertyBag bag1 = new PropertyBag();
            MyConfigSection section1 = this.GetMyConfigSectionSection();
            bag1[0] = section1.MyConfigSectionBoolProperty;
            bag1[1] = (int)section1.MyConfigSectionEnumProperty;
            bag1[2] = (TimeSpan)section1.MyNonCollection.MyNonCollectionTimeSpanProperty;
            bag1[3] = section1.IsLocked;
            return bag1;
        }

        [ModuleServiceMethod]
        public void UpdateSettings(PropertyBag updatedSettings)
        {
            if (updatedSettings == null)
            {
            }
        }
    }
}
```

Listing 7-88: (continued)

```
        throw new ArgumentNullException("updatedSettings");

        MyConfigSection section1 = this.GetMyConfigSectionSection();
        IEnumerator enumerator1 = updatedSettings.ModifiedKeys.GetEnumerator();
        try
        {
            while (enumerator1.MoveNext())
            {
                switch (((int)enumerator1.Current))
                {
                    case 0:
                        section1.MyConfigSectionBoolProperty = (bool)updatedSettings[0];
                        break;
                    case 1:
                        section1.MyConfigSectionEnumProperty =
                            (MyConfigSectionEnum)updatedSettings[1];
                        break;
                    case 2:
                        section1.MyNonCollection.MyNonCollectionTimeSpanProperty =
                            (TimeSpan)updatedSettings[2];
                        break;
                    case 3:
                        section1.MyCollection.MyCollectionIntProperty =
                            (int)updatedSettings[3];
                        break;
                }
            }
        }
        finally
        {
            IDisposable disposable1 = enumerator1 as IDisposable;
            if (disposable1 != null)
                disposable1.Dispose();
        }
        base.ManagementUnit.Update();
    }

    [ModuleServiceMethod]
    public PropertyBag GetCollectionItems()
    {
        MyConfigSection section1 = this.GetMyConfigSectionSection();
        ArrayList list = new ArrayList();
        PropertyBag bag;
        foreach (MyCollectionItem item in section1.MyCollection)
        {
            bag = new PropertyBag();
            bag[0] = item.MyCollectionItemIdentifier;
            bag[1] = item.MyCollectionItemBoolProperty;
            list.Add(bag);
        }
    }
}
```

(Continued)

Listing 7-88: (continued)

```
PropertyBag bag2 = new PropertyBag();
bag2[0] = list;
bag2[1] = section1.IsLocked;
return bag2;
}

[ModuleServiceMethod]
public void AddCollectionItem(PropertyBag bag)
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    string myCollectionItemIdentifier = (string)bag[0];
    bool myCollectionItemBoolValue = (bool)bag[1];

    if (ItemExists(section1.MyCollection, myCollectionItemIdentifier))
        base.RaiseException(
            "An item with the specified identifier already exists!");

    section1.MyCollection.Add(myCollectionItemIdentifier,
                             myCollectionItemBoolValue);
    base.ManagementUnit.Update();
}

[ModuleServiceMethod]
public void DeleteCollectionItem(PropertyBag bag)
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    string myCollectionItemIdentifier = (string)bag[0];
    if (!ItemExists(section1.MyCollection, myCollectionItemIdentifier))
        base.RaiseException(
            "The item with the specified identifier does not exist!");

    section1.MyCollection.Remove(
        section1.MyCollection[myCollectionItemIdentifier]);
    base.ManagementUnit.Update();
}

[ModuleServiceMethod]
public void UpdateCollectionItem(PropertyBag bag)
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    string oldIdentifier = (string)bag[0];
    string newIdentifier = (string)bag[1];
    bool boolValue = (bool)bag[2];

    if (!this.ItemExists(section1.MyCollection, oldIdentifier))
        base.RaiseException("Item with the specified identifier does not exist!");

    if (this.ItemExists(section1.MyCollection, newIdentifier))
        base.RaiseException(
            "An item with the specified identifier already exists!");
}
```

Listing 7-88: (continued)

```
MyCollectionItem item = section1.MyCollection[oldIdentifier];
item.MyCollectionItemIdentifier = newIdentifier;
item.MyCollectionItemBoolProperty = boolValue;
base.ManagementUnit.Update();
}

[ModuleServiceMethod(PassThrough = true)]
public bool UpdateCollectionItemIdentifier(string oldIdentifier,
                                         string newIdentifier)
{
    if (string.IsNullOrEmpty(oldIdentifier))
        throw new ArgumentNullException("Old identifier is required!");

    if (string.IsNullOrEmpty(newIdentifier))
        throw new ArgumentNullException("New identifier is required!");

    MyConfigSection section1 = this.GetMyConfigSectionSection();

    if (!this.ItemExists(section1.MyCollection, oldIdentifier))
        base.RaiseException("Item with the specified identifier does not exist!");

    if (this.ItemExists(section1.MyCollection, newIdentifier))
        base.RaiseException(
            "An item with the specified identifier already exists!");

    MyCollectionItem item = section1.MyCollection[oldIdentifier];
    item.MyCollectionItemIdentifier = newIdentifier;
    base.ManagementUnit.Update();
    return true;
}

private bool ItemExists(MyCollection collection, string identifier)
{
    if (collection != null)
    {
        for (int i = 0; i < collection.Count; i++)
        {
            if (string.Equals(collection[i].MyCollectionItemIdentifier, identifier,
                              StringComparison.OrdinalIgnoreCase))
                return true;
        }
    }
    return false;
}
}
```

Note that all methods of the `MyConfigSectionModuleService` module service are marked with the `ModuleServiceMethodAttribute` metadata attribute, except for the `GetMyConfigSectionSection` method. Only those methods of a custom module service marked with this metadata attribute are accessible from the proxy. The following sections present and discuss the implementation of the methods of the `MyConfigSectionModuleService` module service.

GetMyConfigSectionSection

The `GetMyConfigSectionSection` method's main responsibility is to return the `MyConfigSection` object that represents the `<myConfigSection>` configuration section (see Listing 7-89).

Listing 7-89: The `GetMyConfigSectionSection` Method

```
private MyConfigSection GetMyConfigSectionSection()
{
    if (base.ManagementUnit.Configuration != null)
    {
        MyConfigSection section1 =
            (MyConfigSection)base.ManagementUnit.Configuration.GetSection(
                "system.webServer/myConfigSection", typeof(MyConfigSection));
        if (section1 == null)
            base.RaiseException("MyConfigSectionConfigurationError");

        return section1;
    }

    base.RaiseException("MyConfigSectionConfigurationError");
    return null;
}
```

The `ModuleService` base class exposes an important property of type `ManagementUnit` named `ManagementUnit`, which encapsulates the logic that determines the configuration hierarchy level at which the current user is working, and the configuration file from which the configuration settings are read and into which the configuration settings are stored. Therefore, the `GetMyConfigSectionSection` method doesn't need to worry about what the current configuration hierarchy level and configuration file are.

The `ManagementUnit` class exposes a property of type `ManagementConfiguration` named `Configuration` that features a method named `GetSection`. As the name implies, this method returns the `ConfigurationSection` object that represents a configuration section with the specified name and type. The `GetSection` method takes two arguments. The first argument is the fully qualified name of the configuration section being accessed, including its complete group hierarchy. The second argument is the `Type` object that represents the type of the class that represents the configuration section. The `GetSection` method under the hood uses .NET reflection and this `Type` object to dynamically instantiate an instance of the specified configuration section class and populates it with associated configuration settings.

In our case, the `MyConfigSection` class represents the `<myConfigSection>` configuration section, which means that the `GetSection` method will automatically return an instance of this class populated with the required configuration settings:

```
MyConfigSection section1 =
    (MyConfigSection)base.ManagementUnit.Configuration.GetSection(
        "system.webServer/myConfigSection",
        typeof(MyConfigSection));
```

GetSettings

Listing 7-90 contains the implementation of the `GetSettings` method.

Listing 7-90: The `GetSettings` Method

```
[ModuleServiceMethod]
public PropertyBag GetSettings()
{
    PropertyBag bag1 = new PropertyBag();
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    bag1[0] = section1.MyConfigSectionBoolProperty;
    bag1[1] = (int)section1.MyConfigSectionEnumProperty;
    bag1[2] = (TimeSpan)section1.MyNonCollection.MyNonCollectionTimeSpanProperty;
    bag1[3] = section1.IsLocked;
    return bag1;
}
```

`GetSettings` first creates a `PropertyBag` object:

```
PropertyBag bag1 = new PropertyBag();
```

Next, it calls the `GetMyConfigSectionSection` method discussed in the previous section to return the `MyConfigSection` object that represents the `<myConfigSection>` configuration section:

```
MyConfigSection section1 = this.GetMyConfigSectionSection();
```

As discussed in the previous chapters, the `MyConfigSection` class exposes the configuration settings of the `<myConfigSection>` configuration section as strongly-typed properties. `GetSettings` stores the values of these properties in the `PropertyBag` object and returns the object to the client:

```
bag1[0] = section1.MyConfigSectionBoolProperty;
bag1[1] = (int)section1.MyConfigSectionEnumProperty;
bag1[2] = (TimeSpan)section1.MyNonCollection.MyNonCollectionTimeSpanProperty;
bag1[3] = section1.IsLocked;
```

UpdateSettings

Listing 7-91 presents the code for the `UpdateSettings` method.

Listing 7-91: The `UpdateSettings` Method

```
[ModuleServiceMethod]
public void UpdateSettings(PropertyBag updatedSettings)
{
    if (updatedSettings == null)
        throw new ArgumentNullException("updatedSettings");

    MyConfigSection section1 = this.GetMyConfigSectionSection();
    IEnumerator enumerator1 = updatedSettings.ModifiedKeys.GetEnumerator();
    try
    {
        while (enumerator1.MoveNext())
```

(Continued)

Listing 7-91: (continued)

```
{
    switch (((int)enumerator1.Current))
    {
        case 0:
            section1.MyConfigSectionBoolProperty = (bool)updatedSettings[0];
            break;
        case 1:
            section1.MyConfigSectionEnumProperty =
                (MyConfigSectionEnum)updatedSettings[1];
            break;
        case 2:
            section1.MyNonCollection.MyNonCollectionTimeSpanProperty =
                (TimeSpan)updatedSettings[2];
            break;
        case 3:
            section1.MyCollection.MyCollectionIntProperty = (int)updatedSettings[3];
            break;
    }
}

finally
{
    IDisposable disposable1 = enumerator1 as IDisposable;
    if (disposable1 != null)
        disposable1.Dispose();
}

base.ManagementUnit.Update();
}
```

UpdateSettings first calls the `GetMyConfigSectionSection` method to access the `MyConfigSection` object that contains the configuration settings of the `<myConfigSection>` configuration section:

```
MyConfigSection section1 = this.GetMyConfigSectionSection();
```

Next, `UpdateSettings` calls the `GetEnumerator` method of the `ModifiedKeys` property of the `PropertyBag` object that contains the updated configuration settings. Recall that the `PropertyBag` class exposes a property of type `IDictionary` named `ModifiedKeys`, which contains those indexes of the `PropertyBag` whose associated values have changed. The `GetEnumerator` method returns an `IEnumerator` object that you can use to iterate through these indexes in a generic fashion:

```
IEnumerator enumerator1 = updatedSettings.ModifiedKeys.GetEnumerator();
```

`UpdateSettings` iterates through these indexes, retrieves the associated values from the `PropertyBag`, and assigns them to the respective properties of the `MyConfigSection` object. Finally, `UpdateSettings` calls the `Update` method of the `ManagementUnit` to commit the change to the underlying configuration file.

GetCollectionItems

The main responsibility of the `GetCollectionItems` method is to retrieve the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of all collection items and return them to the client (see Listing 7-92).

Listing 7-92: The `GetCollectionItems` Method

```
[ModuleServiceMethod]
public PropertyBag GetCollectionItems()
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    ArrayList list = new ArrayList();
    PropertyBag bag;
    foreach (MyCollectionItem item in section1.MyCollection)
    {
        bag = new PropertyBag();
        bag[0] = item.MyCollectionItemIdentifier;
        bag[1] = item.MyCollectionItemBoolProperty;
        list.Add(bag);
    }

    PropertyBag bag2 = new PropertyBag();
    bag2[0] = list;
    bag2[1] = section1.IsLocked;
    return bag2;
}
```

The first order of business is to access the `MyConfigSection` object that provides programmatic access to the `<myConfigSection>` configuration section:

```
MyConfigSection section1 = this.GetMyConfigSectionSection();
```

Next, `GetCollectionItems` creates an `ArrayList`:

```
ArrayList list = new ArrayList();
```

Recall from the previous chapters that the `MyConfigSection` class exposes a collection property named `MyCollection` that contains one `MyCollectionItem` object for each collection item. `GetCollectionItems` iterates through these `MyCollectionItem` objects and takes the following actions for each enumerated object:

1. Creates a `PropertyBag`:

```
bag = new PropertyBag();
```

2. Populates the `PropertyBag` with the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes of the collection item. Recall from the previous chapters that the `MyCollectionItem` class exposes these attributes as strongly-typed properties named `MyCollectionItemIdentifier` and `MyCollectionItemBoolProperty`:

```
bag[0] = item.MyCollectionItemIdentifier;
bag[1] = item.MyCollectionItemBoolProperty;
```


Chapter 7: Extending the Integrated Graphical Management System

3. Adds the PropertyBag object to the ArrayList:

```
list.Add(bag);
```

Next, `GetCollectionItems` creates a `PropertyBag` and stores the `ArrayList` into it:

```
PropertyBag bag2 = new PropertyBag();  
bag2[0] = list;
```

Finally, it stores the value of the `isLocked` attribute of the `<myConfigSection>` section into this `PropertyBag` and returns the `PropertyBag` to its caller:

```
bag2[1] = section1.IsLocked;  
return bag2;
```

AddCollectionItem

The `AddCollectionItem` method adds a new collection item with the specified `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attribute values (see Listing 7-93).

Listing 7-93: The AddCollectionItem Method

```
[ModuleServiceMethod]  
public void AddCollectionItem(PropertyBag bag)  
{  
    MyConfigSection section1 = this.GetMyConfigSectionSection();  
    string myCollectionItemIdentifier = (string)bag[0];  
    bool myCollectionItemBoolValue = (bool)bag[1];  
  
    if (ItemExists(section1.MyCollection, myCollectionItemIdentifier))  
        base.RaiseException("An item with the specified identifier already exists!");  
  
    section1.MyCollection.Add(myCollectionItemIdentifier, myCollectionItemBoolValue);  
    base.ManagementUnit.Update();  
}
```

`AddCollectionItem` first accesses the `MyConfigSection` object as usual:

```
MyConfigSection section1 = this.GetMyConfigSectionSection();
```

Next, it retrieves the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes from the `PropertyBag` that it has received from the client:

```
string myCollectionItemIdentifier = (string)bag[0];  
bool myCollectionItemBoolValue = (bool)bag[1];
```

Then, it invokes another method named `ItemExists` to determine whether the `MyCollection` collection property of the `MyConfigSection` object contains an item with a specified identifier. If so, it raises an exception:

```
if (ItemExists(section1.MyCollection, myCollectionItemIdentifier))  
    base.RaiseException("An item with the specified identifier already exists!");
```

Chapter 7: Extending the Integrated Graphical Management System

Next, it passes the values of the `myCollectionItemBoolAttr` and `myCollectionItemIdentifier` attributes into the `Add` method of the `MyCollection` property of the `MyConfigSection` object. Recall from the previous chapters that the `Add` method of the `MyCollection` class creates a new `MyCollectionItem` object with the specified `MyCollectionItemBoolProperty` and `MyCollectionItemIdentifier` properties, and adds this object to the `MyCollection` collection:

```
section1.MyCollection.Add(myCollectionItemIdentifier, myCollectionItemBoolValue);
```

Finally, `AddCollectionItem` calls the `Update` method of the `ManagementUnit` to commit the changes to the underlying configuration file:

```
base.ManagementUnit.Update();
```

DeleteCollectionItem

As Listing 7-94 shows, `DeleteCollectionItem` retrieves the value of the `myCollectionItemIdentifier` attribute of the collection item being deleted from the `PropertyBag` that it received from the client, and passes that value into the `Remove` method.

Listing 7-94: The DeleteCollectionItem Method

```
[ModuleServiceMethod]
public void DeleteCollectionItem(PropertyBag bag)
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    string myCollectionItemIdentifier = (string)bag[0];
    if (!ItemExists(section1.MyCollection, myCollectionItemIdentifier))
        base.RaiseException("The item with the specified identifier does not exist!");

    section1.MyCollection.Remove(section1.MyCollection[myCollectionItemIdentifier]);
    base.ManagementUnit.Update();
}
```

UpdateCollectionItem

Listing 7-95 presents the implementation of the `UpdateCollectionItem` method.

Listing 7-95: The UpdateCollectionItem Method

```
[ModuleServiceMethod]
public void UpdateCollectionItem(PropertyBag bag)
{
    MyConfigSection section1 = this.GetMyConfigSectionSection();
    string oldIdentifier = (string)bag[0];
    string newIdentifier = (string)bag[1];
    bool boolValue = (bool)bag[2];

    if (!this.ItemExists(section1.MyCollection, oldIdentifier))
        base.RaiseException("Item with the specified identifier does not exist!");

    if (this.ItemExists(section1.MyCollection, newIdentifier))
        base.RaiseException("An item with the specified identifier already exists!");
}
```

(Continued)

Chapter 7: Extending the Integrated Graphical Management System

Listing 7-95: *(continued)*

```
MyCollectionItem item = section1.MyCollection[oldIdentifier];
item.MyCollectionItemIdentifier = newIdentifier;
item.MyCollectionItemBoolProperty = boolValue;
base.ManagementUnit.Update();
}
```

UpdateCollectionItem accesses the MyConfigSection object as usual:

```
MyConfigSection section1 = this.GetMyConfigSectionSection();
```

Next, it retrieves the old and new identifiers and the Boolean value of the collection item being updated from the PropertyBag that it has received from the client:

```
string oldIdentifier = (string)bag[0];
string newIdentifier = (string)bag[1];
bool boolValue = (bool)bag[2];
```

Next, it raises an exception if the MyCollection collection of the MyConfigSection object does not contain a MyCollectionItem with the old identifier:

```
if (!this.ItemExists(section1.MyCollection, oldIdentifier))
    base.RaiseException("Item with the specified identifier does not exist!");
```

Then, it raises an exception if the MyCollection collection already contains a MyCollectionItem with the new identifier:

```
if (this.ItemExists(section1.MyCollection, newIdentifier))
    base.RaiseException("An item with the specified identifier already exists!");
```

Next, it uses the old identifier as an index into the MyCollection collection to return a reference to the MyCollectionItem with the old identifier. This MyCollectionItem object represents the collection item being updated:

```
MyCollectionItem item = section1.MyCollection[oldIdentifier];
```

Then, it assigns the new identifier and Boolean value to the MyCollectionItemIdentifier and MyCollectionItemBoolProperty properties of this MyCollectionItem object. Recall that these two properties map to the myCollectionItemIdentifier and myCollectionItemBoolAttr attributes of the collection item being updated:

```
item.MyCollectionItemIdentifier = newIdentifier;
item.MyCollectionItemBoolProperty = boolValue;
```

Finally, it invokes the Update method to commit the changes to the underlying configuration file:

```
base.ManagementUnit.Update();
```

Module Provider

The previous sections showed you how to implement a custom module service and a custom module. Recall that the proxy invokes the methods of the custom module service to interact with the underlying configuration file. The custom module, on the other hand, is used to register a custom module page with the IIS7 Manager. This raises the following question: Who registers the custom module service and custom module? For example, in our case, who registers the `MyConfigSectionModuleService` custom module service and the `MyConfigSectionModule` module?

The answer is a direct or indirect subclass of the `ModuleProvider` base class. In other words, you have to write a custom module provider that directly or indirectly inherits from the `ModuleProvider` base class to register your custom module service and custom module. This custom module provider in our case is a class named `MyConfigSectionModuleProvider`.

The main responsibility of the `MyConfigSectionModuleProvider` custom module provider is to register the `MyConfigSectionModuleService` and `MyConfigSectionModule` as shown in Listing 7-96. `MyConfigSectionModuleProvider` inherits from `ConfigurationModuleProvider`, which is a subclass of the `ModuleProvider` base class. As mentioned, you can derive your custom module provider from any subclass of the `ModuleProvider` base class.

Listing 7-96: The `MyConfigSectionModuleProvider` Custom Module Provider

```
using Microsoft.Web.Management.Server;
using System;

namespace MyNamespace.GraphicalManagement.Server
{
    class MyConfigSectionModuleProvider : ConfigurationModuleProvider
    {
        public override ModuleDefinition GetModuleDefinition(
            IManagementContext context)
        {
            return new ModuleDefinition(base.Name,
                typeof(Client.MyConfigSectionModule).AssemblyQualifiedName);
        }

        public override bool SupportsScope(ManagementScope scope)
        {
            return true;
        }

        protected sealed override string ConfigurationSectionName
        {
            get { return "system.webServer/myConfigSection"; }
        }

        public override string FriendlyName
        {
            get { return "myConfigSection"; }
        }
    }
}
```

(Continued)

Listing 7-96: (continued)

```
public override Type ServiceType
{
    get { return typeof(MyConfigSectionModuleService); }
}
}
```

`MyConfigSectionModuleProvider` overrides the following members of the `ConfigurationModuleProvider` class:

- ❑ **GetModuleDefinition:** The IIS7 and ASP.NET 3.5 integrated infrastructure comes with a class named `ModuleDefinition`. As the name implies, this class represents or defines a module. Your custom module provider's implementation of the `GetModuleDefinition` method must call the `ModuleDefinition` constructor, passing in the following parameters:
 - ❑ The name of the custom module provider, for example, "MyConfigSectionModuleProvider."
 - ❑ The assembly qualified name of the type of the custom module being registered, which consists of five different parts: the fully qualified name of the type of the custom module provider including its namespace containment hierarchy (for example, `Client.MyConfigSectionModule`), assembly name, assembly version, assembly culture, and assembly public key token. The `Type` class exposes a method named `AssemblyQualifiedName` that returns this five-part information.

```
return new ModuleDefinition(base.Name,
                           typeof(Client.MyConfigSectionModule).AssemblyQualifiedName);
```

- ❑ As you can see, you register your custom module (for example, `MyConfigSectionModule`) by overriding the `GetModuleDefinition` method as just described.
- ❑ **SupportsScope:** This property determines the supported configuration hierarchy level. Listing 7-96 returns `true` to signal that it supports all levels.
- ❑ **ConfigurationSectionName:** This property specifies the fully qualified name of configuration section including its complete group hierarchy. Listing 7-96 returns "system.webServer/myConfigSection".
- ❑ **ServiceType:** This property returns the `Type` object that represents the custom module service being registered. Listing 7-96 returns `typeof(MyConfigSectionModuleService)`. As you can see, you register your custom module service (for example, `MyConfigSectionModuleService`) by overriding the `ServiceType` property as just described.

Deployment

The last two steps of our recipe for implementing the server-side code are as follows:

- ❑ Compile the `MyConfigSectionModuleProvider` custom module provider into a strongly-named assembly and load this assembly into the GAC. This is necessary because the IIS7 picks up assemblies from the GAC. It won't pick up your assembly if it's anywhere other than the GAC.
- ❑ Register the `MyConfigSectionModuleProvider` custom module provider with the `administration.config` file.

Because only strongly-named assemblies can be added to the GAC, you need to compile `MyConfigSectionModuleProvider` into a strongly-named assembly. Next, I show you how to configure Visual Studio to generate a strongly-named assembly. Right-click the `MyConfigSection` project in the Solution Explorer panel and select the Properties option from the popup menu to launch the Properties dialog. Switch to the Signing tab in this dialog, toggle on the “Sign the assembly” checkbox, open the “Choose a strong name key file” combo box, and select `New...` as shown in Figure 7-16. This will launch the dialog shown in Figure 7-17.

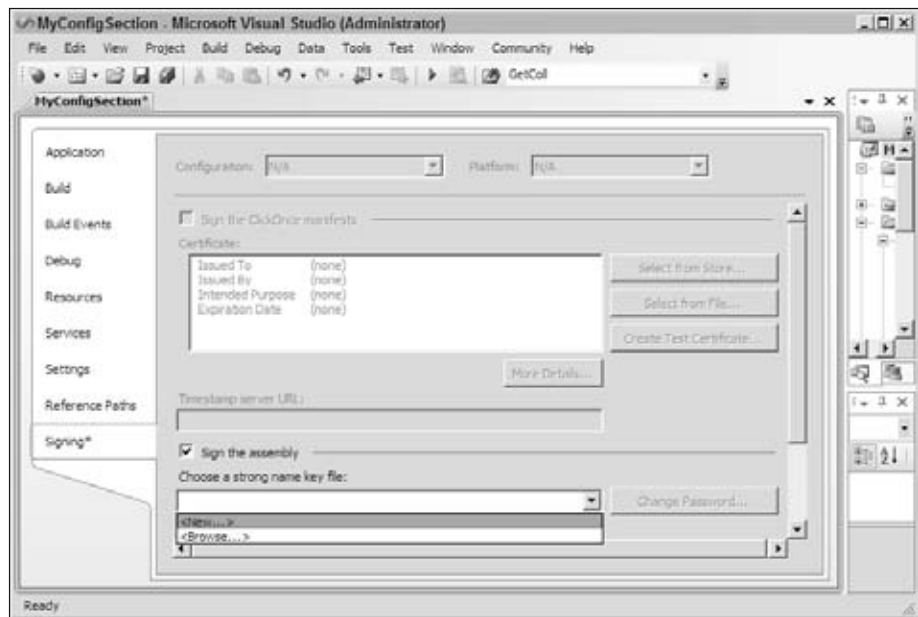


Figure 7-16

Enter `MyConfigSectionKey.snk` in the Key file name textbox, toggle off the “Protect my key file with a password” checkbox, and click OK. This will automatically generate a private and a public key, store them in a key file named `MyConfigSectionKey.snk`, and add the key file to your project. When you build the project, Visual Studio will automatically sign the assembly with this private key.



Figure 7-17

Now configure Visual Studio to automatically add the compiled assembly to the Global Assembly Cache (GAC). Select Build Events as shown in Figure 7-18. Scroll down to the “Post-build event command line” textbox and enter the following command line:

```
"c:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin\gacutil.exe" /if  
"$ (TargetPath) "
```

If the `gacutil.exe` tool is located in a different folder in your machine, you need to use the path to that folder. Visual Studio automatically runs whatever script you enter in the “Post-build event command line” textbox after it compiles the assembly. This command line uses the `gacutil.exe` tool to add the compiled assembly to the GAC automatically. The alternative to this automatic scheme is to manually add the assembly to GAC.

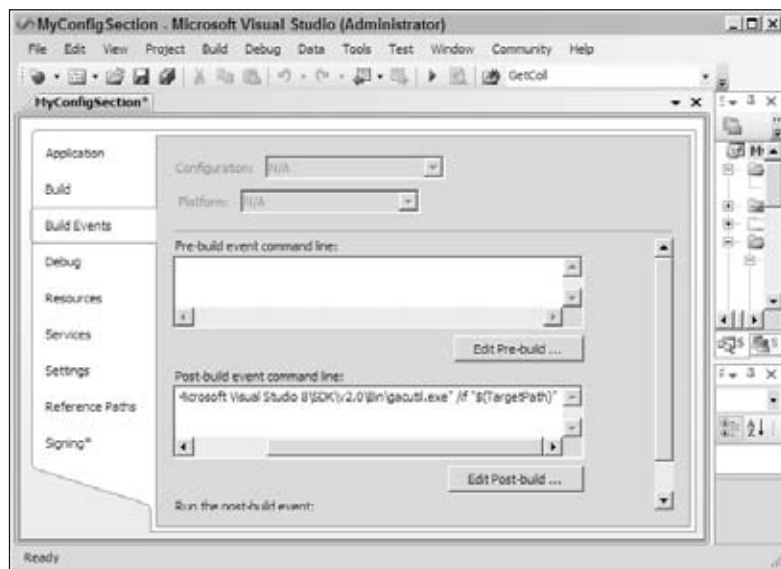


Figure 7-18

Chapter 7: Extending the Integrated Graphical Management System

Another neat thing you can do is to have Visual Studio automatically run the IIS7 Manager every time you run your program so you can see the effects of your code changes immediately without having to run the IIS7 Manager manually to test your code. Switch to the Debug tab as shown in Figure 7-19, toggle on the “Start external program” checkbox, and enter the following command line:

```
C:\Windows\System32\inetmgr\InetMgr.exe
```

Make sure you save everything.

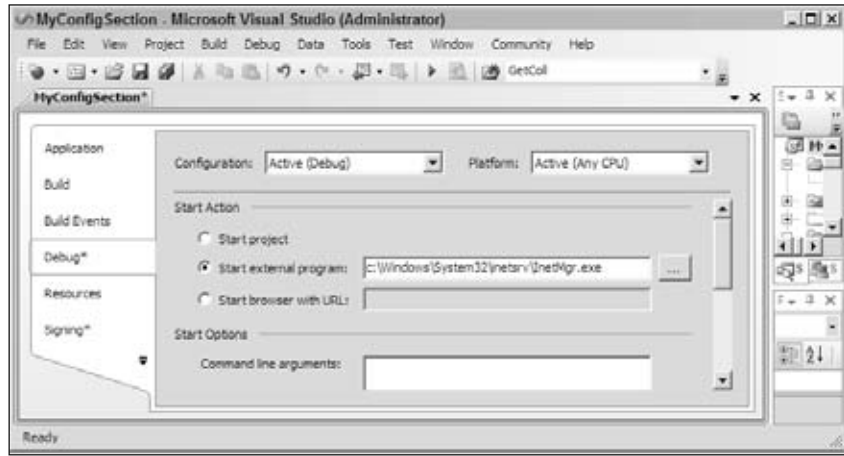


Figure 7-19

The last step of the recipe for implementing the server-side code is to register the `MyConfigSectionModuleProvider` module provider with the `administration.config` file. Open this file and add the boldfaced portions of Listing 7-97 to the `<moduleProviders>` and `<modules>` sections of this file. You must replace the value of the `PublicKeyToken` parameter shown in Listing 7-97 with a different value as discussed later in this section.

It's important that you add entries to both sections. If you don't add the entry to the `<modules>` sections, you will only be able to set the configuration settings for your custom configuration section from the machine configuration hierarchy level. In other words, you won't be able to do it in the site or application level.

Listing 7-97: The `administration.config` File

```
<configuration>
  <moduleProviders>
    <add name="MyConfigSectionModuleProvider"
    type="MyNamespace.Server.MyConfigSectionModuleProvider, MyConfigSection,
    Version=1.0.0.0, Culture=neutral, PublicKeyToken=ce416d19f343c56f" />
    . . .
  </moduleProviders>

  <location path=". ">
```

(Continued)

Listing 7-97: (continued)

```
<modules>
  <add name="MyConfigSectionModuleProvider"/>
  . . .
</modules>
</location>
</configuration>
```

Note that the `<add>` element in the `<moduleProviders>` section features two attributes named `name` and `type`. The `name` attribute contains the friendly name of your custom module provider. You can use any name you want as long as it's unique; that is, no other module provider in the `<moduleProviders>` section has the same name. The only restriction is that the `name` attribute for the `<add>` element in the `<modules>` section must match the `name` attribute used in the `<moduleProviders>` section. The `type` attribute consists of a comma-separated list of five items. The first item is the fully qualified name of the type of your custom module provider, which is `MyNamespace.Server.MyConfigSectionModuleProvider` in our case. The last four items specify the assembly that contains the `MyConfigSectionModuleProvider`. Note that the assembly information includes the assembly public key token. To access the public key token of the assembly, launch the Windows Explorer and navigate to the assembly directory where you can view the contents of the GAC. Right-click the `MyConfigSection.dll` assembly and select the Properties option to launch the Properties dialog shown in Figure 7-20. Copy the public key token shown in this dialog and paste the value into the `administration.config` file.



Figure 7-20

Summary

Extending the IIS7 and ASP.NET 3.5 integrated graphical management system requires writing two sets of managed code: client- and server-side code. I presented a recipe for writing the client-side managed code and a recipe for writing the server-side managed code. Then, I used these recipes to extend the IIS7 and ASP.NET 3.5 integrated graphical management system to add graphical management support for the `MyConfigSection` configuration section to allow the clients of this configuration section to view and to edit this configuration section directly from the IIS7 Manager.

As should be clear by now, the configuration section is the unit of extensibility in the IIS7 and ASP.NET integrated configuration system. The feature modules that make up the IIS7 and ASP.NET integrated request processing pipeline are normally associated with configuration sections from which they can be configured. The next chapter shows you how to extend the IIS7 and ASP.NET integrated request processing pipeline to implement and plug in fully configurable custom module features to add support for custom request processing capabilities.

8

Extending the Integrated Request Processing Pipeline

This chapter shows you how to implement and how to plug *configurable* managed handlers, handler factories, and modules into the IIS 7 and ASP.NET integrated request processing pipeline to extend this integrated pipeline to add support for custom configurable request processing capabilities. I present the discussions in the context of several practical examples that you can use in your own Web applications.

Extending the Integrated Pipeline through Managed Code

One of the great architectural advantages of the new IIS 7 and ASP.NET integrated request processing pipeline is its extensibility through managed code. In general, there are three main ways to extend the IIS 7 and ASP.NET integrated request processing pipeline through managed code: you can write a managed module, handler, or handler factory and plug it into the IIS 7 and ASP.NET integrated request processing pipeline to extend the pipeline to add support for new request processing capabilities.

A managed handler is an ASP.NET object responsible for handling or processing requests for ASP.NET resources with particular file extensions. For example, the managed handler that processes requests for an ASP.NET page (a resource with file extension `.aspx`) is an instance of a class that directly or indirectly derives from the ASP.NET `Page` class.

Chapter 8: Extending the Integrated Request Processing Pipeline

The ASP.NET parser automatically parses the requested ASP.NET page into a dynamically generated class that inherits from the ASP.NET Page class. ASP.NET then dynamically compiles this dynamically generated class into an assembly, loads the assembly into the application domain that contains the current ASP.NET application, instantiates an instance of this compiled class, and hands the request over to the instance for processing. This instance is responsible for generating the HTML markup and client-side code that is sent back to the browser as the server response. In other words, the instance is the managed handler that handles the request for the requested ASP.NET page.

A managed module, on the other hand, is an ASP.NET object responsible for *pre-processing* or *post-processing* ASP.NET requests. For example, ASP.NET authentication modules such as `FormsAuthenticationModule` help to establish the identity of the requester before the request is handed over to the managed handler for processing. As you can see, a managed module does not process or handle the request. Instead it pre-processes the request before it is processed by the handler, or post-processes the request after it is processed by the handler and before the response is sent back to the client.

As a result, the same module may pre-process or post-process requests for resources with different file extensions. As a matter of fact, IIS 7 allows you to use a managed module to pre-process or post-process requests for both ASP.NET and non-ASP.NET contents even though requests for ASP.NET contents are handled by managed handlers and requests for non-ASP.NET contents are handled by unmanaged handlers. In other words, a managed module can pre-process and post-process a request regardless of whether the request is handled by a managed or unmanaged handler.

A managed handler factory is an ASP.NET object responsible for instantiating and initializing the handler responsible for processing requests for resources with particular file extensions. For example, the `PageHandlerFactory` is the managed handler factory responsible for instantiating and initializing the handler responsible for processing requests for ASP.NET pages.

As a matter of fact, the PageHandlerFactory handler factory contains the logic that: a) parses the requested ASP.NET page into a dynamically generated class that derives from the ASP.NET Page class, b) compiles this dynamically generated class into an assembly, c) loads this assembly into the application domain that contains the current ASP.NET application, and d) instantiates this compiled class.

The rest of this chapter uses practical examples to show you how to implement your own custom managed module, handler, and handler factories and how to plug them into the IIS 7 and ASP.NET integrated request processing pipeline to extend this pipeline to add support for custom request processing capabilities.

Managed Handlers

All managed handlers implement an ASP.NET interface named `IHandler` as defined in Listing 8-1.

Listing 8-1: The `IHandler` Interface

```
public interface IHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}
```

This interface exposes the following members:

- ❑ **ProcessRequest**: As the name suggests, this method is responsible for processing the current request and generating the markup text that is sent back to the requester. Note that this method takes a single argument of type `HttpContext`, which refers to the current `HttpContext` object. (I discuss the `HttpContext` class shortly.) Your custom HTTP handler's implementation of the `ProcessRequest` method must include whatever logic you need to generate the appropriate markup text, which is then sent to the requester. The nature of this markup text depends on the type of ASP.NET resource. For example, the `ProcessRequest` method of the handler that processes a request for an ASP.NET page generates the HTML markup and client-side code that represents the page. The `ProcessRequest` method of the handler that processes a request for an ASP.NET Web service (with the file extension `.asmx`), on the other hand, generates a SOAP response message, which is then sent back to the requester. In this chapter, you develop a managed handler whose `ProcessRequest` method will generate an RSS document.
- ❑ **IsReusable**: As the name suggests, this read-only property returns a Boolean value that specifies whether the same handler can be used for processing different requests for the same ASP.NET resource. Your implementation of this property should normally return `false`.

As the name implies, the current `HttpContext` object defines the context within which the current request is pre-processed, processed, and post-processed. The current `HttpContext` object springs into life when the current request arrives in ASP.NET and is disposed of when the server response is sent back to the requester. In other words, the current `HttpContext` object lives as long as the current request. As such, anything that you store in the current `HttpContext` object will be disposed of at the end of the current request.

The current `HttpContext` object exposes properties that reference the well-known ASP.NET objects such as `Request`, `Response`, `Server`, and so on. Therefore, you can use the current `HttpContext` object to access these ASP.NET objects from the `ProcessRequest` method of your custom handler. Recall that ASP.NET passes a reference to the current `HttpContext` object into the `ProcessRequest` method of your custom handler when it invokes this method.

Developing Custom Managed Handlers

In this section, I present and discuss the implementation of a custom managed handler named `RssHandler`, which will process the requests for resources with the file extension `.rss`. Really Simple Syndication (RSS) is a format for syndicating news (or other frequently updated content). Before diving into the implementation of `RssHandler`, I briefly describe the RSS 2.0 format. Listing 8-2 shows an example of an RSS document.

Listing 8-2: An Example of an RSS Document

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>New Articles On localhost</title>
    <link>http://localhost/RssHandlerCh8</link>
    <description>The list of newly published articles on localhost</description>
    <item>
      <title>What's new in ASP.NET?</title>
      <description>Describes the new ASP.NET features</description>
```

(Continued)

Listing 8-2: (continued)

```
<link>http://localhost/RssHandlerCh8/Smith.aspx</link>
</item>
<item>
  <title>XML Programming</title>
  <description>Reviews .NET 2.0 XML programming features</description>
  <link>http://localhost/RssHandlerCh8/Carey.aspx</link>
</item>
<item>
  <title>XSLT in ASP.NET Applications</title>
  <description>Shows how to use XSLT in your ASP.NET applications</description>
  <link>http://localhost/RssHandlerCh8/Smith.aspx</link>
</item>
</channel>
</rss>
```

As you can see, RSS is an XML document with a document element named `<rss>`, which has a mandatory attribute named `version`. I cover only version 2.0 here. The `<rss>` document element has a single child element named `<channel>`, which has three required elements named `<title>`, `<link>`, and `<description>`. The `<channel>` element may also contain zero or more `<item>` elements. Listing 8-2 shows three child elements of the `<item>` element: `<title>`, `<description>`, and `<link>`.

To generate RSS for your application, you could use a data-bound control such as `Repeater` to generate the RSS document. The problem with this approach is that every time the user accesses the document, the request goes through the typical page life cycle even though the RSS document doesn't contain any HTML markup text. To avoid the overhead of normal ASP.NET request processing, this section implements a custom HTTP handler named `RssHandler` to replace the normal page handler.

The `RssHandler` generates the required RSS document from a database table named `Articles` with data fields named `Title`, `Abstract`, and `AuthorName` (see Figure 8-1). Each record of the `Articles` table stores the title, abstract, and author name for a particular article. As you'll see shortly, each record of the `Articles` table maps into a particular `<item>` XML child element of the `<channel>` XML element of the RSS document; the `<title>`, `<description>`, and `<link>` subelements of the `<item>` element display the values of the `Title` and `Abstract` data fields and the formatted value of the `AuthorName` data field of the associated data record, as shown in Figure 8-2.

	Column Name	Data Type	Allow Nulls
1	ArticleID	int	☐
2	Title	varchar(50)	☐
3	AuthorName	varchar(50)	☐
4	Abstract	varchar(255)	☐

Figure 8-1

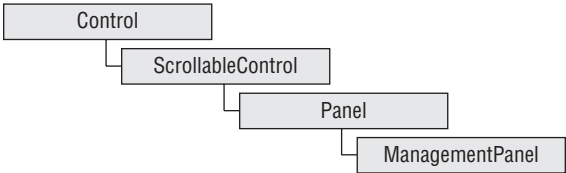


Figure 8-2

Chapter 8: Extending the Integrated Request Processing Pipeline

Now launch Visual Studio. Add a blank solution named `ProIIS7AspNetIntegProgCh8` and add a new Class Library project named `RssHandlerProj` to this solution. Right-click `RssHandlerProj` in the Solution Explorer and select the Properties option. Switch to the Application tab and enter **ProIIS7AspNetIntegProgCh8** into the “Assembly name” and “Default namespace” textboxes and save the changes. Add references to the `System.Configuration.dll` and `System.Web.dll` assemblies to the `RssHandlerProj` project.

Use the approach discussed in Chapter 7 to configure Visual Studio to generate a strongly-named assembly for the `RssHandlerProj` project and to have Visual Studio automatically deploy this assembly to the Global Assembly Cache (GAC). Another alternative is to use Windows Explorer to manually add your strongly-named assembly to GAC.

Listing 8-3 presents the implementation of the `RssHandler` HTTP handler. Now add a new source file named `RssHandler.cs` to the `RssHandlerProj` Class Library project and add the code shown in this code listing to this source file.

Listing 8-3: The `RssHandler` HTTP Handler

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Data.SqlClient;
using System.IO;
using System.Collections;

namespace ProIIS7AspNetIntegProgCh8
{
    public class RssHandler : IHttpHandler
    {
        string channelTitle;
        string channelLink;
        string channelDescription;
        string itemTitleField;
        string itemDescriptionField;
        string itemLinkField;
        string itemLinkFormatString;
        string connectionStringName;
        string commandText;
        CommandType commandType;

        public RssHandler()
        {
            channelTitle = "New Articles On localhost";
            channelLink = "http://localhost/RssHandlerCh8";
            channelDescription = "The list of newly published articles on localhost";
            itemTitleField = "Title";
            itemDescriptionField = "Abstract";
            itemLinkField = "AuthorName";
            itemLinkFormatString = "http://localhost/ RssHandlerCh8/{0}.aspx";
            connectionStringName = "MyConnectionString";
            commandText = "Select * From Articles";
        }
    }
}
```

(Continued)

Listing 8-3: (continued)

```
        commandType = CommandType.Text;
    }

    bool IHttpHandler.IsReusable
    {
        get { return false; }
    }

    SqlDataReader GetDataReader()
    {
        SqlConnection con = new SqlConnection();
       ConnectionStringSettings settings =
            ConfigurationManager.ConnectionStrings[connectionStringName];
        con.ConnectionString = settings.ConnectionString;
        SqlCommand com = new SqlCommand();
        com.Connection = con;
        com.CommandText = commandText;
        com.CommandType = commandType;
        con.Open();
        return com.ExecuteReader(CommandBehavior.CloseConnection);
    }

    public void LoadRss(Stream stream)
    {
        SqlDataReader reader = GetDataReader();

        ArrayList items = new ArrayList();
        Item item;
        while (reader.Read())
        {
            item = new Item();
            item.Title = (string)reader[itemTitleField];
            item.Link = (string)reader[itemLinkField];
            item.Description = (string)reader[itemDescriptionField];
            item.LinkFormatString = itemLinkFormatString;
            items.Add(item);
        }
        reader.Close();

        Channel channel = new Channel();
        channel.Title = channelTitle;
        channel.Link = channelLink;
        channel.Description = channelDescription;

        RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
    }

    void IHttpHandler.ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "text/xml";
        LoadRss(context.Response.OutputStream);
    }
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

I discuss the implementation of the members of the `RssHandler` HTTP handler in the following sections. However, before diving into the implementations of these members, I present the implementations of two classes named `Channel` and `Item` that the implementation of the `RssHandler` HTTP handler uses. The `RssHandler` HTTP handler's implementation uses an instance of the `Channel` class to represent the channel information. The following code fragment contains the code for this class. Add a new source file named `Channel.cs` to the `RssHandlerProj` Class Library project and add the code shown here to this source file:

```
namespace ProIIS7AspNetIntegProgCh8
{
    public class Channel
    {
        private string title;
        private string description;
        private string link;

        public string Title
        {
            get { return title; }
            set { title = value; }
        }

        public string Description
        {
            get { return description; }
            set { description = value; }
        }

        public string Link
        {
            get { return link; }
            set { link = value; }
        }
    }
}
```

The `RssHandler` HTTP handler also uses an instance of the `Item` class to represent each RSS item. The following code listing contains the implementation of this class. Add a new source file named `Item.cs` to the `RssHandlerProj` Class Library project and add the code shown here:

```
namespace ProIIS7AspNetIntegProgCh8
{
    public class Item
    {
        private string title;
        private string description;
        private string link;
        private string linkFormatString;

        public string Title
        {
            get { return title; }
            set { title = value; }
        }
    }
}
```

```
    }

    public string Description
    {
        get { return description; }
        set { description = value; }
    }

    public string Link
    {
        get { return link; }
        set { link = value; }
    }

    public string LinkFormatString
    {
        get { return linkFormatString; }
        set { linkFormatString = value; }
    }
}
}
```

IsReusable

The `RssHandler` HTTP handler, like any other ASP.NET HTTP handler, implements the `IHttpHandler` interface. The `RssHandler` HTTP handler's implementation of the `IsReusable` read-only property simply returns `false` to specify that the same handler must not be used to handle different requests for the resources with the `.rss` extension:

```
bool IHttpHandler.IsReusable
{
    get { return false; }
}
```

Constructor

Note that the constructor of `RssHandler` simply initializes the private fields of the handler:

- ❑ `channelTitle`: This private field specifies the text that goes within the opening and closing tags of the `title` element.
- ❑ `channelLink`: This private field specifies the link that goes within the opening and closing tags of the `link` element.
- ❑ `channelDescription`: This private field specifies the text that goes within the opening and closing tags of the `description` element.
- ❑ `itemTitleField`: This private field specifies the name of the database field whose values are displayed within the opening and closing tags of the `<title>` elements.
- ❑ `itemDescriptionField`: This private field specifies the name of the database field whose values are displayed within the opening and closing tags of the `<description>` elements.
- ❑ `itemLinkField`: This private field specifies the name of the database field whose values are displayed within the opening and closing tags of the `<link>` elements.

- ❑ `itemLinkFormatString`: This private field specifies the string format that will be used to format the values of the database field given by the `itemLinkField` before they're displayed within the opening and closing tags of the `<link>` elements. For example, the `itemLinkField` in the following code fragment is set to the database field named `AuthorName` and the `itemLinkFormatString` is set to the format string that uses the value of the `AuthorName` database field as the name of the ASP.NET page that contains the information about the specified author.

```
itemLinkField = "AuthorName";
itemLinkFormatString = "http://localhost/WebSite23/{0}.aspx";
```

- ❑ `connectionStringName`: This private field specifies the value of the name attribute of the `<add>` subelement of the `<connectionStrings>` section of the `web.config` file whose `connectionString` attribute contains the actual connection string. Here is an example:

```
<configuration>
  <connectionStrings>
    <add name="MyConnectionString" providerName="System.Data.SqlClient"
      connectionString="data source=serverName;Initial Catalog=Providers;
        integrated security=SSPI"/>
  </connectionStrings>
</configuration>
```

Connection strings are some of the most important resources of your application and you have to take all the necessary precautions to ensure that hackers do not get a hold of your connection strings. Keep in mind that a connection string contains the complete information to connect to your database. Imagine how much damage hackers can cause if they get a hold of your connection strings. To avoid such security problems, the `web.config` file comes with a section named `<connectionStrings>` where you can store your connection strings under arbitrary friendly names. Your pages will then include only the friendly names as opposed to the connection strings themselves. Putting your connection strings in the `web.config` file provides two benefits. First, the ASP.NET Framework rejects the request directly made for downloading the `web.config` file. Second, the ASP.NET Framework allows you to encrypt and electronically sign selected sections of the `web.config` file such as `<connectionStrings>`. Therefore, even if the hackers somehow manage to download the `web.config` file, they will not be able to read the content of the `<connectionStrings>` section without the required keys.

- ❑ `commandText`: This private field specifies the SQL Select statement or stored procedure that selects the required records from the underlying database table.
- ❑ `commandType`: This private field specifies whether the `commandText` private field contains the actual SQL Select statement or a stored procedure.

ProcessRequest

The following excerpt from Listing 8-3 presents the `RssHandler`'s implementation of the `ProcessRequest` method of the `IHttpHandler` interface:

```
void IHttpHandler.ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/xml";
    LoadRss(context.Response.OutputStream);
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

As you can see, the `ProcessRequest` method first uses the `HttpContext` object passed into it to access the ASP.NET `Response` object. Recall that the ASP.NET `Response` object represents the current HTTP response and exposes properties that map into the response parameters. The `Response` object exposes a property named `ContentType` that maps into the `Content-Type` HTTP header of the server response. The `ProcessRequest` method sets the value of this property to `"text/xml"` to signal the client that the response contains an XML document. Keep in mind that the RSS document is an XML document:

```
context.Response.ContentType = "text/xml";
```

Next, the `ProcessRequest` method invokes a method named `LoadRss`, passing in a reference to the response output stream. As you'll see in the next section, `LoadRss` generates the RSS document and writes the document into the response output stream.

```
LoadRss(context.Response.OutputStream);
```

LoadRss

Listing 8-4 presents the implementation of the `LoadRss` method.

Listing 8-4: The LoadRss Method

```
public void LoadRss(Stream stream)
{
    SqlDataReader reader = GetDataReader();

    ArrayList items = new ArrayList();
    Item item;
    while (reader.Read())
    {
        item = new Item();
        item.Title = (string)reader[itemTitleField];
        item.Link = (string)reader[itemLinkField];
        item.Description = (string)reader[itemDescriptionField];
        item.LinkFormatString = itemLinkFormatString;
        items.Add(item);
    }
    reader.Close();

    Channel channel = new Channel();
    channel.Title = channelTitle;
    channel.Link = channelLink;
    channel.Description = channelDescription;

    RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
}
```

As you can see from Listing 8-4, `LoadRss` begins by calling a method named `GetDataReader` to return a `SqlDataReader` that contains the data records. Keep in mind that each data record contains information about a particular RSS item.

```
SqlDataReader reader = GetDataReader();
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Next, it instantiates an `ArrayList` that will be populated with the `Item` objects, each an instance of the `Item` class, which exposes four properties named `Title`, `Link`, `Description`, and `LinkFormatString`:

```
ArrayList items = new ArrayList();
```

Then, `LoadRss` iterates through the data records in the `SqlDataReader` and performs these tasks for each enumerated data records:

1. Creates an `Item` object:

```
item = new Item();
```

2. Stores the values of the data fields whose names are given by the `itemTitleField`, `itemLinkField`, and `itemDescriptionField` private fields in the `Title`, `Link`, and `Description` properties of the `Item` object:

```
item.Title = (string)reader[itemTitleField];  
item.Link = (string)reader[itemLinkField];  
item.Description = (string)reader[itemDescriptionField];
```

3. Adds the `Item` object to the `ArrayList`:

```
items.Add(item);
```

Next, `LoadRss` creates a `Channel` object and populates it with the channel-related values:

```
Channel channel = new Channel();  
channel.Title = channelTitle;  
channel.Link = channelLink;  
channel.Description = channelDescription;
```

Finally, `LoadRss` invokes the `GenerateRss` static method on a helper class named `RssHelper`, passing in the `Channel` object, an array that contains the `Item` objects, and the reference to the response output stream. As you'll see later in this chapter, the `GenerateRss` method uses the channel information stored in the `Channel` object and the item information stored in the `Item` array to generate the RSS document and writes this document into the response output stream.

```
RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
```

GetDataReader

As Listing 8-3 shows, `GetDataReader` uses ADO.NET to connect to the underlying database to retrieve the required data. This method returns a `SqlDataReader` that streams out the retrieved data.

GenerateRss

The `GenerateRss` static method is a helper method that generates the actual RSS document. This method takes three parameters. The first parameter refers to the `Channel` object that contains the channel information, the second parameter is an array of `Item` objects that each contain the information about a particular RSS item, and the last parameter refers to the response output stream. The main responsibility of this method is to generate the RSS document and to write this document into the response output stream. Listing 8-5 presents the implementation of this method. Now add a new source file named `RssHelper.cs` to the `RssHandlerProj` Class Library project and add the code shown in this code listing to this source file.

Listing 8-5: The GenerateRss Method of the RssHelper Class

```
using System;
using System.Configuration;
using System.Collections.Specialized;
using System.IO;
using System.Xml;

namespace ProIIS7AspNetIntegProgCh8
{
    public class RssHelper
    {
        public static void GenerateRss(Channel channel, Item[] items, Stream stream)
        {
            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;

            using (XmlWriter writer = XmlWriter.Create(stream, settings))
            {
                writer.WriteStartDocument();
                writer.WriteStartElement("rss");
                writer.WriteAttributeString("version", "2.0");
                writer.WriteStartElement("channel");
                writer.WriteElementString("title", channel.Title);
                writer.WriteElementString("link", channel.Link);
                writer.WriteElementString("description", channel.Description);
                foreach (Item item in items)
                {
                    writer.WriteStartElement("item");
                    writer.WriteElementString("title", item.Title);
                    writer.WriteElementString("description", item.Description);
                    writer.WriteElementString("link",
                        string.Format(item.LinkFormatString, item.Link));
                    writer.WriteEndElement();
                }
                writer.WriteEndElement();
                writer.WriteEndElement();
                writer.WriteEndDocument();
            }
        }
    }
}
```

As Listing 8-5 shows, this method begins by instantiating an `XmlWriterSettings` object, and setting its `Indent` property to request the `XmlWriter` to indent the RSS document:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
```

Next, it instantiates the `XmlWriter` that will be used to write the RSS document into the response output stream:

```
using (XmlWriter writer = XmlWriter.Create(stream, settings))
```

Chapter 8: Extending the Integrated Request Processing Pipeline

The `XmlWriter` class implements the `IDisposable` interface, which exposes a single method named `Dispose`. The `using` construct automatically invokes the `Dispose` method of the specified object when the object goes out of scope. The `Dispose` method of the `XmlWriter` internally calls the `Close` method on the `XmlWriter` to close the writer. As you can see, the `using` construct saves you from having to explicitly invoke the `Close` method. This also avoids a common bug where the developer forgets to invoke the `Close` method to close the writer. The same argument applies to `XmlReader`. Therefore it is highly recommended that you instantiate and use `XmlWriter` and `XmlReader` within the context of a `using` construct.

Next, the `GenerateRss` method invokes the `WriteStartDocument` method on the `XmlWriter` to mark the beginning of the RSS document and to write the XML declaration into the response output stream:

```
writer.WriteStartDocument();
```

Then, it calls the `WriteStartElement` method on the `XmlWriter` to write the opening tag of the `rss` document element (`<rss>`) into the response output stream:

```
writer.WriteStartElement("rss");
```

Next, it calls the `WriteAttributeString` method on the `XmlWriter` to write the version attribute and its value on the opening tag of the `rss` document element (`<rss version="2.0">`) into the response output stream:

```
writer.WriteAttributeString("version", "2.0");
```

Then, it calls the `WriteStartElement` method on the `XmlWriter` to write out the opening tag of the channel child element (`<channel>`):

```
writer.WriteStartElement("channel");
```

Next, it calls the `WriteElementString` method to write the title child element of the channel element and its content (`<title>...</title>`) into the response output stream:

```
writer.WriteElementString("title", channel.Title);
```

Then, it invokes the `WriteElementString` method to write the link child element and its content (`<link>...</link>`) into the response output stream:

```
writer.WriteElementString("link", channel.Link);
```

Next, it calls the `WriteElementString` method once again to write out the description child element and its content (`<description>...</description>`):

```
writer.WriteElementString("description", channel.Description);
```

Then, it iterates through the `Item` objects in the `items` collection and performs these tasks for each enumerated `Item` object:

1. Invokes the `WriteStartElement` method to write out the opening tag of the item child element (`<item>`):

```
writer.WriteStartElement("item");
```


Chapter 8: Extending the Integrated Request Processing Pipeline

2. Calls the `WriteElementString` method to write out the `title` child element and its content (`<title>...</title>`):

```
writer.WriteElementString("title", item.Title);
```

3. Invokes the `WriteElementString` once again to write out the `description` child element and its content (`<description>...</description>`):

```
writer.WriteElementString("description", item.Description);
```

4. Calls the `WriteElementString` once more to write out the `link` child element and its content (`<link>...</link>`). Note that it uses the `Format` method, passing in the format string and link to format the link before it writes it into the stream:

```
writer.WriteElementString("link",  
    string.Format(item.LinkFormatString, item.Link));
```

5. Invokes the `WriteEndElement` to write out the closing tag of the `item` child element (`</item>`):

```
writer.WriteEndElement();
```

Next, it invokes the `WriteEndElement` method twice to write out the closing tags of the `channel` and `rss` elements, that is, (`</channel>` and `</rss>`):

```
writer.WriteEndElement();  
writer.WriteEndElement();
```

Finally, it invokes the `WriteEndDocument` method to mark the end of the RSS document:

```
writer.WriteEndDocument();
```

As you can see, the `XmlWriter` writes XML in streaming fashion. This is in contrast to the DOM and the `XPathNavigator` random-access XML APIs.

Plugging Custom Managed Handlers into the Integrated Request Processing Pipeline

The previous section showed you how to implement your own custom managed handler. This section shows you how to plug your custom managed handlers into the IIS 7 and ASP.NET integrated request processing pipeline to extend the pipeline to add support for custom requesting processing capabilities. You'll plug your `RssHandler` HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline to enable this pipeline to generate and to send an RSS document in response to a request for a resource with the file extension `.rss`.

Take these steps to plug your custom HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline:

1. Choose the level at which you want to register your custom HTTP handler:
 - ☐ The IIS 7 Web server level: Register your custom HTTP handler with the IIS 7 Web server level if you want all Web sites running on your server to use your custom HTTP handler.

- ☐ A specific ASP.NET Web site, application, or virtual directory level: Register your custom HTTP handler with a particular ASP.NET Web site, application, or virtual directory if you want that particular Web site, application, or virtual directory to use your custom HTTP handler.
2. Compile your custom HTTP handler into an assembly. How you compile your HTTP handler depends on the level at which you want to register it:
 - ☐ If you want to register your custom HTTP handler at the IIS 7 Web server level, you must compile your HTTP handler into a strongly-named assembly and deploy it to the Global Assembly Cache (GAC) because IIS 7 only picks up assemblies deployed to the GAC. It does not pick up assemblies deployed anywhere else such as the `bin` directory of a particular Web site or Web application.
 - ☐ If you want to register your custom HTTP handler at a particular ASP.NET Web site, application, or virtual directory level, you have several compilation options. One option is to compile your HTTP handler into a strongly-named assembly and deploy this assembly to the Global Assembly Cache (GAC). Another option is to add the source files of your HTTP handler to the `App_Code` directory of the Web site or Web application with which you want to register your HTTP handler, and have the ASP.NET compilation infrastructure automatically compile your HTTP handler into a dynamic assembly. Yet another option is to manually compile your HTTP handler into an assembly and add this assembly to the `bin` directory of the Web site or Web application with which you want to register your HTTP handler.
 3. Add a reference to the assembly containing your custom HTTP handler to the ASP.NET Web site or Web application with which you want to register your custom HTTP handler. Obviously, this step applies only if you want to register your custom HTTP handler with a particular ASP.NET Web site or Web application.

Because the IIS 7 Web server expects all the referenced assemblies to be in the GAC, there is no need to take extra steps to add a reference to the assembly containing your custom HTTP handler if you want to register your custom HTTP handler with the IIS 7 Web server.
 4. Register your custom HTTP handler with the IIS 7 Web server or the ASP.NET Web site or Web application of interest. You have several different registration options:
 - ☐ Declarative registration: Declarative registration allows you to register your HTTP handler directly from a configuration file.
 - ☐ Graphical registration: Graphical registration allows you to register your HTTP handler directly from the IIS 7 Manager.
 - ☐ Imperative registration: Imperative registration allows you to register your HTTP handler directly from managed code.

Next, you use this recipe to plug the `RssHandler` HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline. The first order of business is to decide the level at which you want to register the `RssHandler` HTTP handler. First, I cover the IIS 7 Web server-level registration.

The next order of business is to decide how you want to compile the `RssHandler` HTTP handler. You've already taken care of this step because you've configured Visual Studio to compile the `RssHandler` HTTP handler into a strongly-named assembly and to deploy this assembly to the GAC. The next step of

Chapter 8: Extending the Integrated Request Processing Pipeline

the recipe requires you to add a reference to this assembly. Obviously this step does not apply to the case at hand because you want to register the `RssHandler` HTTP handler with the IIS 7 Web server.

The last step of the recipe is where you do the actual registration. I cover all three declarative, graphical, and imperative registration options. The declarative registration allows you to register the `RssHandler` HTTP handler directly from a configuration file. This configuration file in this case is the `applicationHost.config` file because you're registering the `RssHandler` HTTP handler at the IIS 7 Web server level. Open this file in your favorite editor and add the boldfaced portion of Listing 8-6 to this file. Don't forget to set the `PublicKeyToken` attribute to the actual value of the public key token of the assembly that contains the `RssHandler` HTTP handler. Chapter 7 showed you how to access the public key token of an assembly.

Listing 8-6: The Portion of the `applicationHost.config` File That Registers `RssHandler`

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="Ch8_RssHandler" path="*.rss" verb="*"
        type="ProIIS7AspNetIntegProgCh8.RssHandler, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=369d834a77" preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

As you can see, you must add an `<add>` child element to the `handlers` section of the `<system.webServer>` group and set the following attributes on this `<add>` child element to register your custom HTTP handler:

- ❑ Set the `path` attribute to the file extension that your HTTP handler supports. Use a wildcard character (*) to specify that this handler supports requests for all resources with the specified file extension. In this case, you've set the `path` attribute to `*.rss` to specify that this handler processes requests for all resources with the file extension `.rss`.
- ❑ Set the `name` attribute to the friendly name of your HTTP handler. You can choose any friendly name you want as long as it is unique, that is, as long as no other handler has the same friendly name. The friendly name of your handler is used to locate and access your handler among other handlers in the `handlers` subsection of the `system.webServer` section. In this case, you've used `Ch8_RssHandler` as the friendly name of your `RssHandler` HTTP handler.
- ❑ Set the `verb` attribute to a comma-separated list of HTTP verbs that your HTTP handler supports. Use a wildcard character if your handler supports all types of HTTP verbs.
- ❑ Set the `type` attribute to a comma-separated list of up to five parts. Only the first part is mandatory, and must contain the fully qualified name of the type of the handler, including its complete namespace containment hierarchy. The last four parts must specify the assembly that contains the type, including the assembly's name, version, culture, and public key token.
- ❑ Set the `preCondition` attribute to `integratedMode` if you want your handler to be used only when IIS 7 is running in integrated mode.

Next, I show you how to register the `RssHandler` HTTP handler directly from the IIS 7 Manager. Launch the IIS 7 Manager and select the Web server node in the Connections pane. You should see the result shown in Figure 8-3.

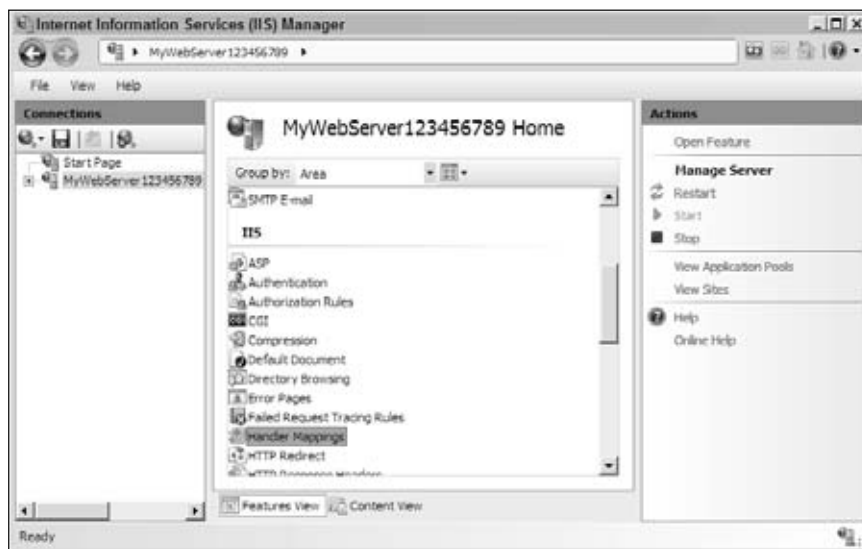


Figure 8-3

Note that the workspace in Figure 8-3 contains an item named Handler Mappings. Either double-click this item or select this item and click the Open Feature link in the task panel to navigate to the Handler Mappings module page shown in Figure 8-4.



Figure 8-4

Chapter 8: Extending the Integrated Request Processing Pipeline

The Handler Mappings module page allows you to register a new handler for handling or processing requests for a specific file extension. Now click the Add Managed Handler link in the task panel associated with this module page to launch the Add Managed Handler task form shown in Figure 8-5.

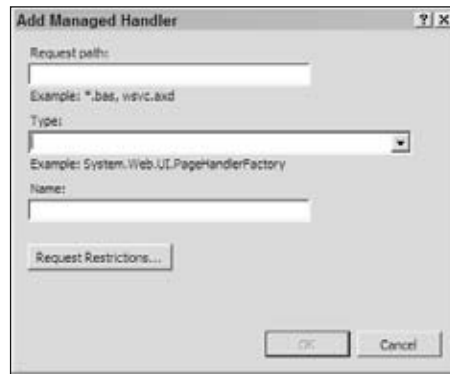


Figure 8-5

The Add Managed Handler task form contains a combo box labeled Type that displays the list of all HTTP handlers in all assemblies deployed to the Global Assembly Cache (GAC). As Figure 8-6 shows, this combo box also contains the `RssHandler` HTTP handler. This is because when you build the `RssHandlerProj` Class Library project, Visual Studio automatically deploys the `ProIIS7AspNetIntegProgCh8` assembly, which contains your `RssHandler` HTTP handler, to the GAC.

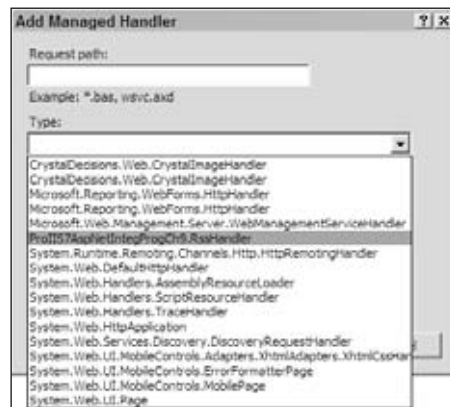


Figure 8-6

Now enter `*.rss` in the “Request path” textbox, select `ProIIS7AspNetIntegProgCh8.RssHandler` from the Type combo box, enter `Ch8_RssHandler` as the friendly name of the `RssHandler` HTTP handler in the “Name” textbox, as shown in Figure 8-7, and click OK. The callback for this button under the hood uses the appropriate proxy to add the boldfaced portion of Listing 8-6 to the `applicationHost.config` file.

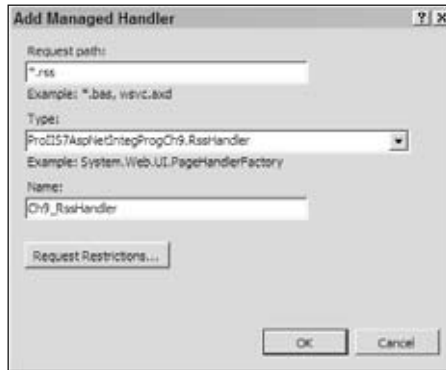


Figure 8-7

As Figure 8-8 shows, after you click OK on the Add Managed Handler task form to commit the changes to the underlying configuration file, the Handler Mappings module page is automatically updated to display the `RssHandler` HTTP handler.



Figure 8-8

When you click the Add Managed Handler link button in the task form associated with the Handler Mappings module page shown in Figure 8-4 to launch the Add Managed Handler task form shown in Figure 8-5, the `onLoad` method of this task form under the hood invokes the appropriate method of the underlying proxy to download the list of available managed handlers from the server. Listing 8-7 presents a simplified version of the logic that the server-side code uses to retrieve the list of available managed handlers.

Listing 8-7: A Simplified Version of the Server-Side Logic That Retrieves the List of Managed Handlers

```
public virtual string[] GetManagedHandlers()
{
    ArrayList managedHandlers = new ArrayList();
    ICollection assemblyNames = Gac.GetAssembliesInGAC();
    foreach (string assemblyName in assemblyNames)
    {
        Assembly assembly = Assembly.Load(assemblyName);
        Type[] types = assembly.GetExportedTypes();
        for (int i = 0; i < types.Length; i++)
        {
            if (typeof(IHttpHandler).IsAssignableFrom(types[i]) &&
                type.IsClass && !type.IsAbstract)
                managedHandlers.Add(type.FullName);
        }
    }

    return managedHandlers.ToArray();
}
```

As Listing 8-7 shows, this logic first instantiates an `ArrayList`, which will be populated with the fully qualified names of available managed handlers:

```
ArrayList managedHandlers = new ArrayList();
```

Then, it invokes a static method named `GetAssembliesInGac` on an internal class named `Gac` to retrieve the list of assemblies in the Global Assembly Cache (GAC):

```
ICollection assemblyNames = Gac.GetAssembliesInGAC();
```

Next, it iterates through this list and takes these steps for each enumerated assembly name:

1. Invokes a static method named `Load` on a .NET class named `Assembly` passing in the enumerated assembly name. This method returns an `Assembly` object, which represents the assembly with the specified name.

```
Assembly assembly = Assembly.Load(assemblyName);
```

2. Invokes the `GetExportedTypes` method on this `Assembly` object. This method returns a collection of `Type` objects that represent the public types defined in this assembly, which are visible outside the assembly.

```
Type[] types = assembly.GetExportedTypes();
```

3. Searches through the collection of `Type` objects returned from the `GetExportedTypes` method for those types that meet all the following conditions:
 - ☐ They implement the `IHttpHandler` interface.
 - ☐ They are classes.
 - ☐ They are not abstract classes.

and adds the fully-qualified names of these types to the `managedHandlers` `ArrayList`.

```
if (typeof(IHttpHandler).IsAssignableFrom(types[i]) && type.IsClass &&
    !type.IsAbstract)
    managedHandlers.Add(type.FullName);
```

4. Finally, it loads the content of the `managedHandlers` `ArrayList` into an array and returns the array to its caller. In other words, the Add Managed Handler task form receives an array, which contains the fully qualified names of available managed HTTP handlers, from the server and displays them in the Type combo box.

So far, I've discussed two different ways to register the custom `RssHandler` HTTP handler with the IIS 7 Web server: declarative through the configuration file, and graphical through the IIS 7 Manager. The third way to register a custom HTTP handler with the IIS 7 Web server is through managed code as follows.

Add a new Console Application named `RssHandlerConsoleApplication` to the `ProIIS7AspNetIntegProgCh8` solution. Add the code shown in Listing 8-8 to the `Program.cs` file. Keep in mind that Visual Studio automatically adds this file to the console application.

Listing 8-8: The Content of the Program.cs File

```
using System;
using Microsoft.Web.Administration;

namespace RssHandlerConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            ServerManager mgr = new ServerManager();
            Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
            ConfigurationSection handlersSection =
                appHostConfig.GetSection("system.webServer/handlers");

            ConfigurationElementCollection handlers = handlersSection.GetCollection();

            ConfigurationElement rssHandler = handlers.CreateElement("add");
            rssHandler.SetAttributeValue("name", "Ch8_RssHandler");
            rssHandler.SetAttributeValue("path", "/*.rss");
            rssHandler.SetAttributeValue("verb", "*");
            rssHandler.SetAttributeValue("type", "ProIIS7AspNetIntegProgCh8.RssHandler");
            rssHandler.SetAttributeValue("preCondition", "integratedMode");
            handlers.Add(rssHandler);
            mgr.CommitChanges();
        }
    }
}
```

Run the `RssHandlerConsoleApplication` project and open the `applicationHost.config` file in your favorite editor. You should see the boldfaced portion shown in Listing 8-6, which registers your `RssHandler` HTTP handler with the IIS 7 Web server. Next, I walk you through the implementation of the `Main` method shown in Listing 8-8.

Chapter 8: Extending the Integrated Request Processing Pipeline

This method basically uses the IIS 7 and ASP.NET integrated imperative management API to register the `RssHandler` HTTP handler directly from managed code. As you can see, this method begins by instantiating a `ServerManager` instance, which is always the first step when you're using the IIS 7 and ASP.NET integrated imperative management API:

```
ServerManager mgr = new ServerManager();
```

Next, it calls the `GetApplicationHostConfiguration` method on the `ServerManager` instance to load the content of the `applicationHost.config` file into a `Configuration` instance:

```
Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
```

Then, it invokes the `GetSection` method on this `Configuration` instance to return a reference to the `ConfigurationSection` instance that provides imperative access to the `<handlers>` configuration section of the `<system.webServer>` configuration section group:

```
ConfigurationSection handlersSection =  
    appHostConfig.GetSection("system.webServer/handlers");
```

Next, it invokes the `GetCollection` method on this `ConfigurationSection` instance to return a reference to the `ConfigurationElementCollection` collection that contains the `<add>` elements, which register handlers with IIS 7:

```
ConfigurationElementCollection handlers = handlersSection.GetCollection();
```

Then, it calls the `CreateElement` method on this collection to create a new `ConfigurationElement` instance that will represent the new `<add>` element, which will register the `RssHandler` HTTP handler with IIS 7:

```
ConfigurationElement rssHandler = handlers.CreateElement("add");
```

Next, it invokes the `SetAttributeValue` method five times to set the values of the name, path, verb, type, and precondition attributes on this `<add>` element to `"Ch8_RssHandler"`, `"*.rss"`, `"*"`, `"ProIIS7AspNetIntegProgCh8.RssHandler"`, and `"integratedMode"`, respectively:

```
rssHandler.SetAttributeValue("name", "Ch8_RssHandler");  
rssHandler.SetAttributeValue("path", "*.rss");  
rssHandler.SetAttributeValue("verb", "*");  
rssHandler.SetAttributeValue("type", "ProIIS7AspNetIntegProgCh8.RssHandler");  
rssHandler.SetAttributeValue("preCondition", "integratedMode");
```

Then, it adds this `ConfigurationElement` instance to the `ConfigurationElementCollection` collection:

```
handlers.Add(rssHandler);
```

Finally, it invokes the `CommitChanges` method on the `ServerManager` instance to commit the changes to the underlying `applicationHost.config` file:

```
mgr.CommitChanges();
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Keep in mind that you've been following the four-step recipe to plug the `RssHandler` HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline. As discussed, this involves deciding whether you want to register your HTTP handler at the IIS 7 Web server level or a particular ASP.NET Web site, Web application, or virtual directory level. So far, I've covered the case involving handler registration at the IIS 7 Web server level. Next, I discuss the case involving handler registration at a particular ASP.NET Web site, Web application, or virtual directory level.

First, you need to unregister the `RssHandler` HTTP handler with the IIS 7 Web server. Keep in mind that if you register your HTTP handler with IIS 7, you don't need to repeat the registration for the Web sites or Web applications running on your server because they all inherit the HTTP handler from the server. For the next step of this exercise you need to first unregister the `RssHandler` HTTP handler with IIS 7.

Now add a new Web application named `RssHandlerCh8` to the `ProIIS7AspNetIntegProg` solution. Next, you'll follow the four-step recipe to plug the `RssHandler` HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline, but this time around you will register your `RssHandler` HTTP handler with the `RssHandlerCh8` application.

You're already done with the first step of the recipe because you've decided you want to register the `RssHandler` HTTP handler with the `RssHandlerCh8` application as opposed to IIS 7. You're already done with the second step of the recipe as well because you've configured Visual Studio to compile the `RssHandler` into a strongly-named assembly and to deploy this assembly to the GAC.

The third step of the recipe requires you to add a reference to the assembly containing your `RssHandler` HTTP handler to the ASP.NET Web application with which you want to register your handler. This application in this case is the `RssHandlerCh8` application. Adding a reference simply adds a new entry to the `<assemblies>` collection XML element of the `<compilation>` configuration section of the `web.config` configuration file of this application as shown in the following code:

```
<configuration>
  <system.web>
    <compilation debug="true">
      <assemblies>
        <add assembly="ProIIS7AspNetIntegProgCh8, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=369d834a770f1f59"/>
        . . .
      </assemblies>
    </compilation>
  </system.web>
</configuration>
```

The fourth step of the recipe requires you to register your `RssHandler` HTTP handler with the `RssHandlerCh8` application. Recall that when you registered the `RssHandler` HTTP handler with IIS 7, I discussed three approaches to do this: declarative through the configuration file, graphical through the IIS 7 Manager, and imperative through the managed code. The same three approaches are applicable when you're registering your custom HTTP handler with a particular ASP.NET Web site or Web application.

First, I cover the declarative approach. You still have to add the boldfaced portion of Listing 8-6 to the configuration file as you did when you registered the `RssHandler` HTTP handler with IIS 7. The only difference is that when you're registering your `RssHandler` HTTP handler with the `RssHandlerCh8`

Chapter 8: Extending the Integrated Request Processing Pipeline

application, you should add this boldfaced portion to the `web.config` file of this application as opposed to the `applicationHost.config` file.

Next, I cover the graphical approach. You still have to go through pretty much the same steps as you did when you registered the `RssHandler` HTTP handler with IIS 7. The main difference is the starting point of this registration process. To graphically register the `RssHandler` HTTP handler with IIS 7, you had to first select the server node from the Connections pane to view the module page (see Figure 8-3), which contains the Handler Mappings item. To graphically register the `RssHandler` HTTP handler with the `RssHandlerCh8` application, on the other hand, you have to first select the `RssHandlerCh8` application node from the Connections pane to view the module page (see Figure 8-9), which contains the Handler Mappings item.



Figure 8-9

When you double-click the Handler Mappings item, the IIS 7 Manager navigates to the Handler Mappings module page shown in Figure 8-10.

Now if you click the Add Managed Handler link in the task panel associated with this Handler Mappings module page, this module page will launch the Add Managed Handler task form shown in Figure 8-11. Now compare the list of managed handlers shown in the Type combo box in Figure 8-6 with the list of managed handlers shown in the Type combo box in Figure 8-11. Note that they are not the same. This is because the combo box shown in Figure 8-6 displays the managed handlers in the assemblies deployed to the GAC, whereas the combo box shown in Figure 8-11 displays managed handlers in the assemblies referenced by the `RssHandlerCh8` application. The assemblies referenced by the `RssHandlerCh8` application are the assemblies registered with the `<assemblies>` Collection XML element of the `<compilation>` configuration section of the `web.config` file of this application and the higher-level configuration files.



Figure 8-10

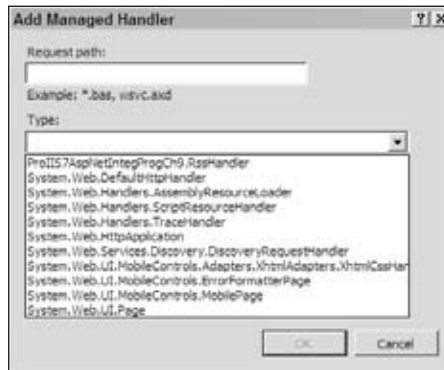


Figure 8-11

Listing 8-9 shows how to use the imperative approach to register the `RssHandler` HTTP handler with the `RssHandlerCh8` application.

Listing 8-9: The Revised Content of the Program.cs File

```
using System;
using Microsoft.Web.Administration;

namespace RssHandlerConsoleApplication
{
```

(Continued)

Listing 8-9: (continued)

```
class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration appHostConfig =
            mgr.GetWebConfiguration("Default Web Site", "/RssHandlerCh8");
        ConfigurationSection handlersSection =
            appHostConfig.GetSection("system.webServer/handlers");

        ConfigurationElementCollection handlers = handlersSection.GetCollection();

        ConfigurationElement rssHandler = handlers.CreateElement("add");
        rssHandler.SetAttributeValue("name", "Ch8_RssHandler");
        rssHandler.SetAttributeValue("path", "/*.rss");
        rssHandler.SetAttributeValue("verb", "**");
        rssHandler.SetAttributeValue("type", "ProIIS7AspNetIntegProgCh8.RssHandler");
        rssHandler.SetAttributeValue("preCondition", "integratedMode");
        handlers.Add(rssHandler);
        mgr.CommitChanges();
    }
}
```

Using the *RssHandler HTTP Handler*

Add a text file with the extension `.rss` to the `RssHandlerCh8` application. This file does not have to contain anything because the `RssHandler` will intercept the request and process the request with no regard to the content of this file. You also need to create a new database named `ArticlesDB` that contains a table named `Articles` as shown in Figure 8-1 and add the boldfaced portion of the following listing to the `<connectionStrings>` section of the `web.config` file of the `RssHandlerCh8` application:

```
<configuration>
  <connectionStrings>
    <add name="MyConnectionString"
      connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\
ArticlesDB.mdf;Integrated Security=True;User Instance=false" />
    </connectionStrings>
  </configuration>
```

Now access this RSS file from your browser. You should see the result shown in Figure 8-12.

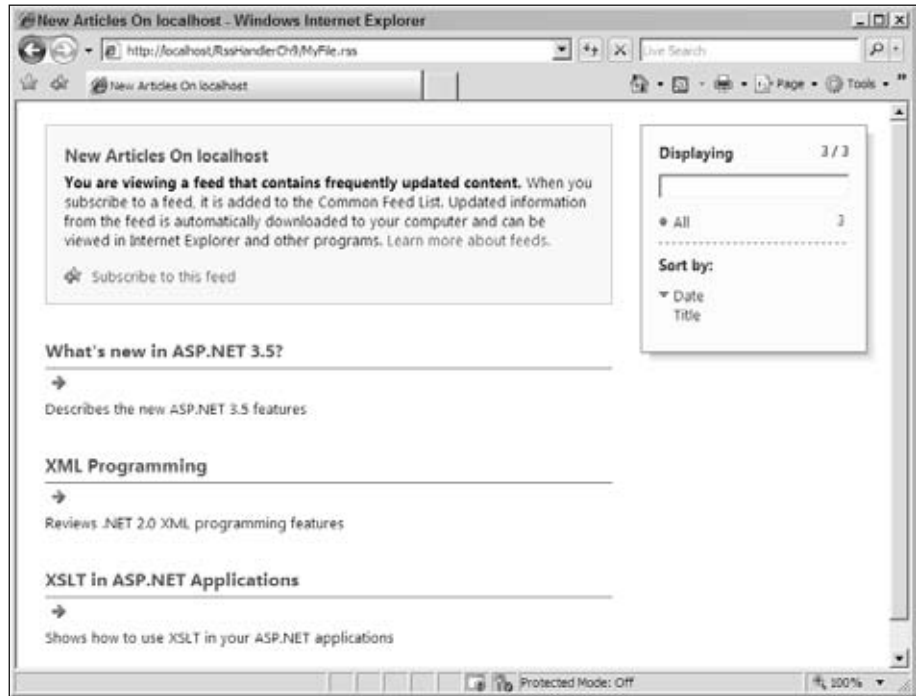


Figure 8-12

Managed Modules

All managed modules implement the ASP.NET `IHttpModule` interface, as defined in Listing 8-10. This interface exposes the following methods:

- ❑ **Init:** This method takes an instance of a class named `HttpApplication`, which represents the current ASP.NET application. I discuss `HttpApplication` shortly. Your custom module's implementation of this method must register one or more event handlers for one or more events of the `HttpApplication` object passed into it as its argument.
- ❑ **Dispose:** This method takes no arguments and returns no value. Your custom module's implementation of this method must perform its final cleanup such as releasing the resources that it is holding (if any) before your module is disposed of.

Listing 8-10: The `IHttpModule` Interface

```
public interface IHttpModule
{
    void Dispose();
    void Init(HttpApplication app);
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

ASP.NET uses a pool of `HttpApplication` objects for processing requests for a specified ASP.NET application. This allows ASP.NET to process multiple requests simultaneously to improve the throughput of an application. When a request for a resource belonging to an ASP.NET application arrives, ASP.NET picks an `HttpApplication` object from the pool and assigns the object to the task of processing the request. If the pool does not contain any `HttpApplication` objects, or if all the `HttpApplication` objects of the pool are busy processing other requests, ASP.NET automatically instantiates and initializes a new `HttpApplication` object to process the current request. Keep in mind that a given `HttpApplication` object cannot process more than one request at a time. One great thing about the pool is that when an `HttpApplication` object is done with processing a request, it is returned back to the pool for another request. Such reuse of `HttpApplication` objects reduces the overhead of instantiating and initializing new `HttpApplication` objects.

As an `HttpApplication` object is processing a request, it fires these events in the specified order:

1. **BeginRequest:** The `HttpApplication` object raises the `BeginRequest` event when it begins processing the current request. A managed module can register an event handler for this event to perform tasks that must be performed at the beginning of the request. For example, this is a good place for a managed module to perform URL rewriting.
2. **AuthenticateRequest:** The `HttpApplication` object raises the `AuthenticateRequest` event to allow authentication managed modules to authenticate the current request.
3. **PostAuthenticateRequest:** The `HttpApplication` object fires the `PostAuthenticateRequest` event after the request is authenticated. A managed module can register an event handler for this event to perform tasks that must be performed after the current request is authenticated.
4. **AuthorizeRequest:** The `HttpApplication` object fires the `AuthorizeRequest` event to allow authorization managed modules to authorize the current request.
5. **PostAuthorizeRequest:** The `HttpApplication` object fires the `PostAuthorizeRequest` event after the request is authorized. A managed module can register an event handler for this event to perform tasks that must be performed after the current request is authorized.
6. **ResolveRequestCache:** The `HttpApplication` object fires the `ResolveRequestCache` event to allow caching managed modules to service the current request from the cache, bypassing the execution of the current HTTP handler and consequently improving the performance of the application.
7. **PostResolveRequestCache:** The `HttpApplication` object fires the `PostResolveRequestCache` event after caching managed modules service the current request from the cache, bypassing the execution of the current HTTP handler.
8. **MapRequestHandler:** The `HttpApplication` object fires the `MapRequestHandler` event right before the handler responsible for processing the current request is specified and assigned to the `Handler` property of the current `HttpContext` object.
9. **PostMapRequestHandler:** The `HttpApplication` object fires the `PostMapRequestHandler` event after the managed handler responsible for handling the current request is instantiated and assigned to the `Handler` property of the current `HttpContext` object.
10. **AcquireRequestState:** The `HttpApplication` object fires the `AcquireRequestState` event to allow the state managed modules to acquire the request state from the underlying data store.

- 11.** `PostAcquireRequestState`: The `HttpApplication` object fires the `PostAcquireRequestState` event after the request state is acquired.
- 12.** `PreRequestHandlerExecute`: The `HttpApplication` object fires the `PreRequestHandlerExecute` event before the execution of the managed handler responsible for handling the current request begins. A managed module can register an event handler for this event to perform tasks that must be performed right before the managed handler is executed, that is, right before the `ProcessRequest` method of the managed handler is invoked. Recall that every managed handler implements the `ProcessRequest` method of the `IHttpHandler` interface.
- 13.** `PostRequestHandlerExecute`: The `HttpApplication` object fires the `PostRequestHandlerExecute` event after the execution of the managed handler responsible for handling the current request completes, that is, after the `ProcessRequest` method of the managed handler returns.
- 14.** `ReleaseRequestState`: The `HttpApplication` object fires the `ReleaseRequestState` event to allow state managed modules to release or store the request state into the underlying data store.
- 15.** `PostReleaseRequestState`: The `HttpApplication` object fires the `PostReleaseRequestState` event right after the request state is stored into the underlying data store to allow the interested managed modules and application code to run logic that must be run after the request state is saved.
- 16.** `UpdateRequestCache`: The `HttpApplication` object fires the `UpdateRequestCache` event to allow caching managed modules to cache the current response.
- 17.** `PostUpdateRequestCache`: The `HttpApplication` object fires the `PostUpdateRequestCache` event after the current response is cached.
- 18.** `LogRequest`: The `HttpApplication` object fires the `LogRequest` event to allow interested managed modules and application code to log the request data. The `HttpApplication` object fires this event only when the current application pool is running in integrated mode.
- 19.** `PostLogRequest`: The `HttpApplication` object fires the `PostLogRequest` event after all interested managed modules and application code have logged the request data. The `HttpApplication` object fires this event only when the current application pool is running in the integrated mode.
- 20.** `EndRequest`: The `HttpApplication` object fires the `EndRequest` event right after the current request is completely processed.
- 21.** `PreSendRequestHeaders`: The `HttpApplication` object fires the `PreSendRequestHeaders` event before the HTTP headers are sent to the client.
- 22.** `PreSendRequestContent`: The `HttpApplication` object fires the `PreSendRequestContent` event before the content is sent to the client.

The `HttpApplication` object also raises an event named `Error`. The main difference between this event and other events is that this event can occur any time during the processing of the current request. Your HTTP module can register an event handler for the `Error` event to log any diagnostic data that will help understand why the error occurred.

Developing Custom Managed Modules

This section implements a managed module named `UrlRewriterModule` that will extend the IIS 7 and ASP.NET integrated request processing pipeline to add support for a feature known as URL rewriting. URL rewriting allows users to access a page using a URL other than the actual URL of the page. In other words, URL rewriting allows you to virtualize the URLs of your Web application's pages. URL rewriting especially comes to the rescue in the following two situations:

- ❑ Restructuring a Web application normally requires the administrator to move some of the existing pages to different URLs. This causes problems for users who have bookmarked your pages, or are used to the old URLs. URL rewriting allows you to transparently map the old URLs to new ones, giving the users the illusion that they are still accessing the old URLs.
- ❑ Most Web applications use query strings to transfer data from one page to another. This leads into URLs that are very hard for users to remember and to use.

The `UrlRewriterModule` managed module will allow users to use memorable URLs to access your Web application pages. The main responsibility of this managed module is to map and to rewrite these memorable URLs to the actual URLs that your application expects.

Suppose you have a Web application that allows users to access the articles written by a particular author. Typically, these kinds of applications pass the author's name or id as part of the URL to the page that displays the list of the articles written by the author. Call this page `Articles.aspx`. Therefore, the URL for accessing the articles written by an author named Smith will look something like this:

```
http://localhost/Articles/Articles.aspx?AuthorName=Smith
```

Such a URL is not memorable and causes all kinds of usability issues. You'll make life easier on the visitors of your site if you allow them to use the following URL to access the same information:

```
http://localhost/Articles/Smith.aspx
```

This is a much more memorable and usable URL than the previous one. Behind the scenes, the `UrlRewriterModule` managed module transparently rewrites the URL that the user enters (`http://localhost/Articles/Smith.aspx`) back to the actual URL (`http://localhost/Articles/Articles.aspx?AuthorName=Smith`) that your application expects.

There are two important things about managed modules that you should keep in mind:

- ❑ A managed module is a component that responds to one or more events of the `HttpApplication` object, which represents the current ASP.NET application. Therefore, every managed module must register one or more event handlers for one or more events of the current `HttpApplication` object. The managed module executes its logic in the event handlers that it registers for the events of the `HttpApplication` object. For example, the `FormsAuthenticationModule` managed module registers an event handler for the `AuthenticateRequest` event, where it runs the logic that determines whether the current user is authenticated and that redirects unauthenticated users to the configured login page for authentication.
- ❑ An event handler registered by a managed module must execute the logic it is designed for and update the current `HttpContext` object accordingly. In other words, the current `HttpContext` object is passed from one module to another as the `HttpApplication` object fires its events and invokes the event handlers that the managed modules have registered for these events.

Chapter 8: Extending the Integrated Request Processing Pipeline

Because the `HttpApplication` object fires its events in the order specified earlier in this chapter, the managed modules that register event handlers for later events see the changes made to the `HttpContext` object by managed modules that register event handlers for earlier events. In other words, if a managed module that registers event handlers for earlier events changes the values of one or more properties of the current `HttpContext` object, the managed modules that register event handlers for later events are forced to use the new values of these properties of the current `HttpContext` object.

Therefore, when you're deciding for which events of the `HttpApplication` object your custom managed module must register its event handlers, you must take into account the effects that the changes made to the `HttpContext` object will have on later managed modules.

As discussed earlier, the `UrlRewriterModule` managed module's main job is to rewrite the memorable URL that the end user enters in the browser's address bar to the actual URL that your application expects. Obviously, rewriting the URL will affect those managed modules that execute after the `UrlRewriterModule` module rewrites the URL, if the logic they execute uses the URL.

Here is an example. Suppose the `UrlRewriterModule` managed module registers its event handler for the `BeginRequest` or `AuthenticateRequest` event of the `HttpApplication` object. Because the `FormsAuthenticationModule` managed module executes its logic when the `AuthenticateRequest` event is fired, the `FormsAuthenticationModule` managed module will only see the rewritten URL. To see what impact rewriting the URL will have on the `FormsAuthenticationModule` module, we need to study how this module operates. This module first checks whether the current requester is authenticated. If not, it redirects the user to the specified login page. After the user successfully logs in to the system, the user is redirected back to the originally requested URL, which is the rewritten version of the URL. This means that the user will still see the rewritten URL in the browser's address bar instead of the memorable one.

You can easily fix this problem by having the `UrlRewriterModule` managed module register its event handler for the `AuthorizeRequest` event instead of the `BeginRequest` or `AuthenticateRequest` event. This will allow the `FormsAuthenticationModule` module to see the original memorable URL that the user entered in the browser's address bar. Therefore, when the user is redirected back to the original requested URL after going through the authentication process, the user's browser's address bar will still show the memorable URL.

However, the authentication modules such as the `FormsAuthenticationModule` module are not the only managed modules that rely on the request URL; the authorization modules such as `FileAuthorizationModule` do as well. The `FileAuthorizationModule` module executes when the `HttpApplication` fires its `AuthorizeRequest` event. This means that this authorization module will perform its authorization on the original memorable URL instead of the actual URL as it should. Keep in mind that the `FileAuthorizationModule` module is used when Windows authentication is enabled.

Therefore, if your application is not using any form of authentication and authorization, your `UrlRewriterModule` managed module can register its event handler for any of the `BeginRequest`, `AuthenticateRequest`, or `AuthorizeRequest` events. If your application is using Forms authentication, your `UrlRewriterModule` module must register its event handler for the `AuthorizeRequest` event. If your application is using Windows authentication, your `UrlRewriterModule` module must register its event handler for the `BeginRequest` or `AuthenticateRequest` event.

The point I'm trying to get across here is that it may matter to other later HTTP modules what your custom module does. You should consider this when you're choosing an event to register an event handler for.

Chapter 8: Extending the Integrated Request Processing Pipeline

Add a new Class Library project named `UrlWriterProj` to the `ProIIS7AspNetIntegProgCh8` solution. Right-click this project in Solution Explorer and select Properties from the popup menu to launch the Properties dialog. Switch to the Application tab in this dialog and enter **ProIIS7AspNetIntegProgCh8_1** into the “Assembly name” and “Default namespace” textboxes. Next, follow the steps discussed in Chapter 7 to configure Visual Studio to compile the `UrlWriterProj` Class Library project into a strongly-named assembly and to deploy this assembly to the GAC after each build. Finally, add a reference to the `System.Web.dll` assembly to the `UrlWriterProj` Class Library project.

Listing 8-11 presents the implementation of the `UrlRewriterModule` HTTP module. Add a new source file named `UrlRewriterModule.cs` to the `UrlRewriterProj` project and add the code shown in this code listing to this source file.

Listing 8-11: The `UrlRewriterModule` HTTP Module

```
using System;
using System.Web;
using System.Text.RegularExpressions;

namespace ProIIS7AspNetIntegProgCh8_1
{
    public class UrlRewriterModule: IHttpModule
    {
        void IHttpModule.Init(HttpApplication app)
        {
            app.BeginRequest += new EventHandler(App_BeginRequest);
        }

        void App_BeginRequest(object sender, EventArgs e)
        {
            HttpApplication app = sender as HttpApplication;
            HttpContext context = app.Context;

            Regex regex = new Regex(@"Articles/(.*)\.aspx", RegexOptions.IgnoreCase);
            Match match = regex.Match(context.Request.Path);

            if (match.Success)
            {
                string newPath =
                    regex.Replace(context.Request.Path, @"Articles.aspx?AuthorName=$1");
                context.RewritePath(newPath);
            }
        }

        void IHttpModule.Dispose() { }
    }
}
```

The `UrlRewriterModule` HTTP module, like any other HTTP module, implements the `IHttpModule` interface. As Listing 8-10 shows, this interface exposes two methods named `Init` and `Dispose`. The `Dispose` method is where your custom HTTP module must release the resources it is holding. This method is automatically called before your module is removed from the pipeline of modules. Because `UrlRewriterModule` doesn’t hold on to any resources, the `Dispose` method of this module doesn’t con-

Chapter 8: Extending the Integrated Request Processing Pipeline

tain any code. Keep in mind that your custom HTTP module must implement both the `Init` and `Dispose` methods even if its implementation of the `Dispose` method does not do anything.

The `Init` method of `UrlRewriterModule` registers a method named `App_BeginRequest` as an event handler for the `BeginRequest` event of the `HttpApplication` object to rewrite the URL right at the beginning of the request:

```
app.BeginRequest += new EventHandler(App_BeginRequest);
```

As discussed, the `HttpApplication` object raises the `BeginRequest` event and automatically calls the `App_BeginRequest` method. This method casts its sender argument to `HttpApplication`. Recall that the sender argument refers to the object that raised the event, which is the `HttpApplication` object in this case.

```
HttpApplication app = sender as HttpApplication;
```

The method then uses the `Context` property of the `HttpApplication` object to access the `HttpContext` object. Recall that the `HttpContext` object contains the complete information about the request and response and exposes this information in the form of convenient managed objects such as `Request`, `Response`, and so on.

```
HttpContext context = app.Context;
```

The method then creates an instance of the `Regex` class:

```
Regex regex = new Regex(@"Articles/(.*)\.aspx", RegexOptions.IgnoreCase);
```

The argument passed into the constructor of the `Regex` class is the regular expression pattern to match.

The method then calls the `Match` method of the `Regex` instance and passes in the URL of the requested resource. This is the memorable URL that the end user uses, which is `http://localhost/Articles/Smith.aspx` in this example. You can use the `Path` property of the `Request` object to access this URL. The `Match` method of the `Regex` instance searches this URL for a substring that matches the `"Articles/(.*)\.aspx"` regular expression pattern. For example, this regular expression pattern will pick the `Articles/Smith.aspx` substring from the `http://localhost/Articles/Smith.aspx` URL.

```
Match match = regex.Match(context.Request.Path);
```

If the `Regex` instance finds a substring of the URL that matches the `"Articles/(.*)\.aspx"` regular expression pattern, it calls the `Replace` method of the `Regex` instance to replace the substring with `"Articles.aspx?AuthorName=$1"`.

```
string newPath = regex.Replace(context.Request.Path, @"Articles.aspx?AuthorName=$1");
```

Finally, the `App_BeginRequest` method calls the `Rewrite` method of the `HttpContext` object to rewrite the URL:

```
context.RewritePath(newPath);
```

Plugging Custom Managed Modules into the Integrated Request Processing Pipeline

The previous section showed you how to implement your own custom managed module. This section shows you how to plug your custom managed modules into the IIS 7 and ASP.NET integrated request processing pipeline to extend the pipeline to add support for custom requesting processing capabilities. You'll plug your `UrlRewriterModule` HTTP handler into the IIS 7 and ASP.NET integrated request processing pipeline to enable this pipeline to rewrite the user's memorable URLs to the actual URLs that your application expects.

Take these steps to plug your custom HTTP module into the IIS 7 and ASP.NET integrated request processing pipeline:

1. Use the same criteria discussed for HTTP handlers to determine whether to register your custom HTTP module with the IIS 7 Web server or a particular Web site, Web application, or virtual directory.
2. Use the same criteria discussed for HTTP handlers to determine how to compile your custom HTTP module into an assembly.
3. Add a reference to this assembly to the ASP.NET Web site or Web application with which you want to register your custom HTTP module.
4. Use one of the following registration procedures to register your custom HTTP module with the IIS 7 Web server or the desired ASP.NET Web site, Web application, or virtual directory:
 - ☐ Declaratively from a configuration file
 - ☐ Graphically from the IIS 7 Manager
 - ☐ Imperatively from managed code

Next, you use this recipe to plug the `UrlRewriterModule` HTTP module into the IIS 7 and ASP.NET integrated request processing pipeline. The first step of the recipe requires you to decide whether you want to register your `UrlRewriterModule` HTTP module with the IIS 7 Web server or a particular Web site, Web application, or virtual directory. I cover both cases beginning with the IIS 7 Web server. You're already done with the second step of the recipe because you've configured Visual Studio to compile `UrlRewriterModule` HTTP module into a strongly-named assembly and to deploy this assembly to the GAC.

Next, you use these declarative, graphical, and imperative approaches to register `UrlRewriterModule` HTTP module beginning with the declarative approach, which involves adding the boldfaced portion of Listing 8-12 to the appropriate configuration file. This configuration file in this case is `applicationHost.config` because you're registering your `UrlRewriterModule` HTTP module with the IIS 7 Web server.

Listing 8-12: The Portion of the `applicationHost.config` File That Registers the `UrlRewriterModule` HTTP Module

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <modules>
        <add name="MyUrlRewriterModule"
              type="ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule"
              preCondition="managedHandler" />
      </modules>
    </system.webServer>
  </location>
</configuration>
```

Listing 8-12: (continued)

```
</system.webServer>
</location>
</configuration>
```

Note that Listing 8-12 uses an `<add>` element to add your `UrlRewriterModule` to the `<modules>` section of the `<system.webServer>` section group. As you can see, the `<add>` element exposes an attribute named `name`, which is set to the friendly name of the HTTP module being registered. In this case, you're using `MyUrlRewriterModule` as the friendly name. The `<add>` element also exposes an attribute named `type`, which specifies the necessary type information about the module being registered. This type information consists of up to five different pieces of information, but only the first piece, which specifies the fully qualified name of the type, is required.

Also note that the `<add>` element features an attribute named `preCondition`. Set the `preCondition` attribute to `managedHandler` if you want your module to be used only for requests made for ASP.NET resources. These are the requests that are handled by managed HTTP handlers. As you can see, by simply setting the value of this attribute you can have the IIS 7 and ASP.NET integrated request processing pipeline use your HTTP module for requests made for both ASP.NET and non-ASP.NET resources. This is a departure from the earlier versions of IIS, where managed HTTP modules could only participate in processing requests made for ASP.NET resources. This means that you can now use the `UrlRewriterModule` HTTP module to perform URL rewriting not only for ASP.NET resources, but also for non-ASP.NET resources such as classic ASP pages.

Next, you use the graphical approach to register the `UrlRewriterModule` HTTP module from the IIS 7 Manager. Launch the IIS 7 Manager as usual and select the Web server node in the **Connections** pane because you want to register your module with the IIS 7 Web server. You should see the result shown in Figure 8-13.

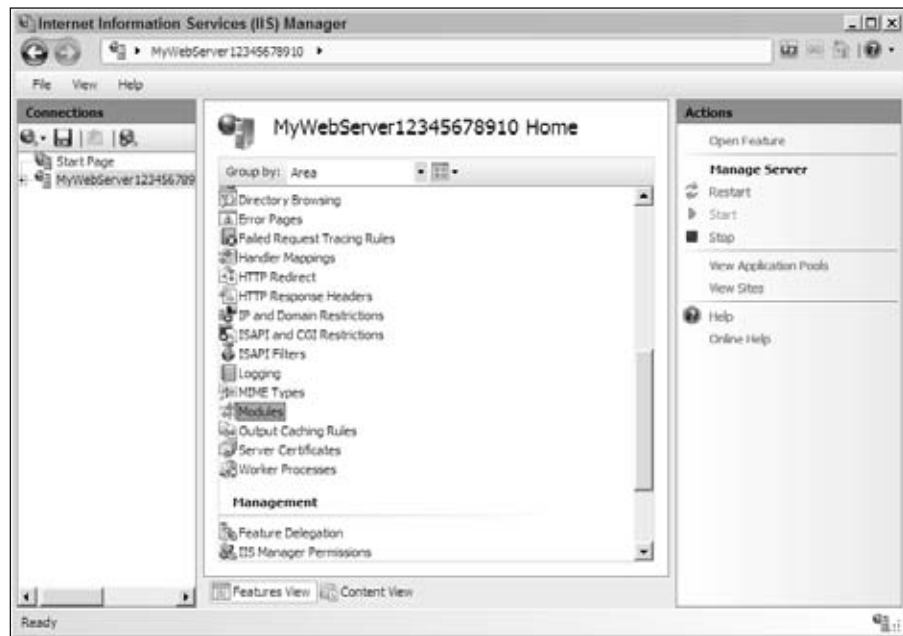


Figure 8-13

Chapter 8: Extending the Integrated Request Processing Pipeline

Double-click the Modules item shown in the IIS 7 Manager’s workspace (see Figure 8-13) or select this item and click the Open Feature link in the task panel to navigate to the Modules module list page (see Figure 8-14).

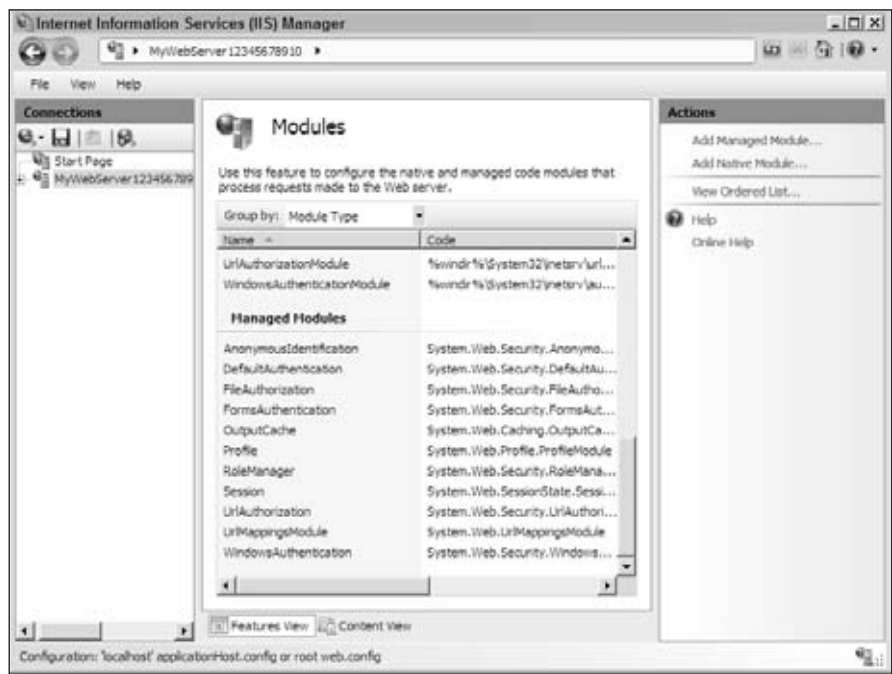


Figure 8-14

Note that the Modules module list page displays the list of both managed and native modules registered with the IIS 7 Web server. If you click the Add Managed Module link in the task panel (see Figure 8-14), it will launch the Add Managed Module task form shown in Figure 8-15.

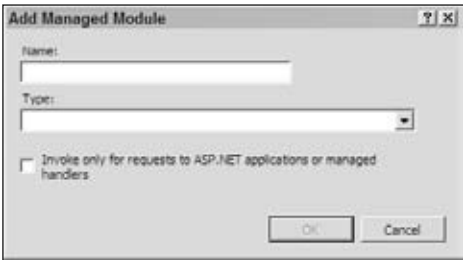


Figure 8-15

Chapter 8: Extending the Integrated Request Processing Pipeline

The Add Managed Module task form contains the following GUI elements:

- ❑ A textbox labeled Name where you must enter a friendly name for the HTTP module being registered, which is `UrlRewriterModule` in this case.
- ❑ A combo box labeled Type, which displays the list of HTTP modules in assemblies deployed to the GAC. As Figure 8-16 shows, this combo box also contains the `UrlRewriterModule` HTTP module. This shouldn't come as a surprise because you configured Visual Studio to deploy the assembly containing `UrlRewriterModule` to the GAC.
- ❑ A checkbox that you can toggle on to specify that the HTTP module being registered must only participate in processing requests made for ASP.NET resources, which are requests handled by managed HTTP handlers.



Figure 8-16

Next, use `MyUrlRewriterModule` as the friendly name. Select `ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule` from the Type combo box, check the checkbox to specify that you want the `UrlRewriterModule` HTTP module to participate in processing only requests made for ASP.NET resources, and click the OK button (see Figure 8-17). The callback for this button under the hood uses the appropriate proxy to add the boldfaced portion of Listing 8-12 to the `applicationHost.config` file.

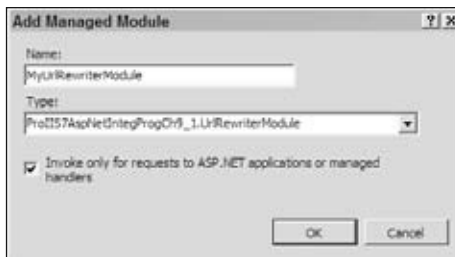


Figure 8-17

Note that the list of modules that the Modules module list page displays is automatically updated to include the newly registered `UrlRewriterModule` HTTP module (see Figure 8-18).

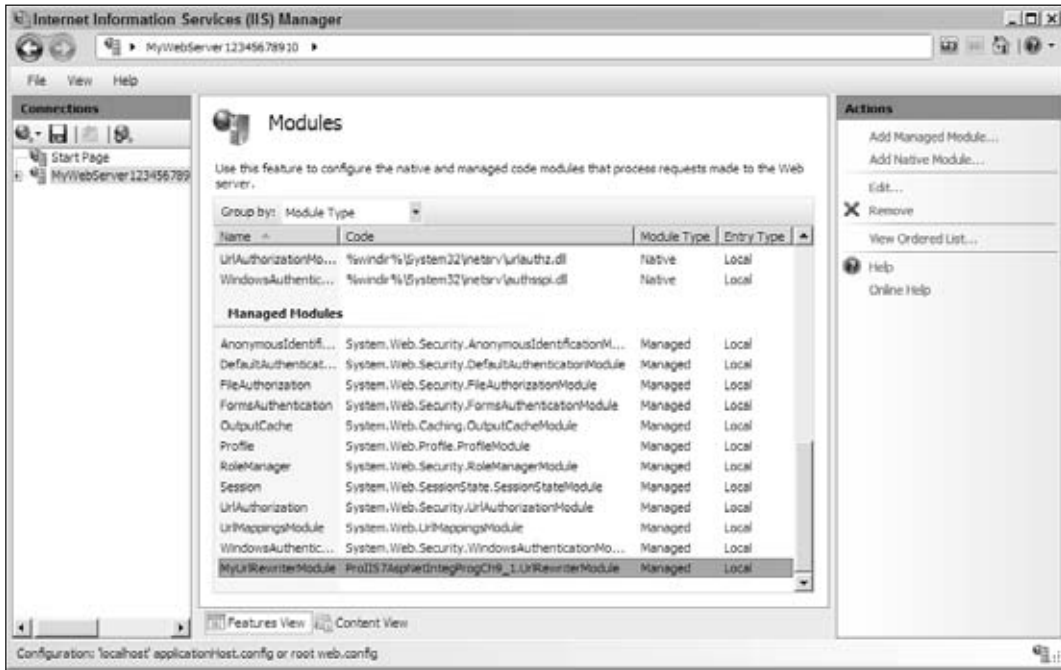


Figure 8-18

Next, I take you under the hood to show you how the Type combo box of the Add Managed Module task form displays the list of available HTTP modules, including the `UrlRewriterModule` module. This task form, like any other task form, inherits a method named `OnLoad`, which is automatically invoked when the task form is loaded. The Add Managed Module task form's implementation of this method uses the appropriate proxy to invoke the appropriate server-side method to download the list of available HTTP modules.

Listing 8-13 presents a simplified version of this server-side method. Note that this code listing is very similar to Listing 8-7 with one major difference. This time around the `IsAssignableFrom` method is invoked on the `IHttpModule` interface to determine whether the specified type implements this interface. Recall that all HTTP modules implement this interface.

Listing 8-13: A Simplified Version of the Server-Side Logic That Retrieves the List of Managed HTTP modules

```
public virtual string[] GetManagedHandlers()
{
    ArrayList managedModules = new ArrayList();
    ICollection assemblyNames = Gac.GetAssembliesInGAC();
    foreach (string assemblyName in assemblyNames)
    {
        Assembly assembly = Assembly.Load(assemblyName);
        Type[] types = assembly.GetExportedTypes();
        for (int i = 0; i < types.Length; i++)
```

Listing 8-13: *(continued)*

```
        {
            if (typeof(IHttpModule).IsAssignableFrom(types[i]) &&
                type.IsClass && !type.IsAbstract)
                managedModules.Add(type.FullName);
        }
    }

    return managedModules.ToArray();
}
```

As you can see, the Add Managed Module task form basically receives an array, which contains the fully qualified names of available managed HTTP modules, from the server and displays them in the Type combo box.

So far, you've seen two different approaches to register an HTTP module with the IIS 7 Web server. Next, I discuss the third approach where you learn how to use managed code to register the `UrlRewriterModule` HTTP module with the IIS 7 Web server.

Now go ahead and add a new Console Application named `UrlRewriterModuleConsoleApplication` to the `ProIIS7AspNetIntegProgCh8` solution and add the code shown in Listing 8-14 to the `Program.cs` file.

Listing 8-14: The Content of the Program.cs File

```
using System;
using Microsoft.Web.Administration;

namespace UrlRewriterModuleConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            ServerManager mgr = new ServerManager();
            Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
            ConfigurationSection modulesSection =
                appHostConfig.GetSection("system.webServer/modules");

            ConfigurationElementCollection modules = modulesSection.GetCollection();

            ConfigurationElement urlRewriterModule = modules.CreateElement("add");
            urlRewriterModule.SetAttributeValue("name", "MyUrlRewriterModule");
            urlRewriterModule.SetAttributeValue("type",
                "ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule");
            urlRewriterModule.SetAttributeValue("preCondition", "managedHandler");
            modules.Add(urlRewriterModule);
            mgr.CommitChanges();
        }
    }
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Next run the `UrlRewriterModuleConsoleApplication` application. This will automatically add the boldfaced portion of Listing 8-12 to the `applicationHost.config` file. As you can see, the `Main` method shown in Listing 8-14 performs this task. First, it instantiates a `ServerManager` instance as usual:

```
ServerManager mgr = new ServerManager();
```

Next, it loads the content of the `applicationHost.config` file into a `Configuration` instance:

```
Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
```

Then, it accesses the `ConfigurationSection` instance that represents the `<modules>` configuration section:

```
ConfigurationSection modulesSection =  
    appHostConfig.GetSection("system.webServer/modules");
```

Next, it accesses the `ConfigurationElementCollection` instance that represents the collection containing the registered modules:

```
ConfigurationElementCollection modules = modulesSection.GetCollection();
```

Then, it instantiates a `ConfigurationElement` that represents an `<add>` element, which will be used to register the `UrlRewriterModule` HTTP module:

```
ConfigurationElement urlRewriterModule = modules.CreateElement("add");
```

Next, it invokes the `SetAttributeValue` method on this `ConfigurationElement` to specify `MyUrlRewriterModule` as the friendly name of your `UrlRewriterModule` HTTP module:

```
urlRewriterModule.SetAttributeValue("name", "MyUrlRewriterModule");
```

Then, it calls the `SetAttributeValue` method once again to specify `ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule` as the fully qualified name of the type of the `UrlRewriterModule` HTTP module:

```
urlRewriterModule.SetAttributeValue("type",  
    "ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule");
```

Next, it invokes the `SetAttributeValue` once more to specify that your `UrlManagedModule` HTTP module can only be used to process requests made for ASP.NET resources:

```
urlRewriterModule.SetAttributeValue("preCondition", "managedHandler");
```

Then, it adds the newly instantiated `ConfigurationElement` instance to the `ConfigurationElementCollection` instance and invokes `CommitChanges` to add your `UrlManagedModule` HTTP module to the underlying `applicationHost.config` file:

```
modules.Add(urlRewriterModule);  
mgr.CommitChanges();
```

Chapter 8: Extending the Integrated Request Processing Pipeline

So far, you've learned how to use the four-step recipe to plug your `UrlRewriterModule` HTTP module into the IIS 7 and ASP.NET integrated pipeline where you registered your module with the IIS 7 Web server. Next, I show you how to use the four-step recipe to plug your module into the integrated pipeline, but this time around you will register your module with a Web site, Web application, or virtual directory.

Because registering an HTTP module with IIS 7 makes the module available to all Web sites and Web applications running on the server, and because the previous exercise has registered the `UrlRewriterModule` HTTP module with IIS 7, you need to unregister the module before you move on to the next exercise. To do so, you need to go back to the `Modules` module list page shown in Figure 8-18, select `MyUrlRewriterModule`, which is the friendly name of `UrlRewriterModule`, from the list, and click the `Remove` link button in the task form associated with the `Modules` module list page.

Because you want to register the `UrlRewriterModule` HTTP module with a Web application, first you need to create a sample Web application. Add a new Web application named `UrlRewriterModuleCh8` to the `ProIIS7AspNetIntegProg` solution. Your goal is to use the four-step recipe to register your `UrlRewriterModule` HTTP module with the `UrlRewriterModuleCh8` Web application. Obviously you don't have to worry about the first step. The second step is also taken care of because you've already configured Visual Studio to compile your `UrlRewriterModule` HTTP module.

According to the third step of the recipe, you need to add a reference to the assembly containing the `UrlRewriterModule` HTTP module to the `UrlRewriterModuleCh8` Web application. Adding this reference will simply add the boldfaced portion of the following listing to the `web.config` configuration file of the `UrlRewriterModuleCh8` Web application:

```
<configuration>
  <system.web>
    <compilation debug="true">
      <assemblies>
        <add assembly="ProIIS7AspNetIntegProgCh8_1, Version=1.0.0.0,
          Culture=neutral, PublicKeyToken=369d834a770f1f59"/>
        . . .
      </assemblies>
    </compilation>
  </system.web>
</configuration>
```

According to the fourth step of the four-step recipe, you need to use one of the declarative, graphical, or imperative approaches to register the `UrlRewriterModule` HTTP module with the `UrlRewriterModuleCh8` Web application. I discuss all these three approaches, starting with the declarative approach.

This declarative approach is very similar to the declarative approach you used to register the `UrlRewriterModule` HTTP module with IIS 7 in that you still have to add the boldfaced portion of Listing 8-12 to the configuration file as you did before. The only difference is that when you're registering the `UrlRewriterModule` HTTP module with the `UrlRewriterModuleCh8` application, you should add this boldfaced portion to the `web.config` file of this application as opposed to the `applicationHost.config` file.

Next, I show you how to use graphical approach to register your `UrlRewriterModule` HTTP module with the `UrlRewriterModuleCh8` Web application from the IIS 7 Manager. First, you need to select the `UrlRewriterModuleCh8` node from the `Connections` pane to view this application's home page as shown in Figure 8-19.

Chapter 8: Extending the Integrated Request Processing Pipeline

Next, you need to either double-click the Modules item shown in Figure 8-19 or select this item and click the Open Features link button in the task panel to navigate to the Modules module list page shown in Figure 8-20. This module list page displays the list of both managed and native modules available to the `UrlRewriterModuleCh8` Web applications. Because you haven't registered any modules, managed or native, with this Web application, you may be wondering where all the modules shown in Modules module list page (see Figure 8-20) come from. The answer to this question lies in the hierarchical nature of the IIS 7 and ASP.NET integrated configuration system. Thanks to the hierarchical characteristic of this configuration system, lower-level configurations inherit configuration settings from the higher-level configurations.

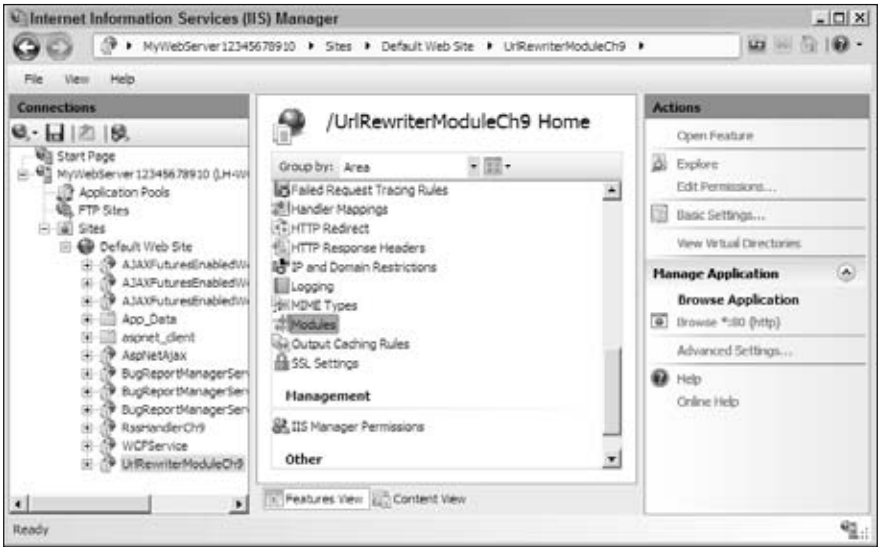


Figure 8-19

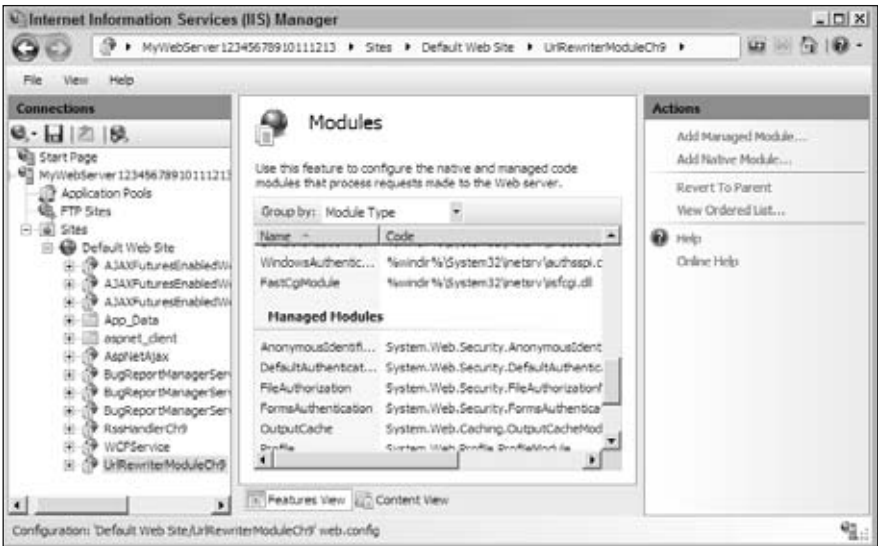


Figure 8-20

Next, click the Add Managed Module link in the task panel associated with the Modules module list page shown in Figure 8-20 to launch the Add Managed Module task form shown in Figure 8-21. If you compare the content of the Type combo box of this task form with the content of the Type combo box of the task form shown in Figure 8-16, you'll notice that they're not the same. The difference lies in the fact that these two Type combo boxes get their contents from different sources. The Type combo box in Figure 8-21 gets its content from assemblies that the `UrlRewriterModuleCh8` Web application references, whereas the Type combo box in Figure 8-22 gets its content from assemblies in the GAC. Because the `UrlRewriterModuleCh8` Web application references some of the GAC assemblies, these two Type combo boxes share some HTTP modules.



Figure 8-21

Listing 8-15 shows how to use the imperative approach to register your `UrlRewriterModule` HTTP module with the `UrlRewriterModuleCh8` application.

Listing 8-15: The Revised Content of the Program.cs File

```
using System;
using Microsoft.Web.Administration;

namespace UrRewriterModuleConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            ServerManager mgr = new ServerManager();
            Configuration appHostConfig =
                mgr.GetWebConfiguration("Default Web Site", "/UrlRewriterModuleCh8");
            ConfigurationSection modulesSection =
                appHostConfig.GetSection("system.webServer/modules");

            ConfigurationElementCollection modules = modulesSection.GetCollection();

            ConfigurationElement urlRewriterModule = modules.CreateElement("add");
            urlRewriterModule.SetAttributeValue("name", "MyUrlRewriterModule");
            urlRewriterModule.SetAttributeValue("type",
                "ProIIS7AspNetIntegProgCh8_1.UrlRewriterModule");
        }
    }
}
```

(Continued)

Listing 8-15: (continued)

```
        urlRewriterModule.SetAttributeValue("preCondition", "managedHandler");
        modules.Add(urlRewriterModule);
        mgr.CommitChanges();
    }
}
```

Using the *UrlRewriterModule HTTP Module*

Add a new Web Form named `Articles.aspx` to the `UrlRewriterModuleCh8` application and add the code shown in Listing 8-16 to this Web Form.

Listing 8-16: The `Articles.aspx` Page

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    string GetAuthorLatestArticle(string authorName)
    {
        return "This is <b>" + authorName + "</b>'s article!";
    }

    void Page_Load(object sender, EventArgs e)
    {
        string authorName = Request.QueryString["AuthorName"];
        ArticleContentLabel.Text = GetAuthorLatestArticle(authorName);
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label runat="server" ID="ArticleContentLabel" />
    </form>
</body>
</html>
```

As you can see, `Articles.aspx` is a very simple page that consists of a `Label` control named `ArticleContentLabel` that displays the content of the latest article written by a specified author. The `Page_Load` method simply extracts the name of the author from the query string and passes it into the `GetAuthorLatestArticle` method to retrieve the author's latest article. The `GetAuthorLatestArticle` method includes the logic that queries the underlying data store for the latest article written by the specified author. To keep our discussions focused, the `GetAuthorLatestArticle` method hard-codes and returns a simple string that contains the name of the author.

Now enter the following URL into the address bar of your browser and press Enter:

```
http://localhost/UrlRewriterModuleCh8/Articles/Smith.aspx
```

You should see the result shown in Figure 8-22.



Figure 8-22

Managed Handler Factories

All managed handler factories implement an ASP.NET interface named `IHttpHandlerFactory` as defined in Listing 8-17.

Listing 8-17: The `IHttpHandlerFactory` Interface

```
public interface IHttpHandlerFactory
{
    IHttpHandler GetHandler(HttpContext context, string requestType,
                           string url, string pathTranslated);

    void ReleaseHandler(IHttpHandler handler);
}
```

As you can see, this interface exposes the following members:

- ❑ **GetHandler:** As the name suggests, this method is responsible for instantiating and returning the managed handler that knows how to handle or process the current request. For example, the `PageHandlerFactory` handler factory's implementation of the `GetHandler` method of the `IHttpHandlerFactory` interface contains the logic that:
 - ❑ Parses the requested ASP.NET page into a dynamically generated class, which inherits from the ASP.NET `Page` class.
 - ❑ Compiles this class into an assembly.
 - ❑ Loads this assembly into the application domain that contains the current ASP.NET applications.
 - ❑ Instantiates and returns an instance of this compiled class. As discussed earlier, this instance is the managed handler that knows how to process the current request.

- ❑ The `WebServiceHandlerFactory` handler factory's implementation of the `GetHandler` method of the `IHttpHandlerFactory` interface, on the other hand, contains the logic that instantiates and returns the managed handler that knows how to process the current request.
- ❑ As Listing 8-17 shows, the `GetHandler` method takes four arguments as follows:
 - ❑ `context`: References the current `HttpContext` object.
 - ❑ `requestType`: Specifies the HTTP verb (POST or GET) used to make the current request.
 - ❑ `url`: Specifies the virtual path of the requested resource.
 - ❑ `pathTranslated`: Specifies the physical path of the requested resource.
- ❑ `ReleaseHandler`: As Listing 8-17 shows, the `ReleaseHandler` method takes an argument of type `IHttpHandler`. When ASP.NET invokes the `ReleaseHandler` method of a managed handler factory, it passes the managed handler instance that the `GetHandler` method returns as an argument into the `ReleaseHandler` method. This allows the managed handler factory to reuse the same handler for the next request if the factory chooses to do so.

Developing Custom Managed Handler Factories

Each managed handler factory is specifically designed to create a specific type of managed handler. For example, the `PageHandlerFactory` handler factory is specifically designed to instantiate and return a managed handler that inherits from the ASP.NET `Page` class, that is, a managed handler that knows how to process requests for resources with the file extension `.aspx`.

This section develops a custom managed handler factory named `UrlRewriterHandlerFactory` and plugs this handler factory into the IIS 7 and ASP.NET integrated request processing pipeline to add support for URL rewriting. As you can see, the IIS 7 and ASP.NET integrated request processing pipeline can be extended in two different ways to add support for URL rewriting. The previous sections discussed the first approach where this was done through a managed module named `UrlRewriterModule`. This section discusses the second approach where this is done through a managed handler factory named `UrlRewriterHandlerFactory`.

Now add a new Class Library project named `UrlRewriterHandlerFactoryProj` to the `ProIIS7AspNetIntegProgCh8` solution. Right-click this project in Solution Explorer and select Properties from the popup menu to launch the Properties dialog. Switch to the Application tab and enter `ProIIS7AspNetIntegProgCh8_2` into the "Assembly name" and "Default namespace" textboxes. Next, follow the steps discussed in Chapter 7 to configure Visual Studio to compile this project into a strongly-named assembly and to deploy this assembly to the Global Assembly Cache (GAC) after each build. Finally, add a reference to the `System.Web.dll` assembly to this project.

`UrlRewriterHandlerFactory`, like any other HTTP handler factory, implements the `IHttpHandlerFactory` interface defined in Listing 8-17. Listing 8-18 contains the implementation of `UrlRewriterHandlerFactory`. Add a new source file named `UrlRewriterHandlerFactory.cs` to the `UrlRewriterHandlerFactoryProj` project and add the code shown in this code listing to this source file.

Listing 8-18: The `UrlRewriterHandlerFactory` HTTP Handler Factory

```
using System.Web;
using System.Web.UI;
using System.Text.RegularExpressions;

namespace ProIIS7AspNetIntegProgCh8_2
{
    public class UrlRewriterHandlerFactory: IHttpHandlerFactory
    {
        IHttpHandler IHttpHandlerFactory.GetHandler(HttpContext context,
                                                    string requestType, string url, string pathTranslated)
        {
            Regex regex =
                new Regex(@"\/Articles\/(.*)\.aspx", RegexOptions.IgnoreCase);
            Match match = regex.Match(url);
            string newPath = string.Empty;

            if (match.Success)
            {
                newPath = regex.Replace(url, @"Articles.aspx?AuthorName=$1");
                context.RewritePath(newPath);
            }

            return PageParser.GetCompiledPageInstance(url,
                                                    context.Server.MapPath("Articles.aspx"), context);
        }

        void IHttpHandlerFactory.ReleaseHandler(IHttpHandler handler) {}
    }
}
```

Note that the implementation of the `GetHandler` method of `UrlRewriterHandlerFactory` is the same as the `App_BeginRequest` method, except for the last line:

```
PageParser.GetCompiledPageInstance(url,
                                    context.Server.MapPath("Articles.aspx"), context);
```

To help you understand the role of this line of code, you need to understand what type of managed handler the `GetHandler` method of `UrlRewriterHandlerFactory` is supposed to return. Recall that the main responsibility of the `GetHandler` method of a handler factory is to return a reference to an instance of a managed handler that knows how to process the current request.

Because the current request is a request for an `.aspx` file, the `GetHandler` method of `UrlRewriterHandlerFactory` must return an instance of the managed handler that knows how to process requests for resources with the file extension `.aspx`. As discussed earlier, such a managed handler is a dynamically generated class that inherits from the ASP.NET `Page` class. The ASP.NET Framework comes with a class named `PageParser` that exposes a static method named `GetCompiledPageInstance` that contains the logic that instantiates and returns an instance of the managed handler that knows how to process requests for resources with the file extension `.aspx`. As a matter of fact, the `GetHandler` method of the `PageHandlerFactory` handler factory uses the `GetCompiledPageInstance` static method under the hood to instantiate the required managed handler.

Plugging Custom Managed Handler Factories into the Integrated Request Processing Pipeline

The previous section showed you how to implement your own custom managed handler factory. Plugging a managed handler factory into the IIS 7 and ASP.NET integrated request processing pipeline is just like plugging a managed handler into this integrated pipeline. Therefore, all the same approaches discussed earlier in this chapter for plugging managed handlers into this integrated pipeline equally apply to managed handler factories.

Extending the Integrated Pipeline with Configurable Managed Components

The previous sections of this chapter showed you how to implement your own custom managed module, handler, and handler factory and how to plug them into the IIS 7 and ASP.NET integrated request processing pipeline to extend this pipeline to add support for new request processing capabilities, such as rewriting URLs and generating RSS documents.

The most important characteristic of a managed module, handler, or handler factory is its *configurability*. As a matter of fact, the *configurability* of the managed modules, handlers, and handler factories, which make up the IIS 7 and ASP.NET integrated request processing pipeline, is the corner-stone of the entire IIS 7 and ASP.NET integrated architecture, including its integrated request processing pipeline, integrated configuration system, integrated imperative management system, and integrated graphical management system.

The rest of this chapter and the next chapter walk you through practical examples where you will learn specific techniques for developing *configurable* managed modules, handlers, and handler factories. As you'll see, the configurability of these managed modules, handlers, and handler factories will allow you to extend:

- ❑ The IIS 7 and ASP.NET integrated configuration system to add support for new configuration sections for these configurable managed modules, handlers, and handler factories. This will allow clients to configure these managed modules, handlers, and handler factories directly from configuration files.
- ❑ The IIS 7 and ASP.NET integrated imperative management system to add imperative management support for these configurable managed modules, handlers, or handler factories. This will allow clients to configure these managed modules, handlers, or handler factories from managed code in a strongly-typed fashion, where they can benefit from the Visual Studio IntelliSense support, the compiler type-checking capabilities, and the well-known object-oriented programming benefits.
- ❑ The IIS 7 and ASP.NET integrated graphical management system to add graphical management support for these configurable managed modules, handlers, or handler factories. This will allow clients to configure these managed modules, handlers, or handler factories from the IIS 7 Manager.

The rest of this chapter focuses on the configurability of managed modules and handler factories and leaves the discussion of the configurability of managed handlers to the next chapter, where it is covered in the context of the IIS 7 and ASP.NET integrated providers extensibility model.

The current implementations of the `UrlRewriterModule` managed module and `UrlRewriterHandlerFactory` managed handler factory suffer from two fundamental problems. First, as Listings 8-11 and 8-18 show, the current implementations of this managed module and handler factory hard-code the following two important pieces of information:

- ❑ The regular expression that defines the pattern that the `Regex` object looks for in the memorable URL:

```
Regex regex = new Regex(@"Articles/(.*)\.aspx", RegexOptions.IgnoreCase);
```

- ❑ The regular expression that defines the replacement string:

```
string newPath = regex.Replace(context.Request.Path,  
                              @"Articles.aspx?AuthorName=$1");
```

Second, the current implementations of the `UrlRewriterModule` managed module and `UrlRewriterHandlerFactory` managed handler factory support only one pair of regular expressions.

In the remainder of this chapter, I present and discuss a new version of the `UrlRewriterModule` managed module that does not hard-code these two regular expressions. Instead it allows clients to specify them in the configuration files. The new version will also allow clients to specify an unlimited number of pairs of regular expressions. The first regular expression instructs the `Regex` object what pattern to look for in the memorable URL, and the second provides the `Replace` method of the `Regex` object with the required replacement string. Because the implementation of the new version of the `UrlRewriterHandlerFactory` handler factory is very similar to the implementation of the new version of the `UrlRewriterModule` managed module, I cover only the implementation of the new version of the `UrlRewriterModule` managed module.

The first order of business is to use the IIS 7 and ASP.NET integrated declarative schema extension markup language to extend the IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section named `urlRewriter`.

Configuration Support for the URL Rewriting Managed Module

Chapter 5 presented a six-step recipe for extending the IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section as follows:

1. Write down a representative example of the configuration section including all its XML elements and attributes.
2. Identify the following portions of the configuration section:
 - ❑ The Containing XML element, and the names, data types, and default values of its attributes
 - ❑ The Non-collection XML elements, and the names, data types, and default values of their attributes

Chapter 8: Extending the Integrated Request Processing Pipeline

- ❑ The Collection XML elements, and the names, data types, and default values of their attributes
- ❑ The child elements of each Collection element that perform the add, remove, and clear operations, and the names, data types, and default values of their attributes

3. Create a new XML file in the following directory on your machine:

```
%WINDIR%\system32\inetsrv\config\schema
```

4. Decide on the section group hierarchy where you want to add your configuration section.
5. Use the IIS 7 and ASP.NET integrated declarative schema extension markup language to implement the schema that defines the XML elements and attributes that make up your custom configuration section. Store this schema in the XML file you created in step 3.
6. Register your custom configuration section with the `<configSections>` section of the `applicationHost.config` file.

Next, you use this recipe to extend the IIS 7 and ASP.NET integrated configuration system to add support for your `urlRewriter` configuration section. Following this recipe, first you need to write down a representative example of your `urlRewriter` configuration section as shown in Listing 8-19.

Listing 8-19: The `<urlRewriter>` Configuration Section

```
<urlRewriter>
  <urlRewriterRules>
    <clear/>
    <add patternToMatch="Articles/(.*)\.aspx"
        replacement="Articles.aspx?AuthorName=$1" />
    <remove patternToMatch=" " />

  </urlRewriterRules>
</urlRewriter>
```

The second step requires you to identify different parts of your configuration section. The `<urlRewriter>` configuration section contains a Collection element named `<urlRewriterRules>`. This Collection element contains one or more `<add>` child elements that exposes a string attribute named `patternToMatch` and a string attribute named `replacement`. The client of your `UrlRewriterModule` uses an `<add>` element to add a new URL rewriter rule to the collection of URL rewriter rules. The `patternToMatch` attribute contains the regular expression pattern to match, whereas the `replacement` attribute contains the replacement string that will be passed into the `Replace` method of the underlying `Regex` object as discussed earlier.

Following the third step of the recipe, add an XML file named `URLREWRITER_schema.xml` to the schema directory on your machine. Next, you need to decide on the section group hierarchy to which you want to add the `<urlRewriter>` configuration section. In this case, add the configuration section to the `<system.webServer>` section group.

Next, you need to use the IIS 7 and ASP.NET integrated declarative schema extension markup language to implement the schema for the `urlRewriter` configuration section and store this schema in the `URLREWRITER_schema.xml` file.

Listing 8-20 presents the content of the `URLREWRITER_schema.xml` file.

Listing 8-20: The Content of the URLREWRITER_schema.xml File

```
<configSchema>
  <sectionSchema name="system.webServer/urlRewriter">
    <element name="urlRewriterRules">
      <collection addElement="add" clearElement="clear" removeElement="remove" >
        <attribute name="patternToMatch" type="string" isUniqueKey="true" />
        <attribute name="replacement" type="string" />
      </collection>
    </element>
  </sectionSchema>
</configSchema>
```

Listing 8-20 uses a `<sectionSchema>` element to define the Containing XML element of the `urlRewriter` configuration section. Note that the `name` attribute of the `<sectionSchema>` element is set to the fully qualified name of the configuration section, including its complete section group hierarchy, that is, `system.webServer/urlRewriter`.

```
<sectionSchema name="system.webServer/urlRewriter">
```

Next Listing 8-20 uses an `<element>` element with the `name` attribute value of `"urlRewriterRules"` to define the `<urlRewriterRules>` Collection element of the `<urlRewriter>` configuration section:

```
<element name="urlRewriterRules">
```

This `<element>` element contains a `<collection>` child element because it represents a Collection element. As you can see from Listing 8-20, the `addElement`, `removeElement`, and `clearElement` attributes of this `<collection>` element define the `<add>`, `<remove>`, and `<clear>` child elements of the `<urlRewriterRules>` Collection element. Recall that these three child elements respectively add a URL rewriter rule to, remove a URL rewriter rule from, and clear all URL rewriter rules from the collection of URL rewriter rules that the `<urlRewriterRules>` Collection element represents.

The `<collection>` element features two child `<attribute>` elements that define the `patternToMatch` and `replacement` attributes of the `<add>` child element of the `<urlRewriterRules>` Collection element. Note that the `isUniqueKey` attribute on the `<attribute>` element that defines the `patternToMatch` attribute is set to `true` to specify the `patternToMatch` attribute as the key attribute. This key attribute uniquely identifies one URL rewriter rule among other URL rewriter rules. This key is necessary because it will allow the `<urlRewriter>` configuration section in a lower-level configuration file to use the `<remove>` element to remove a URL rewriter rule with a specified pattern to match.

Finally, you need to register the `<urlRewriter>` configuration section with the `<configSections>` of the `applicationHost.config` file as shown in Listing 8-21.

Listing 8-21: Registering the <urlRewriter> Configuration Section

```
<configSections>
  <sectionGroup name="system.webServer">
    <section name="urlRewriter" allowDefinition="Everywhere"
      overrideModeDefault="Allow" />
  </sectionGroup>
  . . .
</configSections>
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Because the `<urlRewriter>` configuration section belongs to the `<system.webServer>` section group, Listing 8-21 registers this configuration section in the `<sectionGroup>` element with the name attribute value of `"system.webServer"`. Note that Listing 8-21 uses a `<section>` element to register the `urlRewriter` custom configuration section and sets the values of the attributes of this `<section>` element as follows:

- ❑ Sets the name attribute to the name of the configuration section being registered, which is `urlRewriter` in this case.
- ❑ Sets the `allowDefinition` attribute to `Everywhere` to allow the `<urlRewriter>` configuration section to be used in configuration files at all configuration hierarchy levels, that is, server, site, application, and virtual directory levels.
- ❑ Sets the `overrideModeDefault` attribute to `Allow` to allow lower-level configuration files to override the configuration settings specified in the `<urlRewriter>` configuration section of the `applicationHost.config` file.

Imperative Management Support for the URL Rewriting Managed Module

In this section, you extend the IIS 7 and ASP.NET integrated imperative management system with new managed classes to add imperative management support for the `urlRewriter` configuration section. This will allow page developers to program against this `urlRewriter` configuration section in a strongly-typed fashion, where they can benefit from the Visual Studio IntelliSense support for strongly-typed objects and properties, compiler type-checking support for strongly-typed objects and properties, and the well-known object-oriented programming benefits. I discuss these new managed classes in the following sections.

Before diving into the implementation of these managed classes, you need to set up the Visual Studio project that will contain these classes and the rest of the managed classes discussed in the rest of this chapter. Add a new Class Library project named `UrlRewriterProj2` to the `ProIIS7AspNetIntegProgCh8` solution. Add a reference to the `Microsoft.Web.Administration.dll` assembly. Follow the steps discussed in Chapter 7 to configure Visual Studio to:

- ❑ Compile the `UrlRewriterProj2` into a strongly-named assembly.
- ❑ Automatically add this assembly to the Global Assembly Cache (GAC).
- ❑ Automatically launch the IIS 7 Manager every time the `UrlRewriterProj2` is built.

Right-click `UrlRewriterProj2` in the Solution Explorer and select `Properties` to launch the `Properties` dialog shown in Figure 8-23. Select the `Application` tab in this dialog and specify `UrlRewriter` and `UrlRewriting` as the Assembly name and Default namespace. Don't forget to use the `File` ⇨ `SaveAll` option to save these changes.

Add a new directory named `ImperativeManagement` to the `UrlRewriterProj2` project. This directory will contain all the imperative management classes.

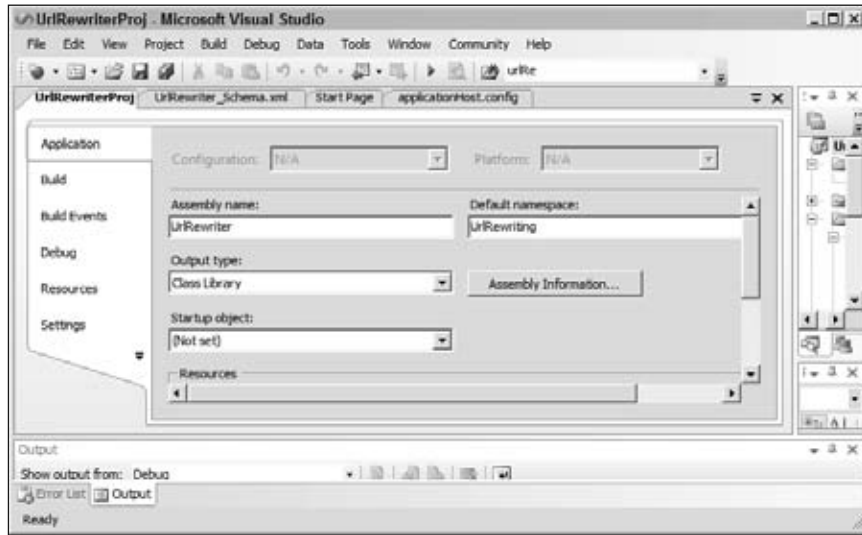


Figure 8-23

UrlRewriterRule

In this section, I design a class named `UrlRewriterRule` whose instances represent the URL rewriter rules in the collection that the `<urlRewriterRules>` element represents. Listing 8-22 presents the implementation of the `UrlRewriterRule` class. Now add a new source file named `UrlRewriterRule.cs` to the `ImperativeManagement` directory and add the code shown in this code listing to this source file.

Listing 8-22: The `UrlRewriterRule` Class

```
using Microsoft.Web.Administration;

namespace UrlRewriting.ImperativeManagement
{
    public class UrlRewriterRule : ConfigurationElement
    {
        public string PatternToMatch
        {
            get { return (string)base["patternToMatch"]; }
            set { base["patternToMatch"] = value; }
        }

        public string Replacement
        {
            get { return (string)base["replacement"]; }
            set { base["replacement"] = value; }
        }
    }
}
```


Chapter 8: Extending the Integrated Request Processing Pipeline

The `UrlRewriterRule` class inherits from the `ConfigurationElement` base class and exposes the `patternToMatch` and `replacement` attributes of the `<add>` element as strongly-typed properties named `PatternToMatch` and `Replacement`, respectively.

UrlRewriterRules

Listing 8-23 presents a collection class named `UrlRewriterRules` that represents the `<urlRewriterRules>` Collection element. Add a new source file named `UrlRewriterRules.cs` to the `ImperativeMangement` directory and add the code shown in this code listing to this source file.

Listing 8-23: The `UrlRewriterRules` Class

```
using System;
using Microsoft.Web.Administration;

namespace UrlRewriting.ImperativeManagement
{
    public class UrlRewriterRules :
        ConfigurationElementCollectionBase<UrlRewriterRule>
    {
        public UrlRewriterRule Add(string patternToMatch, string replacement)
        {
            UrlRewriterRule urlRewriterRule = base.CreateElement();
            urlRewriterRule.PatternToMatch = patternToMatch;
            urlRewriterRule.Replacement = replacement;
            base.Add(urlRewriterRule);
            return urlRewriterRule;
        }

        protected override UrlRewriterRule CreateNewElement(string elementTagName)
        {
            return new UrlRewriterRule();
        }

        public new UrlRewriterRule this[string patternToMatch]
        {
            get
            {
                for (int i = 0; i < base.Count; i++)
                {
                    if (string.Equals(base[i].PatternToMatch,
                                    patternToMatch, StringComparison.OrdinalIgnoreCase))
                        return base[i];
                }
                return null;
            }
        }
    }
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

The `UrlRewriterRules` class, like any other collection class in the IIS 7 and ASP.NET integrated imperative management system, inherits from the `ConfigurationElementCollectionBase` generic class and performs the following tasks:

- ❑ Exposes an indexer that returns the `UrlRewriterRule` object with the specified `PatternToMatch` property value:

```
public new UrlRewriterRule this[string patternToMatch]
{
    get
    {
        for (int i = 0; i < base.Count; i++)
        {
            if (string.Equals(base[i].PatternToMatch,
                              patternToMatch, StringComparison.OrdinalIgnoreCase))
                return base[i];
        }
        return null;
    }
}
```

- ❑ Exposes an `Add` method that creates a `UrlRewriterRule` object with the specified `PatternToMatch` and `Replacement` property values and adds the object to the collection:

```
public UrlRewriterRule Add(string patternToMatch, string replacement)
{
    UrlRewriterRule urlRewriterRule = base.CreateElement();
    urlRewriterRule.PatternToMatch = patternToMatch;
    urlRewriterRule.Replacement = replacement;
    base.Add(urlRewriterRule);
    return urlRewriterRule;
}
```

- ❑ Overrides the `CreateNewElement` method of its base class to instantiate and to return a `UrlRewriterRule` object:

```
protected override UrlRewriterRule CreateNewElement(string elementTagName)
{
    return new UrlRewriterRule();
}
```

UrlRewriterSection

The `UrlRewriterSection` class represents the outermost element of the configuration section, that is, the `<urlRewriter>` element, as presented in Listing 8-24. Now add a new source file named `UrlRewriterSection.cs` to the `ImperativeManagement` directory and add the code shown in this code listing to this source file.

Listing 8-24: The `UrlRewriterSection` Class

```
using Microsoft.Web.Administration;

namespace UrlRewriting.ImperativeManagement
```

Listing 8-24: *(continued)*

```
{
    public class UrlRewriterSection : ConfigurationSection
    {
        private UrlRewriterRules urlRewriterRules;
        public UrlRewriterRules UrlRewriterRules
        {
            get
            {
                if (urlRewriterRules == null)
                {
                    urlRewriterRules =
                        (UrlRewriterRules)base.GetCollection("UrlRewriterRules",
                                                            typeof(UrlRewriterRules));
                }
                return urlRewriterRules;
            }
        }
    }
}
```

The `UrlRewriterSection` class, like any other configuration section in the IIS 7 and ASP.NET integrated imperative management system, inherits from the `ConfigurationSection` base class. As you can see from Listing 8-24, this class exposes a property of type `UrlRewriterRules` named `UrlRewriterRules` that references the `UrlRewriterRules` object that represents the `<urlRewriterRules>` Collection element of the configuration section.

Testing the Managed Classes

The previous sections extended the IIS 7 and ASP.NET integrated imperative management system to add support for new imperative management classes that represent your configuration section and its constituent elements and attributes. Now it's time to put these classes to test. Launch Visual Studio, add a new console application named `UrlRewriterConsoleApplication` to the `ProIIS7AspNetIntegProgCh8` solution, add a reference to the `Microsoft.Web.Administration.dll` assembly to this console application, and add the code shown in Listing 8-25 to the `Program.cs` file. You also need to add a reference to the `UrlRewriterProj2` project, which contains your new imperative management classes.

Listing 8-25: The Program.cs File for Testing the New Managed Classes

```
using Microsoft.Web.Administration;
using UrlRewriting.ImperativeManagement;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
        UrlRewriterSection urlRewriterSection =
            (UrlRewriterSection)appHostConfig.GetSection(
```

Listing 8-25: (continued)

```
                                "system.webServer/urlRewriter",
                                typeof (UrlRewriterSection));
urlRewriterSection.UrlRewriterRules.Add(@"Articles/(.*)\.aspx",
                                @"Articles.aspx?AuthorName=$1");
mgr.CommitChanges();
}
}
```

Run the program and open the `applicationHost.config` file in your favorite editor. The result should look like the following:

```
<configuration>
  <system.webServer>
    <urlRewriter>
      <urlRewriterRules>
        <add patternToMatch="Articles/(.*)\.aspx"
              replacement="Articles.aspx?AuthorName=$1" />
      </urlRewriterRules>
    </urlRewriter>
  </system.webServer>
</configuration>
```

Now let's dissect Listing 8-25. The listing begins by creating a `ServerManager` object and calls its `GetApplicationHostConfiguration` method to load the `applicationHost.config` file into a `Configuration` object:

```
ServerManager mgr = new ServerManager();
Configuration appHostConfig = mgr.GetApplicationHostConfiguration();
```

Next, it calls the `GetSection` method to return the `UrlRewriterSection` object that provides programmatic access to the `<urlRewriter>` configuration section in a strongly-typed manner. Note that the `Type` object representing the type of the `UrlRewriterSection` class is passed into the `GetSection` method. Under the hood, this method uses .NET reflection and this type information to dynamically generate an instance of the `UrlRewriterSection` class.

```
UrlRewriterSection urlRewriterSection =
    (UrlRewriterSection)appHostConfig.GetSection(
                                "system.webServer/urlRewriter",
                                typeof (UrlRewriterSection));
```

Finally, it accesses the `UrlRewriterRules` property of the `UrlRewriterSection` object and adds a new `UrlRewriterRule` object to the collection. These operations are all performed in a strongly-typed manner:

```
urlRewriterSection.UrlRewriterRules.Add(@"Articles/(.*)\.aspx",
                                @"Articles.aspx?AuthorName=$1");
mgr.CommitChanges();
```

Graphical Management Support for the URL Rewriter Managed Module

In this section, you extend the IIS 7 and ASP.NET integrated graphical management system to add graphical management support for the `urlRewriter` configuration section. This will allow you to configure the URL rewriter managed module from the IIS 7 Manager.

As discussed earlier, you have to write two sets of managed code to extend the IIS 7 Manager: client-side managed code and server-side managed code. Go ahead and add a new directory named `GraphicalManagement` to the `UrlRewriterProj2` project. Then add two subdirectories named `Client` and `Server` to the `GraphicalManagement` directory. The `Client` and `Server` directories will respectively contain the client-side and server-side managed code.

Client-Side Managed Code

The client-side managed code is the code that you have to implement to extend the user interface of the IIS 7 Manager to add user interface support for your URL rewriter module. Before diving into the details of the implementation of this client-side managed code, let's see what this user interface looks like in action. As Figure 8-24 shows, the Web server home page includes a new item named `UrlRewriterPage`.

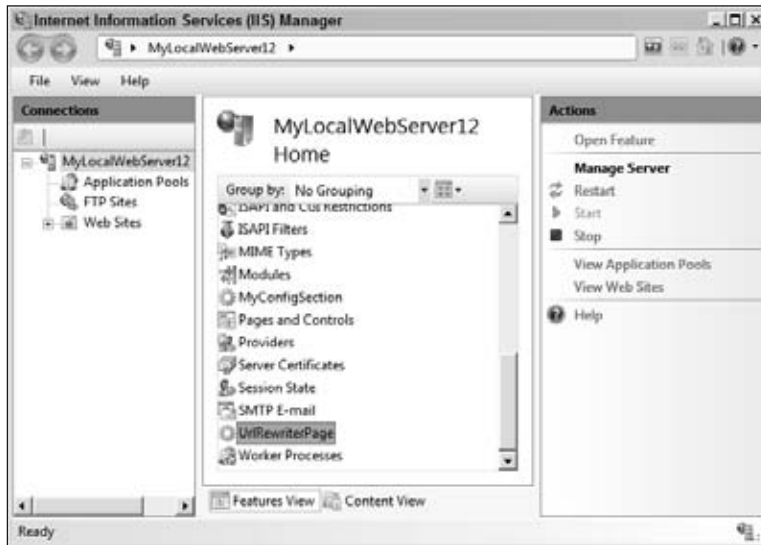


Figure 8-24

Chapter 8: Extending the Integrated Request Processing Pipeline

When the end user double-clicks this item or alternatively clicks the Open Feature link on the Actions panel, the IIS 7 Manager navigates to a module page named Url Rewriter Page as shown in Figure 8-25.

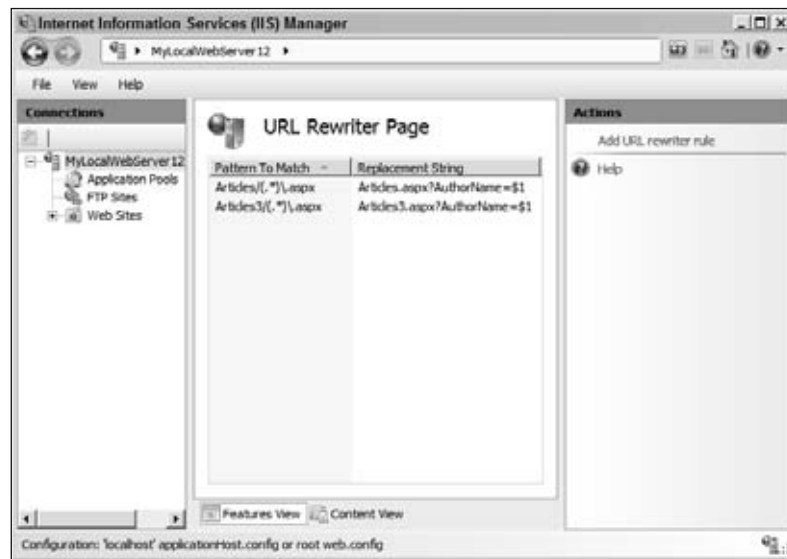


Figure 8-25

As you can see, this module page consists of a list of two columns. The first column displays the patterns to match and the second column displays their associated replacement strings.

Notice that the Actions panel contains a link named "Add URL rewriter rule." When the user clicks this link, the task form named `UrlRewriterRuleTaskForm` pops up, as shown in Figure 8-26.

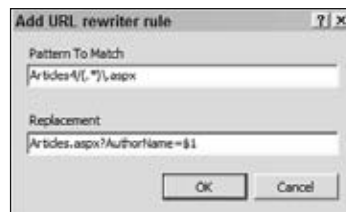


Figure 8-26

This task form allows the user to specify the pattern to match and the replacement string for the URL rewriter rule being added. As Figure 8-27 shows, when the end user selects an item from the list, the Actions panel displays two new links named "Update URL rewriter rule" and "Delete URL rewriter rule."

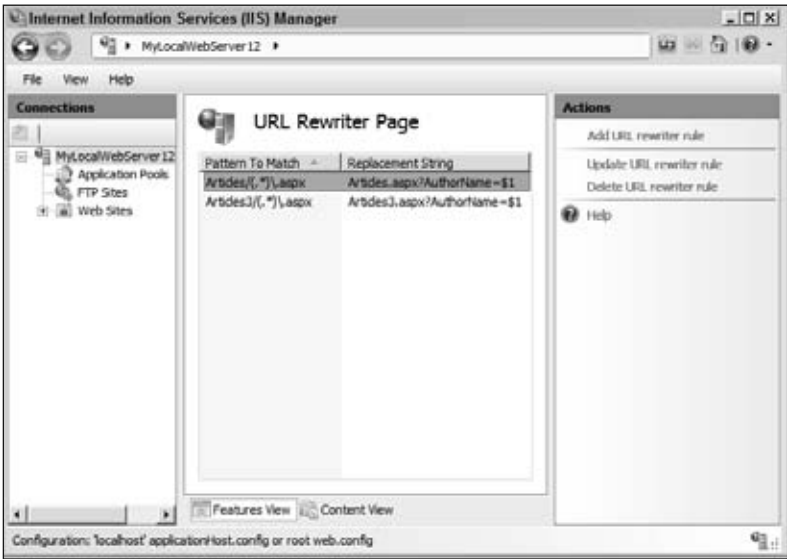


Figure 8-27

The user clicks the “Delete URL rewriter rule” link to delete the selected URL rewriter rule, and the “Update URL rewriter rule” link to launch the task form shown in Figure 8-28. This task form allows the user to modify the pattern to match and the replacement string of the selected URL rewriter rule. The user can also launch this task form by double-clicking the selected URL rewriter rule.



Figure 8-28

Communications with the Back-End Server

As the previous discussions show, you need to implement a module page named `UrlRewriterPage` and a task form named `UrlRewriterRuleTaskForm`. Obviously, the `UrlRewriterPage` module page and the `UrlRewriterRuleTaskForm` task form need to communicate with the back-end server. Let’s study the details of these communications.

First, I discuss the communication scenarios between the back-end server and the `UrlRewriterPage` module page. When the end user clicks the `UrlRewriterPage` item in the middle panel of Figure 8-24 and navigates to the `UrlRewriterPage` module page shown in Figure 8-25, this module page must invoke the appropriate method of a server-side class to retrieve the URL rewriter rules and the value

of the `isLocked` attribute on the `<urlRewriter>` configuration section. As you'll see later, in this case the server-side class is a class named `UrlRewriterModuleService` that exposes a method named `GetUrlRewriterSettings` that returns the available URL rewriter rules and the `isLocked` attribute value. Recall that the `isLocked` attribute of a configuration section specifies whether the configuration section is locked down and consequently its settings cannot be overridden. The `UrlRewriterPage` module page must disable its editing capabilities if the `GetUrlRewriterSettings` method of the server-side `UrlRewriterModuleService` returns `true` as the value of the `isLocked` attribute.

The other communication scenario involving the `UrlRewriterPage` module page is when the user selects a URL rewriter rule from the list shown in Figure 8-25 and clicks the "Delete URL rewriter rule" link in the task panel. In this case, the `UrlRewriterPage` module page must call the appropriate method of the server-side `UrlRewriterModuleService` class to delete the selected URL rewriter rule from the underlying configuration file. In this case, the server-side `UrlRewriterModuleService` class exposes a method named `DeleteUrlRewriterRule` that takes the value of the `patternToMatch` attribute as its argument and deletes the associated URL rewriter rule from the configuration file.

Next, I discuss the communications between the back-end `UrlRewriterModuleService` server-side class and the `UrlRewriterRuleTaskForm` task form. There are two communication scenarios involving this task form. The first occurs when the end user clicks the "Add URL rewriter rule" link in the task panel in Figure 8-25 and launches the task form shown in Figure 8-26 to add a new URL rewriter rule. After the end user specifies the values of the `patternToMatch` and `replacement` attributes of the new URL rewriter rule and clicks the OK button on the `UrlRewriterRuleTaskForm` task form shown in Figure 8-26, this task form must call the appropriate method of the server-side `UrlRewriterModuleService` class to add a new URL rewriter rule with the specified attribute values to the underlying configuration file. In this case, the server-side `UrlRewriterModuleService` class features a method named `AddUrlRewriterRule` that takes these attribute values and adds the new URL rewriter rule.

The second communication scenario involving the `UrlRewriterRuleTaskForm` task form occurs when the end user selects an item from the list shown in Figure 8-27 and launches this task form (see Figure 8-28) to update the current values of the `patternToMatch` and `replacement` attributes of the selected URL rewriter rule. When the end user is done with editing and clicks the OK button, the `UrlRewriterRuleTaskForm` task form must call the appropriate method of the `UrlRewriterModuleService` server-side class to update the attribute values of the selected URL rewriter rule in the configuration file. In this case, the server-side class exposes a method named `UpdateUrlRewriterRule` that takes the new attribute values and updates the underlying attributes.

The IIS 7 Manager architecture encapsulates the logic that deals with the details of the communications with the back-end server-side class into a standard client-side base class named `ModuleServiceProxy`. The methods and properties of this base class define the API that the `UrlRewriterPage` module page and `UrlRewriterRuleTaskForm` task form can use to indirectly communicate with the back-end `UrlRewriterModuleService` class and invoke its methods without getting involved in the dirty little communication details. All you have to do is to implement a client-side class known as a proxy that inherits from the `ModuleServiceProxy` base class and exposes methods with the same signatures as the methods of the back-end server-side `UrlRewriterModuleService` class. The implementation of these methods of this client-side class must use the `Invoke` method of the `ModuleServiceProxy` base class to invoke the associated methods of the server-side class. This proxy class in this case is a class named `UrlRewriterModuleServiceProxy`, as shown in Listing 8-26. Now add a new source file named `UrlRewriterModuleServiceProxy.cs` to the `Client` subdirectory of the `GraphicalManagement` directory and add the code shown in this code listing to the source file. You also need to add a reference

Chapter 8: Extending the Integrated Request Processing Pipeline

to the `Microsoft.Web.Management.dll` assembly to the `UrlRewriterProj2 Class Library` project. This assembly is located in the following directory on your machine:

```
%windir%\System32\inetsvc
```

Listing 8-26: The `UrlRewriterModuleServiceProxy` Class

```
using Microsoft.Web.Management.Client;
using Microsoft.Web.Management.Server;

namespace UrlRewriting.GraphicalManagement.Client
{
    public class UrlRewriterModuleServiceProxy : ModuleServiceProxy
    {
        public PropertyBag GetUrlRewriterSettings()
        {
            return (PropertyBag)base.Invoke("GetUrlRewriterSettings", new object[0]);
        }

        public void UpdateUrlRewriterRule(PropertyBag urlRewriterRuleToUpdate)
        {
            base.Invoke("UpdateUrlRewriterRule",
                new object[] { urlRewriterRuleToUpdate });
        }

        public void DeleteUrlRewriterRule(PropertyBag urlRewriterRuleToDelete)
        {
            base.Invoke("DeleteUrlRewriterRule",
                new object[] { urlRewriterRuleToDelete });
        }

        public void AddUrlRewriterRule(PropertyBag urlRewriterRuleToAdd)
        {
            base.Invoke("AddUrlRewriterRule", new object[] { urlRewriterRuleToAdd });
        }
    }
}
```

Note that each method of your `UrlRewriterModuleServiceProxy` custom proxy class meets the following two important requirements:

- ❑ It has the same signature as the corresponding server-side method. Recall that the signature of a method includes its name, return type, and its parameter types. As you'll see later, the server-side `UrlRewriterModuleService` class exposes the methods shown in Listing 8-27. If you compare this code listing with Listing 8-26 you'll notice that the `UrlRewriterModuleServiceProxy` proxy class exposes methods with the same signatures as the server-side `UrlRewriterModuleService` class.
- ❑ It calls the `Invoke` method of the `ModuleServiceProxy` base class and passes the following two parameters into it:
 - ❑ The name of the corresponding server-side method.

- ❑ An array of objects where each object contains the value of a particular parameter of the corresponding server-side method. The objects in this array must be in the same order as their corresponding parameters.

For example, the `UpdateUrlRewriterRule` method of the `UrlRewriterModuleServiceProxy` proxy class (see Figure 8-26) calls the `Invoke` method of the `ModuleServiceProxy` base class and passes the following two parameters into it:

- ❑ The string value `"UpdateUrlRewriterRule"`, which is the name of the corresponding server-side method (see Listing 8-27)
- ❑ An array that contains a single object of type `PropertyBag`, which is the value of the parameter of the `UpdateUrlRewriterRule` server-side method (see Listing 8-18)

```
public void UpdateUrlRewriterRule(PropertyBag urlRewriterRuleToUpdate)
{
    base.Invoke("UpdateUrlRewriterRule", new object[] {urlRewriterRuleToUpdate});
}
```

Listing 8-27: The `UrlRewriterModuleService` Server-Side Class

```
public class UrlRewriterModuleService: ModuleService
{
    [ModuleServiceMethod]
    public PropertyBag GetUrlRewriterSettings();

    [ModuleServiceMethod]
    public void AddUrlRewriterRule(PropertyBag urlRewriterRuleToAdd);

    [ModuleServiceMethod]
    public void DeleteUrlRewriterRule(PropertyBag urlRewriterRuleToDelete);

    [ModuleServiceMethod]
    public void UpdateUrlRewriterRule(PropertyBag urlRewriterRuleToUpdate);
}
```

As Listings 8-26 and 8-27 show, the client-side `UrlRewriterModuleServiceProxy` proxy class and the server-side `UrlRewriterModuleService` class use a `PropertyBag` object to exchange data. For example, the `AddUrlRewriterRule` method of the `UrlRewriterModuleServiceProxy` proxy class sends the values of the `patternToMatch` and `replacement` attributes of the new URL rewriter rule through a `PropertyBag` object.

UrlRewriterPage

As discussed, when the end user clicks the `UrlRewriterPage` item shown in Figure 8-24, the IIS 7 Manager uses the navigation service to navigate to the `UrlRewriterPage` module page. Listing 8-28 presents the implementation of the `UrlRewriterPage` class. I walk through this implementation in the following sections. Now add a new source file named `UrlRewriterPage.cs` to the `GraphicalManagement/Client` directory and add the code shown in Listing 8-28 to this source file. You also need to add a reference to the `System.Windows.Forms.dll` assembly to the `UrlRewriterProj2 Class Library` project.

Listing 8-28: The UrlRewriterPage Class

```
using Microsoft.Web.Management.Client.Win32;
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System.Windows.Forms;
using System.ComponentModel;
using System;
using System.Collections;

namespace UrlRewriting.GraphicalManagement.Client
{
    public class UrlRewriterPage : ModuleListPage
    {
        private ColumnHeader replacementColumnHeader;
        private ColumnHeader patternToMatchColumnHeader;
        private UrlRewriterModuleServiceProxy serviceProxy;
        private bool errorGetUrlRewriterSettings;
        private PropertyBag bag;
        private bool readOnly;

        protected override void InitializeListPage()
        {
            patternToMatchColumnHeader = new ColumnHeader();
            patternToMatchColumnHeader.Text = "Pattern To Match";
            patternToMatchColumnHeader.Width = 90;
            replacementColumnHeader = new ColumnHeader();
            replacementColumnHeader.Text = "Replacement String";
            replacementColumnHeader.Width = 90;
            base.ListView.Columns.Clear();
            base.ListView.Columns.AddRange(
                new ColumnHeader[] { patternToMatchColumnHeader,
                                    replacementColumnHeader });

            base.ListView.LabelEdit = false;
            base.ListView.MultiSelect = false;
            base.ListView.SelectedIndexChanged +=
                new EventHandler(OnListViewSelectedIndexChanged);
            base.ListView.DoubleClick += new EventHandler(OnListViewDoubleClick);
            base.ListView.KeyUp += new KeyEventHandler(OnListViewKeyUp);
        }

        private void OnListViewDoubleClick(object sender, EventArgs e)
        {
            if ((this.SelectedUrlRewriterRule != null) && !this.ReadOnly)
                this.UpdateUrlRewriterRule();
        }

        private void OnListViewKeyUp(object sender, KeyEventArgs e)
        {
            if ((this.SelectedUrlRewriterRule != null) && (e.KeyData == Keys.Delete))
                this.DeleteUrlRewriterRule();
        }

        private void OnListViewSelectedIndexChanged(object sender, EventArgs e)
```

Listing 8-28: (continued)

```
{
    base.Update();
}

protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (UrlRewriterModuleServiceProxy)base.CreateProxy(
                                                                    typeof(UrlRewriterModuleServiceProxy));
        this.GetUrlRewriterSettings();
    }
}

private void GetUrlRewriterSettings()
{
    base.StartAsyncTask("Getting URL rewriter settings",
        new DoWorkEventHandler(this.OnWorkerGetUrlRewriterSettings),
        new RunWorkerCompletedEventHandler(
            this.OnWorkerGetUrlRewriterSettingsCompleted));
}

private void OnWorkerGetUrlRewriterSettings(object sender, DoWorkEventArgs e)
{
    e.Result = this.serviceProxy.GetUrlRewriterSettings();
}

protected override bool ReadOnly
{
    get
    {
        return this.readOnly;
    }
}

private void OnWorkerGetUrlRewriterSettingsCompleted(object sender,
                                                    RunWorkerCompletedEventArgs e)
{
    base.ListView.BeginUpdate();
    try
    {
        if (e.Result != null)
        {
            base.ListView.Items.Clear();
            this.bag = (PropertyBag)e.Result;
            ArrayList list1 = (ArrayList)this.bag[0];
            this.readOnly = (bool)this.bag[1];

            if (list1 != null)
            {
                for (int num1 = 0; num1 < list1.Count; num1++)
```

(Continued)

Listing 8-28: (continued)

```
        {
            UrlRewriterRuleInfo info1 =
                new UrlRewriterRuleInfo((PropertyBag)list1[num1]);
            this.AddItem(info1, false);
        }
    }
    this.errorGetUrlRewriterSettings = false;
}

catch (Exception exception1)
{
    base.StopProgress();
    base.DisplayErrorMessage(exception1.Message,
        "GetUrlRewriterSettingsCompleted");
    this.errorGetUrlRewriterSettings = true;
    return;
}

finally
{
    base.ListView.EndUpdate();
}
}

private void AddItem(UrlRewriterRuleInfo urlRewriterRuleInfo, bool isSelected)
{
    UrlRewriterRuleListViewItem item1 =
        new UrlRewriterRuleListViewItem(urlRewriterRuleInfo);
    base.ListView.Items.Add(item1);

    if (isSelected)
    {
        item1.Selected = true;
        item1.Focused = true;
        base.ListView.EnsureVisible(base.ListView.Items.IndexOf(item1));
    }
}

private PageTaskList taskList;
protected override TaskListCollection Tasks
{
    get
    {
        if (this.taskList == null)
            this.taskList = new PageTaskList(this);

        TaskListCollection collection1 = base.Tasks;
        collection1.Add(this.taskList);
        return collection1;
    }
}
```

Listing 8-28: (continued)

```
private void AddUrlRewriterRule()
{
    using (UrlRewriterRuleTaskForm form1 =
        new UrlRewriterRuleTaskForm(base.Module, this.serviceProxy))
    {
        if (base.ShowDialog(form1) == DialogResult.OK)
        {
            PropertyBag bag1 = new PropertyBag();
            bag1[0] = form1.PatternToMatch;
            bag1[1] = form1.Replacement;
            this.AddItem(new UrlRewriterRuleInfo(bag1), true);
        }
    }
}

private void DeleteUrlRewriterRule()
{
    UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;
    if (item1 != null)
    {
        DialogResult result1 = base.ShowMessage(
            "Do you really want to delete this URL rewriter rule?",
            MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Question, MessageBoxDefaultButton.Button1,
            "Removed");
        if (result1 == DialogResult.Yes)
        {
            try
            {
                Cursor.Current = Cursors.WaitCursor;
                PropertyBag bag = new PropertyBag();
                bag[0] = item1.UrlRewriterRuleInfo.PatternToMatch;
                this.serviceProxy.DeleteUrlRewriterRule(bag);
                base.ListView.Items.Remove(item1);
            }
            catch (Exception exception1)
            {
                base.DisplayErrorMessage(exception1, null);
                return;
            }
            finally
            {
                Cursor.Current = Cursors.Default;
            }
        }
    }
}

private void UpdateUrlRewriterRule()
{
    UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;
```

(Continued)

Listing 8-28: (continued)

```
        if (item1 != null)
        {
            UrlRewriterRuleInfo info1 = item1.UrlRewriterRuleInfo;
            using (UrlRewriterRuleTaskForm form1 =
                new UrlRewriterRuleTaskForm(base.Module, this.serviceProxy,
                                            info1.PatternToMatch,
                                            info1.Replacement))
            {
                if ((base.ShowDialog(form1) == DialogResult.OK) && form1.HasChanges)
                {
                    info1.PatternToMatch = form1.PatternToMatch;
                    info1.Replacement = form1.Replacement;
                    this.ReplaceItem(item1, info1);
                }
            }
        }
    }

    private void ReplaceItem(UrlRewriterRuleListViewItem item,
                           UrlRewriterRuleInfo urlRewriterRuleInfo)
    {
        base.ListView.Items.Remove(item);
        this.AddItem(urlRewriterRuleInfo, true);
    }

    private UrlRewriterRuleListViewItem SelectedUrlRewriterRule
    {
        get
        {
            if (base.ListView.SelectedItems.Count > 0)
                return (UrlRewriterRuleListViewItem)base.ListView.SelectedItems[0];

            return null;
        }
    }

    #region Nested Types

    private sealed class UrlRewriterRuleListViewItem : ListViewItem
    {
        public UrlRewriterRuleListViewItem(UrlRewriterRuleInfo urlRewriterRuleInfo)
            : base(urlRewriterRuleInfo.PatternToMatch)
        {
            this.urlRewriterRuleInfo = urlRewriterRuleInfo;
            base.SubItems.Add(new ListViewItem.ListViewSubItem(this,
                                                                urlRewriterRuleInfo.Replacement.ToString()));
        }

        public UrlRewriterRuleInfo UrlRewriterRuleInfo
        {
            get { return this.urlRewriterRuleInfo; }
        }
    }
}
```

Listing 8-28: (continued)

```
        private UrlRewriterRuleInfo urlRewriterRuleInfo;
    }

    private sealed class PageTaskList : TaskList
    {
        public PageTaskList(UrlRewriterPage owner)
        {
            this.owner = owner;
        }

        public void AddUrlRewriterRule()
        {
            this.owner.AddUrlRewriterRule();
        }

        public void DeleteUrlRewriterRule()
        {
            this.owner.DeleteUrlRewriterRule();
        }

        public void UpdateUrlRewriterRule()
        {
            this.owner.UpdateUrlRewriterRule();
        }

        public override ICollection GetTaskItems()
        {
            ArrayList list1 = new ArrayList();
            if (!owner.ReadOnly && !owner.errorGetUrlRewriterSettings)
            {
                list1.Add(new MethodTaskItem("AddUrlRewriterRule",
                                             "Add URL rewriter rule", "Add"));
                if (owner.SelectedUrlRewriterRule != null)
                {
                    list1.Add(new MethodTaskItem("UpdateUrlRewriterRule",
                                                 "Update URL rewriter rule", "Tasks"));
                    list1.Add(new MethodTaskItem("DeleteUrlRewriterRule",
                                                 "Delete URL rewriter rule", "Tasks"));
                }
            }
            if (owner.errorGetUrlRewriterSettings)
                list1.Add(new MessageTaskItem(MessageTaskItemType.Error, "Error",
                                             "Info", "Error"));

            foreach (TaskItem item2 in list1)
            {
                if (!(item2 is MessageTaskItem) && !(item2 is TextTaskItem))
                    item2.Enabled = !owner.InProgress;
            }
            return (TaskItem[])list1.ToArray(typeof(TaskItem));
        }
    }
```

(Continued)

Listing 8-28: *(continued)*

```
        private UrlRewriterPage owner;
    }

    #endregion
}
}
```

As Listing 8-28 shows, the `UrlRewriterPage` module page inherits from `ModuleListPage`, which in turn inherits from `ModulePage`. The following code listing presents those members of the `ModuleListPage` base class that `UrlRewriterPage` overrides:

```
public abstract class ModuleListPage : ModulePage
{
    protected abstract void InitializeListPage();
    protected override void OnActivated(bool initialActivation);
}
```

Listing 8-29 presents those members of the `ModulePage` base class that the `UrlRewriterPage` overrides.

Listing 8-29: The ModulePage Class

```
public abstract class ModulePage : ContainerControl, IModulePage, IDisposable
{
    protected virtual void Refresh();
    protected virtual TaskListCollection Tasks { get; }
    protected virtual bool CanRefresh { get; }
    protected virtual bool ReadOnly { get; }
}
```

Now let's get down to the implementation of the members of the `UrlRewriterPage` class.

InitializeListPage

Every module page that inherits from the `ModuleListPage` base class must implement the `InitializeListPage` method because this method is marked as abstract. Listing 8-30 contains the `UrlRewriterPage` module list page's implementation of the `InitializeListPage` method.

Listing 8-30: The InitializeListPage Method

```
protected override void InitializeListPage()
{
    patternToMatchColumnHeader = new ColumnHeader();
    patternToMatchColumnHeader.Text = "Pattern To Match";
    patternToMatchColumnHeader.Width = 90;
    replacementColumnHeader = new ColumnHeader();
    replacementColumnHeader.Text = "Replacement String";
    replacementColumnHeader.Width = 90;
    base.ListView.Columns.Clear();
}
```

Listing 8-30: (continued)

```
base.ListView.Columns.AddRange(new ColumnHeader[] { patternToMatchColumnHeader,
                                                    replacementColumnHeader });

base.ListView.LabelEdit = false;
base.ListView.MultiSelect = false;
base.ListView.SelectedIndexChanged +=
    new EventHandler(OnListViewSelectedIndexChanged);
base.ListView.DoubleClick += new EventHandler(OnListViewDoubleClick);
base.ListView.KeyUp += new KeyEventHandler(OnListViewKeyUp);
}
```

InitializeListPage creates two columns to display the values of the patternToMatch and replacement attributes of the URL rewriter rules and adds them to the list of the columns. It then registers the OnListViewSelectedIndexChanged, OnListViewDoubleClick, and OnListViewKeyUp methods as event handlers for the SelectedIndexChanged, DoubleClick, and KeyUp events, respectively:

```
base.ListView.SelectedIndexChanged +=
    new EventHandler(OnListViewSelectedIndexChanged);
base.ListView.DoubleClick += new EventHandler(OnListViewDoubleClick);
base.ListView.KeyUp += new KeyEventHandler(OnListViewKeyUp);
```

OnActivated

The OnActivated method is invoked when the UrlRewriterPage module list page is accessed (see Listing 8-31). If the module page is being accessed for the first time, the method calls the CreateProxy method of the ModulePage base class to instantiate an instance of the UrlRewriterModuleServiceProxy proxy class and stores this instance in a private field named serviceProxy for future reference. Note that OnActivated calls the GetUrlRewriterSettings method to retrieve the URL rewriter rules and the value of the isLocked attribute from the server.

Listing 8-31: The OnActivated Method

```
protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (UrlRewriterModuleServiceProxy)base.CreateProxy(
                                                                    typeof(UrlRewriterModuleServiceProxy));
        this.GetUrlRewriterSettings();
    }
}
```

GetUrlRewriterSettings

The GetUrlRewriterSettings method calls the StartAsyncTask method of the base class and passes two delegates into it (see Listing 8-32). The first delegate is a DoWorkEventHandler delegate that wraps a method named OnWorkerGetUrlRewriterSettings. The second delegate is a RunWorkerCompletedEventHandler delegate that encapsulates a method named OnWorkerGetUrlRewriterSettingsCompleted.

Listing 8-32: The `GetUrlRewriterSettings` Method

```
private void GetUrlRewriterSettings()
{
    base.StartAsyncTask("Getting URL rewriter settings",
        new DoWorkEventHandler(this.OnWorkerGetUrlRewriterSettings),
        new RunWorkerCompletedEventHandler(
            this.OnWorkerGetUrlRewriterSettingsCompleted));
}
```

OnWorkerGetUrlRewriterSettings

The `OnWorkerGetUrlRewriterSettings` method simply calls the `GetUrlRewriterSettings` method of the proxy to retrieve the URL rewriter rules and the `isLocked` attribute value from the server (see Listing 8-33). As discussed earlier, the `GetUrlRewriterSettings` method of the proxy calls the `GetUrlRewriterSettings` method of the server-side class.

Listing 8-33: The `OnWorkerGetUrlRewriterSettings` Method

```
private void OnWorkerGetUrlRewriterSettings(object sender, DoWorkEventArgs e)
{
    e.Result = this.serviceProxy.GetUrlRewriterSettings();
}
```

OnWorkerGetUrlRewriterSettingsCompleted

The `OnWorkerGetUrlRewriterSettingsCompleted` method is automatically called right after the URL rewriter settings are retrieved from the server (see Listing 8-34).

Listing 8-34: The `OnWorkerGetUrlRewriterSettingsCompleted` Method

```
private void OnWorkerGetUrlRewriterSettingsCompleted(object sender,
                                                    RunWorkerCompletedEventArgs e)
{
    base.ListView.BeginUpdate();
    try
    {
        if (e.Result != null)
        {
            base.ListView.Items.Clear();
            this.bag = (PropertyBag)e.Result;
            ArrayList list1 = (ArrayList)this.bag[0];
            this.readOnly = (bool)this.bag[1];

            if (list1 != null)
            {
                for (int num1 = 0; num1 < list1.Count; num1++)
                {
                    UrlRewriterRuleInfo info1 =
                        new UrlRewriterRuleInfo((PropertyBag)list1[num1]);
                    this.AddItem(info1, false);
                }
            }
        }
    }
}
```

Listing 8-34: (continued)

```
        this.errorGetUrlRewriterSettings = false;
    }
}

catch (Exception exception1)
{
    base.StopProgress();
    base.DisplayErrorMessage(exception1.Message,
                            "GetUrlRewriterSettingsCompleted");
    this.errorGetUrlRewriterSettings = true;
    return;
}

finally
{
    base.ListView.EndUpdate();
}
}
```

This method first clears the list of displayed URL rewriter rules:

```
base.ListView.Items.Clear();
```

Then, it stores the `PropertyBag` that contains the retrieved URL rewriter settings in a private `PropertyBag` field named `bag` for future reference:

```
this.bag = (PropertyBag)e.Result;
```

As you'll see later, the `GetUrlRewriterSettings` method of the server-side class creates one `PropertyBag` object for each URL rewriter rule, stores the values of the `patternToMatch` and `replace` attributes of the URL rewriter rule in this `PropertyBag`, and adds the `PropertyBag` object into an `ArrayList`. It then creates a `PropertyBag` object, stores the `ArrayList` as the first element in this `PropertyBag`, stores the value of the `isLocked` attribute as the second element in this `PropertyBag`, and sends this `PropertyBag` to the client. Therefore, the `OnWorkerGetUrlRewriterSettingsCompleted` method first accesses the `ArrayList` and the `isLocked` attribute value.

```
ArrayList list1 = (ArrayList)this.bag[0];
this.readOnly = (bool)this.bag[1];
```

Note that the method assigns the value of the `isLocked` attribute to the `readOnly` Boolean field of the `UrlRewriterPage` module page. This module page overrides the `ReadOnly` property that it inherits from the `ModulePage` base class to return the value of the `readOnly` Boolean field. The `ReadOnly` property specifies whether the `UrlRewriterPage` module page is editable.

```
protected override bool ReadOnly
{
    get { return this.readOnly; }
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Now back to the implementation of the `OnWorkerGetUrlRewriterSettingsCompleted` method. The method then iterates through the `PropertyBag` objects in the `ArrayList`, creates one `UrlRewriterRuleInfo` object for each enumerated `PropertyBag` object, and calls the `AddItem` passing in the `UrlRewriterRuleInfo` object. I discuss the `UrlRewriterRuleInfo` class and `AddItem` in the next sections.

```
if (list1 != null)
{
    for (int num1 = 0; num1 < list1.Count; num1++)
    {
        UrlRewriterRuleInfo info1 = new
            UrlRewriterRuleInfo((PropertyBag)list1[num1]);
        this.AddItem(info1, false);
    }
}
```

UrlRewriterRuleInfo

The `UrlRewriterRuleInfo` class exposes the contents of the `PropertyBag` object passed into its constructor as strongly-typed properties (see Listing 8-35) to allow us to benefit from the Visual Studio IntelliSense support, the compiler's type-checking support, and well-known benefits of object-oriented programming. Now add a new source file named `UrlRewriterRuleInfo.cs` to the `Client` subdirectory of the `GraphicalManagement` directory and add the code shown in Listing 8-35 to this source file.

Listing 8-35: The `UrlRewriterRuleInfo` Class

```
using Microsoft.Web.Management.Server;

namespace UrlRewriting.GraphicalManagement.Client
{
    internal sealed class UrlRewriterRuleInfo
    {
        internal UrlRewriterRuleInfo(PropertyBag bag)
        {
            this.bag = bag;
        }

        public string PatternToMatch
        {
            get { return (string)this.bag[0]; }
            set { this.bag[0] = value; }
        }

        public string Replacement
        {
            get { return (string)this.bag[1]; }
            set { this.bag[1] = value; }
        }

        private PropertyBag bag;
    }
}
```

UrlRewriterRuleListViewItem

When you write a custom module page that inherits from the `ModuleListPage` base class, you must also implement a class that inherits the `ListViewItem` class. This class must be private and nested within your custom module page (see Listing 8-36). The instances of this class will represent the items that your custom module page displays in its user interface.

Following this pattern, Listing 8-36 implements a class named `UrlRewriterRuleListViewItem` that inherits from the `ListViewItem` class. The instances of this class will display the URL rewriter rules that the `UrlRewriterPage` module list page will display. Note that the constructor of this class passes the value of the `PatternToMatch` property of the `UrlRewriterRuleInfo` object passed into it to the constructor of the `ListViewItem` base class. The base class uses this value to uniquely identify each URL rewriter rule in the list of displayed URL rewriter rules.

Listing 8-36: The UrlRewriterRuleListViewItem Class

```
public class UrlRewriterPage : ModuleListPage
{
    . . .
    private sealed class UrlRewriterRuleListViewItem : ListViewItem
    {
        public UrlRewriterRuleListViewItem(UrlRewriterRuleInfo urlRewriterRuleInfo)
            : base(urlRewriterRuleInfo.PatternToMatch)
        {
            this.urlRewriterRuleInfo = urlRewriterRuleInfo;
            base.SubItems.Add(new ListViewItem.ListViewSubItem(this,
                urlRewriterRuleInfo.Replacement.ToString()));
        }

        public UrlRewriterRuleInfo UrlRewriterRuleInfo
        {
            get { return this.urlRewriterRuleInfo; }
        }

        private UrlRewriterRuleInfo urlRewriterRuleInfo;
    }
}
```

AddItem

Recall from Listing 8-34 that the `OnWorkerGetUrlRewriterSettingsCompleted` method iterates through the `ArrayList` of `PropertyBag` objects that it has received from the server, creates a `UrlRewriterRuleInfo` object for each enumerated `PropertyBag` object, and calls the `AddItem` method, passing in the `UrlRewriterRuleInfo` object. As you can see from Listing 8-37, the main responsibility of this method is to create a `UrlRewriterRuleListViewItem` to represent the associated `UrlRewriterRuleInfo` object and add this `UrlRewriterRuleListViewItem` list view item to the `Items` collection of the list view. This collection contains the list view items that represent the displayed items.

Listing 8-37: The AddItem Method

```
private void AddItem(UrlRewriterRuleInfo urlRewriterRuleInfo, bool isSelected)
{
    UrlRewriterRuleListViewItem item1 =
        new UrlRewriterRuleListViewItem(urlRewriterRuleInfo);
    base.ListView.Items.Add(item1);

    if (isSelected)
    {
        item1.Selected = true;
        item1.Focused = true;
        base.ListView.EnsureVisible(base.ListView.Items.IndexOf(item1));
    }
}
```

Adding Support for New Task Items

Next, you need to add three new links titled “Add URL rewriter rule,” “Update URL rewriter rule,” and “Delete URL rewriter rule” to the task panel associated with the `UrlRewriterPage` module page (see Figure 8-27). As discussed in previous chapters, it takes three steps to add these new links:

1. Implement a class named `PageTaskList` that inherits from the `TaskList` base class. This class must be private and nested within the `UrlRewriterPage` module list page.
2. Override the `Tasks` property of the `UrlRewriterPage` module list page.
3. Implement an event handler associated with each new link.

PageTaskList

Listing 8-38 presents the implementation of the `PageTaskList` class. Note that this class is a nested type within the `UrlRewriterPage` module page.

Listing 8-38: The PageTaskList Class

```
namespace UrlRewriting.GraphicalManagement.Client
{
    public class UrlRewriterPage : ModuleListPage
    {
        . . .
        private sealed class PageTaskList : TaskList
        {
            public PageTaskList(UrlRewriterPage owner)
            {
                this.owner = owner;
            }

            public void AddUrlRewriterRule()
            {
                this.owner.AddUrlRewriterRule();
            }
        }
    }
}
```

Listing 8-38: (continued)

```
public void DeleteUrlRewriterRule()
{
    this.owner.DeleteUrlRewriterRule();
}

public void UpdateUrlRewriterRule()
{
    this.owner.UpdateUrlRewriterRule();
}

public override ICollection GetTaskItems()
{
    ArrayList list1 = new ArrayList();
    if (!owner.ReadOnly && !owner.errorGetUrlRewriterSettings)
    {
        list1.Add(new MethodTaskItem("AddUrlRewriterRule",
                                     "Add URL rewriter rule", "Add"));
        if (owner.SelectedUrlRewriterRule != null)
        {
            list1.Add(new MethodTaskItem("UpdateUrlRewriterRule",
                                         "Update URL rewriter rule", "Tasks"));
            list1.Add(new MethodTaskItem("DeleteUrlRewriterRule",
                                         "Delete URL rewriter rule", "Tasks"));
        }
    }

    if (owner.errorGetUrlRewriterSettings)
        list1.Add(new MessageTaskItem(MessageTaskItemType.Error, "Error",
                                       "Info", "Error"));

    foreach (TaskItem item2 in list1)
    {
        if (!(item2 is MessageTaskItem) && !(item2 is TextTaskItem))
            item2.Enabled = !owner.InProgress;
    }
    return (TaskItem[])list1.ToArray(typeof(TaskItem));
}

private UrlRewriterPage owner;
}
}
```

As Listing 8-38 shows, the `PageTaskList` task list overrides the `GetTaskItems` method that it inherits from the `TaskList` base class. Next, I walk through the implementation of the `GetTaskItems` method. This method first creates an `ArrayList`:

```
ArrayList list1 = new ArrayList();
```


Chapter 8: Extending the Integrated Request Processing Pipeline

Next, it checks whether both of the following conditions are met:

- ❑ The `UrlRewriterPage` module list page that owns the `PageTaskList` task list is editable. Recall that the `ReadOnly` property of the `UrlRewriterPage` module page simply returns the value of the `isLocked` attribute on the `<urlRewriter>` configuration section in the underlying configuration file. The user sets the value of this attribute to true to lock the `<urlRewriter>` configuration section.
- ❑ The `UrlRewriterPage` module list page that owns the `PageTaskList` task list did not have problems downloading the URL rewriter settings from the server in the first place.

If both of these conditions are met, the `GetTaskItems` method creates a `MethodTaskItem` to represent the “Add URL rewriter rule” link. Note that it specifies the `AddUrlRewriterRule` method as the event handler for this link. As Listing 8-38 shows, this method calls the `AddUrlRewriterRule` method of the `UrlRewriterPage` module list page:

```
list1.Add(new MethodTaskItem("AddUrlRewriterRule",  
                             "Add URL rewriter rule", "Add"));
```

The `GetTaskItems` method then checks whether the user has selected a URL rewriter rule from the list of displayed URL rewriter rules. If so, it creates two more `MethodTaskItems` to represent the “Update URL rewriter rule” and “Delete URL rewriter rule” links. This means that these two links are rendered only when an item is selected from the list. Notice that these two method task items respectively register the `UpdateUrlRewriterRule` and `DeleteUrlRewriterRule` methods as event handlers for these two links. As Listing 8-38 shows, these two methods call the `UpdateUrlRewriterRule` and `DeleteUrlRewriterRule` methods of the `UrlRewriterPage` module page, respectively.

```
if (owner.SelectedUrlRewriterRule != null)  
{  
    list1.Add(new MethodTaskItem("UpdateUrlRewriterRule",  
                                "Update URL rewriter rule", "Tasks"));  
    list1.Add(new MethodTaskItem("DeleteUrlRewriterRule",  
                                "Delete URL rewriter rule", "Tasks"));  
}
```

`GetTaskItems` then checks whether the `UrlRewriterPage` module list page had trouble retrieving the URL rewriter settings from the server. If so, it creates a `MessageTaskItem` to display an error message in the Alerts panel.

```
if (owner.errorGetUrlRewriterSettings)  
    list1.Add(new MessageTaskItem(MessageTaskItemType.Error, "Error", "Info", "Error"));
```

Tasks

The `UrlRewriterPage` module page overrides the `Tasks` property, as shown in Listing 8-39. It first instantiates the `PageTaskList` if it hasn’t already been instantiated. Next, it adds this `PageTaskList` object to the `Tasks` collection property of its base class.

Listing 8-39: The Tasks Property

```
protected override TaskListCollection Tasks  
{
```

Listing 8-39: (continued)

```
get
{
    if (this.taskList == null)
        this.taskList = new PageTaskList(this);

    TaskListCollection collection1 = base.Tasks;
    collection1.Add(this.taskList);
    return collection1;
}
}
```

AddUrlRewriterRule

Listing 8-40 contains the implementation of the AddUrlRewriterRule method of the UrlRewriterPage module list page.

Listing 8-40: The AddUrlRewriterRule Method

```
private void AddUrlRewriterRule()
{
    using (UrlRewriterRuleTaskForm form1 =
        new UrlRewriterRuleTaskForm(base.Module, this.serviceProxy))
    {
        if (base.ShowDialog(form1) == DialogResult.OK)
        {
            PropertyBag bag1 = new PropertyBag();
            bag1[0] = form1.PatternToMatch;
            bag1[1] = form1.Replacement;
            this.AddItem(new UrlRewriterRuleInfo(bag1), true);
        }
    }
}
```

This method first instantiates and launches a UrlRewriterRuleTaskForm task form to allow user to specify the patternToMatch and replacement attributes of the URL rewriter rule being added. As you'll see later, when the user clicks the OK button on the task form, the event handler for the button uses the proxy to add the new URL rewriter rule to the underlying configuration file. When the task form returns, AddUrlRewriterRule takes these steps to add the new URL rewriter rule to the list of displayed URL rewriter rules:

1. Creates a PropertyBag:

```
PropertyBag bag1 = new PropertyBag();
```

2. Populates the PropertyBag with the patternToMatch and replacement attributes of the new URL rewriter rule. As you'll see later, the UrlRewriterRuleTaskForm task form exposes these two values as strongly-typed PatternToMatch and Replacement properties.

```
bag1[0] = form1.PatternToMatch;
bag1[1] = form1.Replacement;
```

Chapter 8: Extending the Integrated Request Processing Pipeline

3. Creates a `UrlRewriterRuleInfo` object, passing in the `PropertyBag`. Recall that the `UrlRewriterRuleInfo` object exposes the content of the `PropertyBag` as strongly-typed properties. Finally it calls the `AddItem` method to add the new URL rewriter rule to the list of displayed URL rewriter rules:

```
this.AddItem(new UrlRewriterRuleInfo(bag1), true);
```

DeleteUrlRewriterRule

Listing 8-41 contains the code for the `DeleteUrlRewriterRule` method.

Listing 8-41: The `DeleteUrlRewriterRule` Method

```
private void DeleteUrlRewriterRule()
{
    UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;
    if (item1 != null)
    {
        DialogResult result1 = base.ShowMessage(
            "Do you really want to delete this URL rewriter rule?",
            MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Question, MessageBoxDefaultButton.Button1, "Removed");
        if (result1 == DialogResult.Yes)
        {
            try
            {
                Cursor.Current = Cursors.WaitCursor;
                PropertyBag bag = new PropertyBag();
                bag[0] = item1.UrlRewriterRuleInfo.PatternToMatch;
                this.serviceProxy.DeleteUrlRewriterRule(bag);
                base.ListView.Items.Remove(item1);
            }
            catch (Exception exception1)
            {
                base.DisplayErrorMessage(exception1, null);
                return;
            }
            finally
            {
                Cursor.Current = Cursors.Default;
            }
        }
    }
}
```

This method first accesses the `UrlRewriterRuleListViewItem` object that represents the selected URL rewriter rule:

```
UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;
```

Chapter 8: Extending the Integrated Request Processing Pipeline

It then launches a message box to double-check whether the end user indeed wants to delete the selected URL rewriter rule from the underlying configuration file:

```
DialogResult result1 = base.ShowMessage(  
    "Do you really want to delete this URL rewriter rule?",  
    MessageBoxButtons.YesNoCancel,  
    MessageBoxIcon.Question, MessageBoxDefaultButton.Button1, "Removed");
```

If the user confirms the deletion, the `DeleteUrlRewriterRule` takes these steps:

1. Creates a `PropertyBag`:

```
PropertyBag bag = new PropertyBag();
```

2. Adds the pattern to match of the URL rewriter rule being deleted to the `PropertyBag`:

```
bag[0] = item1.ItemInfo.PatternToMatch;
```

3. Calls the `DeleteUrlRewriterRule` method of the proxy, passing in the `PropertyBag` to delete the URL rewriter rule from the underlying configuration file:

```
this.serviceProxy.DeleteUrlRewriterRule(bag);
```

4. Removes the deleted URL rewriter rule from the list of displayed URL rewriter rules:

```
base.ListView.Items.Remove(item1);
```

UpdateUrlRewriterRule

Listing 8-42 presents the implementation of the `UpdateUrlRewriterRule` method of the `UrlRewriterPage` module page.

Listing 8-42: The `UpdateUrlRewriterRule` Method

```
private void UpdateUrlRewriterRule()  
{  
    UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;  
    if (item1 != null)  
    {  
        UrlRewriterRuleInfo info1 = item1.UrlRewriterRuleInfo;  
        using (UrlRewriterRuleTaskForm form1 =  
            new UrlRewriterRuleTaskForm(base.Module, this.serviceProxy,  
                info1.PatternToMatch,  
                info1.Replacement))  
        {  
            if ((base.ShowDialog(form1) == DialogResult.OK) && form1.HasChanges)  
            {  
                info1.PatternToMatch = form1.PatternToMatch;  
                info1.Replacement = form1.Replacement;  
                this.ReplaceItem(item1, info1);  
            }  
        }  
    }  
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

`UpdateUrlRewriterRule` first accesses the `UrlRewriterRuleListViewItem` object that represents the selected URL rewriter rule. Recall that the users must first select the URL rewriter rule that they want to update:

```
UrlRewriterRuleListViewItem item1 = this.SelectedUrlRewriterRule;
```

Then, it launches the `UrlRewriterRuleTaskForm` task form to allow the user to update the pattern to match and replacement string of the URL rewriter rule being updated:

```
using (UrlRewriterRuleTaskForm form1 =  
    new UrlRewriterRuleTaskForm(base.Module, this.serviceProxy,  
                                info1.PatternToMatch,  
                                info1.Replacement))
```

As you'll see later, after the end user updates the values and clicks the OK button on the task form, the event handler for this button calls the `UpdateUrlRewriterRule` method of the proxy to update the values of the corresponding URL rewriter rule in the underlying configuration file. If the end user has indeed changed the current values, `UpdateUrlRewriterRule` updates the selected URL rewriter rule in the list of displayed URL rewriter rules in the `UrlRewriterPage` module page:

```
info1.PatternToMatch = form1.PatternToMatch;  
info1.Replacement = form1.Replacement;  
this.ReplaceItem(item1, info1);
```

Here is the implementation of the `ReplaceItem` method of the `UrlRewriterPage` module page:

```
private void ReplaceItem(UrlRewriterRuleListViewItem item,  
                        UrlRewriterRuleInfo urlRewriterRuleInfo)  
{  
    base.ListView.Items.Remove(item);  
    this.AddItem(urlRewriterRuleInfo, true);  
}
```

OnListViewSelectedIndexChanged

The following code listing presents the implementation of the `OnListViewSelectedIndexChanged` method that the `UrlRewriterPage` module page registers as the event handler for the `SelectedIndexChanged` event:

```
private void OnListViewSelectedIndexChanged(object sender, EventArgs e)  
{  
    base.Update();  
}
```

OnListViewDoubleClick

The following code fragment contains the implementation of the `OnListViewDoubleClick` method. Recall from Listing 8-28 that the `UrlRewriterPage` module page registers this method as an event handler for the `DoubleClick` event:

```
private void OnListViewDoubleClick(object sender, EventArgs e)  
{
```

```
if ((this.SelectedUrlRewriterRule != null) && !this.ReadOnly)
    this.UpdateUrlRewriterRule();
}
```

As you can see, `OnListViewDoubleClick` first checks whether both of the following conditions are met:

- ❑ The user has selected a URL rewriter rule from the list of displayed URL rewriter rules. This condition is bound to be met because double-clicking an item automatically selects the item.
- ❑ The `ReadOnly` property returns false. Recall that this property reflects the value of the `isLocked` attribute on the underlying `<urlRewriter>` configuration section.

If both of these conditions are met, `OnListViewDoubleClick` invokes the `UpdateUrlRewriterRule` method, which was discussed earlier.

OnListViewKeyUp

The following code listing presents the implementation of the `OnListViewKeyUp` method, which the `UrlRewriterPage` module page registers for the `KeyUp` event (see Listing 8-28):

```
private void OnListViewKeyUp(object sender, KeyEventArgs e)
{
    if ((this.SelectedUrlRewriterRule != null) && (e.KeyData == Keys.Delete))
        this.DeleteUrlRewriterRule();
}
```

This method first checks whether both of the following conditions are met:

- ❑ The end user has selected a URL rewriter rule from the list of displayed URL rewriter rules because the end users must first select the desired URL rewriter rule before they can delete it.
- ❑ The end user has clicked the Delete button.

If both of these conditions are met, `OnListViewKeyUp` invokes the `DeleteUrlRewriterRule` method to delete the selected URL rewriter rule.

UrlRewriterRuleTaskForm

Listing 8-43 presents the implementation of the `UrlRewriterRuleTaskForm` task form. I walk through the implementation of this task form in the following sections. Add a new source file named `UrlRewriterRuleTaskForm.cs` to the `GraphicalManagement/Client` directory and add the code shown in Listing 8-43 to this source file. You also need to add a reference to `System.Drawing.dll` assembly to the `UrlRewriterProj2` Class Library project.

Listing 8-43: The `UrlRewriterRuleTaskForm` Class

```
using Microsoft.Web.Management.Client.Win32;
using Microsoft.Web.Management.Server;
using System.Windows.Forms;
using System.ComponentModel;
```

(Continued)

Listing 8-43: (continued)

```
using System;
using System.Drawing;

namespace UrlRewriting.GraphicalManagement.Client
{
    internal sealed class UrlRewriterRuleTaskForm : TaskForm
    {
        private string originalPatternToMatch;
        private string patternToMatch;
        private string replacement;
        private bool inModificationMode;
        private TextBox patternToMatchTextBox;
        private TextBox replacementTextBox;
        private bool hasChanges;
        private UrlRewriterModuleServiceProxy serviceProxy;
        private ManagementPanel contentPanel;
        private Label patternToMatchLabel;
        private Label replacementLabel;

        public bool HasChanges
        {
            get { return this.hasChanges; }
        }

        public string PatternToMatch
        {
            get { return this.patternToMatch; }
        }

        public string Replacement
        {
            get { return this.replacement; }
        }

        public UrlRewriterRuleTaskForm(IServiceProvider serviceProvider,
                                       UrlRewriterModuleServiceProxy proxy)
            : this(serviceProvider, proxy, string.Empty, string.Empty) { }

        public UrlRewriterRuleTaskForm(IServiceProvider serviceProvider,
                                       UrlRewriterModuleServiceProxy proxy,
                                       string patternToMatch,
                                       string replacement)
            : base(serviceProvider)
        {
            this.serviceProxy = proxy;
            InitializeComponent();
            this.inModificationMode = !string.IsNullOrEmpty(patternToMatch);
            if (this.inModificationMode)
            {
                this.originalPatternToMatch = patternToMatch;
                this.patternToMatchTextBox.Text = patternToMatch;
                this.replacementTextBox.Text = replacement;
            }
        }
    }
}
```

Listing 8-43: (continued)

```
        this.Text = "Update URL rewriter rule";
    }
    else
        this.Text = "Add URL rewriter rule";

    UpdateUIState();
    this.hasChanges = false;
}

private void InitializeComponent()
{
    this.contentPanel = new ManagementPanel();
    this.patternToMatchLabel = new Label();
    this.patternToMatchTextBox = new TextBox();
    this.replacementLabel = new Label();
    this.replacementTextBox = new TextBox();
    this.contentPanel.SuspendLayout();
    base.SuspendLayout();
    this.contentPanel.Controls.Add(patternToMatchLabel);
    this.contentPanel.Controls.Add(patternToMatchTextBox);
    this.contentPanel.Controls.Add(replacementLabel);
    this.contentPanel.Controls.Add(replacementTextBox);

    this.contentPanel.Dock = DockStyle.Fill;
    this.contentPanel.Location = new Point(0, 0);
    this.contentPanel.Name = "contentPanel";
    this.contentPanel.Size = new Size(0x114, 110);
    this.contentPanel.TabIndex = 0;
    this.patternToMatchLabel.Location = new Point(0, 0);
    this.patternToMatchLabel.Name = "_nameLabel";
    this.patternToMatchLabel.AutoSize = true;
    this.patternToMatchLabel.TabIndex = 0;
    this.patternToMatchLabel.TextAlign = ContentAlignment.MiddleLeft;
    this.patternToMatchLabel.Text = "Pattern To Match";
    this.patternToMatchTextBox.Anchor =
        AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
    this.patternToMatchTextBox.Location = new Point(0, 0x10);
    this.patternToMatchTextBox.Name = "_nameTextBox";
    this.patternToMatchTextBox.Size = new Size(0x114, 0x15);
    this.patternToMatchTextBox.TabIndex = 1;
    this.patternToMatchTextBox.TextChanged +=
        new EventHandler(this.OnPatternToMatchTextBoxTextChanged);

    this.replacementLabel.Location = new Point(0, 0x3b);
    this.replacementLabel.Name = "_valueTextBox";
    this.replacementLabel.Text = "Replacement";
    this.replacementLabel.AutoSize = true;
    this.replacementLabel.TabIndex = 0;
    this.replacementLabel.TextAlign = ContentAlignment.MiddleLeft;
    this.replacementTextBox.Location = new Point(0, 0x4c);
    this.replacementTextBox.Name = "_nameTextBox";
    this.replacementTextBox.Size = new Size(0x114, 0x15);
```

(Continued)

Listing 8-43: (continued)

```
this.replacementTextBox.TabIndex = 2;
this.replacementTextBox.Anchor =
    AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
this.replacementTextBox.TextChanged +=
    new EventHandler(this.OnReplacementTextBoxTextChanged);
base.ClientSize = new Size(300, 160);
base.AutoScaleMode = AutoScaleMode.Font;
base.Name = "UrlRewriterRuleTaskForm";
this.contentPanel.ResumeLayout(false);
this.contentPanel.PerformLayout();
base.SetContent(contentPanel);
base.ResumeLayout(false);
}

private void OnPatternToMatchTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void OnReplacementTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void UpdateUIState()
{
    this.hasChanges = true;
}

protected override void OnAccept()
{
    this.patternToMatch = this.patternToMatchTextBox.Text.Trim();
    this.replacement = this.replacementTextBox.Text.Trim();
    base.StartAsyncTask(new DoWorkEventHandler(this.OnWorkerDoWork),
        new RunWorkerCompletedEventHandler(this.OnWorkerCompleted));
    base.UpdateTaskForm();
}

private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
{
    if (this.hasChanges)
    {
        PropertyBag bag = new PropertyBag();
        if (!this.inModificationMode)
        {
            bag[0] = this.patternToMatch;
            bag[1] = this.replacement;
            this.serviceProxy.AddUrlRewriterRule(bag);
        }
    }
    else
    {

```

Listing 8-43: (continued)

```
        bag[0] = this.originalPatternToMatch;
        bag[1] = this.patternToMatch;
        bag[2] = this.replacement;
        this.serviceProxy.UpdateUrlRewriterRule(bag);
    }
}

private void OnWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    base.UpdateTaskForm();
    if (e.Error != null)
        this.DisplayErrorMessage(e.Error, null);

    else
    {
        base.DialogResult = DialogResult.OK;
        base.Close();
    }
}
}
```

Constructors

The following code listing contains the implementation of the constructors of `UrlRewriterRuleTaskForm` task form:

```
public UrlRewriterRuleTaskForm(IServiceProvider serviceProvider,
                               UrlRewriterModuleServiceProxy proxy)
    : this(serviceProvider, proxy, string.Empty, string.Empty) { }

public UrlRewriterRuleTaskForm(IServiceProvider serviceProvider,
                               UrlRewriterModuleServiceProxy proxy,
                               string patternToMatch,
                               string replacement)
    : base(serviceProvider)
{
    this.serviceProxy = proxy;
    InitializeComponent();
    this.inModificationMode = !string.IsNullOrEmpty(patternToMatch);
    if (this.inModificationMode)
    {
        this.originalPatternToMatch = patternToMatch;
        this.patternToMatchTextBox.Text = patternToMatch;
        this.replacementTextBox.Text = replacement;
        this.Text = "Update URL rewriter rule";
    }
    else
        this.Text = "Add URL rewriter rule";
}
```

```
UpdateUIState();
this.hasChanges = false;
}
```

As you can see, the first constructor delegates to the second constructor. The second constructor calls the `InitializeComponent` method to create the user interface of the `UrlRewriterRuleTaskForm` task form. Because the same task form is used for both updating and adding a URL rewriter rule, the value of the `patternToMatch` parameter is used to determine whether the end user is trying to update or add a URL rewriter rule. If the user is updating a URL rewriter rule, the constructor initializes the user interface of the task form with the current values of the pattern to match and replacement string of the URL rewriter rule being updated:

```
patternToMatchTextBox.Text = patternToMatch;
replacementTextBox.Text = replacement;
```

InitializeComponent

The main responsibility of the `InitializeComponent` method is to create the user interface of the `UrlRewriterRuleTaskForm` task form (see Listing 8-44).

Listing 8-44: The `InitializeComponent` Method

```
private void InitializeComponent()
{
    this.contentPanel = new ManagementPanel();
    this.patternToMatchLabel = new Label();
    this.patternToMatchTextBox = new TextBox();
    this.replacementLabel = new Label();
    this.replacementTextBox = new TextBox();
    this.contentPanel.SuspendLayout();
    base.SuspendLayout();
    this.contentPanel.Controls.Add(patternToMatchLabel);
    this.contentPanel.Controls.Add(patternToMatchTextBox);
    this.contentPanel.Controls.Add(replacementLabel);
    this.contentPanel.Controls.Add(replacementTextBox);

    this.contentPanel.Dock = DockStyle.Fill;
    this.contentPanel.Location = new Point(0, 0);
    this.contentPanel.Name = "contentPanel";
    this.contentPanel.Size = new Size(0x114, 110);
    this.contentPanel.TabIndex = 0;
    this.patternToMatchLabel.Location = new Point(0, 0);
    this.patternToMatchLabel.Name = "_nameLabel";
    this.patternToMatchLabel.AutoSize = true;
    this.patternToMatchLabel.TabIndex = 0;
    this.patternToMatchLabel.TextAlign = ContentAlignment.MiddleLeft;
    this.patternToMatchLabel.Text = "Pattern To Match";
    this.patternToMatchTextBox.Anchor =
        AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
    this.patternToMatchTextBox.Location = new Point(0, 0x10);
    this.patternToMatchTextBox.Name = "_nameTextBox";
    this.patternToMatchTextBox.Size = new Size(0x114, 0x15);
    this.patternToMatchTextBox.TabIndex = 1;
```

Listing 8-44: (continued)

```
this.patternToMatchTextBox.TextChanged +=
    new EventHandler(this.OnPatternToMatchTextBoxTextChanged);

this.replacementLabel.Location = new Point(0, 0x3b);
this.replacementLabel.Name = "_valueTextBox";
this.replacementLabel.Text = "Replacement";
this.replacementLabel.AutoSize = true;
this.replacementLabel.TabIndex = 0;
this.replacementLabel.TextAlign = ContentAlignment.MiddleLeft;
this.replacementTextBox.Location = new Point(0, 0x4c);
this.replacementTextBox.Name = "_nameTextBox";
this.replacementTextBox.Size = new Size(0x114, 0x15);
this.replacementTextBox.TabIndex = 2;
this.replacementTextBox.Anchor =
    AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top;
this.replacementTextBox.TextChanged +=
    new EventHandler(this.OnReplacementTextBoxTextChanged);
base.ClientSize = new Size(300, 160);
base.AutoScaleMode = AutoScaleMode.Font;
base.Name = "UrlRewriterRuleTaskForm";
this.contentPanel.ResumeLayout(false);
this.contentPanel.PerformLayout();
base.SetContent(contentPanel);
base.ResumeLayout(false);
}
```

`InitializeComponent` uses a `ManagementPanel` control as the container for the entire user interface of the task form. The `ManagementPanel` is a scrollable panel control.

As Listing 8-44 shows, the `InitializeComponent` method instantiates the `ManagementPanel` control and adds two labels and two textbox controls. The textbox controls are used to display or specify the pattern to match and replacement string of the associated URL rewriter rule.

`InitializeComponent` then registers the `OnPatternToMatchTextBoxTextChanged` and `OnReplacementTextBoxTextChanged` methods as event handlers for the `TextChanged` events of the `patternToMatchTextBox` and `replacementTextBox` textboxes, respectively:

```
this.patternToMatchTextBox.TextChanged +=
    new EventHandler(this.OnPatternToMatchTextBoxTextChanged);

this.replacementTextBox.TextChanged +=
    new EventHandler(this.OnReplacementTextBoxTextChanged);
```

The following code listing presents the implementation of these two methods:

```
private void OnPatternToMatchTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}
```

Chapter 8: Extending the Integrated Request Processing Pipeline

```
private void OnReplacementTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}
```

As you can see, these two methods simply call the `UpdateUIState` method:

```
private void UpdateUIState()
{
    this.hasChanges = true;
}
```

In this case, the `UpdateUIState` method only sets the `hasChanged` field to `true` to specify that the task form has changes to store in the underlying configuration file. However, you may have a situation where the other GUI elements in your task form may have to be informed when the state of another GUI element changes. In these situations, the `UpdateUIState` method must also contain the logic that informs other GUI elements of the new changes.

OnAccept

Recall that `UrlRewriterRuleTaskForm` inherits from the `TaskForm` base class. This base class exposes an overridable method named `OnAccept` that its subclasses must override to add the code that they want to run when the user clicks the OK button of the task form. Listing 8-45 presents the `UrlRewriterRuleTaskForm` class's implementation of the `OnAccept` method.

Listing 8-45: The `OnAccept` Method

```
protected override void OnAccept()
{
    this.patternToMatch = this.patternToMatchTextBox.Text.Trim();
    this.replacement = this.replacementTextBox.Text.Trim();
    base.StartAsyncTask(new DoWorkEventHandler(this.OnWorkerDoWork),
                       new RunWorkerCompletedEventHandler(this.OnWorkerCompleted));
    base.UpdateTaskForm();
}
```

`OnAccept` first retrieves the new values of the pattern to match and replacement string of the associated URL rewriter rule from the `patternToMatchTextBox` and `replacementTextBox` textbox controls and respectively stores them in the `patternToMatch` and `replacement` fields for future reference:

```
this.patternToMatch = this.patternToMatchTextBox.Text.Trim();
this.replacement = this.replacementTextBox.Text.Trim();
```

It then calls the `StartAsyncTask` method, passing in the `DoWorkEventHandler` and `RunWorkerCompletedEventHandler` delegates that respectively represent the `OnWorkerDoWork` and `OnWorkerCompleted` methods. These two delegates and the `StartAsyncTask` method were thoroughly discussed in the previous chapter.

OnWorkerDoWork

Listing 8-46 demonstrates the implementation of the `OnWorkerDoWork` method.

Listing 8-46: The OnWorkerDoWork Method

```
private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
{
    if (this.hasChanges)
    {
        PropertyBag bag = new PropertyBag();
        if (!this.inModificationMode)
        {
            bag[0] = this.patternToMatch;
            bag[1] = this.replacement;
            this.serviceProxy.AddUrlRewriterRule(bag);
        }

        else
        {
            bag[0] = this.originalPatternToMatch;
            bag[1] = this.patternToMatch;
            bag[2] = this.replacement;
            this.serviceProxy.UpdateUrlRewriterRule(bag);
        }
    }
}
```

This method first checks the value of the `hasChanges` field to ensure that the task form indeed has changes to commit to the underlying configuration file. Then, it creates a `PropertyBag`, which will be used to transfer data to the back end server:

```
PropertyBag bag = new PropertyBag();
```

It then checks whether the `UrlRewriterRuleTaskForm` task form is being used to add a new URL rewriter rule. If so, it populates the `PropertyBag` with the values of the pattern to match and replacement string of the URL rewriter rule being added, and calls the `AddUrlRewriterRule` method of the proxy, passing in the `PropertyBag` to add a new URL rewriter rule with the specified `patternToMatch` and `replacement` attribute values to the underlying configuration file:

```
bag[0] = this.patternToMatch;
bag[1] = this.replacement;
this.serviceProxy.AddUrlRewriterRule(bag);
```

If the task form is being used to update an existing URL rewriter rule, `OnWorkerDoWork` populates the `PropertyBag` with the values of the replacement string, original pattern to match, and new pattern to match of the URL rewriter rule being updated, and calls the `UpdateUrlRewriterRule` method of the proxy, passing in the `PropertyBag` to update the respective URL rewriter rule in the underlying configuration file. Notice that the `PropertyBag` passed into the `UpdateUrlRewriterRule` must also contain the original pattern to match to allow the server-side `UpdateUrlRewriterRule` method to identify the URL rewriter rule being updated.

```
bag[0] = this.originalPatternToMatch;
bag[1] = this.patternToMatch;
bag[2] = this.replacement;
this.serviceProxy.UpdateUrlRewriterRule(bag);
```

OnWorkerCompleted

Listing 8-47 contains the code for the `OnWorkerCompleted` method.

Listing 8-47: The `OnWorkerCompleted` Method

```
private void OnWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    base.UpdateTaskForm();
    if (e.Error != null)
        this.DisplayErrorMessage(e.Error, null);

    else
    {
        base.DialogResult = DialogResult.OK;
        base.Close();
    }
}
```

`OnWorkerCompleted` checks whether everything went fine. If an error occurs, it displays the error message to the end user.

UrlRewriterModule

The previous sections showed you how to implement the `UrlRewriterPage` module page. Implementing your custom module page is just the first step. Next, you need to register your module page with the IIS 7 Manager so it gets instantiated and called. As discussed in previous chapters, the IIS 7 Manager comes with a base class named `Module` that defines the API that you need to implement to register your custom module pages.

In this section, you implement a custom module named `UrlRewriterModule` to register the `UrlRewriterPage` module page with the IIS 7 Manager (see Listing 8-48). Now add a new source file named `UrlRewriterModule.cs` to the `GraphicalManagement/Client` directory and add the code shown in Listing 8-48 to this source file.

Listing 8-48: The `UrlRewriterModule` Module

```
using System;
using Microsoft.Web.Management.Client;
using Microsoft.Web.Management.Server;

namespace UrlRewriting.GraphicalManagement.Client
{
    public class UrlRewriterModule : Module
    {
        protected override void Initialize(IServiceProvider serviceProvider,
                                           ModuleInfo moduleInfo)
        {
            base.Initialize(serviceProvider, moduleInfo);
            IControlPanel panel1 = (IControlPanel)GetService(typeof(IControlPanel));
            ModulePageInfo info1 = new ModulePageInfo(this, typeof(UrlRewriterPage),
                                                    "URL Rewriter Page", "Displays URL rewriter page");
        }
    }
}
```

Listing 8-48: (continued)

```
        panel1.RegisterPage(info1);
    }
}
```

`UrlRewriterModule` overrides the `Initialize` method of the `Module` base class. First, it calls the `Initialize` method of the base class to allow the base class to do its own initialization. Next, it calls the `GetService` method of the `Module` base class, passing in the `Type` object that represents the `IControlPanel` interface to access the control panel service. This service exposes a method named `RegisterPage` that you can use to register your module page.

The `RegisterPage` method takes the `ModulePageInfo` object that represents the module page being registered. The `ModulePageInfo` object encapsulates the complete information about the module page that it represents and exposes this information through its properties as discussed in the previous chapters.

As Listing 8-48 shows, the `Initialize` method instantiates the `ModulePageInfo` object that represents the `UrlRewriterPage` module page and passes this object into the `RegisterPage` method of the control panel service to register the module page:

```
ModulePageInfo info1 = new ModulePageInfo(this, typeof(UrlRewriterPage),
                                           "UrlRewriterPage", "Displays URL rewriter page");
panel1.RegisterPage(info1);
```

Server-Side Managed Code

As mentioned earlier, extending the IIS 7 Manager requires writing two sets of code: client-side and server-side. So far, we've only covered the client-side code. This section shows you how to implement the necessary server-side code to enable the back-end server to communicate with your `UrlRewriterPage` module page and `UrlRewriterRuleTaskForm` task form.

Take these steps to write the server-side code:

1. Implement a custom module service named `UrlRewriterModuleService` with the methods with the same signature as the proxy and mark these methods with the `ModuleServiceMethodAttribute` metadata attribute.
2. Implement a custom module provider named `UrlRewriterModuleProvider` to register the `UrlRewriterModule` module and `UrlRewriterModuleService` module service.
3. Register the `UrlRewriterModuleProvider` custom module provider with the `administration.config` file located in the following directory on your machine:

```
%windir%\system32\inetsrv\config
```


UrlRewriterModuleService

A custom module service is a class that inherits from the `ModuleService` base class and exposes methods that interact with the underlying configuration file to retrieve, add, delete, or update configuration settings. Listing 8-49 presents the members of a custom module service named `UrlRewriterModuleService`, which like any other module service, extends the `ModuleService` base class. Now go ahead and add a new source file named `UrlRewriterModuleService.cs` to the `GraphicalManagement/Server` directory and add the code shown in this code listing to this source file.

Listing 8-49: The UrlRewriterModuleService Server-Side Class

```
using Microsoft.Web.Management.Server;
using UrlRewriting.ImperativeManagement;
using System.Collections;
using System;

namespace UrlRewriting.GraphicalManagement.Server
{
    public class UrlRewriterModuleService : ModuleService
    {
        private UrlRewriterSection GetUrlRewriterSection()
        {
            if (base.ManagementUnit.Configuration != null)
            {
                ManagementConfiguration config = base.ManagementUnit.Configuration;
                UrlRewriterSection section1 = null;
                try
                {
                    section1 =
                        (UrlRewriterSection)config.GetSection(
                            "system.webServer/urlRewriter", typeof(UrlRewriterSection));
                }
                catch (Exception ex)
                {
                }

                if (section1 == null)
                    base.RaiseException("UrlRewriterSectionConfigurationError");

                return section1;
            }

            base.RaiseException("UrlRewriterSectionConfigurationError");
            return null;
        }

        [ModuleServiceMethod]
        public PropertyBag GetUrlRewriterSettings()
        {
            UrlRewriterSection section1 = this.GetUrlRewriterSection();
            ArrayList list = new ArrayList();
            PropertyBag bag;
```

Listing 8-49: (continued)

```
foreach (UrlRewriterRule urlRewriterRule in section1.UrlRewriterRules)
{
    bag = new PropertyBag();
    bag[0] = urlRewriterRule.PatternToMatch;
    bag[1] = urlRewriterRule.Replacement;
    list.Add(bag);
}

PropertyBag bag2 = new PropertyBag();
bag2[0] = (object)list;
bag2[1] = (bool)section1.IsLocked;
return bag2;
}

[ModuleServiceMethod]
public void AddUrlRewriterRule(PropertyBag bag)
{
    UrlRewriterSection section1 = this.GetUrlRewriterSection();
    section1.UrlRewriterRules.Add((string)bag[0], (string)bag[1]);
    base.ManagementUnit.Update();
}

[ModuleServiceMethod]
public void DeleteUrlRewriterRule(PropertyBag bag)
{
    UrlRewriterSection section1 = this.GetUrlRewriterSection();
    section1.UrlRewriterRules.Remove(section1.UrlRewriterRules[(string)bag[0]]);
    base.ManagementUnit.Update();
}

[ModuleServiceMethod]
public void UpdateUrlRewriterRule(PropertyBag bag)
{
    UrlRewriterSection section1 = this.GetUrlRewriterSection();
    UrlRewriterRule item = section1.UrlRewriterRules[(string)bag[0]];
    item.PatternToMatch = (string)bag[1];
    item.Replacement = (string)bag[2];
    base.ManagementUnit.Update();
}
}
```

Note that all methods of the `UrlRewriterModuleService` module service are marked with the `ModuleServiceMethodAttribute` metadata attribute except for the `GetUrlRewriterSection` method. Only those methods of a custom module service marked with this metadata attribute are visible to the proxy. In other words, a proxy can only call those methods of a module service that are marked with this metadata attribute. The following sections present and discuss the implementation of the methods of the `UrlRewriterModuleService` module service.

GetUrlRewriterSection

The `GetUrlRewriterSection` method's main responsibility is to return the `UrlRewriterSection` object that represents the `<urlRewriter>` configuration section (see Listing 8-50).

Listing 8-50: The `GetUrlRewriterSection` Method

```
private UrlRewriterSection GetUrlRewriterSection()
{
    if (base.ManagementUnit.Configuration != null)
    {
        ManagementConfiguration config = base.ManagementUnit.Configuration;
        UrlRewriterSection section1 = null;
        try
        {
            section1 =
                (UrlRewriterSection)config.GetSection(
                    "system.webServer/urlRewriter", typeof(UrlRewriterSection));
        }
        catch (Exception ex)
        {
        }

        if (section1 == null)
            base.RaiseException("UrlRewriterSectionConfigurationError");

        return section1;
    }

    base.RaiseException("UrlRewriterSectionConfigurationError");
    return null;
}
```

The `ModuleService` base class exposes an important property of type `ManagementUnit` named `ManagementUnit`, which encapsulates the logic that determines the configuration hierarchy level at which the current user is working and the configuration file from which the configuration settings are read and into which the configuration settings are stored. Therefore, the `GetUrlRewriterSection` method doesn't need to worry about what the current configuration hierarchy level and configuration file are.

The `ManagementUnit` property exposes a property of type `ManagementConfiguration` named `Configuration` that features a method named `GetSection`. As the name implies, this method returns the `ConfigurationSection` object that represents a configuration section with the specified name and type. Note that the `GetSection` method takes two arguments. The first argument is the fully qualified name of the configuration section being accessed, including its complete section group hierarchy. The second argument is the `Type` object that represents the type of the class that represents the configuration section. Under the hood, the `GetSection` method uses .NET reflection and this `Type` object to dynamically instantiate an instance of the specified configuration section class and populates it with the associated configuration settings.

In this case, the `UrlRewriterSection` class represents the `<urlRewriter>` configuration section, which means that the `GetSection` method will automatically return an instance of this class populated with the required configuration settings:

```
UrlRewriterSection section1 =  
    (UrlRewriterSection)base.ManagementUnit.Configuration.GetSection(  
        "system.webServer/urlRewriter", typeof(UrlRewriterSection));
```

GetUrlRewriterSettings

The main responsibility of the `GetUrlRewriterSettings` method is to retrieve the values of the `replacement` and `patternToMatch` attributes of all URL rewriter rules and the `isLocked` attribute of the `<urlRewriter>` configuration section and return them to the client (see Listing 8-51).

Listing 8-51: The `GetUrlRewriterSettings` Method

```
[ModuleServiceMethod]  
public PropertyBag GetUrlRewriterSettings()  
{  
    UrlRewriterSection section1 = this.GetUrlRewriterSection();  
    ArrayList list = new ArrayList();  
    PropertyBag bag;  
    foreach (UrlRewriterRule urlRewriterRule in section1.UrlRewriterRules)  
    {  
        bag = new PropertyBag();  
        bag[0] = urlRewriterRule.PatternToMatch;  
        bag[1] = urlRewriterRule.Replacement;  
        list.Add(bag);  
    }  
  
    PropertyBag bag2 = new PropertyBag();  
    bag2[0] = (object)list;  
    bag2[1] = (bool)section1.IsLocked;  
    return bag2;  
}
```

The first order of business is to access the `UrlRewriterSection` object that provides programmatic access to the `<urlRewriter>` configuration section:

```
UrlRewriterSection section1 = this.GetUrlRewriterSection();
```

Next, `GetUrlRewriterSettings` creates an `ArrayList`:

```
ArrayList list = new ArrayList();
```

Recall from the previous sections that the `UrlRewriterSection` class exposes a collection property named `UrlRewriterRules` that contains one `UrlRewriterRule` object for each URL rewriter rule in the collection. `GetUrlRewriterSettings` iterates through these `UrlRewriterRule` objects and takes the following actions for each enumerated object:

1. Creates a `PropertyBag`:

```
bag = new PropertyBag();
```

2. Populates the `PropertyBag` with the values of the `Replacement` and `PatternToMatch` properties of the `UrlRewriterRule` object. Recall from the previous sections that the `UrlRewriterRule` class exposes the `replacement` and `patternToMatch` attributes of the associated URL rewriter rule as strongly-typed properties named `Replacement` and `PatternToMatch`.

```
bag[0] = urlRewriterRule.PatternToMatch;  
bag[1] = urlRewriterRule.Replacement;
```

3. Adds the `PropertyBag` object to the `ArrayList`:

```
list.Add(bag);
```

Finally, `GetUrlRewriterSettings` creates a `PropertyBag`, stores the `ArrayList` and the value of the `isLocked` attribute of the `<urlRewriter>` section into it, and returns this `PropertyBag` to the client:

```
PropertyBag bag2 = new PropertyBag();  
bag2[0] = list;  
bag2[1] = (bool)section1.IsLocked;  
return bag2;
```

AddUrlRewriterRule

The `AddUrlRewriterRule` method adds a URL rewriter rule with the specified replacement and `patternToMatch` attribute values (see Listing 8-52).

Listing 8-52: The AddUrlRewriterRule Method

```
[ModuleServiceMethod]  
public void AddUrlRewriterRule(PropertyBag bag)  
{  
    UrlRewriterSection section1 = this.GetUrlRewriterSection();  
    section1.UrlRewriterRules.Add((string)bag[0], (string)bag[1]);  
    base.ManagementUnit.Update();  
}
```

`AddUrlRewriterRule` first accesses the `UrlRewriterSection` object as usual:

```
UrlRewriterSection section1 = this.GetUrlRewriterSection();
```

Next, it retrieves the values of the replacement and `patternToMatch` attributes from the `PropertyBag` that it has received from the client and passes these values into the `Add` method of the `UrlRewriterRules` collection property of the `UrlRewriterSection` object. Recall from the previous sections that the `Add` method of the `UrlRewriterRules` class creates a new `UrlRewriterRule` object with the specified `Replacement` and `PatternToMatch` properties, and adds this object to the `UrlRewriterRules` collection.

```
section1.UrlRewriterRules.Add((string)bag[0], (string)bag[1]);
```

Finally, `AddUrlRewriterRule` calls the `Update` method of the `ManagementUnit` to commit the changes to the underlying configuration file:

```
base.ManagementUnit.Update();
```

DeleteUrlRewriterRule

As Listing 8-53 shows, `DeleteUrlRewriterRule` retrieves the value of the `patternToMatch` attribute of the URL rewriter rule being deleted from the `PropertyBag` that it has received from the client and passes that value into the `Remove` method so the remove operation knows which URL rewriter rule to delete.

Listing 8-53: The DeleteUrlRewriterRule Method

```
[ModuleServiceMethod]
public void DeleteUrlRewriterRule(PropertyBag bag)
{
    UrlRewriterSection section1 = this.GetUrlRewriterSection();
    section1.UrlRewriterRules.Remove(section1.UrlRewriterRules[(string)bag[0]]);
    base.ManagementUnit.Update();
}
```

UpdateUrlRewriterRule

The `UpdateUrlRewriterRule` method's implementation is very similar to that of the `AddUrlRewriterRule`, with one notable difference (see Listing 8-54). The `PropertyBag` that the `UpdateUrlRewriterRule` receives from the client contains an extra piece of information, that is, the original value of the `patternToMatch` attribute of the URL rewriter rule being updated. The `UpdateUrlRewriterRule` uses this original attribute value to locate the associated `UrlRewriterRule` object in the `UrlRewriterRules` collection property of the `UrlRewriterSection` object.

Listing 8-54: The UpdateUrlRewriterRule Method

```
[ModuleServiceMethod]
public void UpdateUrlRewriterRule(PropertyBag bag)
{
    UrlRewriterSection section1 = this.GetUrlRewriterSection();
    UrlRewriterRule item = section1.UrlRewriterRules[(string)bag[0]];
    item.PatternToMatch = (string)bag[1];
    item.Replacement = (string)bag[2];
    base.ManagementUnit.Update();
}
```

UrlRewriterModuleProvider

The previous sections showed you how to implement the `UrlRewriterModuleService` custom module service and `UrlRewriterModule` custom module. Recall that the proxy invokes the methods of the `UrlRewriterModuleService` custom module service to interact with the underlying configuration file. The `UrlRewriterModule` custom module, on the other hand, is used to register the `UrlRewriterPage` custom module page with the IIS 7 Manager. This raises the question: Who registers the `UrlRewriterModuleService` custom module service and `UrlRewriterModule` custom module?

The answer is a direct or indirect subclass of the `ModuleProvider` base class. In this section you implement a custom module provider named `UrlRewriterModuleProvider` that inherits from the `ConfigurationModuleProvider` base class to register the `UrlRewriterModuleService` custom module service and `UrlRewriterModule` custom module as shown in Listing 8-55. Keep in mind that the

Chapter 8: Extending the Integrated Request Processing Pipeline

`ConfigurationModuleProvider` base class inherits from another base class named `SimpleDelegatedModuleProvider`, which in turn inherits the `ModuleProvider` base class. Now add a new source file named `UrlRewriterModuleProvider.cs` to the `GraphicalManagement/Server` directory and add the code shown in Listing 8-55 to this source file.

Listing 8-55: The `UrlRewriterModuleProvider` Custom Module Provider

```
using Microsoft.Web.Management.Server;
using System;

namespace UrlRewriting.GraphicalManagement.Server
{
    class UrlRewriterModuleProvider : ConfigurationModuleProvider
    {
        public override ModuleDefinition GetModuleDefinition(
            IManagementContext context)
        {
            Type type =
                typeof(UrlRewriting.GraphicalManagement.Client.UrlRewriterModule);
            return new ModuleDefinition(base.Name, type.AssemblyQualifiedName);
        }

        public override bool SupportsScope(ManagementScope scope)
        {
            return true;
        }

        protected sealed override string ConfigurationSectionName
        {
            get { return "system.webServer/urlRewriter"; }
        }

        public override string FriendlyName
        {
            get { return "urlRewriter"; }
        }

        public override Type ServiceType
        {
            get { return typeof(UrlRewriterModuleService); }
        }
    }
}
```

`UrlRewriterModuleProvider` overrides the following members of the `ConfigurationModuleProvider` class:

- ❑ **GetModuleDefinition:** The IIS 7 and ASP.NET integrated infrastructure comes with a class named `ModuleDefinition`. As the name implies, this class represents or defines a module. Your custom module provider's implementation of the `GetModuleDefinition` method must call the `ModuleDefinition` constructor, passing in the following parameters:
 - ❑ The name of the custom module provider, which is "`UrlRewriterModuleProvider`" in this case.

- ❑ The assembly qualified name of the type of custom module provider, which consists of five different parts: the fully qualified name of the type of the custom module provider (including its complete namespace hierarchy), assembly name, assembly version, assembly culture, and assembly public key token. The `Type` class exposes a method named `AssemblyQualifiedName` that returns this five-part information.

```
Type type = typeof(UrlRewriting.GraphicalManagement.Client.UrlRewriterModule);  
return new ModuleDefinition(base.Name, type.AssemblyQualifiedName);
```

- ❑ `SupportsScope`: This property determines the supported configuration hierarchy level. Listing 8-55 returns true to signal that it supports all levels.
- ❑ `ConfigurationSectionName`: This property specifies the fully qualified name of the configuration section including its complete section group hierarchy. Listing 8-49 returns `"system.webServer/urlRewriter"`.
- ❑ `ServiceType`: This property returns the `Type` object that represents the custom module service being registered. Listing 8-43 returns `typeof(UrlRewriterModuleService)`.

Registering UrlRewriterModuleProvider

The last step of the recipe for implementing the server-side code requires you to register the `UrlRewriterModuleProvider` module provider with the `administration.config` file. Open this file and add the boldfaced portions of Listing 8-56 to the `<moduleProviders>` and `<modules>` sections of this file.

It's important that you add entries to both sections. If you don't add the entry to the `<modules>` sections, you will only be able to set the configuration settings for your custom configuration section from the machine configuration hierarchy level. In other words, you won't be able to do it in the site or application level.

Listing 8-56: The `administration.config` File

```
<configuration>  
  <moduleProviders>  
    <add name="UrlRewriterModuleProvider"  
    type="UrlRewriting.GraphicalManagement.Server.UrlRewriterModuleProvider,  
    UrlRewriter, Version=2.0.0.0, Culture=neutral,  
    PublicKeyToken=01bf1d8e1663132d" />  
    . . .  
  </moduleProviders>  
  
  <location path=".">  
    <modules>  
      <add name="UrlRewriterModuleProvider"/>  
      . . .  
    </modules>  
  </location>  
</configuration>
```


Chapter 8: Extending the Integrated Request Processing Pipeline

Note that the `<add>` element in the `<moduleProviders>` section features two attributes named `name` and `type`. The `name` attribute contains the friendly name of your custom module provider. You can use any name you want as long as it's unique, that is, no other module provider in the `<moduleProviders>` section has the same name. The `type` attribute consists of a comma-separated list of five items. The first item is the fully qualified name of the type of your custom module provider, which is `UrlRewriting.GraphicalManagement.Server.UrlRewriterModuleProvider` in this case. The last four items specify the assembly that contains the `UrlRewriterModuleProvider`. Note that the assembly information includes the assembly public key token.

You must replace the value of the `PublicKeyToken` parameter shown in Listing 8-56 with the actual public key token of the assembly that contains the `UrlRewriterModuleProvider` module provider. Chapter 7 showed you how to access the public key token of an assembly loaded into the GAC.

Configurable UrlRewriterModule

The previous sections extended:

- ❑ The IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section named `urlRewriter` to allow page developers to configure your `UrlRewriterModule` managed module directly from configuration files.
- ❑ The IIS 7 and ASP.NET integrated imperative management system to add support for new managed classes named `UrlRewriterSection`, `UrlRewriterRule`, and `UrlRewriterRules` to allow page developers to configure the `UrlRewriterModule` managed module directly from managed code in a strongly-typed fashion where they can benefit from Visual Studio IntelliSense support, the compiler type-checking supports, and the well-known object-oriented programming benefits.
- ❑ The IIS 7 and ASP.NET integrated graphical management system to add support for a new module page named `UrlRewriterPage`, a new task form named `UrlRewriterRuleTaskForm`, a new proxy class named `UrlRewriterModuleServiceProxy`, a new module named `UrlRewriterModule`, a new module service named `UrlRewriterModuleService`, and new module provider named `UrlRewriterModuleProvider` to allow page developers to configure your `UrlRewriterModule` managed module directly from the IIS 7 Manager.

As you can see, the whole idea behind these three extensions is to allow users to configure the `UrlRewriterModule` managed module. However, as you can see from Listing 8-7, your current implementation of the `UrlRewriterModule` managed module is not configurable because of two fundamental problems. First, as Listing 8-11 shows, the current implementation hard-codes the following two important pieces of information:

- ❑ The regular expression that defines the pattern that the `Regex` object looks for in the memorable URL:

```
Regex regex = new Regex(@"Articles/(.*)\.aspx", RegexOptions.IgnoreCase);
```

- ❑ The regular expression that defines the replacement string:

```
string newPath = regex.Replace(context.Request.Path,
                               @"Articles.aspx?AuthorName=$1");
```

Chapter 8: Extending the Integrated Request Processing Pipeline

Second, as Listing 8-11 shows, the current implementation supports only one pair of regular expressions.

Listing 8-57 presents the implementation of a configurable version of the `UrlRewriterModule` managed module, which fixes the preceding two problems.

Listing 8-57: A Configurable Version of the `UrlRewriterModule` Managed Module

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;
using Microsoft.Web.Administration;
using UrlRewriting.ImperativeManagement;
using System.Text.RegularExpressions;

namespace UrlRewriting
{
    public class UrlRewriterModule : IHttpModule
    {
        void IHttpModule.Init(HttpApplication app)
        {
            app.BeginRequest += new EventHandler(App_BeginRequest);
        }

        void App_BeginRequest(object sender, EventArgs e)
        {
            HttpApplication app = sender as HttpApplication;
            HttpContext context = app.Context;

            ServerManager mgr = new ServerManager();
            Configuration appHostConfig = mgr.GetWebConfiguration("Default Web Site",
                                                                    context.Request.ApplicationPath);

            UrlRewriterSection urlRewriterSection =
                (UrlRewriterSection)appHostConfig.GetSection(
                                                                    "system.webServer/urlRewriter",
                                                                    typeof(UrlRewriterSection));

            UrlRewriterRules urlRewriterRules = urlRewriterSection.UrlRewriterRules;

            Regex regex;
            Match match;
            String newPath;
            foreach (UrlRewriterRule urlRewriterRule in urlRewriterRules)
            {
                regex = new Regex(urlRewriterRule.PatternToMatch, RegexOptions.IgnoreCase);
                match = regex.Match(context.Request.Path);

                if (match.Success)
                {
                    newPath = regex.Replace(context.Request.Path,
                                                                    urlRewriterRule.Replacement);
                    context.RewritePath(newPath);
                    break;
                }
            }
        }
    }
}
```

(Continued)

Listing 8-57: (continued)

```
    }  
    }  
  
    void IHttpModule.Dispose() { }  
    }  
}
```

Next, I walk through the implementation of the `App_BeginRequest` method. This method begins by accessing the current `HttpContext` object. Recall that this object exposes the well-known ASP.NET objects such as `Request`, `Response`, `Server`, and the like.

```
HttpApplication app = sender as HttpApplication;  
HttpContext context = app.Context;
```

Next, it instantiates a `ServerManager` object, which is always the starting point for imperative interaction with the IIS 7 and ASP.NET integrated configuration system:

```
ServerManager mgr = new ServerManager();
```

Then, it invokes the `GetWebConfiguration` method on the `ServerManager` object to load the content of the `web.config` file of the current ASP.NET application into a `Configuration` object:

```
Configuration appHostConfig = mgr.GetWebConfiguration("Default Web Site",  
    context.Request.ApplicationPath);
```

Next, it invokes the `GetSection` method on this `Configuration` object to return a reference to the `UrlRewriterSection` object that provides imperative access to the `<urlRewriter>` configuration section of the `web.config` file of the current ASP.NET application:

```
UrlRewriterSection urlRewriterSection =  
    (UrlRewriterSection)appHostConfig.GetSection(  
        "system.webServer/urlRewriter",  
        typeof(UrlRewriterSection));
```

Then, it accesses the `UrlRewriterRules` collection object that provides imperative access to the URL rewriter rules:

```
UrlRewriterRules urlRewriterRules = urlRewriterSection.UrlRewriterRules;
```

Next, it iterates through the `UrlRewriterRule` objects in the `UrlRewriterRules` collection and performs the same actions for each enumerated `UrlRewriterRule` object (keep in mind that each `UrlRewriterRule` object in this collection represents a URL rewriter rule). First, it uses the constructor of the `Regex` class, passing in the value of the `PatternToMatch` property of the enumerated `UrlRewriterRule` object to instantiate a `Regex` object. Recall that the `PatternToMatch` property of a `UrlRewriterRule` object contains the value of the `patternToMatch` attribute of the associated URL rewriter rule.

```
regex = new Regex(urlRewriterRule.PatternToMatch, RegexOptions.IgnoreCase);
```

Next, it invokes the `Match` method on the `Regex` object to check whether the requested URL contains the specified pattern:

```
match = regex.Match(context.Request.Path);
```

If so, it calls the `Replace` method, passing in the request URL and the replacement string to convert the user's memorable URL to the actual URL that the application expects:

```
newPath = regex.Replace(context.Request.Path,  
                        urlRewriterRule.Replacement);
```

Finally, it invokes the `RewritePath` method on the current `HttpContext` object to rewrite the value of the requested URL to the actual value:

```
context.RewritePath(newPath);
```

Rewriting Non-ASP.NET URLs

One of the great advantages of the IIS 7 and ASP.NET integrated request processing pipeline is that you can plug a managed module into the pipeline and use this module to pre-process or post-process non-ASP.NET content, such as requests for `.asp` and `.php` resources. To enable your configurable `UrlRewriterModule` managed module to rewrite non-ASP.NET URLs, you need to add the following configuration segment to the appropriate configuration file:

```
<system.webServer>  
  <modules runAllManagedModulesForAllRequests="true">  
    <add name="UrlRewriterModule" type="UrlRewriting.UrlRewriterModule" />  
  </modules>  
</system.webServer>
```

These configuration settings provide another benefit besides enabling URL rewriting for non-ASP.NET contents. It also allows you to rewrite URLs such as the following:

```
http://mysite.com/Articles
```

This memorable URL allows users to access all articles. You can have a URL rewriter rule that will instruct your `UrlRewriterModule` managed module to rewrite this memorable URL to something like `http://mysite.com/article.aspx?AllArticles=true`.

Postback Problem with URL Rewriting

As discussed, the `UrlRewriterModule` managed module rewrites the end user's memorable URL back to the actual URL that your application expects. Because URL rewriting occurs before the page handler responsible for processing the current request springs into life, the page handler only deals with the actual URL. This means that when the `HtmlForm` server control enters its rendering phase, it will render the actual URL as the value of the `action` attribute on the `<form runat="server">` HTML element on the current page.

Chapter 8: Extending the Integrated Request Processing Pipeline

Therefore, when the user posts the page back to the server, the browser's address bar will show the actual URL instead of the original memorable URL. To fix this problem, you need to instruct the `HtmlForm` server control to render the original memorable URL as the value of the `action` attribute on the `<form runat="server">` element. Obviously, you need to extend the functionality of the `HtmlForm` server control to enable it to render the original memorable URL as the value of the `action` attribute. There are two ways to achieve this. One way is to write a custom server control that inherits the `HtmlForm` server control. Another approach is to use a control adapter. You'll use the second approach because it does not require any code changes in the existing code.

Every ASP.NET server control is associated with a component called a control adapter. The control adapter associated with an ASP.NET server control allows you to extend the functionality of the server control without having to write a new custom server control that inherits from the server control.

Follow these steps to extend the functionality of a server control such as `HtmlForm` via a control adapter:

1. Implement a new control adapter that inherits from the `ControlAdapter` base class.
2. Add a new `.browser` file to the `App_Browsers` directory of your application to register your custom control adapter.

Following this recipe, Listing 8-58 presents the implementation of a new control adapter named `UrlRewriterControlAdapter`, which extends the `ControlAdapter` base class.

Listing 8-58: The `UrlRewriterControlAdapter`

```
using System.Web.UI;
using System.Web.UI.Adapters;

namespace UrlRewriting
{
    public class UrlRewriterControlAdapter : ControlAdapter
    {
        protected override void Render(HtmlTextWriter writer)
        {
            base.Render(new UrlRewriterHtmlTextWriter(writer));
        }
    }
}
```

As you can see, `UrlRewriterControlAdapter` overrides the `Render` method of its base class to use an instance of a class named `UrlRewriterHtmlTextWriter` as the HTML text writer for rendering the HTML markup of the `HtmlForm` server control. Listing 8-59 presents the implementation of the `UrlRewriterHtmlTextWriter` class.

Listing 8-59: The `UrlRewriterHtmlTextWriter` Class

```
using System.Web.UI;
using System.Web;
using System.IO;

namespace UrlRewriting
{

```

Listing 8-59: (continued)

```
public class UrlRewriterHtmlTextWriter : HtmlTextWriter
{
    public UrlRewriterHtmlTextWriter(HtmlTextWriter writer): base(writer)
    {
        this.InnerWriter = writer.InnerWriter;
    }

    public UrlRewriterHtmlTextWriter(TextWriter writer): base(writer)
    {
        this.InnerWriter = writer;
    }

    public override void WriteAttribute(string name, string value, bool fEncode)
    {
        if (HttpContext.Current.Items["ActionRewrittenToMemorableUrl"] == null &&
            name == "action")
        {
            value = HttpContext.Current.Request.RawUrl;
            HttpContext.Current.Items["ActionRewrittenToMemorableUrl"] = true;
        }

        base.WriteAttribute(name, value, fEncode);
    }
}
```

As you can see, the `UrlRewriterHtmlTextWriter` class overrides the `WriteAttribute` method of its base class to ensure that this method renders the original memorable URL as the value of the action attribute on the `<form runat="server">` HTML element. Keep in mind that the `RawUrl` property of the ASP.NET Request object contains the original memorable URL.

The `WriteAttribute` method takes two important parameters. The first parameter contains the name of the attribute being written. You're only interested in the `action` attribute. The second parameter contains the value of the attribute being written. This value is the actual URL. The `WriteAttribute` method replaces this value with the memorable URL before it calls the `WriteAttribute` method on the base class to write the value of the action attribute.

Finally, you need to add a new `.browser` file named `UrlRewriter.browser` to the `App_Browsers` directory of the application to register the `UrlRewriterControlAdapter` control adapter with ASP.NET in a declarative fashion. Listing 8-60 presents the content of the `UrlRewriter.browser` file.

Listing 8-60: The Content of the `UrlRewriter.browser` File

```
<browsers>
  <browser refID="Default">
    <controlAdapters>
      <adapter controlType="System.Web.UI.HtmlControls.HtmlForm"
        adapterType="UrlRewriterControlAdapter" />
    </controlAdapters>
  </browser>
</browsers>
```

Summary

This chapter showed you how to implement configurable managed modules and handler factories to extend the IIS 7 and ASP.NET integrated request processing pipeline in a configurable fashion where the clients of your configurable managed modules and handler factories can use the IIS 7 and ASP.NET integrated configuration system, integrated imperative management system, and integrated graphical management system to configure these managed modules and handler factories directly from configuration files, managed code, and the IIS 7 Manager, respectively.

The next chapter discusses developing configurable managed handlers in the context of the IIS 7 and ASP.NET Integrated Providers Extensibility model.

Understanding the Integrated Providers Model

A provider-based service is a piece of software that can service a specific type of data from any type of data store. The ASP.NET Framework comes with standard provider-based services such as:

- ❑ User membership service, which services user membership data from any type of data store
- ❑ Role management service, which services role data from any type of data store
- ❑ User profile service, which services user profile data from any type of data store

Provider-based services play a central role in data-driven Web applications where data comes from many different types of data stores, such as SQL Server databases, Oracle databases, XML documents, flat files, and Web services, just to name a few. A provider-based service hides the data-store-specific data access APIs behind a standard API to enable all clients of the service to use the same API to interact with any type of data store. In other words, these clients can write one set of data access code that can interact with all types of data stores without code changes.

The IIS 7 and ASP.NET integrated providers model is an extensible infrastructure that allows you to implement fully configurable provider-based services and plug them into the IIS 7 and ASP.NET integrated infrastructure.

I begin this chapter by discussing why you need provider-based services in the first place. I then use an example to show you the integrated providers model in action before getting into the implementation details. Next, I dive into the internals of the integrated providers model in preparation for the next chapter, where I show you how to extend the integrated providers model to implement fully configurable provider-based services and to plug them into the IIS 7 and ASP.NET integrated infrastructure.

Why You Need Provider-Based Services

The previous chapter developed an HTTP handler named `RssHandler` that services requests for resources with the file extension `.rss`. Listing 9-1 shows the definition of `RssHandler`, which generates an RSS document from a specified SQL Server database. The current implementation of `RssHandler` suffers from the following important shortcomings:

- ❑ As the highlighted portions of Listing 9-1 illustrate, `RssHandler` uses ADO.NET classes to retrieve the data required for generating the RSS document from the underlying SQL Server database. Because ADO.NET classes are the data access API for relational databases, they can't be used to retrieve data from non-relational data stores, such as XML documents, Web services, flat files, and so on. In other words, the implementation of the `RssHandler` shown in Listing 9-1 is tied to relational databases and can't generate RSS documents from non-relational data stores.
- ❑ As the highlighted portions of Listing 9-1 illustrate, `RssHandler` uses ADO.NET SQL Server-specific classes to retrieve the data required for generating the RSS document. As such, the current implementation of `RssHandler` cannot generate RSS documents from other types of relational data stores such as Oracle databases.
- ❑ As you can see from the boldfaced portions of Listing 9-1, the current implementation of `RssHandler` is tied to a SQL Server database with a specific schema. Therefore, this implementation cannot generate RSS documents from SQL Server databases with different schemas.
- ❑ As you can see from Listing 9-1, the current implementation of `RssHandler` hard-codes the channel information, that is, channel title, description, and link.

Listing 9-1: The `RssHandler` Class

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Data.SqlClient;
using System.IO;
using System.Collections;

namespace CustomComponents
{
    public class RssHandler : IHttpHandler
    {
        string channelTitle;
        string channelLink;
        string channelDescription;
        string itemTitleField;
        string itemDescriptionField;
        string itemLinkField;
        string itemLinkFormatString;
        string connectionStringName;
        string commandText;
        CommandType commandType;
```

Listing 9-1: (continued)

```
public RssHandler()
{
    channelTitle = "New Articles On some site";
    channelLink = "http://somesite";
    channelDescription = "The list of newly published articles on some site";
    itemTitleField = "Title";
    itemDescriptionField = "Abstract";
    itemLinkField = "AuthorName";
    itemLinkFormatString = "http://somesite/WebSite23/{0}.aspx";
    connectionStringName = "MyConnectionString";
    commandText = "Select * From Articles";
    commandType = CommandType.Text;
}

bool IHttpHandler.IsReusable
{
    get { return false; }
}

SqlDataReader GetDataReader()
{
    SqlConnection con = new SqlConnection();
    ConnectionSettings settings =
        ConfigurationManager.ConnectionStrings[connectionStringName];
    con.ConnectionString = settings.ConnectionString;
    SqlCommand com = new SqlCommand();
    com.Connection = con;
    com.CommandText = commandText;
    com.CommandType = commandType;
    con.Open();
    return com.ExecuteReader(CommandBehavior.CloseConnection);
}

public void LoadRss(Stream stream)
{
    SqlDataReader reader = GetDataReader();

    ArrayList items = new ArrayList();
    Item item;
    while (reader.Read())
    {
        item = new Item();
        item.Title = (string)reader[itemTitleField];
        item.Link = (string)reader[itemLinkField];
        item.Description = (string)reader[itemDescriptionField];
        item.LinkFormatString = itemLinkFormatString;
        items.Add(item);
    }
    reader.Close();

    Channel channel = new Channel();
    channel.Title = channelTitle;
```

(Continued)

Listing 9-1: (continued)

```
channel.Link = channelLink;
channel.Description = channelDescription;

RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
}

void IHttpHandler.ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/xml";
    LoadRss(context.Response.OutputStream);
}
}
```

One approach to fixing these problems is to make the data access code used in Listing 9-1 generic. The .NET Framework comes with different techniques and tools for writing generic data access code. Making the data access generic has its own downsides; for example, you cannot perform data-store-specific optimizations to improve the performance of your application.

A better approach to fixing the problems with the current implementation of `RssHandler` is to move the data access code from `RssHandler` to a different component known as a *provider*, and have `RssHandler` use this provider to retrieve data from the underlying data store. Because the data access code is data-store-specific and consequently varies from one data store to another, the same provider cannot be used to interact with different types of data stores. As such, simply moving the data access code from `RssHandler` to a provider will not resolve the problems, because you still have to make code changes in `RssHandler` to have it use a new provider to retrieve data from a new data store. In other words, `RssHandler` is still tied to the underlying data store even though it does not directly contain the data access code.

The way to fix this problem is to move the code that switches providers to a different component known as `RssService` and have `RssHandler` interact with `RssService` instead of the provider. As you'll see later, the `RssService` will read the information about the configured provider from the configuration file and use .NET reflection to instantiate an instance of the provider in a generic fashion. What we've been talking about so far is known as the *provider pattern*. This pattern is at the heart of the ASP.NET provider-based services.

The Integrated Providers Model in Action

Before diving into the IIS 7 and ASP.NET integrated providers model and its extensibility points, let's see what this model looks like in action. As Figure 9-1 shows, the IIS 7 Manager server home page contains an item named Providers.



Figure 9-1

If you click the Providers item, it will take you to the module page shown in Figure 9-2. This page is an instance of an internal module list page named `ProviderConfigurationConsolidatedPage`. As the name implies, this module page consolidates the user interface that allows the end user to add, remove, and update providers for any type of provider-based service including your very own custom services. In other words, this consolidated page allows the end user to configure providers for all types of provider-based services from the same module page.

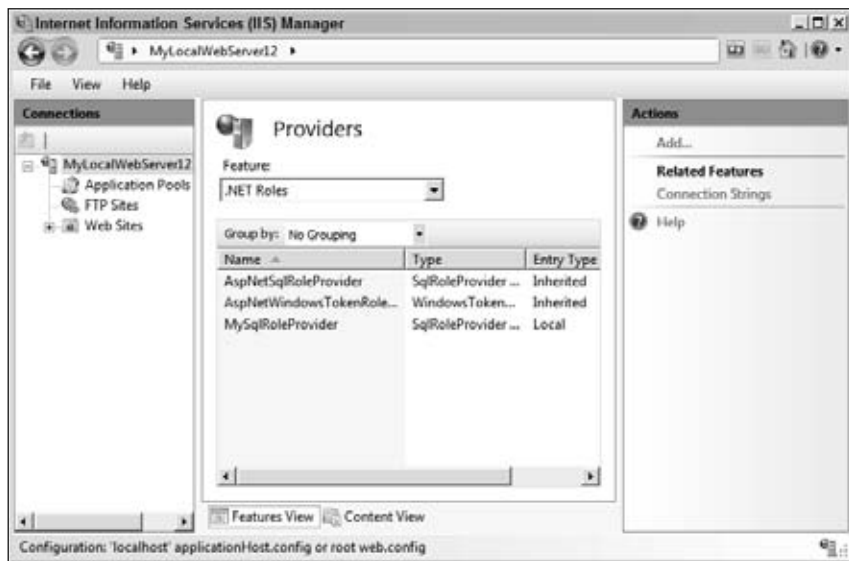


Figure 9-2

Chapter 9: Understanding the Integrated Providers Model

As you can see, the provider configuration consolidated page consists of a combo box labeled “Feature” that displays the list of available provider-based services to choose from and a list view that displays the list of providers registered for the provider-based service that the user has selected from the Feature combo box. For example, the combo box in Figure 9-2 displays the .NET Roles provider-based service, and the list view below this combo box displays all the providers registered for this service. Note that the task panel associated with the provider configuration consolidated page in Figure 9-2 contains two link buttons named “Add” and “Connection Strings.” If you click the Connection Strings link, it will take you to the connection strings page where you can add, remove, or edit connection strings. If you click the Add link, it will pop up the Add Provider task form shown in Figure 9-3. This task form is an instance of a task form named `AddProviderForm`. This task form allows you to register a new provider with the provider-based service.

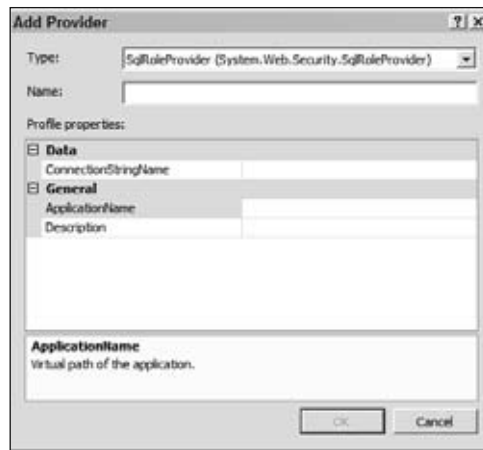


Figure 9-3

As you can see, this task form consists of four main parts. The first part is a combo box that displays the list of available provider types for the specified provider-based service. You need to choose a provider type for the provider you’re about to add. The second part is a textbox where you need to enter a friendly name for the new provider. Notice that as soon as you select a provider type from the combo box and enter a friendly name in the textbox, the OK button is enabled. The third part is a grid where you can specify the settings for your provider. The fourth part is the command bar, which contains the OK and Cancel buttons.

Now back to the IIS 7 Manager shown in Figure 9-1. If you select a provider from the list of displayed providers, the task panel will show a few more link buttons, including Edit, Rename, and Remove, as shown in Figure 9-4.

If you click the Edit link in the task panel, it will launch the Edit Provider task form shown in Figure 9-5. This task form allows you to edit the settings of the selected provider.

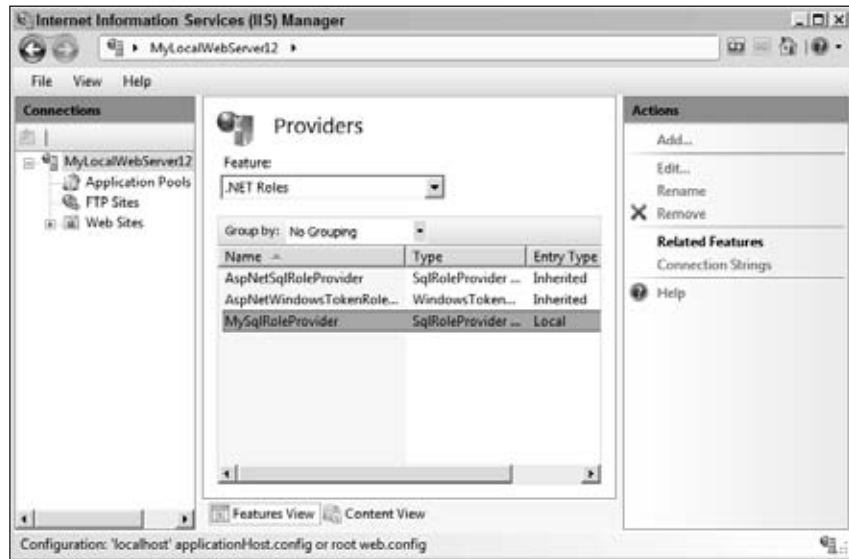


Figure 9-4

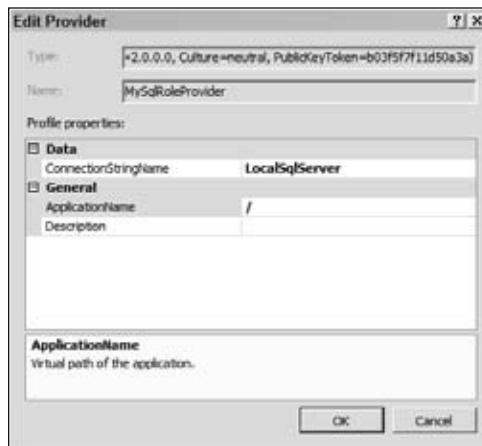


Figure 9-5

Note the main differences between the task forms shown in Figures 9-3 and 9-5. The task form shown in Figure 9-5 is missing the combo box that displays the list of available provider types. Also note that both textbox controls in the top of the Edit Provider task form are grayed out. Both of these make lot of sense because the Edit Provider task form is not adding a new provider. Instead it is editing the settings of the selected provider.

Now back to the IIS 7 Manager shown in Figure 9-4. If you click the Rename link or click the selected provider, you'll get the result shown in Figure 9-6. As you can see, clicking the Rename link button makes the name of the selected provider editable.

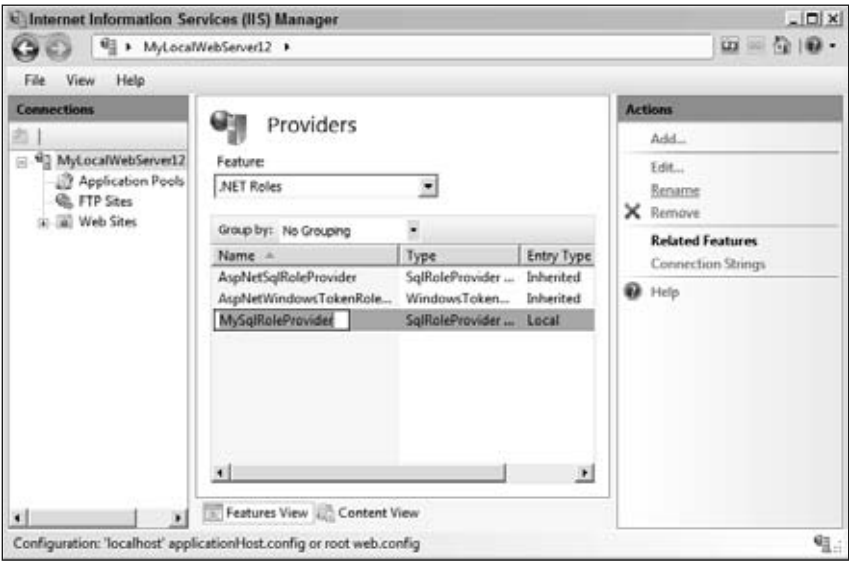


Figure 9-6

The `ProviderConfigurationConsolidatedPage` module list page is where you add, update, remove, or rename providers of a given provider-based service. Adding, updating, removing, and renaming providers are not the only configurable aspects of a provider-based service. Every provider-based service also exposes a module page of its own where you can configure other aspects of the service. Next, I show an example of such a module page. If you click the Default Web Site node in the Connections pane, you'll get the result shown in Figure 9-7.

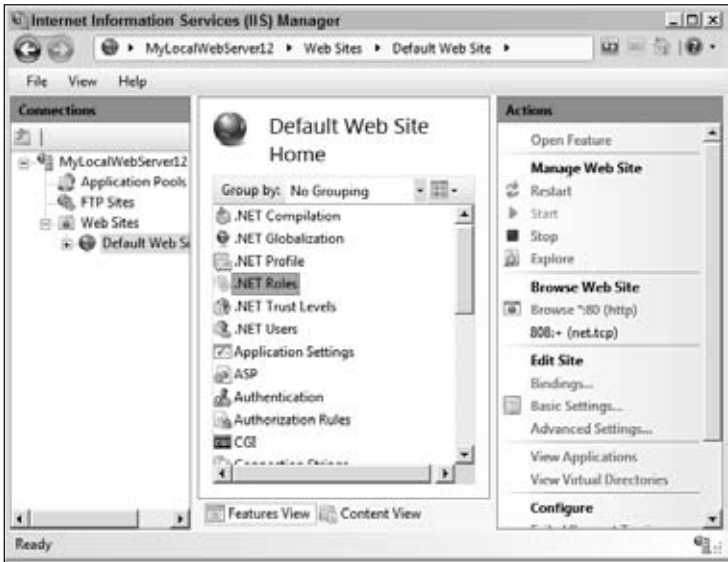


Figure 9-7

If you click the .NET Roles item, it'll take you to the .NET Roles page, shown in Figure 9-8. This is the module page where you can configure aspects of the Roles provider-based service other than adding, removing, updating, and renaming its providers.

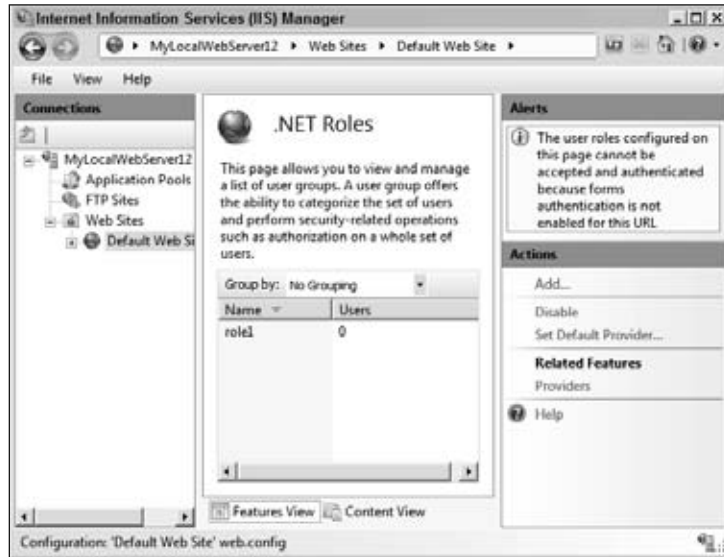


Figure 9-8

Note that the task panel associated with this module page contains a link button named Set Default Provider. If you click this link, it will pop up the task form shown in Figure 9-9. This task form allows you to specify the default provider for the .NET Roles provider-based service. As you can see, this task form contains a combo box that displays the list of available providers to choose from.

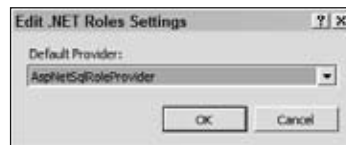


Figure 9-9

Under the Hood of the Integrated Providers Model

The IIS 7 and ASP.NET integrated providers model comes with two important abstract base classes named `ProviderFeature` and `ProviderConfigurationSettings` and an important interface named `IProviderConfigurationService`. Extending the integrated providers model requires a solid

Chapter 9: Understanding the Integrated Providers Model

understanding of these two abstract base classes (and their subclasses) and this interface (and the class that implements this interface).

ProviderFeature

The integrated providers model comes with three standard subclasses of the `ProviderFeature` base class as follows:

- ❑ `RolesProviderConfigurationFeature`: The integrated providers model uses an instance of the `RolesProviderConfigurationFeature` subclass of the `ProviderFeature` abstract base class to represent the `Roles` provider-based service.
- ❑ `MembershipProviderConfigurationFeature`: The integrated providers model uses an instance of the `MembershipProviderConfigurationFeature` subclass of the `ProviderFeature` abstract base class to represent the `Users` provider-based service.
- ❑ `ProfileProviderConfigurationFeature`: The integrated providers model uses an instance of the `ProfileProviderConfigurationFeature` subclass of the `ProviderFeature` abstract base class to represent the `Profile` provider-based service.

As you'll see later in this chapter, you can also implement your own subclass of the `ProviderFeature` abstract base class and have the integrated providers model use an instance of this subclass to represent your own custom provider-based service or feature. For example, you'll implement a subclass of the `ProviderFeature` abstract base class named `RssProviderConfigurationFeature` and have the integrated providers model use an instance of this subclass to represent your RSS provider-based service.

Listing 9-2 presents the internal implementation of the `ProviderFeature` abstract base class.

Listing 9-2: The ProviderFeature Abstract Base Class

```
public abstract class ProviderFeature
{
    public virtual string ConnectionStringAttributeName
    {
        get { return string.Empty; }
    }

    public virtual bool ConnectionStringRequired
    {
        get { return false; }
    }

    public abstract string FeatureName { get; }
    public abstract string ProviderBaseType { get; }
    public abstract string ProviderCollectionPropertyName { get; }
    public abstract string[] ProviderConfigurationSettingNames { get; }
    public abstract string SectionName { get; }
    public abstract string SelectedProvider { get; }
    public abstract string SelectedProviderPropertyName { get; }
    public abstract ProviderConfigurationSettings Settings { get; }
}
```

The abstract properties of the `ProviderFeature` abstract base class define the API that every provider feature must implement, including `RolesProviderConfigurationFeature`, `MembershipProviderConfigurationFeature`, `ProfileProviderConfigurationFeature`, and your very own custom provider feature:

- ❑ **FeatureName:** Your custom provider feature's implementation of this property must return a string that contains your custom provider-based service name. The integrated providers model displays the value of this property in the Feature combo box shown in Figure 9-2. For example, the `RolesProviderConfigurationFeature`'s implementation of this property returns the string `".NET Roles"`, which is listed in the list of features that the Feature combo box displays.
- ❑ **ProviderBaseType:** As you'll see later in this chapter, as part of the implementation of your custom provider-based service or feature, you must implement an abstract class known as a provider base, which will act as the base class for all providers of your custom provider-based service. For example, all Roles providers inherit from an abstract provider base class named `RoleProvider`.
- ❑ Your custom provider feature's implementation of the `ProviderBaseType` property must return a string that contains the complete type information about the provider base from which all the providers of your custom provider-based service inherit. This type information consists of two main parts, but only the first part is required. The first part specifies the fully qualified name of the type of the custom provider base including its complete namespace containment hierarchy. The second part specifies the information about the assembly that contains the custom provider base including the assembly name, version, culture, and public key token.
- ❑ For example, the `RolesProviderConfigurationFeature` provider feature's implementation of the `ProviderBaseType` property returns the string `"System.Web.Security.RoleProvider"`:

```
public override string ProviderBaseType
{
    get { return "System.Web.Security.RoleProvider"; }
}
```

- ❑ **ProviderCollectionPropertyName:** As you'll see later in this chapter, as part of the implementation of your custom provider-based service, you should also extend the IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section to allow the clients of your service to configure your service from a configuration file. This configuration section must contain a Collection XML element (normally named `<providers>`) where the clients of your custom provider-based service can use an Add XML element (normally named `<add>`) to register a new provider with your service.
- ❑ For example, the IIS 7 and ASP.NET integrated configuration system supports a configuration section named `<roleManager>` that contains a Collection XML element named `<providers>` where the clients of the ASP.NET Roles provider-based service can add new providers:

```
<roleManager defaultProvider="SqlProvider" enabled="true" cacheRolesInCookie="true"
cookieName=".ASPROLES" cookieTimeout="30" cookiePath="/" cookieRequireSSL="false"
cookieSlidingExpiration="true" cookieProtection="All" >
  <providers>
    <add name="SqlProvider" type="System.Web.Security.SqlRoleProvider"
connectionStringName="SqlServices" applicationName="MyApplication" />
  </providers>
</roleManager>
```

Chapter 9: Understanding the Integrated Providers Model

- ❑ Your custom provider feature's implementation of the `ProviderCollectionPropertyName` property must return a string that contains the name of this Collection XML element. A typical implementation of the `ProviderCollectionPropertyName` property returns the string "providers". The following code listing presents the `RolesProviderConfigurationFeature` provider feature's implementation of the `ProviderCollectionPropertyName` property:

```
public override string ProviderCollectionPropertyName
{
    get { return "providers"; }
}
```

- ❑ `ProviderConfigurationSettingNames`: Your custom provider feature's implementation of this property must return a string array that contains the names of the attributes on the Add elements that add providers to the collection that the Collection XML element represents. This Add element is normally named <add>. For example, the `RolesProviderConfigurationFeature` provider feature's implementation of the `ProviderConfigurationSettingNames` property returns a string array that contains the strings "applicationName", "description", and "connectionStringName", which are the names of the attributes on the <add> element that registers a new provider with the Roles provider-based service:

```
public override string[] ProviderConfigurationSettingNames
{
    get
    {
        return new string[] { "applicationName", "description",
                               "connectionStringName" };
    }
}
```

- ❑ `SectionName`: Your custom provider feature's implementation of this property must return a string that contains the fully qualified name of the configuration section that configures your custom provider-based service. This name must include the complete group hierarchy of the configuration section. For example, the `RolesProviderConfigurationFeature` provider feature's implementation of the `SectionName` property returns the string "system.web/roleManager", which is the fully qualified name of the configuration section where the clients of the Roles provider-based service can configure the service:

```
public override string SectionName
{
    get { return "system.web/roleManager"; }
}
```

- ❑ `SelectedProvider`: Your custom provider feature's implementation of this property must return a string that contains the friendly name of the default provider of your custom provider-based service.
- ❑ `SelectedProviderPropertyName`: Your custom provider feature's implementation of this property must return a string that contains the name of the attribute whose value specifies the friendly name of the default provider of your custom provider-based service. This property normally returns the string "defaultProvider". For example, the following code listing presents

the `RolesProviderConfigurationFeature` provider feature's implementation of the `SelectedProviderPropertyName` property:

```
public override string SelectedProviderPropertyName
{
    get { return "defaultProvider"; }
}
```

- ❑ **Settings:** As you'll see later in this chapter, as part of the implementation of your custom provider-based service, you must also implement a provider configuration settings class that inherits from the `ProviderConfigurationSettings` base class. For example, the `Roles` provider-based service comes with a provider configuration settings class named `RolesProviderConfigurationSettings`. Your custom provider feature's implementation of the `Settings` property must create and return an instance of the associated provider configuration settings class. For example, the `RolesProviderConfigurationFeature` provider feature's implementation of the `Settings` property instantiates and returns an instance of the `RolesProviderConfigurationSettings`:

```
public override ProviderConfigurationSettings Settings
{
    get { return new RolesProviderConfigurationSettings(this._module); }
}
```

Note that the `ProviderFeature` abstract base class also exposes two virtual properties that its subclasses may choose to override:

- ❑ **ConnectionStringAttributeName:** Your custom provider feature's implementation of this property must return a string that contains the name of the attribute whose value specifies the connection string name. This property normally returns the string `"connectionStringName"`. As Listing 9-2 shows, the `ProviderFeature` abstract base class's implementation of this property returns an empty string. For example, the following code listing presents the `RolesProviderConfigurationFeature` provider feature's implementation of the `ConnectionStringAttributeName` property:

```
public override string ConnectionStringAttributeName
{
    get { return "connectionStringName"; }
}
```

- ❑ **ConnectionStringRequired:** Your custom provider feature's implementation of this property must return a Boolean value that specifies whether the providers of your provider-based service require connection strings. As Listing 9-2 shows, the `ProviderFeature` abstract base class's implementation of this property returns `false`. Therefore, if your custom provider feature does not implement this property, the integrated providers model will assume that the providers of your provider-based service do not require connection strings. As a result, if the end user does not enter a value into the `ConnectionStringName` textbox shown in Figure 9-5 when it is registering a new provider, the integrated providers model will allow the registration process to go through.

As you can see, your custom provider feature provides the integrated providers model with the complete information about your custom provider-based service. For example, as you just saw, the `RolesProviderConfigurationFeature` provider feature provides the integrated providers model

Chapter 9: Understanding the Integrated Providers Model

with the following pieces of information about the Roles provider-based service among other pieces of information:

- ❑ The name of the Roles provider-based service, that is, ".NET Roles". This name appears in the Feature combo box.
- ❑ The fully qualified name of the configuration section, which contains the configuration settings for the Roles provider-based service, that is, "system.web/roleManager".
- ❑ The name of the attribute on the configuration section's containing element, which contains the default provider of the Roles provider-based service, that is, "defaultProvider".
- ❑ The friendly name of the default provider.
- ❑ The name of the Collection XML element that contains the registered providers, that is, "providers".
- ❑ The names of the attributes on the Add XML elements that register providers with the Roles provider-based service, that is, "applicationName", "description", and "connectionStringName".
- ❑ The fully qualified name of the provider base type, that is, "System.Web.Security.RoleProvider" from which all providers of the Roles provider-based service inherit.

Implementing a custom provider feature for your custom provider-based service is just half the story. You must also create an instance of your custom provider feature and make this instance available to the integrated providers model. As you'll see shortly, this model uses this instance to extract the complete information about your provider-based service. As such, this instance is known as an extension because it extends the functionality of the integrated providers model.

You may be wondering how you can make an instance of your custom provider feature available to the integrated providers model. In other words, how can you add an extension to this model?

The configuration settings of your custom provider-based service can be divided into two main groups. The first group includes provider-specific operations such as adding, updating, removing, and renaming providers. These operations must be performed from the `ProviderConfigurationConsolidatedPage` module list page as discussed earlier. The second group includes anything else.

As you'll see later in this chapter, as part of the implementation of your custom provider-based service, you must also implement one or more module pages, which provide the clients of your custom provider-based service with the appropriate user interface to specify the configuration settings of your service other than adding, updating, removing, and renaming its providers. For example, the Roles provider-based service provides its clients with a module page named `RolesPage` to specify the configuration settings other than adding, updating, removing, and renaming providers. Figure 9-8 shows the `RolesPage` module list page in action.

As you saw in the previous chapter, when you implement a custom module page, you must also implement a custom module that inherits from the `Module` base class to register your custom module page with the IIS 7 Manager. As part of this registration process, you must also instantiate an instance of your custom provider feature and register this instance with the integrated providers model. For example, the Roles provider-based service implements a custom module named `RolesModule` to register the `RolesPage` module list page. Listing 9-3 presents the internal implementation of the `Initialize` method of the `RolesModule` module where this registration takes place.

Listing 9-3: The Initialize Method of the RolesModule Module

```
protected override void Initialize(IServiceProvider serviceProvider,
                                ModuleInfo moduleInfo)
{
    base.Initialize(serviceProvider, moduleInfo);
    Connection service = (Connection) serviceProvider.GetService(typeof(Connection));
    ModulePageInfo itemPageInfo = new ModulePageInfo(this, typeof(RolesPage),
                                                    ".NET Roles", ".NET Roles");

    IControlPanel panel =
        (IControlPanel) serviceProvider.GetService(typeof(IControlPanel));
    panel.RegisterPage(ControlPanelCategoryInfo.Security, itemPageInfo);
    panel.RegisterPage(ControlPanelCategoryInfo.AspNet, itemPageInfo);

    IExtensibilityManager manager =
        (IExtensibilityManager) serviceProvider.GetService(
                                                    typeof(IExtensibilityManager));

    if (manager != null)
    {
        RolesProviderConfigurationFeature extension =
            new RolesProviderConfigurationFeature(this);
        manager.RegisterExtension(typeof(ProviderFeature), extension);
    }
}
```

As you can see from Listing 9-3, the `Initialize` method consists of two main parts. The first part registers the `RolesPage` module list page:

```
Connection service = (Connection) serviceProvider.GetService(typeof(Connection));
ModulePageInfo itemPageInfo = new ModulePageInfo(this, typeof(RolesPage),
                                                    ".NET Roles", ".NET Roles");

IControlPanel panel =
    (IControlPanel) serviceProvider.GetService(typeof(IControlPanel));
panel.RegisterPage(ControlPanelCategoryInfo.Security, itemPageInfo);
panel.RegisterPage(ControlPanelCategoryInfo.AspNet, itemPageInfo);
```

The second part adds a new extension to the IIS 7 and ASP.NET integrated infrastructure. First, it accesses a service known as the extensibility manager service. This service, like any other service in the integrated infrastructure, implements an interface. This interface in this case is an interface named `IExtensibilityManager`.

```
IExtensibilityManager manager =
    (IExtensibilityManager) serviceProvider.GetService(
                                                    typeof(IExtensibilityManager));
```

As the name suggests, the extensibility manager service manages the extensions to the integrated infrastructure. Next, the `Initialize` method instantiates an instance of the `RolesProviderConfigurationFeature` provider feature:

```
RolesProviderConfigurationFeature extension =
    new RolesProviderConfigurationFeature(this);
```

Chapter 9: Understanding the Integrated Providers Model

Finally, it invokes a method named `RegisterExtension` on the extensibility manager service to register this `RolesProviderConfigurationFeature` provider feature extension. Note that this extension is registered under the `Type` object that represents the `ProviderFeature` abstract base class.

```
manager.RegisterExtension(typeof(ProviderFeature), extension);
```

ProviderConfigurationSettings

Next, I discuss the `ProviderConfigurationSettings` abstract base class. Recall that the `ProviderFeature` abstract base class exposes a property of type `ProviderConfigurationSettings` named `Settings`. As you'll see later in this chapter, as part of the implementation of your custom provider-based service, you must also implement a custom provider configuration settings class that inherits the `ProviderConfigurationSettings` abstract base class. Your custom provider feature's implementation of the `Settings` property must create and return an instance of this custom provider configuration settings class. Listing 9-4 presents the implementation of the `ProviderConfigurationSettings` base class.

Listing 9-4: The `ProviderConfigurationSettings` Class

```
public abstract class ProviderConfigurationSettings
{
    public IDictionary GetSettings()
    {
        return this.Settings;
    }

    public void LoadSettings(string[] parameters)
    {
        for (int i = 0; i < parameters.Length; i += 2)
        {
            this.Settings[parameters[i]] = parameters[i + 1];
        }
    }

    public abstract bool Validate(out string message);
    protected abstract IDictionary Settings { get; }
}
```

Note that `ProviderConfigurationSettings` is an abstract class. It is the responsibility of each provider-based service to implement a class that inherits from this abstract base class. For example, the ASP.NET Profile, Roles, and Users provider-based services respectively implement the `ProfileProviderConfigurationSettings`, `RolesProviderConfigurationSettings`, and `MembershipProviderConfigurationSettings` classes that inherit the `ProviderConfigurationSettings` abstract base class. Every provider configuration settings class must implement the `Validate` method and `Settings` property of the `ProviderConfigurationSettings` base class because they're both marked as abstract.

Your custom provider configuration setting's implementation of the `Validate` method must contain the logic that performs the necessary validation. This method must return a Boolean value that specifies whether the validation succeeded. It must also populate a string passed into it as its argument with the appropriate error message if the validation fails.

Your custom provider configuration setting's implementation of the `Settings` property must return an `IDictionary` collection that contains the names and values of the attributes of the `Add` element that registers a provider with the specified provider-based service. This `Add` element is normally named `<add>`. The values of these attributes basically specify the configuration settings of the provider that the `<add>` element registers.

Let's consider the implementation of the `RolesProviderConfigurationSettings` class as shown in Listing 9-5.

Listing 9-5: The `RolesProviderConfigurationSettings` Class

```
public sealed class RolesProviderConfigurationSettings :
    ProviderConfigurationSettings
{
    private Hashtable settings;

    public RolesProviderConfigurationSettings()
    {
        this.settings = new Hashtable();
    }

    public override bool Validate(out string message)
    {
        if (this.ConnectionStringName.Length == 0)
        {
            message = "Connection string name is required";
            return false;
        }

        message = string.Empty;
        return true;
    }

    public string ApplicationName
    {
        get
        {
            if (this.settings["applicationName"] != null)
                return (string) this.settings["applicationName"];

            return string.Empty;
        }

        set { this.settings["applicationName"] = value; }
    }

    public string ConnectionStringName
    {
        get
        {
            if (this.settings["connectionStringName"] != null)
                return (string) this.settings["connectionStringName"];
        }
    }
}
```

(Continued)

Listing 9-5: (continued)

```
        return string.Empty;
    }

    set { this.settings["connectionStringName"] = value; }
}

public string Description
{
    get
    {
        if (this.settings["description"] != null)
            return (string) this.settings["description"];

        return string.Empty;
    }

    set { this.settings["description"] = value; }
}

protected override IDictionary Settings
{
    get { return this.settings; }
}
}
```

Follow these steps to implement a custom provider configuration settings class:

1. Derive your provider configuration settings class from the `ProviderConfigurationSettings` base class:

```
public sealed class RolesProviderConfigurationSettings :
    ProviderConfigurationSettings
```

2. Add a private field of type `Hashtable` named `settings` to your provider configuration settings class. You don't have to use a hashtable; any `IDictionary` will do the job.

```
private Hashtable settings;
```

3. Add a default constructor to your provider configuration settings class, which instantiates a `Hashtable` and assigns it to this private field:

```
public RolesProviderConfigurationSettings()
{
    this.settings = new Hashtable();
}
```

4. Override the `Settings` property of the `ProviderConfigurationSettings` base class to return a reference to the `settings` private field:

```
protected override IDictionary Settings
{
    get { return this.settings; }
}
```

5. Add a bunch of read/write properties to your provider configuration settings class, where each property exposes a specific item in the settings Hashtable. For example, the `RolesProviderConfigurationSettings` class exposes three read/write properties named `ApplicationName`, `ConnectionStringName`, and `Description`, which expose the `applicationName`, `connectionStringName`, and `description` contents of the settings Hashtable.

Keep in mind that the contents of the settings Hashtable map to the attributes on the Add XML element that is used to register a provider with your provider-based service. This Add XML element is normally named `<add>`. The attributes on the Add XML element basically specify the configuration settings of the provider being registered.

As you can see, your provider configuration settings class basically exposes the attributes on the Add XML element as strongly-typed properties. We've discussed the benefits of strongly-typed properties on several occasions in the previous chapters. To put it differently, your provider configuration settings class basically exposes the configuration settings of the providers of your provider-based service, hence the name `ProviderConfigurationSettings`.

Now you should clearly see the difference between a provider feature and provider configuration settings. A provider feature describes your provider-based service, whereas provider configuration settings describe the providers of your provider-based service.

6. Override the `Validate` method to include the logic that validates the values assigned to the properties of your provider configuration settings class. For example, the `RolesProviderConfigurationSettings` class's implementation of the `Validate` method contains the logic that determines whether the value of the `ConnectionStringName` property is set.

```
public override bool Validate(out string message)
{
    if (this.ConnectionStringName.Length == 0)
    {
        message = "Connection string name is required";
        return false;
    }

    message = string.Empty;
    return true;
}
```

Putting it All Together

Recall that Figures 9-1 through 9-6 took you through typical workflows that the clients of a provider-based service such as the `Roles` provider-based service use to update, remove, rename, and add providers for the service. Next, I take you under the hood of four of these workflows to help you understand the important roles that the `ProviderFeature` and `ProviderConfigurationSettings` base classes and their subclasses play in the integrated providers model.

Before diving into the internals of these workflows, let's briefly enumerate these four workflows:

- ❑ The workflow that takes you to the `ProviderConfigurationConsolidatedPage` module list page. This workflow consists of one of the following two activities:

Chapter 9: Understanding the Integrated Providers Model

- ❑ Double-clicking the Providers item in the workspace of the IIS 7 Manager (see Figure 9-1)
- ❑ Selecting the Providers item in the workspace and clicking the Open Feature link in the task panel
- ❑ The workflow that allows you to view the providers registered for a particular provider-based service. This workflow consists of a single activity: selecting a provider-based service from the Feature combo box.
- ❑ The workflow that allows you to add a new provider to a particular provider-based service. This workflow consists of the following activities:
 1. Clicking the Add link in the task panel associated with the `ProviderConfigurationConsolidatedPage` module list page to launch the `AddProviderForm` task form.
 2. Entering or selecting the desired configuration settings for the provider being added.
 3. Clicking the OK button on the `AddProviderForm` task form to add the provider to the underlying configuration file.
- ❑ The workflow that allows you to edit a provider of a particular provider-based service. This workflow consists of the following activities:
 1. Clicking the Edit link in the task panel associated with the `ProviderConfigurationConsolidatedPage` module list page to launch the `AddProviderForm` task form.
 2. Editing the desired configuration settings for the provider.
 3. Clicking the OK button on the `AddProviderForm` task form to update the provider in the underling configuration file.

Workflow that Displays the `ProviderConfigurationConsolidatedPage` Module List Page

Let's begin our discussions with Figure 9-1. Recall that the workspace shown in this figure contains an item named Providers. When the end user double-clicks this item to navigate to the `ProviderConfigurationConsolidatedPage` module list page shown in Figure 9-2, the `OnActivated` method of this module list page is automatically invoked. This method takes a single Boolean argument that specifies whether the module list page is being accessed for the first time. Listing 9-6 presents a portion of the `ProviderConfigurationConsolidatedPage` module list page's implementation of the `OnActivated` method.

Listing 9-6: The `OnActivated` Method

```
protected override void OnActivated(bool initialActivation)
{
    . . .

    if (initialActivation)
    {
        . . .
    }
}
```

Listing 9-6: (continued)

```
        IExtensibilityManager extensibilityManager =  
            (IExtensibilityManager)base.GetService(  
                typeof(IExtensibilityManager));  
  
        this.features =  
            extensibilityManager.GetExtensions(typeof(ProviderFeature));  
  
        foreach (ProviderFeature feature in this.features)  
        {  
            this.featuresComboBox.Items.Add(feature.FeatureName);  
        }  
  
        . . .  
    }  
  
    . . .  
}
```

As you can see, the `OnActivated` method first accesses the extensibility manager service:

```
IExtensibilityManager extensibilityManager =  
    (IExtensibilityManager)base.GetService(  
        typeof(IExtensibilityManager));
```

Recall that the extensibility manager service manages the extensions to the IIS 7 and ASP.NET integrated infrastructure. As discussed earlier, as part of the implementation of your custom provider-based service, you must also implement one or more module pages, which provide the clients of your custom provider-based service with the appropriate user interface to specify the configuration settings of your service other than adding, removing, updating, and renaming providers. Also recall that the custom module that registers these module pages (see Listing 9-3) must instantiate an instance of your custom provider feature and register the instance with the extensibility manager service. In other words, this provider feature instance is an extension to the integrated infrastructure. Recall from Listing 9-3 that this provider feature extension is registered with the extensibility manager service under the `Type` object that represents the `ProviderFeature` base class, that is, `typeof(ProviderFeature)`.

The extensibility manager service comes with a method named `GetExtensions` that takes a `Type` object and returns a collection that contains all extensions of the specified type. As you can see from Listing 9-6, the `OnActivated` method of the `ProviderConfigurationConsolidatedPage` module list page uses the `GetExtensions` method to return a collection that contains extensions of type `ProviderFeature`. Note that this method stores this collection in a private field named `features` for future reference.

```
this.features = service.GetExtensions(typeof(ProviderFeature));
```

As you can see, the `features` collection is a collection of provider feature objects where each provider feature object represents a particular provider-based service. As such, the `features` collection contains at least the three provider feature objects that represent the Roles, Users, and Profile provider-based services. These three provider feature objects are instances of the `RolesProviderConfigurationFeature`, `MembershipProviderConfigurationFeature`, and `ProfileProviderConfigurationFeature`, respectively. Recall that these three classes, like any other custom provider feature class, inherit from the

Chapter 9: Understanding the Integrated Providers Model

`ProviderFeature` base class. The `features` collection also contains the provider feature objects that represent your own custom provider-based services if you have taken the steps discussed earlier to register these provider feature objects with the extensibility manager service.

The `OnActivated` method then iterates through the `ProviderFeature` objects in this collection and adds the value of the `FeatureName` property of each object to the `Feature` combo box. This property basically contains the name of the provider-based service that the enumerated `ProviderFeature` object represents:

```
foreach (ProviderFeature feature in this.features)
{
    this.featuresComboBox.Items.Add(feature.FeatureName);
}
```

In summary, the custom module that registers the module pages of your provider-based service instantiates a provider feature that represents the service and registers this provider feature extension with the extensibility manager service. The `OnActivated` method of the `ProviderConfigurationConsolidatedPage` module list page, on the other hand, retrieves this provider feature extension from the extensibility manager service, stores it somewhere for future reference, extracts the feature name (`FeatureName`) from this provider feature extension, and adds this name to the `Feature` combo box.

Workflow for Viewing the Providers of a Provider-Based Service

When the user selects a provider-based service from the `Feature` combo box, the event handler registered for the `SelectedIndexChanged` event of this combo box is automatically invoked. This event handler is a method of the `ProviderConfigurationConsolidatedPage` module list page named `OnFeatureComboBoxSelectedIndexChanged`. Listing 9-7 presents the portion of the internal implementation of this method.

Listing 9-7: The Portion of the Implementation of the `OnFeatureComboBoxSelectedIndexChanged` Event Handler

```
private void OnFeatureComboBoxSelectedIndexChanged(object sender, EventArgs e)
{
    string selectedFeatureName = this.featuresComboBox.Text;
    ProviderFeature selectedFeature;
    foreach (ProviderFeature feature in this.features)
    {
        if (string.Equals(feature.FeatureName, selectedFeatureName,
            StringComparison.OrdinalIgnoreCase))
        {
            selectedFeature = feature;
            break;
        }
    }

    PropertyBag selectedFeatureInfoBag = new PropertyBag();
    selectedFeatureInfoBag[0] = selectedFeature.SectionName;
    selectedFeatureInfoBag[1] = selectedFeature.SelectedProviderPropertyName;
    selectedFeatureInfoBag[2] = selectedFeature.ProviderCollectionPropertyName;
    selectedFeatureInfoBag[3] = selectedFeature.ProviderBaseType;
```

Listing 9-7: (continued)

```
selectedFeatureInfoBag[4] = selectedFeature.ProviderConfigurationSettingNames;
selectedFeatureInfoBag[5] = selectedFeature.ConnectionStringRequired;
selectedFeatureInfoBag[6] = selectedFeature.ConnectionStringAttributeName;

this.DownloadAndDisplayProviders(selectedFeatureInfoBag);
}
```

As you can see, this event handler first retrieves the name of the selected provider-based service from the Feature combo box:

```
string selectedFeatureName = this.featuresComboBox.Text;
```

Next, it searches the `features` collection for the provider feature that represents the provider-based service with the specified name (recall that the `features` collection is a collection of provider feature objects where each provider feature object represents a particular provider-based service):

```
ProviderFeature selectedFeature;
foreach (ProviderFeature feature in this.features)
{
    if (string.Equals(feature.FeatureName, selectedFeatureName,
        StringComparison.OrdinalIgnoreCase))
    {
        selectedFeature = feature;
        break;
    }
}
```

Then, it instantiates a `PropertyBag` collection and populates the collection with the complete information about the selected provider-based service. Recall that the provider feature object that represents a provider-based service exposes the complete information about the service through its strongly-typed properties:

```
PropertyBag selectedFeatureInfoBag = new PropertyBag();
selectedFeatureInfoBag[0] = selectedFeature.SectionName;
selectedFeatureInfoBag[1] = selectedFeature.SelectedProviderPropertyName;
selectedFeatureInfoBag[2] = selectedFeature.ProviderCollectionPropertyName;
selectedFeatureInfoBag[3] = selectedFeature.ProviderBaseType;
selectedFeatureInfoBag[4] = selectedFeature.ProviderConfigurationSettingNames;
selectedFeatureInfoBag[5] = selectedFeature.ConnectionStringRequired;
selectedFeatureInfoBag[6] = selectedFeature.ConnectionStringAttributeName;
```

Finally, it invokes a method named `DownloadAndDisplayProviders` passing in the `PropertyBag` collection to download the providers registered for the selected provider-based service from the Collection XML element — with the name specified in the third item in the `PropertyBag` collection — of the configuration section with the name specified in the first item in the `PropertyBag` collection and to display these providers in the list view below the Feature combo box:

```
this.DownloadAndDisplayProviders(selectedFeatureInfoBag);
```

Workflow for Adding a New Provider to a Provider-Based Service

When the user clicks the Add link button in the task panel associated with the `ProviderConfigurationConsolidatedPage` module list page, this module list page automatically instantiates and launches the `AddProviderForm` task form shown in Figure 9-3. When the `ProviderConfigurationConsolidatedPage` module list page is instantiating the `AddProviderForm` task form, it passes the following two pieces of information into the constructor of this task form:

- ❑ The provider configuration settings object whose properties specify the configuration settings of the provider being added. The `ProviderFeature` object that represents the selected provider-based service exposes a property of type `ProviderConfigurationSettings` named `Settings` that returns a reference to this provider configuration settings object. The `ProviderConfigurationConsolidatedPage` module list page passes the value of this `Settings` property into the constructor of the `AddProviderForm` task form.
- ❑ The `PropertyBag` collection that contains the complete information about the selected provider-based service. As you saw earlier, this `PropertyBag` collection is populated with the values of the properties of the `ProviderFeature` object that represents the selected provider-based service:

```
PropertyBag selectedFeatureInfoBag = new PropertyBag();
selectedFeatureInfoBag[0] = selectedFeature.SectionName;
selectedFeatureInfoBag[1] = selectedFeature.SelectedProviderPropertyName;
selectedFeatureInfoBag[2] = selectedFeature.ProviderCollectionPropertyName;
selectedFeatureInfoBag[3] = selectedFeature.ProviderBaseType;
selectedFeatureInfoBag[4] = selectedFeature.ProviderConfigurationSettingNames;
selectedFeatureInfoBag[5] = selectedFeature.ConnectionStringRequired;
selectedFeatureInfoBag[6] = selectedFeature.ConnectionStringAttributeName;
```

The `AddProviderForm` task form internally invokes a method named `DownloadProviderTypes`, passing in the complete type information about the provider base of the selected provider based-service. As just mentioned, the `ProviderConfigurationConsolidatedPage` module list page passes a `PropertyBag` collection to the `AddProviderForm` task form that contains this complete type information, among other pieces of information. Recall that all providers of a given provider-based service inherit from a base class known as a provider base. For example, all providers of the Roles provider-based service inherit the `RoleProvider` provider base.

As the name suggests, the `DownloadProviderTypes` method downloads all provider types that inherit the specified provider base type. The server-side code under the hood uses .NET reflection and searches through the appropriate assemblies for those types that inherit the specified provider base type and returns the assembly-qualified names of these types to the `AddProviderForm` task form, where this task form displays these names in the Type combo box for the user to choose from. Which assemblies the search is performed through on the server-side depends on which level the user is registering her provider for.

This is very similar to the discussions in the previous chapter regarding the level for which you register an HTTP module or handler. Recall that this registration can be done in one of the following two levels:

- ❑ The IIS 7 Web server level
- ❑ A particular Web site, Web application, or virtual directory level

As discussed in the previous chapter, the Type combo box in the Add Managed Handler (or Add Managed Module) task form displays the list of HTTP handlers (or modules) to choose from. When the Add Managed Handler (or Add Managed Module) task form is launched, this task form under the hood uses the appropriate proxy to download the list of available HTTP handlers (or modules) from the server. As you saw in the previous chapter, the underlying server-side code searches through the assemblies in the Global Assembly Cache (GAC) for those types that implement the `IHttpHandler` (or `IHttpModule`) if you're registering an HTTP handler (or HTTP module) at the IIS 7 Web server level. This server-side code searches through the referenced assemblies for those types that implement the `IHttpHandler` (or `IHttpModule`) if you're registering an HTTP handler (or HTTP module) at a particular Web site, Web application, or virtual directory level. As you saw, this affects how you should go about compiling your custom HTTP handlers, modules, and handler factories. If you want to allow users to register your custom HTTP handler, module, or handler factory at the IIS 7 Web server level, you must compile your custom HTTP handler, module, or handler factory into a strongly-named assembly and deploy this assembly to the GAC. If you want to allow users to register your custom HTTP handler, module, or handler factory at a particular Web site, Web application, or virtual directory level, you have several compilation options but there is one requirement. You must add a reference to the assembly that contains your custom HTTP handler, module, or handler factory to the Web site or Web application that needs to use your custom HTTP handler, module, or handler factory.

The same exact argument applies when you launch the `AddProviderForm` task form to register a new provider for a given provider-based service. This task form under the hood uses the appropriate proxy to download the list of registered providers from the server. The server-side code searches through the assemblies in the GAC for those types that inherit the specified provider base type if you're registering a provider at the IIS 7 Web server level. This server-side code searches through the referenced assemblies for those types that inherit the specified provider base type if you're registering a provider at a particular Web site, Web application, or virtual directory level. Similarly, this affects how you should go about compiling a provider type of your custom provider-based service. If you want to allow users to register providers of the same type as your provider type at the IIS 7 Web server level, you must compile your provider type into a strongly-named assembly and deploy this assembly to the GAC. If you want to allow users to register providers of your provider type at a particular Web site, Web application, or virtual directory level, you have several compilation options but there is one requirement. You must add a reference to the assembly that contains your provider type to the Web site or Web application that needs to use your provider type.

After invoking the `DownloadProviderTypes` method and downloading the provider types of the selected provider-based service, the `AddProviderForm` task form displays these provider types in the Type combo box shown in Figure 9-3.

Recall from Figure 9-3 that the `AddProviderForm` task form contains a grid that displays a bunch of controls, such as textboxes with labels. This grid is an instance of a control named `PropertyGrid`, which exposes a property named `SelectedObject`. The `AddProviderForm` task form assigns the provider configuration settings object that it has received from the `ProviderConfigurationConsolidatedPage` module list page to the `SelectedObject` property of the `PropertyGrid` control.

The `PropertyGrid` control under the hood uses .NET reflection to retrieve the names and values of the properties of the object assigned to its `SelectedObject` property. The control then renders a label and a control such as a textbox for each property of this object and displays the name of the property in the label. Every time the user enters a new value into a textbox, or edits a value, the `PropertyGrid` control automatically updates the value of the object assigned to its `SelectedObject` property, which is the provider configuration settings object in this case.

Chapter 9: Understanding the Integrated Providers Model

To help you understand the significance of the `PropertyGrid` control and how you can customize this control for your own provider-based services, I'll walk you through an exercise. Launch Visual Studio. Add a new Windows Forms Application project named `MyApplication`. This will automatically add a new `Form1.cs` file to the project. Change the name of this file to `MyForm.cs`. Right-click the `MyApplication` node in the Solution Explorer panel and select the Properties option from the popup menu to launch the Properties page. Switch to the Application tab and enter "MyNamespace" into the "Default namespace" textfield.

Add a new source file named `MyClass1.cs` to the `MyApplication` project and add the code shown in Listing 9-8 to this source file.

Listing 9-8: The `MyClass1` Class

```
using System;
using System.ComponentModel;

namespace MyNamespace
{
    [DefaultProperty("MyClass1Property3")]
    class MyClass1
    {
        private bool myClass1Field1;
        private string myClass1Field2;
        private MyEnum myClass1Field3;
        private DateTime myClass1Field4;
        private string[] myClass1Field5;
        private MyClass2[] myClass1Field6;

        [Description("This is MyClass1Property1 property.")]
        [Category("Non-Collection Properties")]
        [DefaultValue(false)]
        public bool MyClass1Property1
        {
            get { return this.myClass1Field1; }
            set { this.myClass1Field1 = value; }
        }

        [Description("This is MyClass1Property2 property.")]
        [Category("Non-Collection Properties")]
        [DefaultValue("")]
        public string MyClass1Property2
        {
            get { return this.myClass1Field2; }
            set { this.myClass1Field2 = value; }
        }

        [Description("This is MyClass1Property3 property.")]
        [Category("Non-Collection Properties")]
        [DefaultValue("MyValue1")]
        public MyEnum MyClass1Property3
        {
            get { return this.myClass1Field3; }
            set { this.myClass1Field3 = value; }
        }
    }
}
```

Listing 9-8: (continued)

```
    }

    [Description("This is MyClass1Property4 property.")]
    [Category("Non-Collection Properties")]
    [DefaultValue("")]
    public DateTime MyClass1Property4
    {
        get { return this.myClass1Field4; }
        set { this.myClass1Field4 = value; }
    }

    [Description("This is MyClass1Property5 property.")]
    [Category("Collection Properties")]
    [DefaultValue("")]
    public string[] MyClass1Property5
    {
        get { return this.myClass1Field5; }
        set { this.myClass1Field5 = value; }
    }

    [Description("This is MyClass1Property6 property.")]
    [Category("Collection Properties")]
    [DefaultValue("")]
    public MyClass2[] MyClass1Property6
    {
        get { return this.myClass1Field6; }
        set { this.myClass1Field6 = value; }
    }
}
}
```

As you can see, `MyClass1` is a simple class with six read/write properties named `MyClass1Property1`, `MyClass1Property2`, `MyClass1Property3`, `MyClass1Property4`, `MyClass1Property5`, and `MyClass1Property6` of types `bool`, `string`, `MyEnum`, `DateTime`, `string[]`, and `MyClass2[]`, respectively. Note that `MyEnum` is a simple enumeration type with four values named `MyValue1`, `MyValue2`, `MyValue3`, and `MyValue4` as shown in Listing 9-9.

Now add a new source file named `MyEnum.cs` to the `MyApplication` project and add the code shown in Listing 9-9 to this source file.

Listing 9-9: The `MyEnum` Enumeration

```
namespace MyNamespace
{
    enum MyEnum
    {
        MyValue1, MyValue2, MyValue3, MyValue4
    }
}
```

Chapter 9: Understanding the Integrated Providers Model

As you can see from Listing 9-10, `MyClass1` exposes a property of type `MyClass2[]` named `MyClass1Property6`. Listing 9-10 presents the implementation of `MyClass2`. Add a new source file to the `MyApplication` project and add the code shown in Listing 9-10 to this source file.

Listing 9-10: The `MyClass2` Class

```
using System;
using System.ComponentModel;

namespace MyNamespace
{
    [DefaultProperty("MyClass2Property2")]
    class MyClass2
    {
        private DateTime myClass2Field1;
        private string[] myClass2Field2;

        [Description("This is MyClass2Property1 property.")]
        [Category("Non-Collection Properties")]
        [DefaultValue("")]
        public DateTime MyClass2Property1
        {
            get { return this.myClass2Field1; }
            set { this.myClass2Field1 = value; }
        }

        [Description("This is MyClass2Property2 property.")]
        [Category("Collection Properties")]
        [DefaultValue("")]
        public string[] MyClass2Property2
        {
            get { return this.myClass2Field2; }
            set { this.myClass2Field2 = value; }
        }
    }
}
```

Now add a `PropertyGrid` control and a `Button` control to the `MyForm` Windows Form as shown in Listing 9-11, which presents the content of the `MyForm.Designer.cs` file.

Listing 9-11: The Content of the `MyForm.Designer.cs` File

```
namespace MyNamespace
{
    partial class MyForm
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
                components.Dispose();
        }
    }
}
```

Listing 9-11: (continued)

```
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

    private void InitializeComponent()
    {
        this.propertyGrid1 = new System.Windows.Forms.PropertyGrid();
        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        this.propertyGrid1.Anchor = ((System.Windows.Forms.AnchorStyles) (
            (System.Windows.Forms.AnchorStyles.Top |
             System.Windows.Forms.AnchorStyles.Left) |
             System.Windows.Forms.AnchorStyles.Right)));
        this.propertyGrid1.Location = new System.Drawing.Point(-1, 1);
        this.propertyGrid1.Name = "propertyGrid1";
        this.propertyGrid1.Size = new System.Drawing.Size(416, 219);
        this.propertyGrid1.TabIndex = 0;
        this.button1.Location = new System.Drawing.Point(167, 235);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(75, 23);
        this.button1.TabIndex = 1;
        this.button1.Text = "OK";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click += new System.EventHandler(this.button1_Click);
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleModeMode.Font;
        this.ClientSize = new System.Drawing.Size(412, 270);
        this.Controls.Add(this.button1);
        this.Controls.Add(this.propertyGrid1);
        this.Name = "MyForm";
        this.Text = "MyForm";
        this.ResumeLayout(false);
    }

    #endregion

    private System.Windows.Forms.PropertyGrid propertyGrid1;
    private System.Windows.Forms.Button button1;
}
}
```

Note that Listing 9-11 also registers a method named `Button1_Click` as the event handler for the Click event of the Button. Listing 9-12 presents the content of the `MyForm.cs` file.

Listing 9-12: The Content of the `MyForm.cs` File

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

(Continued)

Listing 9-12: (continued)

```
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MyNamespace
{
    public partial class MyForm : Form
    {
        MyClass1 myobj = new MyClass1();

        public MyForm()
        {
            InitializeComponent();
            propertyGrid1.SelectedObject = myobj;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            string msg = string.Empty;
            msg += ("MyClass1Property1 = " + myobj.MyClass1Property1.ToString());
            msg += ("\n\nMyClass1Property2 = " + myobj.MyClass1Property2);
            msg += ("\n\nMyClass1Property3 = " + myobj.MyClass1Property3.ToString());
            msg += ("\n\nMyClass1Property4 = " + myobj.MyClass1Property4.ToString());

            msg += "\n\nMyClass1Property5 = { ";
            for (int i = 0; i < myobj.MyClass1Property5.Length; i++)
            {
                msg += myobj.MyClass1Property5[i];
                if (i != myobj.MyClass1Property5.Length - 1)
                    msg += ", ";
            }
            msg += " }";

            msg += "\n\nMyClass1Property6 : \n";
            for (int j=0; j<myobj.MyClass1Property6.Length; j++)
            {
                msg += ("MyClass2Property1 = "+
                    myobj.MyClass1Property6[j].MyClass2Property1.ToShortDateString()+"\n");
                msg += "MyClass2Property2 = { ";
                for (int k=0; k<myobj.MyClass1Property6[j].MyClass2Property2.Length; k++)
                {
                    msg += myobj.MyClass1Property6[j].MyClass2Property2[k];
                    if (k != myobj.MyClass1Property6[j].MyClass2Property2.Length - 1)
                        msg += ", ";
                }
                msg += " }\n";
            }

            MessageBox.Show(msg);
        }
    }
}
```

Note that the `MyForm` class features a private field of type `MyClass1` named `myobj`:

```
MyClass1 myobj = new MyClass1();
```

As Listing 9-12 shows, the constructor of the `MyForm` Windows Form assigns the `myobj` object to the `SelectedObject` property of the `PropertyGrid` control. We claimed earlier that the `PropertyGrid` control is capable of displaying the appropriate user interface to allow you to edit the properties of the object assigned to its `SelectedObject` property. Let's see this in action. Go ahead and run the `MyApplication` application. You should get the result shown in Figure 9-10.

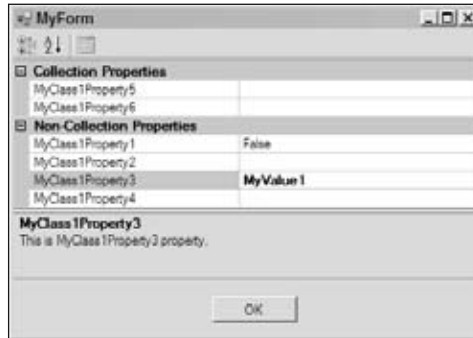


Figure 9-10

Here are a few observations about Figure 9-10:

- ❑ The properties are divided into two groups named “Collection Properties” and “Non-Collection Properties.” You may be wondering who did this grouping. The answer lies in Listing 9-9 where we’ve annotated the `MyClass1Property5` and `MyClass1Property6` properties with the `[Category("Non-Collection Properties")]` metadata attribute and the `MyClass1Property1`, `MyClass1Property2`, `MyClass1Property3`, and `MyClass1Property4` properties with the `[Category("Collection Properties")]` metadata attribute. Decorating a property with the `Category` metadata attribute instructs the `PropertyGrid` control to display the property in the specified category.
- ❑ The label that displays the text “`MyClass1Property3`” is highlighted. You may be wondering why this property in particular is highlighted. The answer lies in Listing 9-9 where we’ve annotated the `MyClass1` class with `[DefaultProperty("MyClass1Property3")]`. Annotating a class with this the `DefaultProperty` metadata attribute instructs the `PropertyGrid` control to highlight the specified property. In other words, this property is treated as the default property of the class.
- ❑ When you click a property, the bottom area of the `PropertyGrid` control displays the name of the property in bold and a short description about the property. This short description comes from Listing 9-9 where we’ve annotated each property of the `MyClass1` class with the `Description` metadata attribute. For example, the `MyClass1Property3` property is annotated with this metadata attribute as follows:

```
[Description("This is MyClass1Property3 property.")]  
[Category("Non-Collection Properties")]
```

```
[DefaultValue("MyValue1")]
public MyEnum MyClass1Property3
{
    get { return this.myClass1Field3; }
    set { this.myClass1Field3 = value; }
}
```

- ❑ This instructs the PropertyGrid control to display the text "This is MyClass1Property3 property" when the user selects the MyClass1Property3 property.

If you click the textfield associated with the MyClass1Property5 property in Figure 9-10, you'll see the result shown in Figure 9-11 where this textfield contains a little button labeled "...". As you can see, the PropertyGrid control uses .NET reflection to determine whether a property is a collection. If so, it automatically displays this little button in the textfield associated with the property.

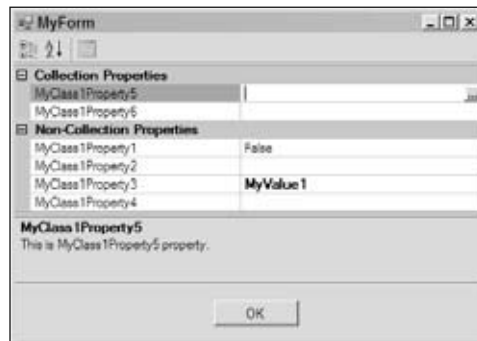


Figure 9-11

If you click this little button, the PropertyGrid control automatically pops up the String Collection Editor shown in Figure 9-12. Again thanks to .NET reflection, the PropertyGrid control has determined that the MyClass1Property5 property is a string collection and consequently pops up the String Collection Editor to allow you to edit the value of this property.

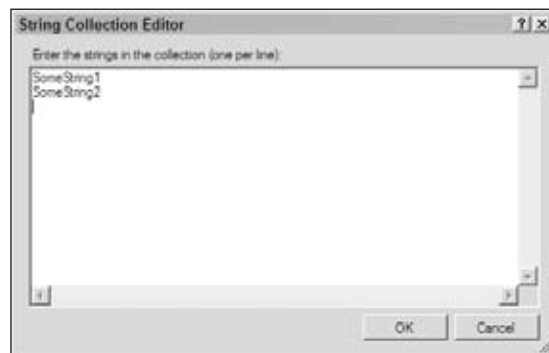


Figure 9-12

If you click the `MyClass1Property6` property in Figure 9-10, you'll see the result shown in Figure 9-13 where the associated textfield contains the little button we discussed earlier. Again this means that the `PropertyGrid` control was able to determine that the `MyClass1Property6` property is a collection.

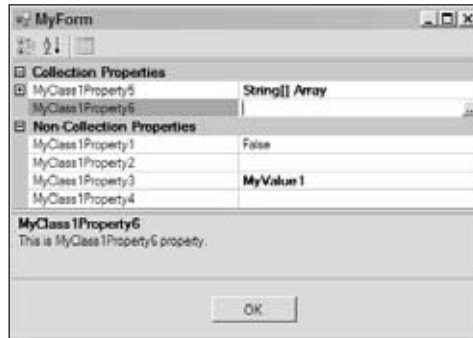


Figure 9-13

Now if you click the button, the `PropertyGrid` control will pop up a different editor, that is, the `MyClass2` Collection Editor, as shown in Figure 9-14. In other words, the `PropertyGrid` control is capable of making a distinction between the `MyClass1Property5` and `MyClass1Property6` collection properties.

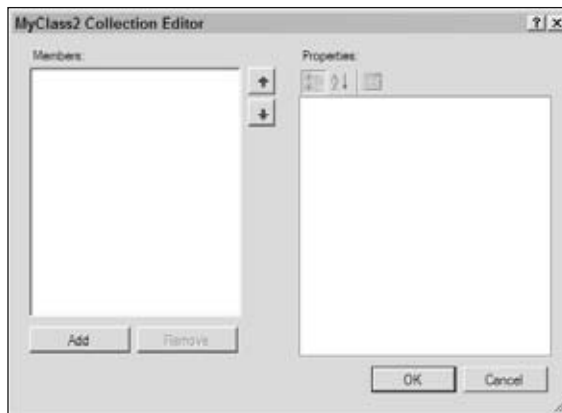


Figure 9-14

Now if you click the `Add` button shown in Figure 9-14, you'll see the result shown in Figure 9-15. The `PropertyGrid` control was able to determine that the `MyClass1Property6` collection property is a collection of `MyClass2` objects and consequently it displays another `PropertyGrid` control where you can edit the properties of the `MyClass2` object being added to the `MyClass1Property6` collection property.

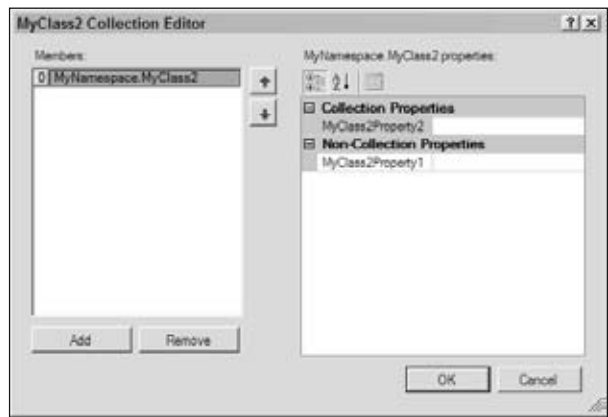


Figure 9-15

If you click the `MyClass1Property3` property in Figure 9-10, you'll see the result shown in Figure 9-16. As you can see, the `PropertyGrid` control automatically uses a combo box to display the legal values of the `MyEnum` enumeration, which is the type of the `MyClass1Property3`.

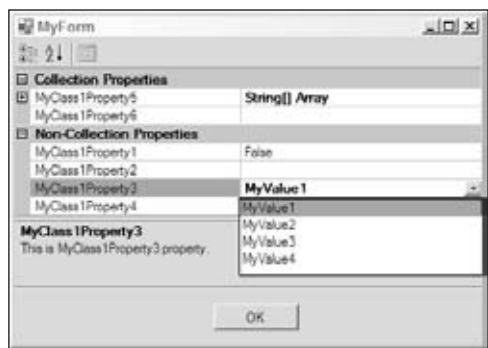


Figure 9-16

If you click the `MyClass1Property4` property in Figure 9-10, you'll see the result shown in Figure 9-17. As you can see, the `PropertyGrid` control uses .NET reflection to determine that this property is of type `DateTime` and pops up a calendar control to allow you to edit the value of this property.

If none of the standard editors such as calendar, string collection editor, and so on that the `PropertyGrid` control uses by default are appropriate for a particular type of property, you can always implement your own custom editor and annotate your property with the appropriate metadata attribute to instruct the `PropertyGrid` control to pop up your custom editor. This topic is beyond the scope of this book.

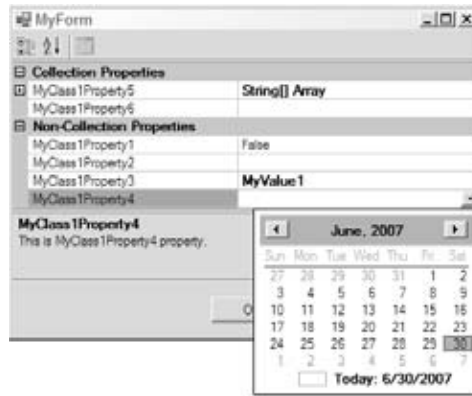


Figure 9-17

We claimed that when you edit a property in the `PropertyGrid` control or one of the editors that this control pops up, these changes are automatically reflected in the object assigned to the `SelectedObject` property of the control. To help you see this in action, I've added an OK button to the `MyForm` Windows Form as shown in Figure 9-10 and registered an event handler named `Button1_Click` for the `Click` event of this button. The following code listing presents the implementation of this event handler:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = string.Empty;
    msg += ("MyClass1Property1 = " + myobj.MyClass1Property1.ToString());
    msg += ("\n\nMyClass1Property2 = " + myobj.MyClass1Property2);
    msg += ("\n\nMyClass1Property3 = " + myobj.MyClass1Property3.ToString());
    msg += ("\n\nMyClass1Property4 = " + myobj.MyClass1Property4.ToString());

    msg += "\n\nMyClass1Property5 = { ";
    for (int i = 0; i < myobj.MyClass1Property5.Length; i++)
    {
        msg += myobj.MyClass1Property5[i];
        if (i != myobj.MyClass1Property5.Length - 1)
            msg += ", ";
    }
    msg += " }";

    msg += "\n\nMyClass1Property6 : \n";
    for (int j=0; j<myobj.MyClass1Property6.Length; j++)
    {
        msg += ("MyClass2Property1 = "+
            myobj.MyClass1Property6[j].MyClass2Property1.ToShortDateString()+"\n");
        msg += "MyClass2Property2 = { ";
        for (int k = 0; k < myobj.MyClass1Property6[j].MyClass2Property2.Length; k++)
        {
            msg += myobj.MyClass1Property6[j].MyClass2Property2[k];
            if (k != myobj.MyClass1Property6[j].MyClass2Property2.Length - 1)
                msg += ", ";
        }
    }
}
```

```
    msg += " }\n";  
}  
  
MessageBox.Show(msg);  
}
```

If you run the `MyApplication` application, use the `PropertyGrid` control and its editors to edit the displayed properties, and click the OK button, the event handler will pop up the message shown in Figure 9-18 where the names and values of the properties of the object assigned to the `SelectedObject` property of the `PropertyGrid` control are displayed.

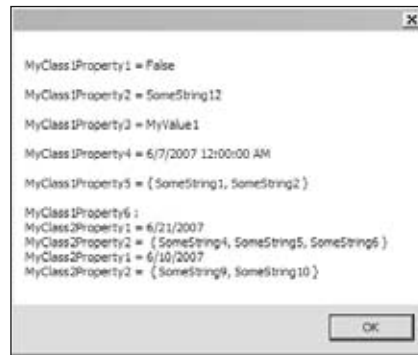


Figure 9-18

Now that you've learned a great deal about the `PropertyGrid` control, let's go back to our original discussion that is the `AddProviderForm` task form where we left off. As Figure 9-3 shows, this task form consists of four different parts:

- ❑ **Type combo box:** As discussed earlier, the `AddProviderForm` task form invokes the `DownloadProviderTypes` method to download the provider types that the selected provider-based service supports from the server and displays them in the Type combo box.
- ❑ **Friendly name textfield:** This is where the user enters a friendly name for the provider being added.
- ❑ **PropertyGrid:** The `AddProviderForm` task form assigns the provider configuration settings object that it receives from the `ProviderConfigurationConsolidatedPage` module list page to the `SelectedObject` property of this `PropertyGrid` control. Recall that the type of this provider configuration settings object depends on the type of the selected provider-based service. For example, if the end user has selected the Roles provider-based service from the Feature combo box of the `ProviderConfigurationConsolidatedPage` module list page, this module list page will pass an instance of the `RolesProviderConfigurationSettings` class into the constructor of the `AddProviderForm` task form.
- ❑ **Therefore, the `PropertyGrid` control displays the properties of the provider configuration settings object attached to its `SelectedObject` property.** When you're implementing a custom provider configuration settings class for your custom provider-based service, you should annotate the properties of this class with the appropriate metadata attributes as discussed earlier to improve the user experience with the `AddProviderForm` task form. As you can see, all of our previous discussions about the `PropertyGrid` control equally apply here.

When the user is finally done with editing the configuration settings of the provider being added and clicks the OK button on the `AddProviderForm` task form, the event handler registered for the `Click` event of this button is automatically invoked. Listing 9-13 presents a portion of the internal implementation of this event handler.

Listing 9-13: The `OnOkButtonClick` Method

```
private void OnOkButtonClick(object sender, EventArgs e)
{
    string message = null;
    if (!this.settings.Validate(out message))
        base.ShowMessage(message);

    else
    {
        string[] settingNamesAndValues =
            new string[this.providerConfigurationSettingNames.Length * 2];
        IDictionary settings = this.settings.GetSettings();
        int index = 0;
        foreach (string settingName in this.providerConfigurationSettingNames)
        {
            string settingValue = (string)settings[settingName];
            if (settingValue == null)
                settingValue = string.Empty;

            settingNamesAndValues[index] = settingName;
            settingNamesAndValues[index + 1] = settingValue;
            index += 2;
        }
        PropertyBag argument = new PropertyBag();
        argument[0] = this.friendlyNameTextBox.Text.Trim();

        if (!this.inModificationMode)
            argument[1] = this.GetProviderTypeFromTypeComboBox();
        else
            argument[1] = this.providerType;

        argument[2] = settingNamesAndValues;
        AddOrUpdateProvider(argument);
    }
}
```

Next, I walk you through the implementation of the `OnOkButtonClick` method. This method first invokes the `Validate` method on the provider configuration settings object:

```
string message = null;
if (!this.settings.Validate(out message))
    base.ShowMessage(message);
```

Note that the `OnOkButtonClick` method passes an out string parameter into the `Validate` method. Your custom provider configuration settings class's implementation of the `Validate` method must include the appropriate validation logic to validate the user inputs and return a Boolean value to specify whether the validation succeeded. If the validation fails, the `Validate` method should assign an error

Chapter 9: Understanding the Integrated Providers Model

message to the out string parameter. As you can see, the `OnOkButtonClick` method uses the `ShowMessage` method to display this error message to the end user.

If the validation succeeds, the `OnOkButtonClick` method first instantiates a new string array, double the size of the array that contains the provider configuration settings names:

```
string[]settingNamesAndValues =  
    new string[this.providerConfigurationSettingNames.Length * 2];
```

Then, it calls the `GetSettings` method on the provider configuration settings object to return the `IDictionary` collection that contains the setting names and values:

```
IDictionary settings = this.settings.GetSettings();
```

Next, it iterates through the strings in the `providerConfigurationSettingNames` array. Recall that this array contains the names of the configuration settings of the providers of the selected provider-based service. Note that the `OnOkButtonClick` method uses these names in this array as an index into the `IDictionary` collection returned from the `GetSettings` method to access their associated values. Thanks to the fact that the `PropertyGrid` control in the `AddProviderForm` task form automatically updates the properties of the provider configuration settings object assigned to its `SelectedObject` property with the user inputs, the `IDictionary` collection that this provider configuration settings object returns contains the values that the end user has entered into the `PropertyGrid` control of the `AddProviderForm` task form.

Note that both the name and value of the properties of the provider configuration settings object are stored in the `settingNamesAndValues`:

```
int index = 0;  
foreach (string settingName in this.providerConfigurationSettingNames)  
{  
    string settingValue = (string)settings[settingName];  
    if (settingValue == null)  
        settingValue = string.Empty;  
  
    settingNamesAndValues[index] = settingName;  
    settingNamesAndValues[index + 1] = settingValue;  
    index += 2;  
}
```

Next, it creates a `PropertyBag` collection:

```
PropertyBag argument = new PropertyBag();
```

Then, it retrieves the provider's friendly name from the associated textbox and stores it in the `PropertyBag` collection:

```
argument[0] = this.friendlyNameTextBox.Text.Trim();
```

Next it calls another method named `GetProviderTypeFromTypeComboBox` to retrieve the selected provider type from the `Type` combo box and stores it in the `PropertyBag` collection:

```
if (!this.inModificationMode)  
    argument[1] = this.GetProviderTypeFromTypeComboBox();
```

Next, it stores the `settingNamesAndValues` array in the `PropertyBag` collection:

```
argument[2] = settingNamesAndValues;
```

Finally, it invokes another method named `AddOrUpdateProvider` passing in the `PropertyBag` collection to add a new provider in the underlying configuration file:

```
AddOrUpdateProvider(argument);
```

Workflow for Updating a Provider of a Provider-Based Service

When the user clicks the Edit link button in the task panel associated with the `ProviderConfigurationConsolidatedPage` module list page, this module list page automatically instantiates and launches the `AddProviderForm` task form shown in Figure 9-5. When the `ProviderConfigurationConsolidatedPage` module list page is instantiating the `AddProviderForm` task form, it passes the following three pieces of information into the constructor of this task form:

- ❑ The provider configuration settings object whose properties specify the configuration settings of the provider being edited.
- ❑ The `PropertyBag` collection that contains the complete information about the selected provider-based service.
- ❑ The `PropertyBag` collection that contains the complete information about the selected provider, which is the provider the user has selected from the list of displayed providers. This `PropertyBag` collection contains the following pieces of information about the selected provider:
 - ❑ Friendly name
 - ❑ Type
 - ❑ Parameters other than the friendly name and type

The `AddProviderForm` task form invokes the `LoadSettings` method on the provider configuration settings object passing in the parameters of the selected provider other than its name and type. As just mentioned, the `ProviderConfigurationConsolidatedPage` module list page passes a `PropertyBag` collection into the constructor of the `AddProviderForm` task form, which contains three items, where the third item contains the parameters of the selected provider.

As you can see from the following excerpt from Listing 9-4, the `LoadSettings` method basically loads the settings collection of the provider configuration settings object with the parameters of the selected provider. Therefore, after this method is invoked, the properties of the provider configuration settings object contain the values of the associated configuration settings of the selected provider.

```
public void LoadSettings(string[] parameters)
{
    for (int i = 0; i < parameters.Length; i += 2)
    {
        this.Settings[parameters[i]] = parameters[i + 1];
    }
}
```

Chapter 9: Understanding the Integrated Providers Model

The `AddProviderForm` task form then assigns this provider configuration settings object to the `SelectedObject` property of the `PropertyGrid` control. This means that the `PropertyGrid` control now displays the current configuration settings of the selected provider.

When the user is finally done with all the editing and clicks the OK button on the `AddProviderForm` task form, the event handler shown in Listing 9-13 is automatically invoked.

IProviderConfigurationService

Next, I discuss the provider configuration service and its important role in the integrated providers model. The provider configuration service, like any other service in the IIS 7 and ASP.NET integrated infrastructure, implements an interface. This interface in this case is an interface named `IProviderConfigurationService`. As Listing 9-14 shows, this interface exposes a single method named `ConfigureProvider` that takes a single argument of type `ProviderFeature`.

Listing 9-14: The `IProviderConfigurationService` Interface

```
public interface IProviderConfigurationService
{
    bool ConfigureProvider(ProviderFeature feature);
}
```

The caller of the `ConfigureProvider` method (I discuss shortly who the caller is) must pass a provider feature representing a particular provider-based service into this method to have this method configure its providers, hence the name `ConfigureProvider`. The integrated providers model comes with a single implementation of the `IProviderConfigurationService` named `ProviderConfigurationModule` as shown in Listing 9-15.

Listing 9-15: The `ProviderConfigurationModule` Class

```
public sealed class ProviderConfigurationModule : Module,
                                                IProviderConfigurationService
{
    protected override void Initialize(IServiceProvider serviceProvider,
                                       ModuleInfo moduleInfo)
    {
        base.Initialize(serviceProvider, moduleInfo);
        IControlPanel service =
            (IControlPanel)serviceProvider.GetService(typeof(IControlPanel));
        ModulePageInfo itemPageInfo =
            new ModulePageInfo(this, typeof(ProviderConfigurationConsolidatedPage),
                              "Providers", "Providers", null, null);
        service.RegisterPage(ControlPanelCategoryInfo.ApplicationDevelopment,
                             itemPageInfo);
        service.RegisterPage(ControlPanelCategoryInfo.AspNet, itemPageInfo);

        IServiceContainer container =
            (IServiceContainer)this.GetService(typeof(IServiceContainer));
        if (container != null)
            container.AddService(typeof(IProviderConfigurationService), this);
    }
}
```

```
bool IProviderConfigurationService.ConfigureProvider(ProviderFeature feature)
{
    Connection service = (Connection)this.GetService(typeof(Connection));
    if (!string.IsNullOrEmpty(feature.SelectedProvider))
    {
        ProviderConfigurationModuleProxy proxy =
            (ProviderConfigurationModuleProxy)service.CreateProxy(this,
                                                                    typeof(ProviderConfigurationModuleProxy));

        PropertyBag infoBag = new PropertyBag();
        infoBag[0] = feature.SectionName;
        infoBag[1] = feature.SelectedProviderPropertyName;
        infoBag[2] = feature.ProviderCollectionPropertyName;
        infoBag[3] = feature.ProviderBaseType;
        infoBag[4] = feature.ProviderConfigurationSettingNames;

        if (proxy.SelectProvider(infoBag, feature.SelectedProvider))
            return true;

        return false;
    }

    ((INavigationService)this.GetService(
        typeof(INavigationService))).Navigate(service,
                                                service.ConfigurationPath,
                                                typeof(ProviderConfigurationConsolidatedPage),
                                                feature);

    return true;
}
}
```

As Listing 9-15 shows, the `ProviderConfigurationModule` class not only implements the `IProviderConfigurationService` interface, but also inherits from the `Module` base class. This means that the `ProviderConfigurationModule` is both a module and a provider configuration service. Recall that a module is a class that inherits from the `Module` base class and registers one or more module pages with the IIS 7 and ASP.NET integrated graphical management system. As you can see from the `ProviderConfigurationModule` module's implementation of the `Initialize` method, `ProviderConfigurationModule` is the module that registers the `ProviderConfigurationConsolidatedPage` module list page. In other words, the same module that registers the `ProviderConfigurationConsolidatedPage` module list page also acts as a provider configuration service.

Next, I walk you through the `ProviderConfigurationModule` class's implementation of the `Initialize` method of the `Module` base class and the `ConfigureProvider` method of the `IProviderConfigurationService` interface. I begin with the coverage of the implementation of the `Initialize` method.

As you can see from Listing 9-15, the `Initialize` method first invokes the `GetService` method to access the control panel service:

```
IControlPanel service =
    (IControlPanel)serviceProvider.GetService(typeof(IControlPanel));
```


Chapter 9: Understanding the Integrated Providers Model

Next, `Initialize` instantiates a `ModulePageInfo` module page info to represent the `ProviderConfigurationConsolidatedPage` module list page:

```
ModulePageInfo itemPageInfo =  
    new ModulePageInfo(this, typeof(ProviderConfigurationConsolidatedPage),  
        "Providers", "Providers", null, null);
```

Then, it invokes the `RegisterPage` method twice on the control panel service to register the `ProviderConfigurationConsolidatedPage` module list page under the application development and ASP.NET categories:

```
service.RegisterPage(ControlPanelCategoryInfo.ApplicationDevelopment, itemPageInfo);  
service.RegisterPage(ControlPanelCategoryInfo.AspNet, itemPageInfo);
```

Next, it invokes the `GetService` method to access the service container:

```
IServiceContainer container =  
    (IServiceContainer)this.GetService(typeof(IServiceContainer));
```

Then, it adds the `ProviderConfigurationModule` module as the provider configuration service to this service container under the `Type` object that represents the `IProviderConfigurationService` interface. As you'll see shortly, this will allow the clients of this provider configuration service to use this `Type` object as the key to access the service.

```
if (container != null)  
    container.AddService(typeof(IProviderConfigurationService), this);
```

Before walking through the `ProviderConfigurationModule` class's implementation of the `ConfigureProvider` method of the `IProviderConfigurationService` interface, you need to have a good understanding of when, where, and why the `ConfigureProvider` method is invoked.

As you'll see later, as part of the implementation of your custom provider-based service, you must implement a module page that provides the clients of your service with the appropriate user interface to graphically configure your service. Figure 9-8 presents an example of such a module page. The module page shown in this figure is a module page named `RolesPage`, which provides the clients of the `Roles` provider-based service with the appropriate user interface to graphically configure this service.

Configuring a provider-based service involves three types of configurations:

- ❑ Adding, removing, renaming, and updating providers of the service
- ❑ Setting the default provider of the service
- ❑ Specifying configuration settings other than adding, removing, renaming, and updating providers and setting the default provider

Here we're only interested in the first two types of configurations because they involve the invocation the `ConfigureProvider` method of the `ProviderConfigurationModule` module, which is the current topic of our discussions. I begin our discussions with the first type of configuration. As discussed earlier, adding, removing, renaming, and updating providers must be done from the `ProviderConfigurationConsolidatedPage` module list page shown in Figure 9-2. Therefore the

module page that provides the clients of a provider-based service with the appropriate user interface to graphically configure the service must include a link button labeled “Providers” in its associated task panel to allow the clients to navigate to the `ProviderConfigurationConsolidatedPage` module list page to add, remove, rename, and update providers of the service.

For example, the task panel associated with the `RolesPage` module page shown in Figure 9-8 contains a `Providers` link button to allow the clients of the `Roles` provider-based service to navigate to the `ProviderConfigurationConsolidatedPage` module list page to add, remove, rename, and update providers of the `Roles` provider-based service.

As you’ll see later in this chapter, the `ConfigureProvider` method of `ProviderConfigurationModule` contains the logic that uses the navigation service to navigate to the `ProviderConfigurationConsolidatedPage` module list page. Therefore, a module page such as `RolesPage` shown in Figure 9-8, which provides the clients of a particular provider-based service with the appropriate user interface to configure the service, can register an event handler for the `Click` event of the `Providers` link button and have this event handler invoke the `ConfigureProvider` method of `ProviderConfigurationModule` to navigate to the `ProviderConfigurationConsolidatedPage` module list page.

Invoking the `ConfigureProvider` method on the `ProviderConfigurationModule` instance requires a module page such as `RolesPage` to have access to this instance. This shouldn’t be a problem because as you saw in Listing 9-15, the `Initialize` method of this instance adds itself as a provider configuration service to the service container. This makes the `ProviderConfigurationModule` instance available to a module page such as `RolesPage` that provides the clients of a particular provider-based service with the appropriate user interface to configure the service:

```
container.AddService(typeof(IProviderConfigurationService), this);
```

For example, the `RolesPage` module page shown in Figure 9-8 registers an event handler named `ConfigureRoleProvider` for the `Click` event of the `Providers` link button. Listing 9-16 presents the internal implementation of this event handler.

Listing 9-16: The `ConfigureRoleProvider` Method

```
private void ConfigureRoleProvider()
{
    ProviderFeature providerFeature = new RolesProviderConfigurationFeature(this);
    this.ProviderConfigurationService.ConfigureProvider(providerFeature);
}
```

As you can see, this event handler first instantiates a `RolesProviderConfigurationFeature` to represent the `Roles` provider-based service and then invokes the `ConfigureProvider` method on the `ProviderConfigurationService` property. As Listing 9-17 shows, the `ProviderConfigurationService` property calls the `GetService` method passing in the `Type` object that represents the `IProviderConfigurationService` interface to access and return the provider configuration service registered under this `Type` object. Because the `Initialize` method registers the `ProviderConfigurationModule` instance as the provider configuration service under this `Type` object (see Listing 9-15), the `GetService` method basically returns a reference to this `ProviderConfigurationModule` instance.

Listing 9-17: The ProviderConfigurationService Property

```
private IProviderConfigurationService providerConfigurationService;
private IProviderConfigurationService ProviderConfigurationService
{
    get
    {
        if (this.providerConfigurationService == null)
            this.providerConfigurationService = (IProviderConfigurationService)
                base.GetService(typeof(IProviderConfigurationService));

        return this.providerConfigurationService;
    }
}
```

Keep in mind that we’re discussing two types of configurations involving the invocation of the `ConfigureProvider` method of the `ProviderConfigurationModule` module. So far, I’ve covered the first type of configuration, which is adding, removing, renaming, and updating providers of a service. Next, I discuss the second type of configuration.

As should be clear by now, you can register as many providers as you want with a given provider-based service. However, the service will use only the provider that is registered as the default provider. Therefore, the module page such as `RolesPage` that provides the clients of a provider-based service with the appropriate user interface to graphically configure the service must contain a link button labeled “Set Default Provider ...” in its associated task panel. When the user clicks this link button, the module page must create and launch a task form that contains a combo box that displays the list of providers registered for the provider-based service.

For example, when you click the “Set Default Provider ...” link button in the task panel associated with the `RolesPage` module page shown in Figure 9-8, this module page launches the `RolesSettingsForm` task form shown in Figure 9-9. Note that the `RolesSettingsForm` task form contains a combo box that displays the list of providers registered for the `Roles` provider-based service.

When the user selects a provider from this combo box and clicks OK, the callback for the OK button must use a proxy to set the value of the default provider attribute on the underlying configuration section to the friendly name of the provider that the user has selected from the combo box.

As you’ll see shortly, the `ConfigureProvider` method of the `ProviderConfigurationModule` module contains the logic that invokes the appropriate method of the appropriate proxy to set the value of the default provider attribute on a specified configuration section to the friendly name of a specified provider. Therefore, the event handler for the `Click` event of the OK button of a task form such as `RolesSettingsForm`, which displays the list of providers registered for a particular provider-based service, can call the `ConfigureProvider` method of the `ProviderConfigurationModule` module to have this method set the value of the default provider attribute on the underlying configuration section to the friendly name of the provider that the user has selected from the combo box.

For example, the event handler registered for the `Click` event of the OK button on the `RolesSettingsForm` is a method named `OnAccept`. Listing 9-18 presents the internal implementation of this event handler.

Listing 9-18: The OnAccept Method

```
protected override void OnAccept()
{
    base.StartAsyncTask(new DoWorkEventHandler(this.OnWorkerDoWork),
                        new RunWorkerCompletedEventHandler(this.OnWorkerCompleted));
    base.UpdateTaskForm();
}
```

As you can see, the `OnAccept` event handler invokes the `StartAsyncTask` method, passing a delegate of type `DoWorkEventHandler`, which wraps a method named `OnWorkerDoWork`, and a delegate of type `RunWorkerCompletedEventHandler`, which wraps a method named `OnWorkerCompleted`. As should be clear by now, the `StartAsyncTask` method allows you to invoke the method wrapped by the `DoWorkEventHandler` delegate in an asynchronous fashion to improve responsiveness and performance. Listing 9-19 presents the internal implementation of the `OnWorkerDoWork` method.

Listing 9-19: The OnWorkerDoWork Method

```
private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
{
    if (this.hasChanges)
    {
        string selectedProvider = (string)this.providerComboBox.SelectedItem;
        ProviderFeature providerFeature =
            new RolesProviderConfigurationFeature(rolesPage, selectedProvider);
        this.providerConfigurationService.ConfigureProvider(providerFeature);
    }
}
```

The `OnWorkerDoWork` method begins by accessing the friendly name of the provider that the user has selected from the combo box. Recall that the `RolesSettingsForm` task form contains a combo box that displays the providers registered for the `Roles` provider-based service (see Figure 9-9):

```
string selectedProvider = (string)this.providerComboBox.SelectedItem;
```

Next, `OnWorkerDoWork` creates a `RolesProviderConfigurationFeature` provider feature to represent the `Roles` provider-based service. Note that `OnWorkerDoWork` passes two parameters into the constructor of the `RolesProviderConfigurationFeature`. The first parameter references the `RolesPage` module page (see Figure 9-8). The second parameter is a string that contains the friendly name of the provider that the user wants to be used as the default provider:

```
ProviderFeature providerFeature =
    new RolesProviderConfigurationFeature(rolesPage, selectedProvider);
```

Finally, `OnWorkerDoWork` invokes the `ConfigureProvider` method on the provider configuration service, passing in the reference to the provider feature that represents the `Roles` provider-based service to have this method set the `defaultProvider` attribute on the `<rolesManager>` configuration section in the underlying configuration file to the friendly name of the selected provider:

```
this.providerConfigurationService.ConfigureProvider(providerFeature);
```

Chapter 9: Understanding the Integrated Providers Model

Needless to say, the `providerConfigurationService` field of the `RolesSettingsForm` task form references the `ProviderConfigurationModule` module.

In summary, there are two occasions where the `ConfigureProvider` method of the `ProviderConfigurationModule` module is invoked. The first occasion is when the user of a provider-based service clicks the Providers link button in the task panel associated with a module page such as `RolesPage`. As just discussed, clicking the Providers link button automatically invokes the `ConfigureProvider` method of the `ProviderConfigurationModule` module to have this method use the navigation service to navigate to the `ProviderConfigurationConsolidatedPage` module list page.

The second occasion is when the user of a provider-based service selects a provider from the list of providers displayed in the combo box of a task form such as `RolesSettingsForm` and clicks the OK button in this task form. As just discussed, clicking OK automatically triggers a call into the `ConfigureProvider` method of the `ProviderConfigurationModule` module to have this method use the proxy to set the default provider attribute on the underlying configuration section to the friendly name of the selected provider.

Now that you have a good understanding of when, where, and why the `ConfigureProvider` method of the `ProviderConfigurationModule` module is invoked, we're ready to dive into the details of the implementation of this method.

As Listing 9-15 shows, this method begins by calling the `GetService` method to access the connection service:

```
Connection service = (Connection)this.GetService(typeof(Connection));
```

Next, the `ConfigureProvider` method checks whether the `SelectedProvider` property of the `ProviderFeature` object passed into the method has been set. Recall from Listing 9-19 that the `SelectedProvider` property of the `ProviderFeature` object is only set when the `ConfigureProvider` method is invoked to set the default provider attribute on the underlying configuration section to the friendly name of the selected provider.

If the `SelectedProvider` property of the `ProviderFeature` object has indeed been set, the `ConfigureProvider` method rightly assumes that its caller is trying to set the default provider attribute on the underlying configuration section to the friendly name of the selected provider. Therefore, the `ConfigureProvider` method takes these steps to set the default provider:

1. Instantiates the `ProviderConfigurationModuleProxy` proxy. This is the proxy that facilitates the communications between the `ProviderConfigurationConsolidatedPage` module list page and the back-end server class.

```
ProviderConfigurationModuleProxy proxy =  
    (ProviderConfigurationModuleProxy)service.CreateProxy(this,  
        typeof(ProviderConfigurationModuleProxy));
```

2. Instantiates a `PropertyBag` collection:

```
PropertyBag infoBag = new PropertyBag();
```

3. Populates the `PropertyBag` collection with:

- ❑ The configuration section name of the provider-based service that the provider feature passed into the `ConfigureProvider` method represents:

```
infoBag[0] = feature.SectionName;
```

- ❑ The name of the attribute on this configuration section that specifies the default provider:

```
infoBag[1] = feature.SelectedProviderPropertyName;
```

- ❑ The name of the Collection XML element of this configuration section that contains the Add XML elements, which register providers for this provider-based service:

```
infoBag[2] = feature.ProviderCollectionPropertyName;
```

- ❑ The fully qualified name of the provider base type from which all providers of the provider-based service inherit:

```
infoBag[3] = feature.ProviderBaseType;
```

- ❑ A string array that contains the names of all attributes (other than the name and type attributes) on the Add XML elements that register providers. These attributes basically specify the configuration settings of these providers.

```
infoBag[4] = feature.ProviderConfigurationSettingNames;
```

4. Invokes the `SelectProvider` method on the proxy, passing in the `PropertyBag` collection and the friendly name of the selected provider to set this provider as the default provider in the underlying configuration file:

```
if (proxy.SelectProvider(infoBag, feature.SelectedProvider))  
    return true;
```

If the `SelectProvider` property of the `ProviderFeature` object passed into the `ConfigureProvider` method has not been set, the method rightly assumes that its caller is trying to navigate to the `ProviderConfigurationConsolidatedPage` module list page. Therefore, the `ConfigureProvider` method needs to navigate from the current module page to the `ProviderConfigurationConsolidatedPage` module list page. First, it invokes the `GetService` method to access the navigation service. Then it calls the `Navigate` method on the navigation service to navigate to the `ProviderConfigurationConsolidatedPage` module list page:

```
((INavigationService)this.GetService(  
    typeof(INavigationService))).Navigate(  
    service, service.ConfigurationPath,  
    typeof(ProviderConfigurationConsolidatedPage),  
    feature);
```

Note that the `ConfigureProvider` method passes the `ProviderFeature` object as the navigation data into the `Navigate` method. The `Navigate` method passes this `ProviderFeature` object to the `Initialize` method of the `ProviderConfigurationConsolidatedListPage` module list page.

Chapter 9: Understanding the Integrated Providers Model

Listing 9-20 presents the internal implementation of the `Initialize` method of the `ProviderConfigurationConsolidatedPage` module list page.

Listing 9-20: The `Initialize` Method of the `ProviderConfigurationConsolidatedPage` Module List Page

```
protected override void Initialize(object navigationData)
{
    if (navigationData != null)
    {
        ProviderFeature feature = navigationData as ProviderFeature;
        if (feature != null)
            this.selectedFeatureName = feature.FeatureName;
    }
}
```

As you can see, this method takes an argument that references the navigation data. The navigation data in this case is the provider feature that represents a provider-based service. The `Initialize` method simply assigns the value of the `FeatureName` property of this provider feature to a private field named `selectedFeatureName`. The `ProviderConfigurationConsolidatedPage` module list page sets the selected provider of the Feature combo box to the provider whose friendly name is given by the `selectedFeatureName`, and consequently the list view underneath this combo box automatically displays the list of providers registered for the selected provider-based service.

This allows the `ProviderConfigurationConsolidatedPage` module list page to automatically display the list of providers for the provider-based service that invoked the `ConfigureProvider` method of the `ProviderConfigurationModule` in the first place. For example, if the end user clicks the Providers link button in the task panel associated with the `RolesPage` module page to navigate to the `ProviderConfigurationConsolidatedPage` module list page, the `ProviderConfigurationConsolidatedPage` module list page will automatically select the Roles option from the Feature combo box and the list view underneath this combo box will automatically display the list of providers registered for the Roles provider-based service.

Summary

This chapter provided in-depth coverage of the IIS 7 and ASP.NET integrated providers model, and you also saw this model in action. The next chapter builds on what you've learned in this chapter to show you how to extend the IIS 7 and ASP.NET integrated providers model to implement fully configurable provider-based services.

10

Extending the Integrated Providers Model

The previous chapter provided in-depth coverage of the IIS 7 and ASP.NET integrated providers model where you learn a great deal about the internals of this model and its constituent components. You also saw this model in action. This chapter builds on what you learned in the previous chapter to teach you how to extend the integrated providers model to implement fully configurable provider-based services.

I begin the chapter by presenting a detailed step-by-step recipe for extending the integrated providers model. Then I use this recipe to implement a fully configurable RSS provider-based service that can generate RSS data from any type of data store.

Recipe

Follow these steps to extend the IIS 7 and ASP.NET integrated providers model to develop a fully configurable provider-based service:

1. Implement a custom provider base class that defines the API through which your provider-based service will interact with its providers. This API isolates your service from the specifics of its providers, allowing it to interact with them in a generic fashion without knowing their real types.
2. Implement a custom provider collection class that acts as a container for providers of your service.

3. Take the following steps to extend the IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section for your service to allow the clients of your service to configure your service directly from configuration files:
 - a. Use the IIS 7 and ASP.NET integrated declarative schema extension markup language to implement this configuration section.
 - b. Register this configuration section with the integrated configuration system.
4. Extend the IIS 7 and ASP.NET integrated imperative management system to add support for a new set of imperative management classes to enable the clients of your service to configure your service directly from managed code in a strongly-type fashion where they can benefit from the Visual Studio IntelliSense and compiler type-checking supports and the well-known object-oriented programming benefits.
5. Implement a service class that performs the following tasks:
 - a. Uses the new imperative management classes to retrieve the required configuration settings, including the registered providers from the configuration section that configures your provider-based service in the specified configuration file at a specified configuration hierarchy level
 - b. Instantiates and initializes the registered providers
 - c. Defines the API that services a specific type of data from any type of data store
6. Implement one or more providers. Recall that your provider-based service uses each provider to service a specific type of data from a specific type of data store. Keep in mind that each provider must inherit your custom provider base class from Step 1.
7. Extend the integrated graphical management system to add graphical management support for your provider-based service to allow the clients of your service to configure your service directly from the IIS 7 Manager. This involves two sets of managed code: client-side and server-side.

Take these steps to implement the client-side managed code:

1. Implement a custom provider configuration settings class that inherits from the `ProviderConfigurationSettings` base class. This custom provider configuration settings class must expose the attributes (other than the name and type attributes) on the Add XML element that registers a provider as strongly-typed properties. These attributes basically specify the configuration settings (other than name and type) of the provider that the Add XML element registers with provider-based service.
2. Implement a custom provider feature class that inherits the `ProviderFeature` base class. Your custom provider feature must override the `Settings` property, among other properties, of the `ProviderFeature` base class to return an instance of your custom provider configuration settings class.
3. Implement a custom class whose constructor takes a `PropertyBag` collection as its argument and exposes the content of this collection as strongly-typed properties. This `PropertyBag` collection contains all configurable aspects of your provider-based service excluding its providers. The client-side managed code normally receives this `PropertyBag` object from the server.
4. Implement a custom module service proxy class that inherits the `ModuleServiceProxy` base class to facilitate the communications between your client-side managed code and the server.

5. Implement a module page that provides the clients of your provider-based service with the appropriate user interface to configure all configurable aspects of your service excluding adding, removing, renaming, and updating providers.
6. Add the following link buttons to the task panel associated with this module page:
 - ☐ A link button labeled “Set Default Provider ...” for configuring the default provider of your provider-based service
 - ☐ A link button for navigating to the IIS 7 Manager’s `ProviderConfigurationConsolidatePage` module list page where the clients can add, remove, rename, and update providers
 - ☐ A link button for enabling or disabling your provider-based service
7. Implement a task form that allows the clients of your provider-based service to specify the default provider for your service. End users click the “Set Default Provider ...” link button to launch this task form.
8. Implement a custom module class that inherits from the `Module` base class to register your module page with the IIS 7 and ASP.NET integrated infrastructure. This custom module class must also create an instance of your custom provider feature and register this instance with the extensibility manager service under the `Type` object that represents the `ProviderFeature` type.

Take these steps to implement the server-side managed code:

1. Implement a custom module service class that inherits the `ModuleService` base class. This class is the server-side class that the client-side managed code interacts with.
2. Implement a custom configuration module provider class that inherits from the `ConfigurationModuleProvider` base class to register your custom module and custom module service classes with the IIS 7 and ASP.NET integrated infrastructure.
3. Register your custom configuration module provider class with the `administration.config` file.

The rest of this chapter uses this recipe to develop a provider-based RSS service that will allow components such as `RssHandler` to generate RSS data from any type of data store. Before diving into the implementation of your custom provider-based RSS service, you need to take care of some preliminary setup.

Launch Visual Studio and add a blank solution named `RssSol`. Next add a new class library project named `Rss` to this solution. Right-click the `Rss` project in the Solution Explorer panel of Visual Studio and select the Properties option from the popup menu to launch the Properties dialog. Select the Application tab and specify `Rss` as both the Assembly name and Default namespace. Then, follow the steps discussed in Chapter 7 to have Visual Studio automatically:

- ☐ Compile the `Rss` project into a strongly-named assembly.
- ☐ Add the assembly to the Global Assembly Cache (GAC).
- ☐ Launch the IIS 7 Manager after each build.

Now add a directory named `Base` to your `Rss` project.

Custom Provider Base Class

Following the recipe, this section develops a custom provider base class named `RssProvider` that derives from the `ProviderBase` base class and defines the API that every RSS provider must implement. As you'll see later, every RSS provider will be specifically designed to generate RSS from a specific type of data store with a specific schema. For example, you will implement the following two RSS providers:

- ❑ An RSS provider named `SqlRssProvider` that generates RSS from a SQL Server database with a specific schema
- ❑ An RSS provider named `XmlRssProvider` that generates RSS from an XML document with a specific schema

Both the `SqlRssProvider` and `XmlRssProvider` RSS providers will derive from the `RssProvider` base class and implement the API that this base class defines. As you'll see later, the custom RSS service will use this API to interact with the configured RSS provider, be it `SqlRssProvider` or `XmlRssProvider`. In other words, the `RssProvider` base class allows your custom RSS service to use the same API to interact with all types of RSS providers.

Listing 10-1 contains the code for the `RssProvider` base class. Add a new source file named `RssProvider.cs` to the `Base` directory and add the code shown in this code listing to this source file.

Listing 10-1: The `RssProvider` Base Class

```
using System.Configuration.Provider;
using System.IO;

namespace Rss.Base
{
    public abstract class RssProvider : ProviderBase
    {
        public abstract void LoadRss(Channel channel, Stream stream);
    }
}
```

As you can see from Listing 10-1, the API that the `RssProvider` base class defines consists of a single method named `LoadRss` that takes two parameters. The first parameter references a `Channel` object and the second parameter references a `Stream` object. Recall from Chapter 8 that this `Stream` object is normally nothing but the server response output stream. It is the responsibility of each RSS provider such as `SqlRssProvider` and `XmlRssProvider` to implement this method to contain the logic that knows how to generate the RSS document from the specified data store. For example, as you'll see later, the `SqlRssProvider` will implement the `LoadRss` method to contain the logic that generates the RSS document from the SQL Server database with a specified schema. Or the `XmlRssProvider` will implement the `LoadRss` method to contain the logic that generates the RSS document from the XML document with a specified schema. It is the responsibility of each RSS provider to load the `Stream` passed into the `LoadRss` method with the RSS document that it generates from the specified data store.

Listing 10-2 presents the implementation of the `Channel` class. Now add a new source file named `Channel.cs` to the `Base` directory and add the code shown in this code listing to this source file.

Listing 10-2: The Channel Class

```
namespace Rss.Base
{
    public class Channel
    {
        private string title;
        private string description;
        private string link;

        public string Title
        {
            get { return title; }
            set { title = value; }
        }

        public string Description
        {
            get { return description; }
            set { description = value; }
        }

        public string Link
        {
            get { return link; }
            set { link = value; }
        }
    }
}
```

Custom Provider Collection

The next order of business is to develop a custom provider collection named `RssProviderCollection` that derives from the `ProviderCollection` base class, as shown in Listing 10-3. Now add a new source file named `RssProviderCollection.cs` to the `Base` directory and add the code shown in this code listing to this source file.

Listing 10-3: The `RssProviderCollection` Class

```
using System.Configuration.Provider;
using System;

namespace Rss.Base
{
    public class RssProviderCollection : ProviderCollection
    {
        public new RssProvider this[string name]
        {
            get { return (RssProvider)base[name]; }
        }
    }
}
```

(Continued)

Listing 10-3: (continued)

```
public override void Add(ProviderBase provider)
{
    if (provider == null)
        throw new ArgumentNullException("provider");

    if (!(provider is RssProvider))
        throw new ArgumentException("Invalid provider type", "provider");

    base.Add(provider);
}
}
```

The `ProviderCollection` base class implements an indexer that returns a reference to the provider with the specified friendly name. However, this base class returns the specified provider as an object of type `ProviderBase`. The `RssProviderCollection` implements a new indexer that returns a reference to the provider with the specified name but with one difference, that is, it casts the provider to `RssProvider` type before it returns it. This allows the clients of the `RssProviderCollection` to return a strongly-typed object from the collection.

The `ProviderCollection` base class exposes a method named `Add` that allows the clients of the collection to add new providers to the collection. This base class's implementation of the `Add` method allows its clients to add any provider of type `ProviderBase` to this collection. As Listing 10-3 shows, the `RssProviderCollection` overrides the `Add` method of the `ProviderCollection` base class to include the logic that raises an exception if the client of the collection attempts to add a provider of a type other than the `RssProvider` type. This ensures that the `RssProviderCollection` only contains providers of type `RssProvider` such as `SqlRssProvider` and `XmlRssProvider`.

Extending the Integrated Configuration System

The clients of your provider-based RSS service must be allowed to perform the following tasks in a specified configuration file at a specified configuration hierarchy level without writing a single line of code:

- ❑ Register new types of RSS providers such as `SqlRssProvider` or `XmlRssProvider` with your provider-based RSS service.
- ❑ Configure your provider-based RSS service to use a specified type of RSS provider.
- ❑ Enable or disable your provider-based RSS service.
- ❑ Specify the channel information, including its title, description, and link. As the highlighted portion of the following excerpt from Listing 9-1 shows, the implementation of the `RssHandler` presented in Chapter 8 hard-codes the channel information. The clients of the provider-based RSS service should be able to declaratively specify the channel information in a configuration file.

```
public RssHandler()
{
```

```
channelTitle = "New Articles On mysite.com";
channelLink = "http://www.mysite.com";
channelDescription = "The list of newly published articles on mysite.com";
    . . .
}
```

To make all the features I discussed possible you need to use the IIS 7 and ASP.NET integrated declarative schema extension markup language to extend the IIS 7 and ASP.NET integrated configuration system to add support for a new configuration section named `<rss>`. You also need to register this configuration section with the IIS 7 and ASP.NET integrated configuration system.

In general, every provider-based service should use the integrated declarative schema extension markup language to extend the integrated configuration system to add support for a configuration section to allow its clients to configure the service from the configuration system.

This configuration section must contain a Collection XML element, which represents the collection that contains the providers registered with the provider-based service. Even though you can choose any name you want for this Collection XML element, it is typically named `<providers>`. This Collection XML element allows the clients of a service to use an Add XML element to register a new provider with the provider-based service. Even though you can choose any name you want for this Add XML element, it is typically named `<add>`.

Listing 10-4 uses the XML constructs of the IIS 7 and ASP.NET integrated declarative schema extension markup language to define your new `<rss>` configuration section. You must store this XML schema definition in an XML file. Following the naming convention discussed in the previous chapters, name this file `RSS_schema.xml`.

Listing 10-4: The Content of the RSS_schema.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<configSchema>
  <sectionSchema name="system.webServer/rss">
    <attribute name="enabled" type="bool" defaultValue="true"/>
    <attribute name="channelTitle" type="string" defaultValue="Unknown"/>
    <attribute name="channelDescription" type="string" defaultValue="Unknown"/>
    <attribute name="channelLink" type="string" defaultValue="Unknown"/>
    <attribute name="defaultProvider" type="string"
      validationType="requireTrimmedString" defaultValue="SqlRssProvider"/>
    <element name="providers">
      <collection addElement="add" removeElement="remove" clearElement="clear"
        allowUnrecognizedAttributes="true">
        <attribute name="name" required="true" isUniqueKey="true" type="string" />
        <attribute name="type" required="true" type="string" />
      </collection>
    </element>
  </sectionSchema>
</configSchema>
```

The `RSS_schema.xml` file, like any other schema file in the integrated configuration system, has a document element named `<configSchema>`, which contains a child element named `<sectionSchema>`, which defines the RSS configuration section. The `RSS_schema.xml` file uses the XML constructs of the

Chapter 10: Extending the Integrated Providers Model

integrated declarative schema extension markup language to define the XML attributes and elements that make up the RSS configuration section as follows:

- ❑ Uses an `<attribute>` XML element with a name attribute value of "enabled", a type attribute value of "bool", and a defaultValue attribute value of "true" to define the enabled attribute of the `<rss>` containing XML element and the type and default value of this attribute. As you can see, the RSS service is enabled by default.

```
<attribute name="enabled" type="bool" defaultValue="true"/>
```

- ❑ Uses an `<attribute>` XML element with a name attribute value of "channelTitle", a type attribute value of "string", and a defaultValue attribute value of "Unknown" to define the channelTitle attribute of the `<rss>` containing XML element and the type and default value of this attribute.

```
<attribute name="channelTitle" type="string" defaultValue="Unknown"/>
```

- ❑ Uses an `<attribute>` XML element with name attribute value of "channelDescription", a type attribute value of "string", and a defaultValue attribute value of "Unknown" to define the channelDescription attribute of the `<rss>` containing XML element and the type and default value of this attribute.

```
<attribute name="channelDescription" type="string" defaultValue="Unknown"/>
```

- ❑ Uses an `<attribute>` XML element with a name attribute value of "channelLink", a type attribute value of "string", and a defaultValue attribute value of "Unknown" to define the channelLink attribute of the `<rss>` containing XML element and the type and default value of this attribute.

```
<attribute name="channelLink" type="string" defaultValue="Unknown"/>
```

- ❑ Uses an `<attribute>` XML element with a name attribute value of "defaultProvider", a type attribute value of "string", and a defaultValue attribute value of "SqlRssProvider" to define the defaultProvider attribute of the `<rss>` containing XML element and the type and default value of this attribute. As you can see, the RSS service uses the `SqlRssProvider` by default. In other words, the RSS service generates RSS documents from the specified SQL Server database with specified schema by default. I discuss this database and its schema later in this chapter.

```
<attribute name="defaultProvider" type="string"
validationType="requireTrimmedString" defaultValue="SqlRssProvider"/>
```

- ❑ Defines the `<providers>` Collection XML element of the `<rss>` containing XML element. First, it uses an `<element>` XML element with a name attribute value of "providers" to define the Providers element itself. Then, it uses a `<collection>` XML element with an `addElement` attribute value of "add", a `removeElement` attribute value of "remove", and a `clearElement` attribute value of "clear" to specify that the Providers element can contain zero or more instances of the `<add>`, `<remove>`, and `<clear>` child elements. Each `<add>` child element adds or registers an RSS provider such as `SqlRssProvider` with the RSS service. A `<remove>` child element removes the RSS provider that a higher-level configuration file has registered with the RSS service. A `<clear>` child element removes all the RSS providers that the higher-level configuration files have registered with the RSS service. Note that the `<collection>` element contains two `<attribute>` XML elements as follows:

- ❑ An `<attribute>` XML element with a name attribute value of "name" to define the name attribute of the `<add>` child element. Note that the required attribute of this

<attribute> XML element is set to `true` to specify that the `name` attribute of an <add> child element is mandatory. Also note that the `isUniqueKey` attribute of this <attribute> XML element is set to `true` to specify that the `name` attribute of the <add> child element uniquely identifies the provider being registered among other providers. This allows the clients of the RSS provider-based service to use the `name` attribute on the <remove> child element to specify the friendly name of the provider to remove.

- ❑ An <attribute> XML element with a `name` attribute value of `"type"` to define the `type` attribute of the <add> child element. Keep in mind that the value of the `type` attribute of an <add> child element is a string that contains a comma-separated list of up to five substrings, where only the first substring is mandatory. The first substring contains the fully qualified name of the type of the provider being registered including its complete namespace hierarchy. The remaining four substrings contain the information about the assembly that contains the specified provider type.

Using the XML constructs of the integrated declarative schema extension markup language to define the XML attributes and elements that make up the RSS configuration section is just half the story. The other half is the registration process, which involves two steps. First, you must move the `RSS_schema.xml` file to the following standard directory on your machine:

```
%WinDir%\System32\inetsrv\config\schema
```

Second, you must add the XML fragment shown in Listing 10-5 to the `applicationHost.xml` file located in the following directory on your machine. You'll need administrative privileges to edit this file.

```
%WinDir%\System32\inetsrv\config
```

Listing 10-5 uses a <section> XML element with the `name` attribute value of `"rss"` to register the RSS configuration section. Note that this <section> XML element is the child element of the <sectionGroup> XML element with the `name` attribute value of `"system.webServer"` to specify that the <rss> configuration section belongs to the <system.webServer> configuration section group. Also note that the `allowDefinition` attribute on this <section> XML element is set to the value of `"Everywhere"` to specify that the <rss> configuration section can be added to configuration files at all configuration hierarchy levels of the integrated configuration system. Also note that the `overrideModeDefault` attribute on this <section> XML element is set to the value of `"Allow"` to specify that lower-level configuration files are allowed to override the RSS configuration settings specified in higher-level configuration files.

Listing 10-5: The XML Fragment That Registers the rss Configuration Section

```
<configuration>
  <configSections>
    <sectionGroup name="system.webServer">
      <section name="rss" allowDefinition="Everywhere"
        overrideModeDefault="Allow" />
    </sectionGroup>
  </configSections>
</configuration>
```


Extending the Integrated Imperative Management System

The previous section used the XML constructs of the integrated declarative schema extension markup language to define the `<rss>` configuration section. Following the recipe, the next order of business is to extend the integrated imperative management system to add support for new imperative management classes that will allow the clients of your provider-based RSS service to configure the service directly from managed code in a strongly-typed fashion.

In this section, you implement the following four imperative management classes:

- ☐ `ProviderSettings`
- ☐ `ProviderSettingsCollection`
- ☐ `ProvidersHelper`
- ☐ `RssSection`

Note that you don't need to implement the first three classes every time you implement a custom provider-based service. In other words, the implementation of these three classes presented in the following sections can be used as is with all types of custom provider-based services. Now add a new subdirectory named `ImperativeManagement` to the `Rss` project. You will add the source files for all four imperative management classes to this subdirectory.

ProviderSettings

Listing 10-6 presents the implementation of the `ProviderSettings` imperative management class. Add a new source file named `ProviderSettings.cs` to the `ImperativeManagement` directory of the `Rss` project and add the code shown in this code listing to this source file. You also need to add a reference to the `Microsoft.Web.Administration.dll` assembly located in the following directory on your machine to this project:

```
%windir%\System32\inetsrv
```

Listing 10-6: The ProviderSettings Class

```
using System.Collections.Specialized;
using Microsoft.Web.Administration;
using System.Collections.Generic;

namespace Rss.ImperativeManagement
{
    public class ProviderSettings : ConfigurationElement
    {
        public string Name
        {
            get { return (string)base["name"]; }
            set { base["name"] = value; }
        }
    }
}
```

Listing 10-6: *(continued)*

```

    public string Type
    {
        get { return (string)base["type"]; }
        set { base["type"] = value; }
    }

    private NameValueCollection parameters;
    public NameValueCollection Parameters
    {
        get
        {
            if (parameters == null)
            {
                parameters = new NameValueCollection();
                IDictionary<string, string> rawAttributes = base.RawAttributes;
                foreach (string attributeName in rawAttributes.Keys)
                {
                    parameters.Add(attributeName, rawAttributes[attributeName]);
                }
            }
            return parameters;
        }
    }
}

```

A `ProviderSettings` instance is an imperative representation of an Add XML element (the Add element in the case of the RSS provider-based service is named `<add>`) that registers a provider with a provider-based service. In other words, a `ProviderSettings` instance provides imperative access to the attributes on the Add XML element that the instance represents. As a result, the `ProviderSettings` class exposes three properties as follows:

- ❑ **Name:** This string property provides imperative access to the `name` attribute on the associated Add XML element.
- ❑ **Type:** This string property provides imperative access to the `type` attribute on the associated Add XML element.
- ❑ **Parameters:** This `NameValueCollection` property provides imperative access to the attributes other than `name` and `type` on the associated Add XML element. You can use the name of each attribute as an index into this `NameValueCollection` to return the value of the attribute.

ProviderSettingsCollection

Listing 10-7 contains the code for the `ProviderSettingsCollection` class. Next add a new source file named `ProviderSettingsCollection.cs` to the `ImperativeManagement` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-7: The ProviderSettingsCollection Class

```
using Microsoft.Web.Administration;
using System;

namespace Rss.ImperativeManagement
{
    public class ProviderSettingsCollection :
        ConfigurationElementCollectionBase<ProviderSettings>
    {
        public ProviderSettings Add(string name, string type)
        {
            ProviderSettings providerSettings = base.CreateElement();
            providerSettings.Name = name;
            providerSettings.Type = type;
            return base.Add(providerSettings);
        }

        protected override ProviderSettings CreateNewElement(string elementTagName)
        {
            return new ProviderSettings();
        }

        public void Remove(string name)
        {
            base.Remove(this[name]);
        }

        public new ProviderSettings this[string name]
        {
            get
            {
                for (int i = 0; i < base.Count; i++)
                {
                    ProviderSettings providerSettings = base[i];
                    if (string.Equals(providerSettings.Name, name,
                        StringComparison.OrdinalIgnoreCase))
                        return providerSettings;
                }
                return null;
            }
        }
    }
}
```

As discussed earlier, the configuration section of every provider-based service must contain a Collection XML element, which is normally named `<providers>`. The `ProviderSettingsCollection` is the imperative representation of this Collection XML element. As a result, it exposes the following members:

- ❑ **Add:** This method takes the name and type of a provider, instantiates a `ProviderSettings` object, and adds this object to the collection. In other words, the `Add` method is the imperative equivalent of using an `Add` XML element to add a provider in the configuration file. The `Add` XML element is normally named `<add>`.

- ❑ **Remove:** This method takes the friendly name of a provider and removes the provider from the collection. This is equivalent to using the Remove XML element in the configuration file. The Remove XML element is normally named `<remove>`.
- ❑ **Indexer:** This indexer returns a reference to a `ProviderSettings` object with the specified name.
- ❑ **CreateNewElement:** `ProviderSettingsCollection` overrides the `CreateNewElement` method of its base class to return an instance of the `ProviderSettings` class because this collection is a collection of `ProviderSettings` objects.

ProvidersHelper

Listing 10-8 demonstrates the implementation of the `ProvidersHelper` helper class. Now add a new source file named `ProvidersHelper.cs` to the `ImperativeManagement` directory of the `Rss` project and add the code shown in this code listing to this source file. You also need to add references to `System.Web.dll` and `System.Configuration.dll` assemblies because they respectively contain the `System.Web.Compilation.BuildManager` and `System.Configuration.Provider.ProviderBase` types.

Listing 10-8: The ProvidersHelper Helper Class

```
using System;
using System.Configuration.Provider;
using System.Collections.Specialized;
using System.Web.Compilation;

namespace Rss.ImperativeManagement
{
    public static class ProvidersHelper
    {
        public static ProviderBase InstantiateProvider(
            ProviderSettings providerSettings,
            Type providerType)
        {
            string providerTypeInfo =
                (providerSettings.Type == null) ? null : providerSettings.Type.Trim();
            if (string.IsNullOrEmpty(providerTypeInfo))
                throw new ArgumentException("Provider's type must be specified.");

            Type providerTypeObj = BuildManager.GetType(providerTypeInfo, true, true);
            if (!providerType.IsAssignableFrom(providerTypeObj))
                throw new ArgumentException("Provider must implement type " +
                    providerType.ToString());

            ProviderBase provider =
                (ProviderBase)Activator.CreateInstance(providerTypeObj);
            NameValueCollection parameters = providerSettings.Parameters;
            NameValueCollection config =
                new NameValueCollection(parameters.Count, StringComparer.Ordinal);
            foreach (string attributeName in parameters)
            {
```

(Continued)

Listing 10-8: (continued)

```
        config[attributeName] = parameters[attributeName];
    }

    provider.Initialize(providerSettings.Name, config);
    return provider;
}

public static void InstantiateProviders(
    ProviderSettingsCollection configProviders,
    ProviderCollection providers,
    Type providerType)
{
    foreach (ProviderSettings settings in configProviders)
    {
        providers.Add(InstantiateProvider(settings, providerType));
    }
}
}
```

As discussed earlier, the configuration section associated with a provider-based service contains a Collection XML element (typically named `<providers>`) that contains one or more Add XML elements (typically named `<add>`), each of which registers a specified provider. The `ProvidersHelper` class is a helper class that contains the logic that iterates through these Add XML elements, extracts the required information from the attributes on each Add XML element, and instantiates and initializes the provider that each Add XML element registers. Keep in mind that every ASP.NET provider directly or indirectly inherits from the `ProviderBase` class. ASP.NET comes with a collection class named `ProviderCollection` that acts as a container for `ProviderBase` objects.

As you can see from Listing 10-8, the `InstantiateProviders` method takes three parameters. The first parameter is of type `ProviderSettingsCollection`, which references the `ProviderSettingsCollection` collection that represents the Collection XML element of the configuration section of the associated provider-based service, which is the RSS provider-based service in this case. As discussed earlier, this collection contains one `ProviderSettings` object for each Add XML element in this Collection XML element. The second parameter of the `InstantiateProviders` method is of type `ProviderCollection`. The third parameter of this method is a `Type` object that represents the type of the provider being instantiated.

As you can see from the following excerpt from Listing 10-8, the `InstantiateProviders` method iterates through the `ProviderSettings` object in the `ProviderSettingsCollection` collection passed into the method as its first argument. For each enumerated `ProviderSettings` object, it invokes another method named `InstantiateProvider`, passing in the reference to the enumerated `ProviderSettings` object to instantiate a `ProviderBase` object with the specified type and specified settings. Next, it adds this `ProviderBase` object to the `ProviderCollection` collection passed into the method as its second argument.

```
public static void InstantiateProviders(
    ProviderSettingsCollection configProviders,
    ProviderCollection providers,
```

```
                                Type providerType)
{
    foreach (ProviderSettings settings in configProviders)
    {
        providers.Add(InstantiateProvider(settings, providerType));
    }
}
```

Next, I walk you through the internal implementation of the `InstantiateProvider` method as shown in Listing 10-8. This method first uses the `Type` property of the `ProviderSetting` object passed into it as its first argument to access the value of the `type` attribute on the `Add` XML element that registers the associated provider. Recall that this attribute contains a comma-separated list of up to five substrings, where only the first substring is mandatory. The first substring contains the fully qualified name of the type of the provider, including its complete containment namespace hierarchy. The remaining substrings specify the assembly that contains this provider type.

```
string providerTypeInfo =
    (providerSettings.Type == null) ? null : providerSettings.Type.Trim();
if (string.IsNullOrEmpty(providerTypeInfo))
    throw new ArgumentException("Provider's type must be specified.");
```

Next, the `InstantiateProvider` method invokes a static method named `GetType` on a standard ASP.NET class named `BuildManager`, passing in the value of the `type` attribute of the `Add` XML element. Under the hood, the `GetType` method loads the assembly that contains the provider type if it hasn't already been loaded, and creates and returns a `Type` object that represents the type of the provider:

```
Type providerTypeObj = BuildManager.GetType(providerTypeInfo, true, true);
if (!providerType.IsAssignableFrom(providerTypeObj))
    throw new ArgumentException("Provider must implement type " +
                                providerType.ToString());
```

Next, it invokes a static method named `CreateInstance` on a standard .NET class named `Activator`, passing in the `Type` object to dynamically instantiate an instance of the specified provider:

```
ProviderBase provider =
    (ProviderBase)Activator.CreateInstance(providerTypeObj);
```

Next, it instantiates a `NameValueCollection` collection and populates this collection with the content of the `Parameters` collection of the `ProviderSettings` object. Recall that the `Parameters` collection contains the names and values of the attributes (other than the name and type attributes) on this `Add` XML element. As discussed earlier, you can use the name of an attribute as an index into this collection to access its value:

```
NameValueCollection parameters = providerSettings.Parameters;
NameValueCollection config =
    new NameValueCollection(parameters.Count, StringComparer.Ordinal);
foreach (string attributeName in parameters)
{
    config[attributeName] = parameters[attributeName];
}
```

Chapter 10: Extending the Integrated Providers Model

Finally, it invokes the `Initialize` method on the provider, passing in two parameters. The first parameter is a string that contains the value of the `name` attribute on the `Add` XML element that registers the specified provider. The second parameter references the `NameValueCollection` collection that contains the names and values of the attributes (other than the `name` and `type` attributes) on this `Add` XML element. As you'll see later, every provider inherits the `Initialize` method from the `ProviderBase` class.

```
provider.Initialize(providerSettings.Name, config);
```

RssSection

Listing 10-9 presents the implementation of the `RssSection` class. Now add a new source file named `RssSection.cs` to the `ImperativeManagement` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-9: The `RssSection` Class

```
using System;
using Microsoft.Web.Administration;

namespace Rss.ImperativeManagement
{
    public class RssSection : ConfigurationSection
    {
        static RssSection()
        {
            RssSection.ProvidersAttribute = "providers";
            RssSection.DefaultProviderAttribute = "defaultProvider";
            RssSection.EnabledAttribute = "enabled";
            RssSection.ChannelTitleAttribute = "channelTitle";
            RssSection.ChannelDescriptionAttribute = "channelDescription";
            RssSection.ChannelLinkAttribute = "channelLink";
        }

        public RssSection() { }

        public string DefaultProvider
        {
            get
            {
                return (string)base[RssSection.DefaultProviderAttribute];
            }
            set
            {
                base[RssSection.DefaultProviderAttribute] = value;
            }
        }
        public bool Enabled
        {
            get
            {
                return (bool)base[RssSection.EnabledAttribute];
            }
        }
    }
}
```

Listing 10-9: *(continued)*

```
        set
        {
            base[RssSection.EnabledAttribute] = value;
        }
    }

    public string ChannelTitle
    {
        get
        {
            return (string)base[RssSection.ChannelTitleAttribute];
        }
        set
        {
            base[RssSection.ChannelTitleAttribute] = value;
        }
    }

    public string ChannelDescription
    {
        get
        {
            return (string)base[RssSection.ChannelDescriptionAttribute];
        }
        set
        {
            base[RssSection.ChannelDescriptionAttribute] = value;
        }
    }

    public string ChannelLink
    {
        get
        {
            return (string)base[RssSection.ChannelLinkAttribute];
        }
        set
        {
            base[RssSection.ChannelLinkAttribute] = value;
        }
    }

    public ProviderSettingsCollection Providers
    {
        get
        {
            if (this._providers == null)
            {
                this._providers = (ProviderSettingsCollection)
                    base.GetCollection(RssSection.ProvidersAttribute,
                                      typeof(ProviderSettingsCollection));
            }
        }
    }
}
```

(Continued)

Listing 10-9: *(continued)*

```
        return this._providers;
    }
}

private ProviderSettingsCollection _providers;
private static readonly string DefaultProviderAttribute;
private static readonly string EnabledAttribute;
private static readonly string ProvidersAttribute;
private static readonly string ChannelTitleAttribute;
private static readonly string ChannelDescriptionAttribute;
private static readonly string ChannelLinkAttribute;
}
}
```

The `RssSection` imperative management class allows imperative code such as C# or Visual Basic to programmatically access and manipulate the XML attributes and elements that make up the `<rss>` configuration section. As you'll see later, this allows imperative code to load the contents of the `<rss>` configuration section of a given configuration file at a given configuration hierarchy level into an instance of the `RssSection` and use the methods and properties of this instance to imperatively access and manipulate the values of these XML attributes and elements.

As you can see from Listing 10-9, this class exposes one property for each XML attribute or element that makes up the `<rss>` configuration section where the `DefaultProvider`, `Enabled`, `ChannelTitle`, `ChannelDescription`, and `ChannelLink` properties allow you to imperatively get and set the values of the `defaultProvider`, `enabled`, `channelTitle`, `channelDescription`, and `channelLink` attributes, respectively, and the `Providers` property allows you to imperatively access the providers specified in the `<providers>` section Collection XML element of the `<rss>` containing XML element.

Implementing the Service Class

Listing 10-10 illustrates the implementation of the `RssService` class. Now add a new source file named `RssService.cs` to the `Base` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-10: The `RssService` Class

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
using Microsoft.Web.Administration;
using Rss.ImperativeManagement;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
    }
}
```

Listing 10-10: (continued)

```

private static RssProviderCollection providers = null;
private static bool IsInitialized = false;

public RssProvider Provider
{
    get { Initialize(); return provider; }
}

public RssProviderCollection Providers
{
    get { Initialize(); return providers; }
}

public static void LoadRss(Stream stream)
{
    Initialize();
    Channel channel = new Channel();
    channel.Title = channelTitle;
    channel.Link = channelLink;
    channel.Description = channelDescription;

    provider.LoadRss(channel, stream);
}

private static string channelTitle;
private static string channelDescription;
private static string channelLink;

private static void Initialize()
{
    if (!IsInitialized)
    {
        ServerManager mgr = new ServerManager();
        Configuration config =
            mgr.GetWebConfiguration("Default Web Site",
                                    HttpContext.Current.Request.ApplicationPath);
        RssSection section =
            (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
        channelDescription = section.ChannelDescription;
        channelLink = section.ChannelLink;
        channelTitle = section.ChannelTitle;

        providers = new RssProviderCollection();
        ProvidersHelper.InstantiateProviders
            (section.Providers, providers, typeof(RssProvider));
        provider = providers[section.DefaultProvider];

        if (provider == null)
            throw new ProviderException("Unable to load default RssProvider");
        IsInitialized = true;
    }
}
}
}

```

Chapter 10: Extending the Integrated Providers Model

The `RssService` class exposes a method named `Initialize` whose main responsibility is to instantiate and initialize the providers registered in the `<providers>` subelement of the `<rss>` configuration section of the specified configuration file. The `Initialize` method first instantiates an instance of the `ServerManager` class:

```
ServerManager mgr = new ServerManager();
```

Next, it invokes the `GetWebConfiguration` method on this `ServerManager` instance to load the contents of the configuration file of the current application into a `Configuration` object. This `Configuration` object allows the `Initialize` method to imperatively access the contents of the specified configuration file:

```
Configuration config =  
    mgr.GetWebConfiguration("Default Web Site",  
        HttpContext.Current.Request.ApplicationPath);
```

Next, the `Initialize` method invokes the `GetSection` method on this `Configuration` object to return an `RssSection` object that provides imperative access to the contents of the `<rss>` configuration section of the configuration file:

```
RssSection section =  
    (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
```

Then, it uses the `RssSection` object to imperatively access the values of the `channelDescription`, `channelLink`, and `channelTitle` attributes on the `<rss>` containing XML element and stores these values in private fields with the same names for future reference:

```
channelDescription = section.ChannelDescription;  
channelLink = section.ChannelLink;  
channelTitle = section.ChannelTitle;
```

Next, it creates an `RssProviderCollection` object and stores this object in a private field named `providers` for future reference:

```
providers = new RssProviderCollection();
```

Then, it invokes the `InstantiateProviders` static method on the `ProvidersHelper` helper class discussed earlier to instantiate the providers in the `<providers>` subelement of the `<rss>` containing XML element and load them into the `RssProviderCollection` collection.

```
ProvidersHelper.InstantiateProviders  
    (section.Providers, providers, typeof(RssProvider));
```

Next, it uses the value of the `defaultProvider` attribute on the `<rss>` containing XML element as an index into the `RssProviderCollection` collection to return a reference to the default provider and stores this reference in a private field named `provider` for future reference:

```
provider = providers[section.DefaultProvider];
```

Chapter 10: Extending the Integrated Providers Model

Note that the `Initialize` method raises an exception if the `<providers>` subelement of the `<rss>` containing XML element does not contain the default provider with the specified name.

```
if (provider == null)
    throw new ProviderException("Unable to load default RssProvider");
```

Also note that the `RssService` class exposes a property of type `RssProvider` named `Provider` that returns a reference to the default provider, that is, it returns the value of the `provider` private field. As you can see from the following excerpt from Listing 10-9, the `RssService` class's implementation of this property first invokes the `Initialize` method to initialize the `provider` private field before it attempts to return its value.

```
public RssProvider Provider
{
    get { Initialize(); return provider; }
}
```

The `RssService` class also exposes a property named `Providers` that returns the `RssProviderCollection` object stored in the `providers` private field. Note that this property first invokes the `Initialize` method to initialize this private field before it attempts to return its value:

```
public RssProviderCollection Providers
{
    get { Initialize(); return providers; }
}
```

Next, I walk you through the implementation of the `LoadRss` method of the `RssService` class as shown in the following excerpt from Listing 10-9:

```
public static void LoadRss(Stream stream)
{
    Initialize();
    Channel channel = new Channel();
    channel.Title = channelTitle;
    channel.Link = channelLink;
    channel.Description = channelDescription;

    provider.LoadRss(channel, stream);
}
```

As you can see, this method takes a single argument of type `Stream`. As discussed earlier, this `Stream` object normally references the server response output stream. This method basically defines the API for servicing RSS data.

You must always implement the methods and properties that make up the API of your custom provider-based service as static members.

The `LoadRss` method first calls the `Initialize` method to ensure that the `RssService` has already been initialized:

```
Initialize();
```

Chapter 10: Extending the Integrated Providers Model

Next, it creates a `Channel` object and populates this object with the channel information:

```
Channel channel = new Channel();
channel.Title = channelTitle;
channel.Link = channelLink;
channel.Description = channelDescription;
```

Finally, it delegates the responsibility of loading RSS data from the underlying data store to the configured default RSS provider:

```
provider.LoadRss(channel, stream);
```

The provider-based RSS service just developed allows components such as the `RssHandler` HTTP handler to retrieve RSS data from any type of data store. This section provides a new implementation of `RssHandler` that delegates the responsibility of generating the RSS document from the underlying data store to `RssService` component. Listing 10-11 contains the code for the new version of the `RssHandler` HTTP handler. Now add a new source file named `RssHandler.cs` to the `Base` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-11: The New Version of the `RssHandler` HTTP Handler That Uses `RssService`

```
using System.Web;

namespace Rss.Base
{
    public class RssHandler : IHttpHandler
    {
        bool IHttpHandler.IsReusable
        {
            get { return false; }
        }

        void IHttpHandler.ProcessRequest(HttpContext context)
        {
            context.Response.ContentType = "text/xml";
            RssService.LoadRss(context.Response.OutputStream);
        }
    }
}
```

The `ProcessRequest` method of `RssHandler` sets the value of the `ContentType` property of the `Response` to the string value `text/xml` to indicate to the requesting browser that the response contains an XML document:

```
context.Response.ContentType = "text/xml";
```

The method then calls the `LoadRss` static method on the `RssService` and passes the `OutputStream` of the `Response` into it. As discussed before, `LoadRss` uses the configured default RSS provider to generate the RSS document from the data retrieved from the data store and to load the document into the `OutputStream`.

```
RssService.LoadRss(context.Response.OutputStream);
```

Implementing Custom Providers

The previous sections implemented the new provider-based RSS service, which is capable of generating RSS data from any type of data store. As discussed, the provider-based RSS service uses each provider to service RSS data from a specific type of data store. In this section, you develop two providers named `SqlRssProvider` and `XmlRssProvider` that will allow the provider-based RSS service to service RSS data from a SQL Server database with a specific schema and an XML document with a specific XML schema, respectively.

SqlRssProvider

This section implements an RSS provider named `SqlRssProvider` that derives from the `RssProvider` base class and generates RSS data from the SQL Server database discussed in the Chapter 8. Listing 10-12 contains the code for the `SqlRssProvider` class. Add a new source file named `SqlRssProvider.cs` to the `Base` directory and add the code shown in this code listing to this source file.

Listing 10-12: The `SqlRssProvider` Class

```
using System;
using System.Data;
using System.Configuration;
using System.Collections.Specialized;
using System.Configuration.Provider;
using System.Data.SqlClient;
using System.IO;
using System.Collections;

namespace Rss.Base
{
    public class SqlRssProvider : RssProvider
    {
        private string itemTitleField;
        private string itemDescriptionField;
        private string itemLinkField;
        private string itemLinkFormatString;
        private string connectionString;
        private string commandText;
        private CommandType commandType;

        public override void Initialize(string name, NameValueCollection config)
        {
            if (config == null)
                throw new ArgumentNullException("config");

            if (string.IsNullOrEmpty(name))
                name = "SqlRssProvider";

            if (string.IsNullOrEmpty(config["description"]))
            {
                config.Remove("description");
                config.Add("description",
                    "Retrieve RSS data from the SQL Server database");
            }
        }
    }
}
```

Listing 10-12: *(continued)*

```
}
base.Initialize(name, config);

string connectionStringName = config["connectionStringName"];
if (string.IsNullOrEmpty(connectionStringName))
    throw new ProviderException("Invalid connection string name");

connectionString =
ConfigurationManager.ConnectionStrings[connectionStringName].ConnectionString;
if (string.IsNullOrEmpty(connectionString))
    throw new ProviderException("Connection string not found");

config.Remove("connectionStringName");

itemTitleField = config["itemTitle"];
if (string.IsNullOrEmpty(itemTitleField))
    throw new ProviderException("Title field not found");
config.Remove("itemTitle");

itemDescriptionField = config["itemDescription"];
if (string.IsNullOrEmpty(itemDescriptionField))
    throw new ProviderException("Description field not found");
config.Remove("itemDescription");

itemLinkField = config["itemLink"];
if (string.IsNullOrEmpty(itemLinkField))
    throw new ProviderException("Link field not found");
config.Remove("itemLink");

itemLinkFormatString = config["itemLinkFormatString"];
config.Remove("itemLinkFormatString");

commandText = config["item"];
if (string.IsNullOrEmpty(commandText))
    throw new ProviderException("Command text not found");
config.Remove("item");

string commandTypeText = config["itemInfo"];
if (string.IsNullOrEmpty(commandTypeText))
    commandType = CommandType.Text;
else if (commandTypeText.ToLower() == "storedprocedure")
    commandType = CommandType.StoredProcedure;
else
    commandType = CommandType.Text;
config.Remove("itemInfo");

if (config.Count > 0)
{
    string key = config.GetKey(0);
    if (!string.IsNullOrEmpty(key))
        throw new ProviderException("Unrecognized attribute");
}
```

Listing 10-12: (continued)

```
    }

    SqlDataReader GetDataReader()
    {
        SqlConnection con = new SqlConnection();
        con.ConnectionString = connectionString;
        SqlCommand com = new SqlCommand();
        com.Connection = con;
        com.CommandText = commandText;
        com.CommandType = commandType;
        con.Open();
        return com.ExecuteReader(CommandBehavior.CloseConnection);
    }

    public override void LoadRss(Channel channel, Stream stream)
    {
        SqlDataReader reader = GetDataReader();

        ArrayList items = new ArrayList();
        Item item;
        while (reader.Read())
        {
            item = new Item();
            item.Title = (string)reader[itemTitleField];
            item.Link = (string)reader[itemLinkField];
            item.Description = (string)reader[itemDescriptionField];
            item.LinkFormatString = itemLinkFormatString;
            items.Add(item);
        }
        reader.Close();
        RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
    }
}
```

Because the `RssProvider` base class derives from the `ProviderBase` class, `SqlRssProvider` implements both the `LoadRss` method of `RssProvider` and the `Initialize` method of `ProviderBase` as discussed in the following sections.

Listing 10-13 presents the implementation of the `Item` class that the `SqlRssProvider`'s implementation uses. Add a new source file named `Item.cs` to the `Base` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-13: The Item Class

```
namespace Rss.Base
{
    public class Item
    {
        private string title;
```

(Continued)

Listing 10-13: *(continued)*

```
private string description;
private string link;
private string linkFormatString;

public string Title
{
    get { return title; }
    set { title = value; }
}

public string Description
{
    get { return description; }
    set { description = value; }
}

public string Link
{
    get { return link; }
    set { link = value; }
}

public string LinkFormatString
{
    get { return linkFormatString; }
    set { linkFormatString = value; }
}
}
```

Listing 10-14 presents the implementation of the `RssHelper` class that the `SqlRssProvider`'s implementation uses. Add a new source file named `RssHelper.cs` to the `Base` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-14: The `RssHelper` Class

```
using System;
using System.Configuration;
using System.Collections.Specialized;
using System.IO;
using System.Xml;

namespace Rss.Base
{
    public class RssHelper
    {
        public static void GenerateRss(Channel channel, Item[] items, Stream stream)
        {
            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
        }
    }
}
```

Listing 10-14: (continued)

```
using (XmlWriter writer = XmlWriter.Create(stream, settings))
{
    writer.WriteStartDocument();
    writer.WriteStartElement("rss");
    writer.WriteAttributeString("version", "2.0");
    writer.WriteStartElement("channel");
    writer.WriteElementString("title", channel.Title);
    writer.WriteElementString("link", channel.Link);
    writer.WriteElementString("description", channel.Description);
    foreach (Item item in items)
    {
        writer.WriteStartElement("item");
        writer.WriteElementString("title", item.Title);
        writer.WriteElementString("description", item.Description);
        writer.WriteElementString("link",
            string.Format(item.LinkFormatString, item.Link));
        writer.WriteEndElement();
    }
    writer.WriteEndElement();
    writer.WriteEndElement();
    writer.WriteEndDocument();
}
}
```

Initialize

As Listing 10-12 shows, the `Initialize` method first performs the four tasks that the `Initialize` method of any provider must perform:

1. It raises an exception if the `NameValueCollection` is null. Recall that this `NameValueCollection` collection contains one item for each attribute on the `<add>` element that registers the provider. As discussed earlier, you can use the name of an attribute as an index into this collection to access its value.

```
if (config == null)
    throw new ArgumentNullException("config");
```

2. It sets the friendly name of the provider if it hasn't already been set.

```
if (string.IsNullOrEmpty(name))
    name = "SqlRssProvider";
```

Your custom provider's implementation of the `Initialize` method mustn't set the value of the friendly name of the provider if the page developer has already specified a value for the name attribute on the `<add>` element that registers the provider. If your custom provider overrides this value, it could break the page developer's code if the code uses the friendly name of the provider to access the provider. Here is an example. Let's say the page developer has assigned

Chapter 10: Extending the Integrated Providers Model

the string "MyProvider" as the value of the name attribute on the <add> element that registers the SqlRssProvider:

```
<rss enabled="true" defaultProvider="MyProvider">
  <providers>
    <add name="MyProvider" type="CustomComponents.SqlRssProvider"
      . . ./>
  </providers>
</rss>
```

Now, let's say the page developer uses this value as an index into the Providers collection of the RssService to access the SqlRssProvider provider:

```
RssService service;
. . .
SqlRssProvider provider = service.Providers["MyProvider"];
```

This code would fail if your custom provider's implementation of the Initialize method overrides the friendly name of the provider.

Breaking the page developer's code is not the only problem. As the following excerpt from Listing 10-10 shows, the Initialize method of the RssService will also fail and raise an exception because it uses the value of the defaultProvider attribute on the <rss> containing XML element as an index into the Providers collection to access the default provider.

```
private static void Initialize()
{
    if (!IsInitialized)
    {
        . . .

        providers = new RssProviderCollection();
        ProvidersHelper.InstantiateProviders
            (section.Providers, providers, typeof(RssProvider));
        provider = providers[section.DefaultProvider];

        if (provider == null)
            throw new ProviderException(
                "Unable to load default RssProvider");
        IsInitialized = true;
    }
}
```

3. It sets the value of the description field if it hasn't already been set:

```
if (string.IsNullOrEmpty(config["description"]))
{
    config.Remove("description");
    config.Add("description", "Retrieve RSS data from the SQL Server database");
}
```

4. It calls the Initialize method of the base class to allow the base class to initialize the name and description properties.

```
base.Initialize(name, config);
```

Your custom provider must always delegate the responsibility of setting the name and description properties to its base class as opposed to overriding the Name and Description properties.

The `Initialize` method of the `SqlRssProvider`, like the `Initialize` method of any other provider, then iterates through each item in the `NameValueCollection` and performs the following tasks for each item. Recall that each item represents an attribute (other than name and type attributes) on the `<add>` element that registers the provider:

1. Uses the name of the attribute as an index into the `NameValueCollection` collection to access the value of the attribute.
2. Uses the value to set the respective property of the provider.
3. Calls the `Remove` method of the `NameValueCollection` to remove the item from the collection.

The `Initialize` method follows this three-step pattern to set the values of the `connectionString`, `commandText`, `commandType`, `channelTitle`, `channelDescription`, `channelLink`, `itemTitleField`, `itemDescriptionField`, `itemLinkField`, and `itemLinkFormatString` private fields of the `SqlRssProvider`. These fields are discussed in the following sections.

Connection String

The `Initialize` method sets the `connectionString` private field of the `SqlRssProvider`. First it uses the `connectionStringName` string to index into the `NameValueCollection` to access the value of the attribute named `connectionStringName` on the `<add>` element that registers the `SqlRssProvider` provider. If the value is null or an empty string, the method raises a `ProviderException` exception because `SqlRssProvider` needs this value to access the underlying data store.

```
string connectionStringName = config["connectionStringName"];
if (string.IsNullOrEmpty(connectionStringName))
    throw new ProviderException("Invalid connection string name");
```

Next, it uses the value it just obtained as an index into the `ConnectionStrings` collection of the `ConfigurationManager` class to access the connection string and assign this connection string to the `connectionString` private field of the `SqlRssProvider` for future reference. The `ConnectionStrings` collection represents the `<connectionStrings>` section of the configuration file. The client's of the RSS provider-based service must:

- ❑ Use an `<add>` element to register the required connection string within the `<connectionStrings>` section of the configuration file.
- ❑ Set the name attribute of the `<add>` element to the friendly name of the connection string.
- ❑ Assign the connection string to the `connectionString` attribute of the `<add>` element that registers the `SqlRssProvider`.

You'll see an example of these three steps later in this chapter.

```
connectionString =
    ConfigurationManager.ConnectionStrings[connectionStringName].ConnectionString;
if (string.IsNullOrEmpty(connectionString))
    throw new ProviderException("Connection string not found");
```

Chapter 10: Extending the Integrated Providers Model

Finally, the `Initialize` method removes the item associated with the connection string from the `NameValueCollection`:

```
config.Remove("connectionStringName");
```

Command Text

The `SqlRssProvider` uses the connection string to connect to the underlying data store. The command text, on the other hand, specifies the SQL `Select` statement or stored procedure that the `SqlRssProvider` must use to retrieve the required data from the data store. To set the `commandText` private field of the `SqlRssProvider`, the `Initialize` method uses the `"item"` string as an index into the `NameValueCollection` to access the value of the `item` attribute on the `<add>` element that registers the `SqlRssProvider` provider and assigns this value to the `commandText` private field. If the value is null or an empty string, it raises a `ProviderException` exception because the `SqlRssProvider` can't operate without this value.

```
commandText = config["item"];
if (string.IsNullOrEmpty(commandText))
    throw new ProviderException("Command text not found");
```

The method then removes the associated item from the `NameValueCollection`:

```
config.Remove("item");
```

Command Type

Because the `commandText` field can contain a SQL `Select` statement or stored procedure, the client of the RSS provider-based service must set the value of the `itemInfo` attribute on the `<add>` element that registers the `SqlRssProvider` to one of the following values:

- ❑ `"Text"` (case insensitive) to specify that the `item` attribute contains a SQL `Select` statement
- ❑ `"StoredProcedure"` (case insensitive) to specify that the `item` attribute contains a stored procedure

The `Initialize` method then follows the same steps as for the `commandText` private field to set the value of the `commandType` private field. The only difference is that the method must map the string value that it retrieves from the `NameValueCollection` to its associated `CommandType` enumeration value as follows:

```
string commandTypeText = config["itemInfo"];
if (string.IsNullOrEmpty(commandTypeText))
    commandType = CommandType.Text;
else if (commandTypeText.ToLower() == "storedprocedure")
    commandType = CommandType.StoredProcedure;
else
    commandType = CommandType.Text;
config.Remove("itemInfo");
```

RSS-Related Fields

SqlRssProvider exposes the following four private fields:

- ❑ `itemTitle`: The name of the datafield whose values are rendered within the opening and closing tags of the `<title>` child elements of the `<item>` elements of the RSS document
- ❑ `itemDescription`: The name of the datafield whose values are rendered within the opening and closing tags of the `<description>` child elements of the `<item>` elements of the RSS document
- ❑ `itemLink`: The name of the datafield whose values are rendered within the opening and closing tags of the `<link>` child elements of the `<item>` elements
- ❑ `itemLinkFormatString`: Formats the values of the datafield whose name is given by `itemLink` before they are rendered within the opening and closing tags of the `<link>` child elements of the `<item>` elements

As the following excerpt from Listing 10-12 shows, the `SqlRssProvider` follows the same steps discussed in the previous sections to retrieve the values of the `itemTitle`, `itemDescription`, `itemLink`, and `itemLinkFormatString` attributes on the `<add>` element that registers the provider and to assign them to its associated private fields:

```
itemTitleField = config["itemTitle"];
if (string.IsNullOrEmpty(itemTitleField))
    throw new ProviderException("Title field not found");
config.Remove("itemTitle");

itemDescriptionField = config["itemDescription"];
if (string.IsNullOrEmpty(itemDescriptionField))
    throw new ProviderException("Description field not found");
config.Remove("itemDescription");

itemLinkField = config["itemLink"];
if (string.IsNullOrEmpty(itemLinkField))
    throw new ProviderException("Link field not found");
config.Remove("itemLink");

itemLinkFormatString = config["itemLinkFormatString"];
config.Remove("itemLinkFormatString");
```

LoadRss

The `LoadRss` method of the `SqlRssProvider` basically contains the data access code that was part of the implementation of the `RssHandler` class presented in the previous chapter. As you can see, this data access code is moved from the client of the RSS service class to the `SqlRssProvider` provider allowing the client to interact with all types of data stores.

Registering SqlRssProvider

Listing 10-15 shows how the client of the provider-based RSS service can declaratively register `SqlRssProvider` with the RSS service without writing a single line of code.

Listing 10-15: Registering SqlRssProvider

```
<configuration>
  <system.webServer>
    <rss enabled="true" defaultProvider="SqlRssProvider" channelTitle="Title1"
      channelDescription="Description1" channelLink="Link1">
      <providers>
        <add name="SqlRssProvider"
          connectionStringName="myConnectionString"
          item="Select * From Articles"
          itemInfo="Text"
          itemTitle="Title" itemDescription="Abstract"
          itemLink="AuthorName"
          type="Rss.Base.SqlRssProvider, Rss, Version=2.0.0.0,
            Culture=Neutral, PublicKeyToken=a31626cc5fbb47c3" />
      </providers>
    </rss>
  </system.webServer>
</configuration>
```

The client of the provider-based RSS service must add a new `<add>` child element within the opening and closing tags of the `<providers>` Collection XML element and set its attribute values as follows:

- ❑ **name:** The client has the freedom of choosing any friendly name for the provider as long as it is different from the friendly names of other registered providers. Listing 10-15 uses the string value `SqlRssProvider` as the friendly name.

```
name="SqlRssProvider"
```

- ❑ **type:** The value of this attribute is a string that contains a comma-separated list of up to five substrings, where only the first substring is mandatory. The first substring contains the fully qualified name of the type of the provider including its complete namespace containment hierarchy, `Rss.Base.SqlRssProvider`.

```
type="Rss.Base.SqlRssProvider"
```

- ❑ **connectionStringName:** The value of this attribute must be set to the value of the name attribute of the `<add>` element that the client adds to the `<connectionStrings>` section of the configuration file as shown in the following code:

```
<configuration>
  <connectionStrings>
    <add name="FriendlyName"
      connectionString="Data Source=ServerName;Initial Catalog=DatabaseName;pwd;uid"
    </connectionStrings>
  </configuration>
```

- ❑ **item:** The client must set the value of this attribute to a string that contains a SQL `Select` statement or stored procedure name that selects the records that contains information about the RSS items:

```
item="Select * From Articles"
```

- ❑ **Item-related attributes:** The client must set the values of the `itemTitle`, `itemDescription`, and `itemLink` attributes to the names of the appropriate datafields:

```
itemTitle="Title" itemDescription="Abstract" itemLink="AuthorName"
```

Registering a provider doesn't automatically tell the RSS service to use the provider, because more than one RSS provider can be registered with the RSS service. The client must set the value of the `defaultProvider` attribute of the `<rss>` element to the friendly name of the desired provider to instruct the RSS service to use the specified provider.

```
<rss defaultProvider="SqlRssProvider">
```

Figures 10-1 and 10-2 present an example of a database that your `SqlRssProvider` supports. As you can see, this database consists of the following columns:

- ❑ **ArticleID:** This is an integer data column that contains the primary key values of the data records.
- ❑ **Title:** This is a varchar column that contains the article titles.
- ❑ **AuthorName:** This is a varchar column that contains the author names.
- ❑ **Abstract:** This is a varchar column that contains the article abstracts.

	Column Name	Data Type	Allow Nulls
PK	ArticleID	int	☐
	Title	varchar(50)	☐
	AuthorName	varchar(50)	☐
	Abstract	varchar(255)	☐

Figure 10-1

	ArticleID	Title	AuthorName	Abstract
Articles	1	What's new in ASP.NET 3.5?	Smith	Describes the new ASP.NET 3.5 feature
Applications	2	XSLT in ASP.NET Applications	Carey	Shows how to use XSLT in your ASP.NET
Tutorials	3	XML programming	Smith	Reviews .NET 2.0 XML programming fea

Figure 10-2

XmlRssProvider

As you saw in the previous section, the client of the provider-based RSS service declaratively plugs the `SqlRssProvider` into the RSS service to have the service generate the RSS document from the specified SQL Server database. This section develops an RSS provider named `XmlRssProvider` that the client can declaratively plug into the RSS service to have the service generate the RSS document from the specified XML file. Listing 10-16 presents the implementation of the `XmlRssProvider`. Add a new source file named `XmlRssProvider.cs` to the `Base` directory of the `Rss` project and add the code shown in this code listing to this source file.

Listing 10-16: The XmlRssProvider Class

```
using System;
using System.Configuration;
using System.Web;
using System.Collections.Specialized;
using System.Xml.XPath;
using System.Configuration.Provider;
using System.Xml;
using System.IO;
using System.Collections;

namespace Rss.Base
{
    public class XmlRssProvider : RssProvider
    {
        private string itemTitleXPath;
        private string itemDescriptionXPath;
        private string itemLinkXPath;
        private string itemLinkFormatString;
        private string dataFile;
        private string itemXPath;

        public override void Initialize(string name, NameValueCollection config)
        {
            if (config == null)
                throw new ArgumentNullException("config");

            if (string.IsNullOrEmpty(name))
                name = "XmlRssProvider";

            if (string.IsNullOrEmpty(config["description"]))
            {
                config.Remove("description");
                config.Add("description", "Retrieve RSS data from an XML document");
            }
            base.Initialize(name, config);

            string connectionStringName = config["connectionStringName"];
            if (string.IsNullOrEmpty(connectionStringName))
                throw new ProviderException("Invalid connection string name");

            dataFile =
                ConfigurationManager.ConnectionStrings[connectionStringName].ConnectionString;
            if (string.IsNullOrEmpty(dataFile))
                throw new ProviderException("Data file not found");

            config.Remove("connectionStringName");

            itemTitleXPath = config["itemTitle"];
            if (string.IsNullOrEmpty(itemTitleXPath))
                throw new ProviderException("Title XPath not found");
            config.Remove("itemTitle");
        }
    }
}
```

Listing 10-16: (continued)

```

        itemDescriptionXPath = config["itemDescription"];
        if (string.IsNullOrEmpty(itemDescriptionXPath))
            throw new ProviderException("Description XPath not found");
        config.Remove("itemDescription");

        itemLinkXPath = config["itemLink"];
        if (string.IsNullOrEmpty(itemLinkXPath))
            throw new ProviderException("Link XPath not found");
        config.Remove("itemLink");

        itemLinkFormatString = config["itemLinkFormatString"];
        config.Remove("itemLinkFormatString");

        itemXPath = config["item"];
        if (string.IsNullOrEmpty(itemXPath))
            throw new ProviderException("Item XPath not found");
        config.Remove("item");

        config.Remove("itemInfo");

        if (config.Count > 0)
        {
            string key = config.GetKey(0);
            if (!string.IsNullOrEmpty(key))
                throw new ProviderException("Unrecognized attribute");
        }
    }

    protected virtual XPathNodeIterator RetrieveData()
    {
        IXPathNavigable document =
            new XPathDocument(HttpContext.Current.Server.MapPath(dataFile));
        XPathNavigator nav = document.CreateNavigator();
        nav.MoveToChild(XPathNodeType.Element);
        return nav.Select(itemXPath);
    }

    public override void LoadRss(Channel channel, Stream stream)
    {
        XPathNodeIterator iter = RetrieveData();
        ArrayList items = new ArrayList();
        Item item;
        while (iter.MoveNext())
        {
            item = new Item();
            item.Title = iter.Current.SelectSingleNode(itemTitleXPath).Value;
            item.Link = iter.Current.SelectSingleNode(itemLinkXPath).Value;
            item.Description =
                iter.Current.SelectSingleNode(itemDescriptionXPath).Value;
            item.LinkFormatString = itemLinkFormatString;
            items.Add(item);
        }
    }

```

(Continued)

Listing 10-16: *(continued)*

```
        RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
    }
}
```

XmlRssProvider, like any other RSS provider, must implement the Initialize method of the ProviderBase class and the LoadRss method of the RssProvider class as discussed in the following sections.

Initialize

As Listing 10-16 shows, the Initialize method basically initializes the private fields of the XmlRssProvider provider. The following table compares the member fields of XmlRssProvider and SqlRssProvider.

SqlRssProvider	XmlRssProvider
connectionString: Contains the information needed to locate and connect to the database.	dataFile: Contains the information needed to locate the XML file.
commandText: The SQL Select statement or stored procedure that selects database records, where each record corresponds to an <item> element. Each database record contains the strings that SqlRssProvider renders within the opening and closing tags of the <title>, <description>, and <link> subelements of its associated <item> element.	itemXPath: The XPath expression that selects XML nodes, where each node corresponds to an <item> element. Each XML node contains the strings that XmlRssProvider renders within the opening and closing tags of the <title>, <description>, and <link> subelements of its associated <item> element.
itemTitleField: Used to select the database field associated with the <title> subelement of the <item> element.	itemTitleXPath: The XPath expression that is used to select the child XML node associated with the <title> subelement of the <item> element.
itemDescriptionField: Used to select the database field associated with the <description> subelement of the <item> element.	itemDescriptionXPath: The XPath expression that is used to select the child XML node associated with the <description> subelement of the <item> element.
itemLinkField: Used to select the database field associated with the <link> subelement of the <item> element.	itemLinkXPath: The XPath expression that is used to select the child XML node associated with the <link> subelement of the <item> element.

Here is an example to help you understand the `dataFile`, `itemTitleXPath`, `itemDescriptionXPath`, and `itemLinkXPath` fields of the `XmlRssProvider` RSS provider. Listing 10-17 shows an example of an XML file that `XmlRssProvider` uses to generate the RSS document.

Listing 10-17: A Sample XML Document That XmlRssProvider Uses

```
<?xml version="1.0" encoding="utf-8" ?>
<Articles>
  <Article title="What's new in ASP.NET 3.5?" authorName="Smith">
    <Abstract>Describes the new ASP.NET 3.5 features</Abstract>
  </Article>
  <Article title="XSLT in ASP.NET Applications" authorName="Carey">
    <Abstract>Shows how to use XSLT in your ASP.NET applications</Abstract>
  </Article>
  <Article title="XML programming" authorName="Smith">
    <Abstract>Reviews .NET 2.0 XML programming features</Abstract>
  </Article>
</Articles>
```

The following table shows examples of XPath expressions that the `itemTitleXPath`, `itemDescriptionXPath`, and `itemLinkXPath` fields can contain. Notice the third column of the table shows the current XML node. This is very important because XPath expressions are always evaluated with respect to the current node.

Field Name	Field Value XPath Expression	Current XML Node
<code>itemXPath</code>	<code>//Article</code>	<code><Articles></code>
<code>itemTitleXPath</code>	<code>@title</code>	<code><Article></code>
<code>itemDescriptionXPath</code>	<code>Abstract/text()</code>	<code><Article></code>
<code>itemLinkXPath</code>	<code>@authorName</code>	<code><Article></code>

Next, I show you the XML nodes that each XPath expression shown in the preceding table selects from the XML document shown in Listing 10-17. The XPath expression `//Article` shown in the first row of the table where the current node is `<Articles>` selects the following XML nodes:

```
<Article title="What's new in ASP.NET 3.5?" authorName="Smith">
  <Abstract>Describes the new ASP.NET 3.5 features</Abstract>
</Article>
<Article title="XSLT in ASP.NET Applications" authorName="Carey">
  <Abstract>Shows to use XSLT in your ASP.NET applications</Abstract>
</Article>
<Article title="XML programming" authorName="Smith">
  <Abstract>Reviews .NET 2.0 XML programming features</Abstract>
</Article>
```

The XPath expression `@title` shown in the second row of the table selects the attribute node `title="ASP.NET "` if the current node is the first `<Article>` element, the attribute node `title="XSLT`

Chapter 10: Extending the Integrated Providers Model

in ASP.NET" if the current node is the second <Article> element, and the attribute node title="XML programming" if the current node is the third <Article> element.

The XPath expression `Abstract/text()` shown in the third row of the table selects the text node "Reviews ASP.NET " if the current node is the first <Article> element, the text node "Overview of XSLT in ASP.NET" if the current node is the second <Article> element, and the text node "Reviews .NET XML programming" if the current node is the third <Article> element.

The XPath expression `@authorName` shown in the fourth row of the table selects the attribute node `authorName="Smith"` if the current node is the first <Article> element, the attribute node `authorName="Carey"` if the current node is the second <Article> element, and the attribute node `authorName="Smith"` if the current node is the third <Article> element.

LoadRss

Listing 10-18 contains the code for the `LoadRss` method.

Listing 10-18: The `LoadRss` Method of `XmlRssProvider`

```
public override void LoadRss(Channel channel, Stream stream)
{
    XPathNodeIterator iter = RetrieveData();
    ArrayList items = new ArrayList();
    Item item;
    while (iter.MoveNext())
    {
        item = new Item();
        item.Title = iter.Current.SelectSingleNode(itemTitleXPath).Value;
        item.Link = iter.Current.SelectSingleNode(itemLinkXPath).Value;
        item.Description =
            iter.Current.SelectSingleNode(itemDescriptionXPath).Value;
        item.LinkFormatString = itemLinkFormatString;
        items.Add(item);
    }

    RssHelper.GenerateRss(channel, (Item[])items.ToArray(typeof(Item)), stream);
}
```

The `LoadRss` method calls the `RetrieveData` method to access the `XPathNodeIterator` that contains the retrieved XML nodes. This method is discussed in the next section. `XPathNodeIterator` allows you to iterate through the retrieved XML nodes.

```
XPathNodeIterator iter = RetrieveData();
```

The method then iterates through the retrieved XML nodes. For each enumerated XML node, `LoadRss` creates an `Item` object:

```
item = new Item();
```

The `LoadRss` method then calls the `SelectSingleNode` method for each node and passes the XPath expression specified in the `itemTitleXPath` field into it. This method locates the XML node that the

XPath expression represents. LoadRss then assigns the value of this XML node to the `Title` property of the `Item` object.

```
item.Title = iter.Current.SelectSingleNode(itemTitleXPath).Value;
```

LoadRss then calls the `SelectSingleNode` method for each node and passes the XPath expression specified in the `itemLinkXPath` field into it. This method locates the XML node that the XPath expression represents. LoadRss then assigns the value of this XML node to the `Link` property of the `Item` object.

```
item.Link = iter.Current.SelectSingleNode(itemLinkXPath).Value;
```

LoadRss also calls the `SelectSingleNode` method for each node and passes the XPath expression specified in the `itemDescriptionXPath` field into it. This method locates the XML node that this XPath expression represents. LoadRss then assigns the value of this XML node to the `Description` property of the `Item` object.

```
item.Description = iter.Current.SelectSingleNode(itemDescriptionXPath).Value;
```

RetrieveData

Listing 10-19 illustrates the implementation of the `RetrieveData` method.

Listing 10-19: The RetrieveData Method

```
protected virtual XPathNodeIterator RetrieveData()
{
    XPathNavigable document =
        new XPathDocument(HttpContext.Current.Server.MapPath(dataFile));
    XPathNavigator nav = document.CreateNavigator();
    nav.MoveToChild(XPathNodeType.Element);
    return nav.Select(itemXPath);
}
```

The `RetrieveData` method loads the XML file into an `XPathDocument`:

```
XPathNavigable document =
    new XPathDocument(HttpContext.Current.Server.MapPath(dataFile));
```

The method then calls the `CreateNavigator` method of the `XPathDocument` object to access its `XPathNavigator`.

```
XPathNavigator nav = document.CreateNavigator();
```

The `XPathNavigator` contains an XPath engine and has been optimized for XPath queries. If you need to do a lot of XPath queries, the `XPathNavigator` is the way to go. However, only classes that implement the `IXPathNavigable` interface expose `XPathNavigator`. Currently .NET contains three concrete implementations of this interface: `XmlDocument`, `XPathDocument`, and `XmlDataDocument`.

Chapter 10: Extending the Integrated Providers Model

All three classes have one thing in common; they all load the entire XML document in memory to make random node access possible. This is not a problem if your XML document is not too large. However, it takes up lot of memory if your XML document is too big. In these cases, you should use the `XmlReader` and `XmlWriter` classes instead.

If you need random read access, you should use `XPathDocument` because it is optimized for XPath queries. You shouldn't use `XmlDocument` because it uses the DOM data model, which is different from the XPath data model. This data model mismatch degrades the XPath query performance of the `XmlDocument`. If you need random read/write access, you should use `XmlDocument`. Because the `XmlRssProvider` doesn't change the XML document, it doesn't need write access. That is why it uses `XPathDocument`.

The `RetrieveData` method moves the navigator to the document (root) element of the XML document. For example, in the case of the XML document shown in Listing 10-17, it moves to the `<Articles>` element.

```
nav.MoveToChild(XPathNodeType.Element);
```

The method then calls the `Select` method of the navigator (keep in mind the navigator is located at the root element, such as the `<Articles>` element) and passes the XPath expression specified in the `itemXPath` field into it. As I mentioned, XPath expressions are always evaluated with respect to the current node. Because the current node is the document or root element, the `Select` method evaluates the XPath expression specified in the `itemXPath` field with respect to the document element.

```
return nav.Select(itemXPath);
```

To help you understand how this works, take another look at the XML document shown in Listing 10-17. The call into the `MoveToNext` method takes the navigator to the `<Articles>` element of this XML document. Suppose the client has set the value of the `itemXPath` field to the XPath expression `//Article`. The call into the `Select("//Article")` method will select and return all the `<Article>` nodes in the XML document:

```
<Article title="What's new in ASP.NET 3.5?" authorName="Smith">
  <Abstract>Describes the new ASP.NET 3.5 features</Abstract>
</Article>
<Article title="XSLT in ASP.NET Applications" authorName="Carey">
  <Abstract>Shows to use XSLT in your ASP.NET applications</Abstract>
</Article>
<Article title="XML programming" authorName="Smith">
  <Abstract>Reviews .NET 2.0 XML programming features</Abstract>
</Article>
```

Registering `XmlRssProvider`

The highlighted portion of Listing 10-20 shows how the client of the RSS provider-based service must register `XmlRssProvider` with the service.

Listing 10-20: Registering `XmlRssProvider`

```
<configuration>
  <system.webServer>
```

Listing 10-20: *(continued)*

```

<rss enabled="true" defaultProvider="XmlRssProvider" channelTitle="Title1"
channelDescription="Description1" channelLink="Link1">
  <providers>
    <add name="XmlRssProvider" connectionStringName="myDataFile"
item="//Article" itemTitle="@title"
itemDescription="Abstract/text()"
itemLink="@authorName"
type="Rss.Base.XmlRssProvider, Rss, Version=2.0.0.0,
Culture=Neutral, PublicKeyToken=a31626cc5fbb47c3" />

    <add name="SqlRssProvider" connectionStringName="myConnectionString"
item="Select * From Articles"
itemTitle="Title" itemDescription="Abstract"
itemLink="AuthorName"
type="Rss.Base.SqlRssProvider, Rss, Version=2.0.0.0,
Culture=Neutral, PublicKeyToken=a31626cc5fbb47c3" />
  </providers>
</rss>
</system.webServer>
</configuration>

```

Extending the Integrated Graphical Management System

In this section, you extend the IIS 7 and ASP.NET integrated graphical management system to add graphical management support for the RSS provider-based service to allow the clients of the service to configure it right from the IIS 7 Manager. However, before diving into the implementation details of these graphical management extensions, let's see what these new extensions look like in action.

The users of your RSS provider-based service will use the following workflow to configure the service from the IIS 7 Manager:

1. Navigate to the `ConnectionStringsPage` module list page to add one or more connection strings. This standard module list page of the IIS 7 Manager provides the appropriate user interface for adding connection strings to, removing connection strings from, and updating connection strings in the `<connectionStrings>` section of the underlying configuration file.
2. Navigate to the `ProviderConfigurationConsolidatedPage` module list page to add one or more RSS providers. This standard module list page of the IIS 7 Manager provides the appropriate user interface for adding providers to, removing providers from, updating providers in, and renaming providers in the `Collection XML` element of the configuration section of a specified provider-based service. This `Collection XML` element is normally named `<providers>`.
3. Navigate to the `RssPage` module dialog page. As you'll see later, this module dialog page provides the appropriate user interface to specify configuration settings other than adding, removing, renaming, and updating providers. Adding, removing, renaming, and updating providers must be performed from the `ProviderConfigurationConsolidatedPage` module list page.

Chapter 10: Extending the Integrated Providers Model

Next, I show you how the user can use this three-step workflow to configure the RSS provider-based service. As Figure 10-3 shows, the workspace of the IIS 7 Manager contains an item named `Connection Strings`.



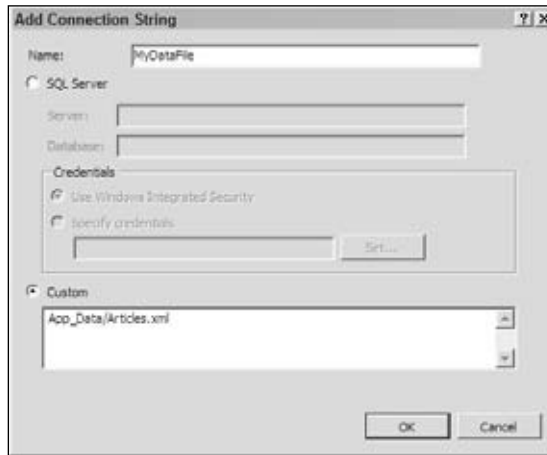
Figure 10-3

If the user double-clicks the `Connection Strings` item or selects this item and clicks the `Open Feature` link in the task panel, the IIS 7 Manager will navigate to the `ConnectionStringsPage` module list page shown in Figure 10-4.



Figure 10-4

If the user clicks the Add link in the task panel associated with the `ConnectionStringsPage` module list page, it will launch the Add Connection String task form shown in Figure 10-5.

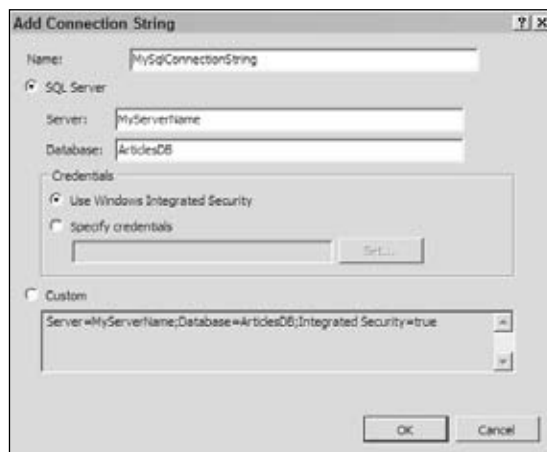


The dialog box is titled "Add Connection String". It has a "Name:" field with the text "MyDataFile". Below this are two radio buttons: "SQL Server" (selected) and "Custom". Under "SQL Server", there are fields for "Server:" and "Database:". Under "Credentials", there are two radio buttons: "Use Windows Integrated Security" (selected) and "Specify credentials". Below "Specify credentials" is a text field and a "Set..." button. Under "Custom", there is a large text field containing "App_Data/Articles.xml". At the bottom right are "OK" and "Cancel" buttons.

Figure 10-5

The Add Connection String task form allows the user to add a new connection string to the `<connectionStrings>` section of the underlying configuration file. Let's say the user enters the text **MyDataFile** as the friendly name for the new connection string, selects the Custom radio button because she wants to add a connection string for the `XmlRssProvider`, which does not use the SQL Server as the back-end data store, and finally enters the text **App_Data/Articles.xml** into the Custom textfield. As you can see, in this case the connection string contains the virtual path of an XML file named `Articles.xml` located in the `App_Data` directory of the Web application that uses your RSS provider-based service to generate the RSS document. Listing 10-17 presents an example of such an XML file. Finally the user clicks the OK button to add the connection string to the `<connectionStrings>` section of the configuration file.

The user then clicks the Add link in the task panel associated with the `ConnectionStringsPage` module list page to launch the Add Connection String task form once more (see Figure 10-6). This time around the user wants to add a connection string for the `SqlRssProvider`.



The dialog box is titled "Add Connection String". It has a "Name:" field with the text "MySqlConnectionString". Below this are two radio buttons: "SQL Server" (selected) and "Custom". Under "SQL Server", there are fields for "Server:" with the text "MyServerName" and "Database:" with the text "ArticlesDB". Under "Credentials", there are two radio buttons: "Use Windows Integrated Security" (selected) and "Specify credentials". Below "Specify credentials" is a text field and a "Set..." button. Under "Custom", there is a large text field containing "Server=MyServerName;Database=ArticlesDB;Integrated Security=true". At the bottom right are "OK" and "Cancel" buttons.

Figure 10-6

Chapter 10: Extending the Integrated Providers Model

Next, the user enters **MySqlConnectionString** as the friendly name for the new connection string as shown in Figure 10-6, the name of the SQL Server that contains the **ArticlesDB** database in the **Server** textfield, and **ArticlesDB** into the **Database** textfield. Figures 10-1 and 10-2 show an example of such database. Finally, the user clicks the OK button to add the new connection string to the <connectionStrings> section of the configuration file.

As Figure 10-7 shows, the **ConnectionStringsPage** module list page now displays both of the newly added connection strings.



Figure 10-7

Keep in mind that we're following a typical user as she's using our three-step workflow to configure the RSS provider-based service. So far I've covered the first step, which is adding the required connection strings to the <connectionStrings> section of the configuration file. Next, I discuss the second step of the workflow. As Figure 10-8 shows, the Default Web Site Home page contains an item named RSS.

If the user double-clicks the RSS item or if the user selects the item and clicks the Open Feature link in the task panel, the IIS 7 Manager will navigate to the **RssPage** module dialog page shown in Figure 10-9.

As you can see, the **RssPage** module dialog page contains three textboxes where the user can enter the channel information. Let's say the user enters the values shown in Figure 10-9 and clicks the Apply link in the task panel associated with the **RssPage** module dialog page. This will add an <rss> configuration section with the specified channel information to the **web.config** file of the Default Web Site. This wraps up the second step of the three-step workflow for configuring your RSS provider-based service. Next, I discuss the third step.

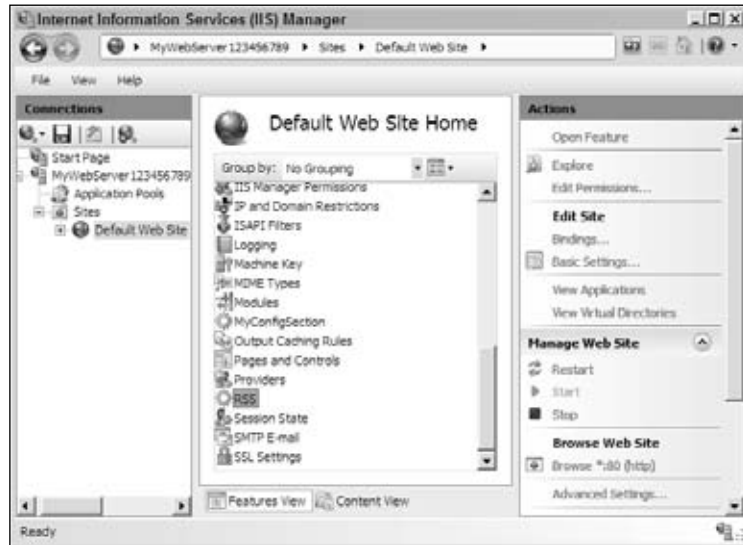


Figure 10-8

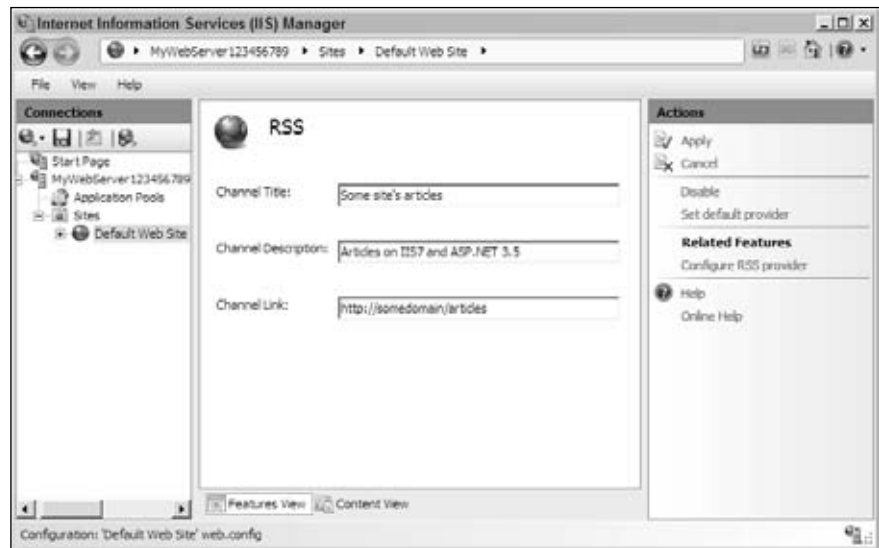


Figure 10-9

Next, the user clicks the Configure RSS provider link in the task panel associated with the `RssPage` module dialog page (see Figure 10-9) to navigate to the `ProviderConfigurationConsolidatedPage` module list page shown in Figure 10-10. If the user selects RSS from the Feature combo box, she'll get the result shown in this figure.

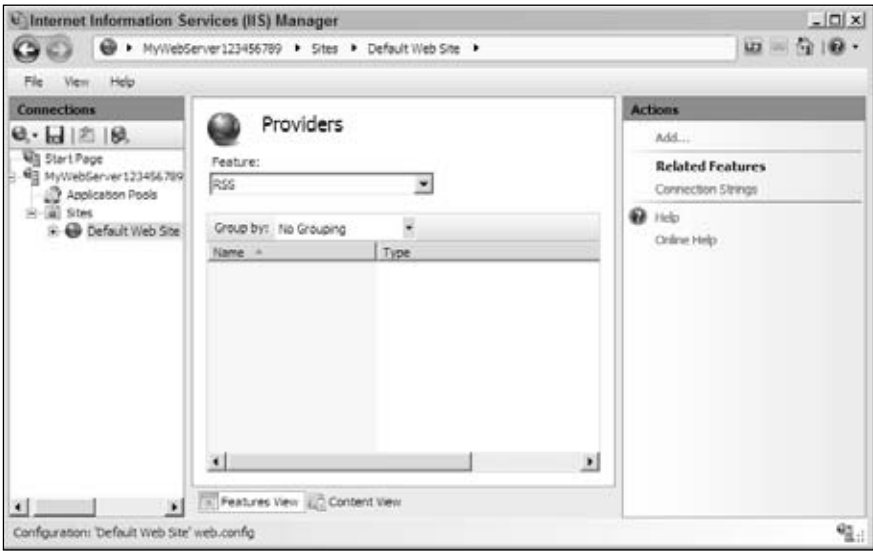


Figure 10-10

As you can see, the list view underneath of the Feature combo box is empty. This makes sense because the user hasn't registered any providers for your RSS provider-based service yet. Now if the user clicks the Add link in the task panel associated with the `ProviderConfigurationConsolidatedPage` module list page, it will launch the `AddProviderForm` task form shown in Figure 10-11.

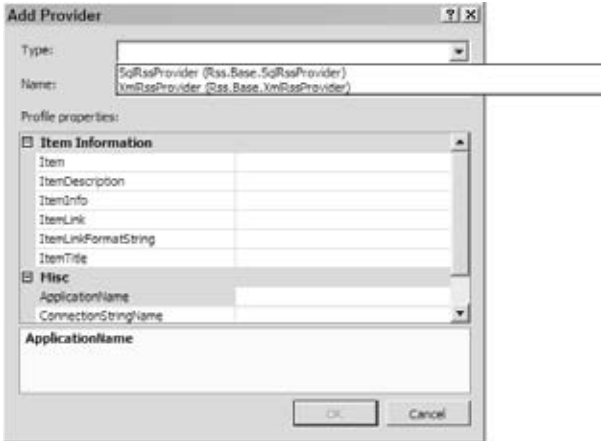


Figure 10-11

As you can see from Figure 10-11, the Type combo box contains the following two items:

- ☐ SqlRssProvider (Rss.Base.SqlRssProvider)
- ☐ XmlRssProvider (Rss.Base.XmlRssProvider)

Let's say the user selects the XmlRssProvider (Rss.Base.XmlRssProvider) item because she wants to add an RSS provider of type XmlRssProvider and enters the text **MyXmlRssProvider** as the friendly name of the new provider into the Name textbox as shown in Figure 10-12.

The dialog box titled "Add Provider" contains a "Type" dropdown menu with "XmlRssProvider (Rss.Base.XmlRssProvider)" selected. Below it is a "Name" text box containing "MyXmlRssProvider". A "Profile properties" section follows, containing two expandable categories: "Item Information" and "Misc". The "Item Information" category is expanded, showing a list of properties: Item (//Article), ItemDescription (Abstract/text()), ItemInfo, ItemLink (@authorname), ItemLinkFormatString, and ItemTitle (@title). The "Misc" category is also expanded, showing ApplicationName and ConnectionStringName (MyDatafile). At the bottom of the dialog are "OK" and "Cancel" buttons.

Figure 10-12

As Figure 10-12 shows, the PropertyGrid control underneath the Name textbox consists of two categories of properties. The first category named Item Information contains item-related properties including Item, ItemInfo, ItemDescription, ItemTitle, ItemLink, and ItemLinkFormatString. The second category named Misc contains miscellaneous properties including ApplicationName, ConnectionStringName, and Description.

As you'll see later, the properties shown in the PropertyGrid control in Figure 10-12 are the properties of an instance of a custom provider configuration settings class named RssProviderConfigurationSettings, which is assigned to the SelectedObject property of this PropertyGrid control. This will all be clear later in this chapter.

Now back to Figure 10-12. Let's say the user enters the values shown in this figure into the appropriate textboxes of the AddProviderForm task form and clicks OK to add the new provider to the <providers> Collection XML element of the <rss> configuration section in the underlying configuration file.

Next the user clicks the Add link in the task panel associated with the ProviderConfigurationConsolidatedPage module list page shown in Figure 10-10 to launch the AddProviderForm task form once again as shown in Figure 10-13.

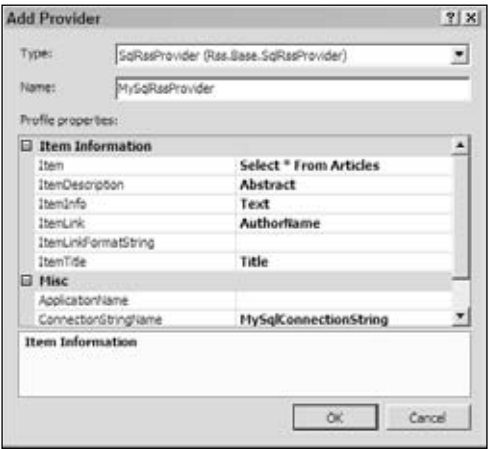


Figure 10-13

The user selects the `SqlRssProvider` option from the Type combo box because this time around she wants to add an RSS provider of type `SqlRssProvider` and enters the text **MySqlRssProvider** as the friendly name of the new provider. Let's say the user enters the values shown in Figure 10-13 into the associated textboxes of the PropertyGrid control shown in Figure 10-13 and clicks OK to add the provider to the underlying configuration file. As Figure 10-14 shows, the `ProviderConfigurationConsolidatedPage` module list page now displays the `MySqlRssProvider` and `MyXmlRssProvider` providers.



Figure 10-14

Next, the user needs to go back to the `RssPage` module dialog page shown in Figure 10-9 to specify a default provider for the RSS provider-based service. The user clicks the Set default provider link in the

task panel associated with the `RssPage` module dialog page to launch the Set default provider task form shown in Figure 10-15, selects the `MyXmlRssProvider` option from the Default Provider combo box, and clicks OK to specify the `MyXmlRssProvider` provider as the default provider of the RSS provider-based service.



Figure 10-15

Now that you've seen these graphical management extensions in action, it's time to dive into the implementation details of these extensions.

As discussed in the previous chapters, extending the IIS 7 and ASP.NET integrated graphical management system requires adding two sets of managed code: client-side and server-side managed code.

Add a new directory named `GraphicalManagement` to the `Rss` project and add two subdirectories named `Client` and `Server` to the `GraphicalManagement` directory. These two subdirectories will respectively contain the classes that will make up the client-side and server-side managed code.

Client-Side Managed Code

Take these steps to implement the client-side managed code:

1. Implement a custom provider configuration settings class to represent the configuration settings of an RSS provider.
2. Implement a custom provider feature class to represent your RSS provider-based service.
3. Implement a custom section `PropertyBag` wrapper class.
4. Implement a custom module service proxy.
5. Implement a custom module page.
6. Implement a custom task form.
7. Implement a custom module.

I walk you through each of these steps in this section.

Implementing a Custom Provider Configuration Settings Class

Following the recipe, first you need to implement a custom provider configuration settings class named `RssProviderConfigurationSettings` that inherits from the `ProviderConfigurationSettings` base class as shown in Listing 10-21. Add a new source file named `RssProviderConfigurationSettings.cs` to the `GraphicalManagement/Client` directory of the `Rss` project and add the code shown in Listing 10-21 to this source file.

Chapter 10: Extending the Integrated Providers Model

You also need to add a reference to the `Microsoft.Web.Management.dll` assembly to the `Rss` project. This assembly is located in the following directory on your machine:

```
%WinDir%\System32\inetsrv
```

Listing 10-21: The `RssProviderConfigurationSettings` Class

```
using Microsoft.Web.Management.Client.Extensions;
using Microsoft.Web.Management.Client;
using System.Collections.Generic;
using System.Collections;
using System.ComponentModel;

namespace Rss.GraphicalManagement.Client
{
    public sealed class RssProviderConfigurationSettings :
        ProviderConfigurationSettings
    {
        private Hashtable settings;
        public RssProviderConfigurationSettings()
        {
            this.settings = new Hashtable();
        }

        public override bool Validate(out string message)
        {
            if (this.ConnectionStringName.Length == 0)
            {
                message = "Connection string name is required";
                return false;
            }

            if (this.Item.Length == 0)
            {
                message = "Item is required";
                return false;
            }

            if (this.ItemTitle.Length == 0)
            {
                message = "Item's title is required";
                return false;
            }

            if (this.ItemDescription.Length == 0)
            {
                message = "Item's description is required";
                return false;
            }

            if (this.ItemLink.Length == 0)
            {
                message = "Item's link is required";
                return false;
            }
        }
    }
}
```

Listing 10-21: *(continued)*

```
    }

    message = string.Empty;
    return true;
}

public string ApplicationName
{
    get
    {
        if (this.settings["applicationName"] != null)
            return (string)this.settings["applicationName"];

        return string.Empty;
    }

    set { this.settings["applicationName"] = value; }
}

public string ConnectionStringName
{
    get
    {
        if (this.settings["connectionStringName"] != null)
            return (string)this.settings["connectionStringName"];

        return string.Empty;
    }

    set { this.settings["connectionStringName"] = value; }
}

public string Description
{
    get
    {
        if (this.settings["description"] != null)
            return (string)this.settings["description"];

        return string.Empty;
    }

    set { this.settings["description"] = value; }
}

[Category("Item Information")]
[Description("Selects the item records from the underlying data store")]
public string Item
{
    get
    {
        if (this.settings["item"] != null)
```

(Continued)

Listing 10-21: *(continued)*

```
        return (string)this.settings["item"];

        return string.Empty;
    }

    set { this.settings["item"] = value; }
}

[Category("Item Information")]
[Description("Provides the select operation with extra information")]
public string ItemInfo
{
    get
    {
        if (this.settings["itemInfo"] != null)
            return (string)this.settings["itemInfo"];

        return string.Empty;
    }

    set { this.settings["itemInfo"] = value; }
}

[Category("Item Information")]
[Description("Selects the title information from an item record")]
public string ItemTitle
{
    get
    {
        if (this.settings["itemTitle"] != null)
            return (string)this.settings["itemTitle"];

        return string.Empty;
    }

    set { this.settings["itemTitle"] = value; }
}

[Category("Item Information")]
[Description("Selects the description information from an item record")]
public string ItemDescription
{
    get
    {
        if (this.settings["itemDescription"] != null)
            return (string)this.settings["itemDescription"];

        return string.Empty;
    }

    set { this.settings["itemDescription"] = value; }
}
```

Listing 10-21: (continued)

```
[Category("Item Information")]
[Description("Selects the link information from an item record")]
public string ItemLink
{
    get
    {
        if (this.settings["itemLink"] != null)
            return (string)this.settings["itemLink"];

        return string.Empty;
    }

    set { this.settings["itemLink"] = value; }
}

[Category("Item Information")]
[Description("Specifies the link format string")]
public string ItemLinkFormatString
{
    get
    {
        if (this.settings["itemLinkFormatString"] != null)
            return (string)this.settings["itemLinkFormatString"];

        return string.Empty;
    }

    set { this.settings["itemLinkFormatString"] = value; }
}

protected override IDictionary Settings
{
    get { return this.settings; }
}
}
```

Note that the constructor of `RssProviderConfigurationSettings` instantiates a `Hashtable` named `settings`. `RssProviderConfigurationSettings` also overrides the `Settings` property of its base class to return a reference to this `settings` private field. Note that `RssProviderConfigurationSettings` exposes the same properties as the `PropertyGrid` control in the `AddProviderForm` task form shown in Figures 10-11 and 10-12. This is no coincidence because when this task form is launched, an instance of the `RssProviderConfigurationSettings` class is assigned to the `SelectedObject` property of this `PropertyGrid` control.

This also means that when the user edits the properties shown in this `PropertyGrid` control, the control automatically updates the corresponding properties of the `RssProviderConfigurationSettings` instance.

Chapter 10: Extending the Integrated Providers Model

Also note that `RssProviderConfigurationSettings` overrides the `Validate` method of the `ProviderConfigurationSettings` base class, to ensure that the end user has specified values for the required properties. As discussed earlier, the event handler for the `Click` event of the OK button of the `AddProviderForm` task form invokes the `Validate` method before it attempts to commit the changes to the underlying configuration file.

Implementing a Custom Provider Feature

The next step in the recipe is to implement a custom provider feature named `RssProviderConfigurationFeature` as shown in Listing 10-22. Next, add a new source file named `RssProviderConfigurationFeature.cs` to the `Client` directory and add the code shown in Listing 10-22 to this source file.

Listing 10-22: The `RssProviderConfigurationFeature` Class

```
using Microsoft.Web.Management.Client.Extensions;
using Microsoft.Web.Management.Client;

namespace Rss.GraphicalManagement.Client
{
    public sealed class RssProviderConfigurationFeature : ProviderFeature
    {
        private RssPage owner;
        private string selectedProvider;

        public RssProviderConfigurationFeature() { }
        internal RssProviderConfigurationFeature(RssPage owner)
        {
            this.owner = owner;
        }

        internal RssProviderConfigurationFeature(RssPage owner,
                                                string selectedProvider)
        {
            this.owner = owner;
            this.selectedProvider = selectedProvider;
        }

        public override string ConnectionStringAttributeName
        {
            get { return "connectionStringName"; }
        }

        public override bool ConnectionStringRequired
        {
            get { return true; }
        }

        public override string FeatureName
        {
            get { return "RSS"; }
        }
    }
}
```

Listing 10-22: *(continued)*

```
public override string ProviderBaseType
{
    get { return "Rss.Base.RssProvider"; }
}

public override string ProviderCollectionPropertyName
{
    get { return "providers"; }
}

public override string[] ProviderConfigurationSettingNames
{
    get
    {
        return new string[] {
            "applicationName", "description", "connectionStringName",
            "item", "itemInfo", "itemTitle", "itemDescription",
            "itemLink", "itemLinkFormatString"
        };
    }
}

public override string SectionName
{
    get { return "system.webServer/rss"; }
}

public override string SelectedProvider
{
    get
    {
        if (this.selectedProvider == null)
            return string.Empty;

        return this.selectedProvider;
    }
}

public override string SelectedProviderPropertyName
{
    get { return "defaultProvider"; }
}

public override ProviderConfigurationSettings Settings
{
    get { return new RssProviderConfigurationSettings(); }
}
}
```

Chapter 10: Extending the Integrated Providers Model

`RssProviderConfigurationFeature` performs the following tasks:

- ❑ Overrides the `ConnectionStringAttributeName` property of its base class to specify "connectionStringName" as the name of the attribute — on an <add> that registers an RSS provider — that specifies the connection string name.
- ❑ Overrides the `ConnectionStringRequired` property of its base class to return `true` to specify that the connection string name is required.
- ❑ Overrides the `FeatureName` property of its base class to specify "RSS" as the feature name. This name will appear in the Feature combo box that displays the list of available features or provider-based services.
- ❑ Overrides the `ProviderBaseType` property of its base class to specify "Rss.Base.RssProvider" as the fully qualified name of the provider base type from which all RSS providers inherit. As discussed earlier, this information is used to populate the Type combo box in the AddProviderForm task form that displays the list of available provider types for a specified provider-based service.
- ❑ Overrides the `ProviderCollectionPropertyName` property of its base class to specify "providers" as the name of the Collection XML element of the <rss> configuration section. This is the Collection XML element that contains the Add XML elements (normally named <add>) that clients use to add new RSS providers.
- ❑ Overrides the `ProviderConfigurationSettingNames` property of its base class to specify "applicationName", "description", "connectionStringName", "item", "itemInfo", "itemDescription", "itemTitle", "itemLink", and "itemLinkFormatString" as the names of the attributes (other than name and type attributes) on the Add XML elements that clients use to add new RSS providers. To put it differently, the `ProviderConfigurationSettingNames` property returns an array that contains the names of the configuration settings of an RSS provider, hence the name `ProviderConfigurationSettingNames`.
- ❑ Overrides the `SectionName` property of its base class to specify "system.webServer/rss" as the fully qualified name of the configuration section associated with the provider-based RSS service.
- ❑ Overrides the `SelectedProvider` property of its base class to return the friendly name of the default provider.
- ❑ Overrides the `SelectedProviderPropertyName` property of its base class to specify "defaultProvider" as the name of the attribute (on the configuration section) that specifies the friendly name of the default provider.
- ❑ Overrides the `Settings` property of its base class to return an instance of the `RssProviderConfigurationSettings` class.

Implementing a Custom Section PropertyBag Wrapper Class

As you'll see later in this chapter, the client-side and server-side managed code use a `PropertyBag` collection to exchange the channel information. The main problem with the `PropertyBag` collection is that you can't program against the collection in a strongly-typed fashion. That is why you need to implement a class named `RssSectionInfo` to expose the content of this `PropertyBag` collection as strongly-typed properties. This class will provide you with the following programming benefits:

- ❑ You can take advantage of Visual Studio IntelliSense support to catch problems as you're coding.

- ❑ You can take advantage of the compiler's type-checking support to catch problems as you're compiling.
- ❑ You can take advantage of the well-known object-oriented benefits.

Note that this `PropertyBag` collection contains the following three pieces of information:

- ❑ A string value that specifies the channel title
- ❑ A string value that specifies the channel description
- ❑ A string value that specifies the channel link

As Listing 10-23 shows, the constructor of the `RssSectionInfo` class takes a `PropertyBag` collection as its argument as you would expect. Add a new source file named `RssSectionInfo.cs` to the `Client` directory and add the code shown in Listing 10-23 to this source file.

Listing 10-23: The `RssSectionInfo` Class

```
using Microsoft.Web.Management.Server;

namespace Rss.GraphicalManagement.Client
{
    public sealed class RssSectionInfo
    {
        private PropertyBag bag;

        public RssSectionInfo(PropertyBag bag)
        {
            this.bag = bag.Clone();
        }

        public string ChannelTitle
        {
            get { return (string)this.bag[0]; }
        }

        public string ChannelDescription
        {
            get { return (string)this.bag[1]; }
        }

        public string ChannelLink
        {
            get { return (string)this.bag[2]; }
        }

        public bool ReadOnly
        {
            get { return (bool)this.bag[3]; }
        }

        public bool Enabled
        {

```

(Continued)

Listing 10-23: *(continued)*

```
        get { return (bool)this.bag[4]; }
    }

}
```

Note that `RssSectionInfo` exposes the first, second, third, fourth, and fifth items in the `PropertyBag` collection as strongly-typed properties named `ChannelTitle`, `ChannelDescription`, `ChannelLink`, `ReadOnly`, and `Enabled`, respectively.

Implementing a Custom Module Service Proxy

Following the recipe, next you need to implement a proxy class named `RssModuleServiceProxy` that inherits from the `ModuleServiceProxy` base class as shown in Listing 10-24. Add a new source file named `RssModuleServiceProxy.cs` to the `Client` directory and add the code shown in this code listing to this source file.

Listing 10-24: The `RssModuleServiceProxy` Class

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System.Web.Configuration;

namespace Rss.GraphicalManagement.Client
{
    class RssModuleServiceProxy : ModuleServiceProxy
    {
        public void EnableRss()
        {
            base.Invoke("EnableRss", new object[0]);
        }

        public void DisableRss()
        {
            base.Invoke("DisableRss", new object[0]);
        }

        public PropertyBag GetSettings()
        {
            return (PropertyBag)base.Invoke("GetSettings", new object[0]);
        }

        public void UpdateChannelSettings(PropertyBag updatedChannelSettings)
        {
            base.Invoke("UpdateChannelSettings",
                new object[] { updatedChannelSettings });
        }
    }
}
```

Listing 10-24: *(continued)*

```
        public PropertyBag GetProviders()
        {
            return (PropertyBag)base.Invoke("GetProviders", new object[0]);
        }
    }
}
```

As you can see, all methods of this proxy class call the `Invoke` method of the base class to invoke the associated methods of the underlying server-side class.

Implementing a Custom Module Page

Listing 10-25 presents the declaration of the members of the `RssPage` module dialog page. Note that the constructor of this class invokes the `InitializeComponent` method to create the user interface of the module dialog page. I discuss the members of this module dialog page in the following sections. Now go ahead and add a new source file named `RssPage.cs` to the `Client` directory and add the code shown in Listing 10-25 to this source file. You also need to add references to the `System.Windows.Forms.dll` and `System.Drawing.dll` assemblies to the `Rss` project.

Listing 10-25: The RssPage Module Page

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System.Web.Configuration;
using Microsoft.Web.Management.Client.Win32;
using System.Windows.Forms;
using System.ComponentModel;
using System.Collections;
using System.Drawing;

namespace Rss.GraphicalManagement.Client
{
    class RssPage : ModuleDialogPage
    {
        public RssPage()
        {
            this.rssEnabled = true;
            InitializeComponent();
        }

        private PropertyBag clone;
        private Label channelTitleLabel;
        private TextBox channelTitleTextBox;
        private Label channelDescriptionLabel;
        private TextBox channelDescriptionTextBox;
        private Label channelLinkLabel;
        private TextBox channelLinkTextBox;
    }
}
```

(Continued)

Listing 10-25: (continued)

```
private bool hasChanges;
private PropertyBag bag;
private RssSectionInfo localInfo;
private bool errorGetSettings;
private bool rssEnabled;
private RssModuleServiceProxy serviceProxy;
private IProviderConfigurationService providerConfigurationService;
private PageTaskList taskList;
private bool readOnly;

private void InitializeComponent();
private void OnChannelTitleTextBoxTextChanged(object sender,
                                                EventArgs e);
private void OnChannelDescriptionTextBoxTextChanged(object sender,
                                                    EventArgs e);
private void OnChannelLinkTextBoxTextChanged(object sender, EventArgs e);
private void UpdateUIState();
protected override bool HasChanges { get; }
protected override bool CanApplyChanges { get; }
protected override void OnActivated(bool initialActivation);
private void GetSettings();
private void OnWorkerGetSettings(object sender, DoWorkEventArgs e);
private void OnWorkerGetSettingsCompleted(object sender,
                                           RunWorkerCompletedEventArgs e);

private void InitializeUI();
private void ClearChannelSettings();
protected override bool ApplyChanges();
private void GetChannelValues();
protected override void CancelChanges();
private sealed class PageTaskList : TaskList { . . . }
private void EnableRss();
private void DisableRss();
private void ConfigureRssProvider();
internal void SetDefaultProvider();
protected override TaskListCollection Tasks { get; }
private IProviderConfigurationService ProviderConfigurationService { get; }
protected sealed override bool ReadOnly { get; }
protected bool RssEnabled { get; }
protected override bool CanRefresh { get; }
protected override void OnRefresh();
}
}
```

InitializeComponent

Listing 10-26 contains the code for the `InitializeComponent` method. The main job of this method is to create the user interface of the `RssPage` module dialog page. Now replace the declaration of the `InitializeComponent` method in the `RssPage.cs` file with the code shown in Listing 10-26.

Listing 10-26: The InitializeComponent Method

```
private void InitializeComponent()
{
    channelTitleLabel = new Label();
    channelTitleTextBox = new TextBox();
    channelDescriptionLabel = new Label();
    channelDescriptionTextBox = new TextBox();
    channelLinkLabel = new Label();
    channelLinkTextBox = new TextBox();

    base.SuspendLayout();

    channelTitleLabel.Location = new Point(0, 30);
    channelTitleLabel.Name = "channelTitleLabel";
    channelTitleLabel.AutoSize = true;
    channelTitleLabel.TabIndex = 0;
    channelTitleLabel.Text = "Channel Title:";
    channelTitleLabel.TextAlign = ContentAlignment.MiddleLeft;

    channelTitleTextBox.Location = new Point(110, 30);
    channelTitleTextBox.Name = "channelTitleTextBox";
    channelTitleTextBox.Width = 250;
    channelTitleTextBox.TabIndex = 1;
    channelTitleTextBox.TextChanged +=
        new EventHandler(OnChannelTitleTextBoxTextChanged);

    channelDescriptionLabel.Location = new Point(0, 80);
    channelDescriptionLabel.Name = "channelDescriptionLabel";
    channelDescriptionLabel.AutoSize = true;
    channelDescriptionLabel.TabIndex = 2;
    channelDescriptionLabel.Text = "Channel Description:";
    channelDescriptionLabel.TextAlign = ContentAlignment.MiddleLeft;

    channelDescriptionTextBox.Location = new Point(110, 80);
    channelDescriptionTextBox.Name = "channelTitleTextBox";
    channelDescriptionTextBox.Width = 250;
    channelDescriptionTextBox.TabIndex = 3;
    channelDescriptionTextBox.TextChanged +=
        new EventHandler(OnChannelDescriptionTextBoxTextChanged);

    channelLinkLabel.Location = new Point(0, 130);
    channelLinkLabel.Name = "channelLinkLabel";
    channelLinkLabel.AutoSize = true;
    channelLinkLabel.TabIndex = 4;
    channelLinkLabel.Text = "Channel Link:";
    channelLinkLabel.TextAlign = ContentAlignment.MiddleLeft;

    channelLinkTextBox.Location = new Point(110, 130);
    channelLinkTextBox.Name = "channelLinkTextBox";
    channelLinkTextBox.Width = 250;
    channelLinkTextBox.TabIndex = 5;
    channelLinkTextBox.TextChanged +=
        new EventHandler(OnChannelLinkTextBoxTextChanged);
}
```

(Continued)

Listing 10-26: (continued)

```
AutoScroll = true;
base.AutoScaleMode = AutoScaleMode.Font;
base.AutoScaleDimensions = new SizeF(6f, 13f);
base.ClientSize = new System.Drawing.Size(0x1d8, 0x228);

base.Controls.Add(channelTitleLabel);
base.Controls.Add(channelTitleTextBox);
base.Controls.Add(channelDescriptionLabel);
base.Controls.Add(channelDescriptionTextBox);
base.Controls.Add(channelLinkLabel);
base.Controls.Add(channelLinkTextBox);

base.ResumeLayout(false);
}
```

Next, I walk you through the implementation of the `InitializeComponent` method. This method begins by creating a textbox that allows the user to specify the channel title, a label to display the text that appears next to this textbox, a textbox that allows the end user to specify the channel description, a label to display the text that appears next to this textbox, a textbox that allows the end user to specify the channel link, and a label to display the text that appears next to this text box.

```
channelTitleLabel = new Label();
channelTitleTextBox = new TextBox();
channelDescriptionLabel = new Label();
channelDescriptionTextBox = new TextBox();
channelLinkLabel = new Label();
channelLinkTextBox = new TextBox();
```

Next, the `InitializeComponent` method registers the `OnChannelTitleTextBoxTextChanged`, `OnChannelDescriptionTextBoxTextChanged`, and `OnChannelLinkTextBoxTextChanged` methods as event handlers for the `TextChanged` events of the three textbox controls it just created:

```
channelTitleTextBox.TextChanged +=
    new EventHandler(OnChannelTitleTextBoxTextChanged);

channelDescriptionTextBox.TextChanged +=
    new EventHandler(OnChannelDescriptionTextBoxTextChanged);

channelLinkTextBox.TextChanged +=
    new EventHandler(OnChannelLinkTextBoxTextChanged);
```

Finally, the `InitializeComponent` method adds the label and textbox controls to the `Controls` collection of the `RssPage` module dialog page:

```
base.Controls.Add(channelTitleLabel);
base.Controls.Add(channelTitleTextBox);
base.Controls.Add(channelDescriptionLabel);
base.Controls.Add(channelDescriptionTextBox);
```

```
base.Controls.Add(channelLinkLabel);
base.Controls.Add(channelLinkTextBox);
```

Listing 10-27 presents the implementation of the callback methods for the `TextChanged` events of the textbox controls. Note that all three methods invoke the `UpdateUIState` method, which in turn sets the `hasChanges` Boolean field to `true` and calls the `Update` method to update the user interface. Replace the declaration of these three methods and the `UpdateUIState` method in the `RssPage.cs` file with the code shown in Listing 10-27.

Listing 10-27: The Callback Methods for the TextChanged Event of the Textbox Controls

```
private void OnChannelTitleTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void OnChannelDescriptionTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void OnChannelLinkTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void UpdateUIState()
{
    this.hasChanges = true;
    base.Update();
}
```

Also note that `RssPage` module dialog page overrides the `HasChanges` and `CanApplyChanges` properties of its base class to return the value of the `hasChanges` field. Replace the declaration of these two properties in the `RssPage.cs` file with the code shown in Listing 10-28.

Listing 10-28: The HasChanges and CanApplyChanges Properties

```
protected override bool HasChanges
{
    get { return this.hasChanges; }
}

protected override bool CanApplyChanges
{
    get { return this.hasChanges; }
}
```

Chapter 10: Extending the Integrated Providers Model

OnActivated

The `RssPage` module dialog page overrides the `OnActivated` method as shown in Listing 10-29. If the module dialog page is being activated for the first time, the method first instantiates the `RssModuleServiceProxy` proxy class and then calls the `GetSettings` method. Replace the declaration of the `OnActivated` method in the `RssPage.cs` file with the code shown in Listing 10-29.

Listing 10-29: The `OnActivated` Method

```
protected override void OnActivated(bool initialActivation)
{
    base.OnActivated(initialActivation);
    if (initialActivation)
    {
        this.serviceProxy = (RssModuleServiceProxy)base.CreateProxy(
                                                                    typeof(RssModuleServiceProxy));
        this.GetSettings();
    }
}
```

As you can see from Listing 10-30, the `GetSettings` method invokes the `StartAsyncTask` method, passing in a delegate of type `DoWorkEventHandler` that represents the `OnWorkerGetSettings` method, and a delegate of type `RunWorkerCompletedEventHandler` that represents the `OnWorkerGetSettingsCompleted` method. Now replace the declaration of the `GetSettings` method in the `RssPage.cs` file with the code shown in Listing 10-30.

Listing 10-30: The `GetSettings` Method

```
private void GetSettings()
{
    base.StartAsyncTask("Getting settings...",
        new DoWorkEventHandler(this.OnWorkerGetSettings),
        new RunWorkerCompletedEventHandler(
            this.OnWorkerGetSettingsCompleted));
    this.hasChanges = false;
}
```

As Listing 10-31 shows, the `OnWorkerGetSettings` method simply invokes the `GetSettings` method on the proxy object. Now replace the declaration of the `OnWorkerGetSettings` method in the `RssPage.cs` file with the code shown in Listing 10-31.

Listing 10-31: The `OnWorkerGetSettings` Method

```
private void OnWorkerGetSettings(object sender, DoWorkEventArgs e)
{
    e.Result = this.serviceProxy.GetSettings();
}
```

When the server response arrives, the `OnWorkerGetSettingsCompleted` method shown in Listing 10-32 is automatically invoked. This method first accesses the `PropertyBag` collection that contains the server data:

```
this.bag = (PropertyBag)e.Result;
```

Next, it instantiates an `RssSectionInfo` object to expose the content of this `PropertyBag` collection as strongly-typed properties and stores this object in a private field named `localInfo`:

```
this.localInfo = new RssSectionInfo(this.bag);
```

Then, it assigns the `ReadOnly` property of the `localInfo` object to the `readOnly` field. Recall that the `ReadOnly` property reflects the value of the `isLocked` attribute on the `<rss>` configuration section:

```
this.readOnly = this.localInfo.ReadOnly;
```

Next, it assigns the `Enabled` property of the `localInfo` object to the `rssEnabled` field. Recall that the `Enabled` property reflects the value of the `enabled` attribute on the `<rss>` configuration section:

```
this.rssEnabled = this.localInfo.Enabled;
```

Finally, it invokes the `InitializeUI` method:

```
this.InitializeUI();
```

Now replace the declaration of the `OnWorkerGetSettingsCompleted` method in the `RssPage.cs` file with the code shown in Listing 10-32.

Listing 10-32: The `OnWorkerGetSettingsCompleted` Method

```
private void OnWorkerGetSettingsCompleted(object sender,
                                         RunWorkerCompletedEventArgs e)
{
    try
    {
        this.bag = (PropertyBag)e.Result;
        this.localInfo = new RssSectionInfo(this.bag);
        this.readOnly = this.localInfo.ReadOnly;
        this.rssEnabled = this.localInfo.Enabled;
        this.errorGetSettings = false;
    }
    catch (Exception exception1)
    {
        base.StopProgress();
        base.DisplayErrorMessage(exception1.Message,
                                "DoWorkerGetSettingsCompleted");
        this.errorGetSettings = true;
    }
    finally
    {
        if (this.bag != null)
            this.InitializeUI();

        this.hasChanges = false;
    }
}
```


Chapter 10: Extending the Integrated Providers Model

As Listing 10-33 shows, the `InitializeUI` method first calls the `ClearChannelSettings` method to clear the textbox controls:

```
ClearChannelSettings();
```

Then, it populates these textbox controls with the channel information received from the server:

```
this.channelTitleTextBox.Text = localInfo.ChannelTitle;
this.channelDescriptionTextBox.Text = localInfo.ChannelDescription;
this.channelLinkTextBox.Text = localInfo.ChannelLink;
```

Now replace the declaration of the `InitializeUI` and `ClearChannelSettings` methods in the `RssPage.cs` file with the code shown in Listing 10-33.

Listing 10-33: The `InitializeUI` and `ClearChannelSettings` Methods

```
private void InitializeUI()
{
    if (localInfo == null)
        return;

    ClearChannelSettings();
    this.channelTitleTextBox.Text = localInfo.ChannelTitle;
    this.channelDescriptionTextBox.Text = localInfo.ChannelDescription;
    this.channelLinkTextBox.Text = localInfo.ChannelLink;
}

private void ClearChannelSettings()
{
    this.channelTitleTextBox.Clear();
    this.channelDescriptionTextBox.Clear();
    this.channelLinkTextBox.Clear();
}
```

ApplyChanges

The `RssPage` module dialog page overrides the `ApplyChanges` method to contain the logic that must be run when the end user clicks the `Apply` link button on the task panel associated with the module dialog page as shown in Listing 10-34. Now replace the declaration of the `ApplyChanges` method in the `RssPage.cs` file with the code shown in this code listing.

Listing 10-34: The `ApplyChanges` Method

```
protected override bool ApplyChanges()
{
    bool flag = false;
    try
    {
        Cursor.Current = Cursors.WaitCursor;
        GetChannelValues();
        this.serviceProxy.UpdateChannelSettings(this.clone());
        this.bag = this.clone();
    }
}
```

Listing 10-34: (continued)

```
        this.localInfo = new RssSectionInfo(this.bag);
        flag = true;
        this.hasChanges = false;
    }
    catch (Exception exception)
    {
        base.DisplayErrorMessage(exception.Message, "ApplyChanges");
    }
    finally
    {
        Cursor.Current = Cursors.Default;
        base.Update();
    }

    return flag;
}
```

ApplyChanges first invokes the GetChannelValues method to extract the channel information from the associated textbox controls:

```
GetChannelValues();
```

Then, it invokes the UpdateChannelSettings method on the proxy passing in the PropertyBag collection that contains the extracted channel information to update the underlying configuration file:

```
this.serviceProxy.UpdateChannelSettings(this.clone);
```

As Listing 10-35 shows, GetChannelValues clones a new PropertyBag collection, extracts the values the user had entered into the associated textbox controls, and stores these values into this PropertyBag collection. Now replace the declaration of the GetChannelValues method in the RssPage.cs file with the code shown in Listing 10-35.

Listing 10-35: The GetChannelValues Method

```
private void GetChannelValues()
{
    this.clone = this.bag.Clone();
    this.clone[0] = this.channelTitleTextBox.Text;
    this.clone[1] = this.channelDescriptionTextBox.Text;
    this.clone[2] = this.channelLinkTextBox.Text;
}
```

CancelChanges

The RssPage module page overrides the CancelChanges method to include the logic that must be run when the end user clicks the Cancel link button on the task panel associated with the module page as shown in Listing 10-36. This method first invokes the InitializeUI method discussed earlier and then calls the Update method to update the user interface. Now replace the declaration of the CancelChanges method in the RssPage.cs file with the code shown in Listing 10-36.

Listing 10-36: The CancelChanges Method

```
protected override void CancelChanges()
{
    this.InitializeUI();
    this.hasChanges = false;
    base.Update();
}
```

PageTaskList

The `RssPage` module dialog page, like any other module page, contains a nested private class named `PageTaskList` that inherits the `TaskList` base class as shown in Listing 10-37. Now replace the declaration of the `PageTaskList` nested class in the `RssPage.cs` file with the code shown Listing 10-37.

Listing 10-37: The PageTaskList Class

```
private sealed class PageTaskList : TaskList
{
    public PageTaskList(RssPage owner)
    {
        this.owner = owner;
    }

    public override ICollection GetTaskItems()
    {
        ArrayList list = new ArrayList();
        if (!this.owner.errorGetSettings)
        {
            if (this.owner.RssEnabled)
            {
                if (!this.owner.ReadOnly)
                    list.Add(new MethodTaskItem("DisableRss", "Disable",
                                                "EnableDisable", "Disables RSS"));
            }

            else
            {
                list.Add(new MessageTaskItem(MessageTaskItemType.Information,
                                                "RssDisabledMessage", "RssDisabled"));
                if (!this.owner.ReadOnly)
                    list.Add(new MethodTaskItem("EnableRss", "Enable",
                                                "EnableDisable", "Enables RSS"));
            }
        }

        if (!this.owner.ReadOnly &&
            (this.owner.ProviderConfigurationService != null))
        {
            list.Add(new MethodTaskItem("SetDefaultProvider", "Set default provider",
                                        "EnableDisable"));
            list.Add(new TextTaskItem("Related Features", "Providers", true));
            list.Add(new MethodTaskItem("ConfigureRssProvider",
```

Listing 10-37: (continued)

```

        "Configure RSS provider", "Providers",
        "Configure RSS provider"));
    }

    return (TaskItem[])list.ToArray(typeof(TaskItem));
}

public void SetDefaultProvider()
{
    this.owner.SetDefaultProvider();
}

public void ConfigureRssProvider()
{
    this.owner.ConfigureRssProvider();
}

public void EnableRss()
{
    this.owner.EnableRss();
}

public void DisableRss()
{
    this.owner.DisableRss();
}

private RssPage owner;
}

```

Next, I walk you through the implementation of the `GetTaskItems` method. This method begins by instantiating an `ArrayList`:

```
ArrayList list = new ArrayList();
```

If the `RssPage` module page had no problem downloading the configuration settings from the server, `GetTaskItems` checks whether the RSS service is enabled. If so, and if the underlying configuration section is not locked, the `GetTaskItems` method creates a `MethodTaskItem` task item to represent the Disable link button on the task panel associated with the `RssPage` module page and adds this `MethodTaskItem` task item to the `ArrayList`. Note that `GetTaskItems` registers the `DisableRss` method as an event handler for the `Click` event of the Disable link button. As Listing 10-37 shows, the `DisableRss` method of `PageTaskList` delegates to the `DisableRss` method of `RssPage` module dialog page.

```

        if (this.owner.rssEnabled)
        {
            if (!this.owner.ReadOnly)
                list.Add(new MethodTaskItem("DisableRss", "Disable",
                                            "EnableDisable", "Disables RSS"));
        }
    }

```

Chapter 10: Extending the Integrated Providers Model

If the RSS service is disabled, `GetTaskItems` first creates an informational `MessageTaskItem` task item to display a message that informs the user that the service is disabled and adds this `MessageTaskItem` task item to the `ArrayList`. Then, if the underlying configuration section is not locked, it creates a `MethodTaskItem` task item to represent the Enable link button on the task panel associated with the `RssPage` module dialog page and adds this `MessageTaskItem` task item to the `ArrayList`. Note that `GetTaskItem` registers the `EnableRss` method as an event handler for `Click` event of the Enable link button. As Listing 10-37 shows, the `EnableRss` method of `PageTaskList` delegates to the `EnableRss` method of `RssPage` dialog page.

```
else
{
    list.Add(new MessageTaskItem(MessageTaskItemType.Information,
                                "RssDisabledMessage", "RssDisabled"));
    if (!this.owner.ReadOnly)
        list.Add(new MethodTaskItem("EnableRss", "EnableTask",
                                    "EnableDisable", "Enables RSS"));
}
```

Next, `GetTaskItem` checks whether both of the following two conditions are met:

- ❑ The underlying configuration section is not locked. Recall that the `ReadOnly` property of the `RssPage` module dialog page reflects the value of the `isLocked` attribute on the `<rss>` configuration section.
- ❑ The `ProviderConfigurationService` property of the `RssPage` module dialog has been set. I discuss the significance of this property and what this property references later in this chapter.

If both of these conditions are met, the `GetTaskItems` method takes these steps:

1. Creates a `MethodTaskItem` task item to represent the “Set default provider” link button and adds this `MethodTaskItem` task item to the `ArrayList`. Note that `GetTaskItems` registers the `SetDefaultProvider` method as an event handler for the `Click` event of the “Set default provider” link button. As Listing 10-37 shows, the `SetDefaultProvider` method delegates to the `SetDefaultProvider` method of the `RssPage` module dialog page.

```
list.Add(new MethodTaskItem("SetDefaultProvider", "Set default provider",
                            "EnableDisable"));
```

2. Creates a `TextTaskItem` to represent the Related Features header text and adds this `TextTaskItem` task item to the `ArrayList`.

```
list.Add(new TextTaskItem("Related Features", "Providers", true));
```

3. Creates a `MethodTaskItem` task item to represent the “Configure RSS provider” link button and adds this `MethodTaskItem` task item to the `ArrayList`. Note that it also registers the `ConfigureRssProvider` method as an event handler for the `Click` event of the “Configure RSS provider” link button. As Listing 10-37 shows, the `ConfigureRssProvider` methods delegate to the `ConfigureRssProvider` methods of the `RssPage` module dialog page.

```
list.Add(new MethodTaskItem("ConfigureRssProvider", "Configure RSS provider",
                            "Providers", "Configure RSS provider"));
```

Finally, `GetTaskItems` loads the content of the `ArrayList` into an array and returns this array, which contains all the task items that the `GetTaskItems` method has added, to its caller.

```
return (TaskItem[])list.ToArray(typeof(TaskItem));
```

EnableRss

As Listing 10-38 shows, the `EnableRss` method of `RssPage` module dialog page invokes the `EnableRss` method on the proxy to enable the RSS service in the underlying configuration file. Now replace the declaration of the `EnableRss` method in the `RssPage.cs` file with the code shown in Listing 10-38.

Listing 10-38: The EnableRss Method

```
private void EnableRss()
{
    try
    {
        Cursor.Current = Cursors.WaitCursor;
        this.serviceProxy.EnableRss();
    }

    catch (Exception exception)
    {
        base.DisplayErrorMessage(exception, null);
    }

    finally
    {
        this.rssEnabled = true;
        Cursor.Current = Cursors.Default;
        this.Refresh();
    }
}
```

DisableRss

As you can see from Listing 10-39, the `DisableRss` method first ensures that the end user does indeed want to disable the RSS service and then invokes the `DisableRss` method on the proxy to disable the service in the underlying configuration file. Now replace the declaration of the `DisableRss` method in the `RssPage.cs` file with the code shown in Listing 10-39.

Listing 10-39: The DisableRss Method

```
private void DisableRss()
{
    if (base.ShowMessage("Do you really want to disable RSS?",
        MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question,
        MessageBoxDefaultButton.Button2) == DialogResult.Yes)
    {
        try
        {
            Cursor.Current = Cursors.WaitCursor;
```

(Continued)

Listing 10-39: *(continued)*

```
        this.serviceProxy.DisableRss();
        this.rssEnabled = false;
    }

    catch (Exception exception)
    {
        base.DisplayErrorMessage(exception, null);
    }

    finally
    {
        Cursor.Current = Cursors.Default;
        this.Refresh();
    }
}
```

ConfigureRssProvider

As Listing 10-40 illustrates, this method instantiates an `RssProviderConfigurationFeature` provider feature and invokes the `ConfigureProvider` method on the `ProviderConfigurationService` property, passing in this provider feature. Now replace the declaration of the `ConfigureRssProvider` method in the `RssPage.cs` file with the code shown Listing 10-40.

Listing 10-40: The ConfigureRssProvider Method

```
private void ConfigureRssProvider()
{
    this.ProviderConfigurationService.ConfigureProvider(
        new RssProviderConfigurationFeature(this));
}
```

SetDefaultProvider

As Listing 10-41 shows, the `SetDefaultProvider` method of the `RssPage` module dialog page instantiates an `RssSettingsForm` task form and displays it to the end user. As you'll see later in this chapter, this task form contains a combo box that displays the list of available providers to choose from. The end user chooses a provider from the list and clicks the OK button on the task form to commit the changes to the underlying configuration file. Now replace the declaration of the `SetDefaultProvider` method in the `RssPage.cs` file with the code shown Listing 10-41.

Listing 10-41: The SetDefaultProvider Method

```
internal void SetDefaultProvider()
{
    using (RssSettingsForm form =
        new RssSettingsForm(base.Module, this, this.serviceProxy,
            this.ProviderConfigurationService))
    {
        if ((base.ShowDialog(form) == DialogResult.OK) && form.HasChanges)
        {
            // ...
        }
    }
}
```

Listing 10-41: *(continued)*

```
        this.Refresh();  
    }  
}
```

Tasks

The `RssPage` module dialog page, like any other module page, overrides the `Tasks` property as shown in Listing 10-42. Now replace the declaration of the `Tasks` property in the `RssPage.cs` file with the code shown in Listing 10-42.

Listing 10-42: The Tasks Property

```
protected override TaskListCollection Tasks  
{  
    get  
    {  
        if (this.taskList == null)  
            this.taskList = new PageTaskList(this);  
  
        TaskListCollection tasks = base.Tasks;  
        tasks.Add(this.taskList);  
        return tasks;  
    }  
}
```

ProviderConfigurationService

The `RssPage` module dialog page exposes a property of type `IProviderConfigurationService` named `ProviderConfigurationService` as shown in Listing 10-43. When this property is accessed for the first time, it automatically calls the `GetService` method to access the provider configuration service and stores this service in a private field. As discussed in the previous chapter, the provider configuration service returned from the `GetService` method references the `ProviderConfigurationModule` instance that registers the `ProviderConfigurationConsolidatedPage` module list page with the IIS 7 and ASP.NET integrated infrastructure (see Listing 9-17). Replace the declaration of the `ProviderConfigurationService` property in the `RssPage.cs` file with the code shown in Listing 10-43.

Listing 10-43: The ProviderConfigurationService Property

```
private IProviderConfigurationService ProviderConfigurationService  
{  
    get  
    {  
        if (this.providerConfigurationService == null)  
            this.providerConfigurationService =  
                (IProviderConfigurationService)base.GetService(  
                    typeof(IProviderConfigurationService));  
  
        return this.providerConfigurationService;  
    }  
}
```


Chapter 10: Extending the Integrated Providers Model

ReadOnly

As you can see from Listing 10-44, the `RssPage` module dialog page overrides the `ReadOnly` property to return the value of the `readOnly` field. Recall that this field reflects the value of the `isLocked` attribute on the underlying `<rss>` configuration section. Now replace the declaration of the `ReadOnly` property in the `RssPage.cs` file with the code shown Listing 10-44.

Listing 10-44: The ReadOnly Property

```
protected sealed override bool ReadOnly
{
    get { return this.readOnly; }
}
```

Refreshing

The `RssPage` module dialog page overrides two refreshing related members as shown in Listing 10-45. The `CanRefresh` property returns `true` to specify that this module dialog page is refreshable. The `OnRefresh` method simply calls the `GetSettings` method discussed earlier. Now replace the declaration of the `CanRefresh` property and `OnRefresh` method in the `RssPage.cs` file with the code shown Listing 10-45.

Listing 10-45: The CanRefresh and OnRefresh Members

```
protected override bool CanRefresh
{
    get { return true; }
}

protected override void OnRefresh()
{
    this.GetSettings();
}
```

RssEnabled

Listing 10-46 presents the implementation of the `RssEnabled` property. Now replace the declaration of the `RssEnabled` property in the `RssPage.cs` file with the code shown in Listing 10-46.

Listing 10-46: The RssEnabled Property

```
protected bool RssEnabled
{
    get { return this.rssEnabled; }
}
```

Implementing a Custom Task Form

Listing 10-47 presents the implementation of the `RssSettingsForm` task form. Next, add a new source file named `RssSettingsForm.cs` to the `Client` directory and add the code shown in Listing 10-47 to this source file.

Listing 10-47: The RssSettingsForm Task Form

```

using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System.Web.Configuration;
using Microsoft.Web.Management.Client.Win32;
using System.Windows.Forms;
using System.ComponentModel;
using System.Collections;
using System.Drawing;

namespace Rss.GraphicalManagement.Client
{
    internal sealed class RssSettingsForm : TaskForm
    {
        private bool canAccept;
        private Panel contentPanel;
        private bool hasChanges;
        private ComboBox providerComboBox;
        private IProviderConfigurationService providerConfigurationService;
        private Label providerLabel;
        private RssPage rssPage;
        private RssModuleServiceProxy serviceProxy;

        public RssSettingsForm(IServiceProvider serviceProvider, RssPage rssPage,
                               RssModuleServiceProxy serviceProxy,
                               IProviderConfigurationService providerConfigurationService)
            : base(serviceProvider)
        {
            this.rssPage = rssPage;
            this.serviceProxy = serviceProxy;
            this.providerConfigurationService = providerConfigurationService;
            this.InitializeComponent();
            this.Text = "Set default provider";
            this.GetProviders();
            this.UpdateUIState();
            this.hasChanges = false;
        }

        private void GetProviders()
        {
            try
            {
                Cursor.Current = Cursors.WaitCursor;
                PropertyBag providers = this.serviceProxy.GetProviders();
                if (providers != null)
                {
                    string b = (string)providers[1];
                    string[] textArray = (string[])providers[2];
                    this.providerComboBox.Enabled = !((bool)providers[0]);
                    foreach (string text2 in textArray)

```

(Continued)

Listing 10-47: (continued)

```
        {
            int num = this.providerComboBox.Items.Add(text2);
            if (string.Equals(text2, b, StringComparison.OrdinalIgnoreCase))
                this.providerComboBox.SelectedIndex = num;
        }
    }
}
catch (Exception exception)
{
    this.DisplayErrorMessage(exception, null);
}

finally
{
    this.Cursor = Cursors.Default;
}
}

private void InitializeComponent()
{
    this.contentPanel = new ManagementPanel();
    this.providerLabel = new Label();
    this.providerComboBox = new ComboBox();
    this.contentPanel.SuspendLayout();
    base.SuspendLayout();
    this.contentPanel.Controls.Add(this.providerLabel);
    this.contentPanel.Controls.Add(this.providerComboBox);
    this.contentPanel.Dock = DockStyle.Fill;
    this.contentPanel.Location = new Point(0, 0);
    this.contentPanel.Name = "contentPanel";
    this.contentPanel.Size = new Size(0x114, 90);
    this.contentPanel.TabIndex = 0;
    this.providerLabel.Location = new Point(0, 0);
    this.providerLabel.Name = "providerLabel";
    this.providerLabel.AutoSize = true;
    this.providerLabel.TabIndex = 0;
    this.providerLabel.TextAlign = ContentAlignment.MiddleLeft;
    this.providerLabel.Text = "Default Provider";
    this.providerComboBox.Anchor = AnchorStyles.Right | AnchorStyles.Left |
                                AnchorStyles.Top;
    this.providerComboBox.Location = new Point(0, 0x10);
    this.providerComboBox.Name = "providerComboBox";
    this.providerComboBox.Size = new Size(0x114, 0x15);
    this.providerComboBox.DropDownStyle = ComboBoxStyle.DropDownList;
    this.providerComboBox.TabIndex = 1;
    this.providerComboBox.SelectedIndexChanged +=
        new EventHandler(this.OnProviderComboBoxSelectedIndexChanged);
    base.ClientSize = new Size(300, 100);
    base.AutoScaleMode = AutoScaleMode.Font;
    base.Name = "RssSettingsForm";
    this.contentPanel.ResumeLayout(false);
    this.contentPanel.PerformLayout();
}
```

Listing 10-47: *(continued)*

```
        base.SetContent(this.contentPanel);
        base.ResumeLayout(false);
    }

    protected override void OnAccept()
    {
        base.StartAsyncTask(new DoWorkEventHandler(this.OnWorkerDoWork),
            new RunWorkerCompletedEventHandler(this.OnWorkerCompleted));
        base.UpdateTaskForm();
    }

    private void OnProviderComboBoxSelectedIndexChanged(object sender, EventArgs e)
    {
        this.UpdateUIState();
    }

    private void OnWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
    {
        base.UpdateTaskForm();
        if (e.Error != null)
            this.DisplayErrorMessage(e.Error, null);

        else
        {
            base.DialogResult = DialogResult.OK;
            base.Close();
        }
    }

    private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
    {
        if (this.hasChanges)
            this.providerConfigurationService.ConfigureProvider(
                new RssProviderConfigurationFeature(this.rssPage,
                    (string)this.providerComboBox.SelectedItem));
    }

    private void UpdateUIState()
    {
        this.hasChanges = true;
        this.canAccept = !string.IsNullOrEmpty(this.providerComboBox.Text);
        base.UpdateTaskForm();
    }

    protected override bool CanAccept
    {
        get
        {
            if (!base.BackgroundJobRunning)
                return this.canAccept;

            return false;
        }
    }
}
```

(Continued)

Listing 10-47: *(continued)*

```
    }  
    }  
  
    protected override bool CanShowHelp  
    {  
        get { return false; }  
    }  
  
    public bool HasChanges  
    {  
        get { return this.hasChanges; }  
    }  
    }  
}
```

Constructor

Next, I walk you through the implementation of the members of the `RssSettingsForm` task form. As Listing 10-47 shows, the constructor of this task form takes four parameters: the first parameter references the service provider, the second parameter references the `RssPage` module dialog page associated with the task form, the third parameter references the `RssModuleServiceProxy` proxy, and the fourth parameter references the provider configuration service. The constructor stores the references to the `RssPage` module dialog page, service proxy, and provider configuration service in private fields for future reference.

```
this.rssPage = rssPage;  
this.serviceProxy = serviceProxy;  
this.providerConfigurationService = providerConfigurationService;
```

Next, it invokes the `InitializeComponent` method to create the user interface of the task form:

```
this.InitializeComponent();
```

Then, it invokes the `GetProviders` method to download the list of available provider types from the underlying configuration file:

```
this.GetProviders();
```

Finally, it invokes the `UpdateUIState` method:

```
this.UpdateUIState();
```

InitializeComponent

As Listing 10-47 shows, the `InitializeComponent` method first creates a `ManagementPanel` control:

```
this.contentPanel = new ManagementPanel();
```

Then, it creates the `Type` combo box that displays the list of available provider types and the label that displays the text that appears next to this combo box:

```
this.providerLabel = new Label();
this.providerComboBox = new ComboBox();
```

Next, it adds this combo box and its associated label to the `Controls` collection of the `ManagementPanel` control:

```
this.contentPanel.Controls.Add(this.providerLabel);
this.contentPanel.Controls.Add(this.providerComboBox);
```

Then, it registers the `OnProviderComboBoxSelectedIndexChanged` method as an event handler for the `SelectedIndexChanged` event of the `Type` combo box:

```
this.providerComboBox.SelectedIndexChanged +=
    new EventHandler(this.OnProviderComboBoxSelectedIndexChanged);
```

Finally, it calls the `SetContent` method to specify the `ManagementPanel` control as the content control:

```
base.SetContent(this.contentPanel);
```

GetProviders

As Listing 10-47 shows, `GetProviders` first invokes the `GetProviders` method on the service proxy to download a `PropertyBag` collection that contains the list of available provider types:

```
PropertyBag providers = this.serviceProxy.GetProviders();
```

Then, it retrieves the friendly name of the default provider from this `PropertyBag` collection:

```
string b = (string)providers[1];
```

Next, it retrieves the string array that contains the names of the available provider types:

```
string[] textArray = (string[])providers[2];
```

Then, it retrieves the Boolean value that specifies whether the underlying configuration section is locked, and uses that to determine whether to enable the `Type` combo box that displays the list of available provider types:

```
this.providerComboBox.Enabled = !((bool)providers[0]);
```

Finally, it iterates through the names of the downloaded provider types and adds each one to the `Items` collection of the `Type` combo box. Note that `GetProviders` ensures that the selected item of the `Type` combo box is the default provider type:

```
foreach (string text2 in textArray)
{
    int num = this.providerComboBox.Items.Add(text2);
}
```

Chapter 10: Extending the Integrated Providers Model

```
        if (string.Equals(text2, b, StringComparison.OrdinalIgnoreCase))
            this.providerComboBox.SelectedIndex = num;
    }
```

OnAccept

The `RssSettingsForm` task form overrides the `OnAccept` method to include the logic that must execute when the end user clicks the OK button on the task form. As you can see, this method invokes the `StartAsyncTask` method, passing in a `DoWorkEventHandler` delegate that wraps the `OnWorkerDoWork` method and a `RunWorkerCompletedEventHandler` delegate that wraps the `OnWorkerCompleted` method.

```
protected override void OnAccept()
{
    base.StartAsyncTask(new DoWorkEventHandler(this.OnWorkerDoWork),
        new RunWorkerCompletedEventHandler(this.OnWorkerCompleted));
    base.UpdateTaskForm();
}
```

The `OnWorkerDoWork` method first checks whether the `RssSettingsForm` task form has any changes to commit to the underlying configuration file. If so, it instantiates an `RssProviderConfigurationFeature` object and passes this object into the `ConfigureProvider` method of the provider configuration service.

```
private void OnWorkerDoWork(object sender, DoWorkEventArgs e)
{
    if (this.hasChanges)
        this.providerConfigurationService.ConfigureProvider(
            new RssProviderConfigurationFeature(this.rssPage,
                (string)this.providerComboBox.SelectedItem));
}
```

Recall from the previous section that the provider configuration service in this case is the module that registers the `ProviderConfigurationConsolidatedPage` module page with the IIS 7 and ASP.NET integrated infrastructure. Note that when the `OnWorkerDoWork` method invokes the `RssProviderConfigurationFeature` constructor to instantiate the `RssProviderConfigurationFeature` object, it passes the value of the `SelectedItem` property of the `Type` combo box into this constructor. As discussed in the previous chapter, specifying the selected provider signals the `ConfigureProvider` method that the caller is trying to set the default provider in the underlying configuration file. As such, this method uses the service proxy to set the default provider of the `<rss>` configuration section to the selected provider. Recall that if the `ConfigureProvider` method is invoked without specifying a selected provider, the method assumes that the caller is trying to navigate to the `ProviderConfigurationConsolidatedPage` module list page. As such, it uses the navigation service to navigate to this page.

Implementing a Custom Module

Following the recipe, next you need to implement a custom module named `RssModule` to register your `RssPage` module dialog page with the IIS 7 and ASP.NET integrated infrastructure as shown in Listing 10-48. Add a new source file named `RssModule.cs` to the `Client` directory and add the code shown in Listing 10-48 to this source file.

Listing 10-48: The RssModule Module

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using Microsoft.Web.Management.Client;
using System.Web.Configuration;
using Microsoft.Web.Management.Client.Extensions;

namespace Rss.GraphicalManagement.Client
{
    public class RssModule : Module
    {
        public RssModule() { }

        protected override void Initialize(IServiceProvider serviceProvider,
                                           ModuleInfo moduleInfo)
        {
            base.Initialize(serviceProvider, moduleInfo);
            Connection service =
                (Connection)serviceProvider.GetService(typeof(Connection));
            ModulePageInfo itemPageInfo =
                new ModulePageInfo(this, typeof(RssPage), "RSS",
                                   "Generates RSS document.");

            IControlPanel panel =
                (IControlPanel)serviceProvider.GetService(typeof(IControlPanel));
            panel.RegisterPage(itemPageInfo);
            IExtensibilityManager manager =
                (IExtensibilityManager)serviceProvider.GetService(
                    typeof(IExtensibilityManager));

            if (manager != null)
            {
                RssProviderConfigurationFeature extension =
                    new RssProviderConfigurationFeature();
                manager.RegisterExtension(typeof(ProviderFeature), extension);
            }
        }

        protected override bool IsPageEnabled(ModulePageInfo pageInfo)
        {
            Connection service = (Connection)this.GetService(typeof(Connection));
            if ((service.ConfigurationPath.PathType != ConfigurationPathType.Site) &&
                (service.ConfigurationPath.PathType !=
                 ConfigurationPathType.Application))
                return false;

            return base.IsPageEnabled(pageInfo);
        }

        public override Version MinimumFrameworkVersion
        {
            get { return Module.FxVersion20; }
        }
    }
}
```


Chapter 10: Extending the Integrated Providers Model

As Listing 10-48 shows, the `RssModule` module overrides the `Initialize` method of the base class. First, it accesses the connection service:

```
Connection service = (Connection) serviceProvider.GetService(typeof(Connection));
```

Next, it instantiates a `ModulePageInfo` module page info to represent the `RssPage` module dialog page:

```
ModulePageInfo itemPageInfo =  
    new ModulePageInfo(this, typeof(RssPage), "RSS", "Generates RSS document.");
```

Then, it accesses the control panel service:

```
IControlPanel panel =  
    (IControlPanel) serviceProvider.GetService(typeof(IControlPanel));
```

Next, it calls the `RegisterPage` method on the control panel service to register your `RssPage` module dialog page:

```
panel.RegisterPage(itemPageInfo);
```

Next, it accesses the extensibility manager service:

```
IExtensibilityManager manager =  
    (IExtensibilityManager) serviceProvider.GetService(  
        typeof(IExtensibilityManager));
```

Then, it instantiates an `RssProviderConfigurationFeature` provider feature and registers it with the extensibility manager service under the key that references the `Type` object that represents the `ProviderFeature` type:

```
RssProviderConfigurationFeature extension = new RssProviderConfigurationFeature();  
manager.RegisterExtension(typeof(ProviderFeature), extension);
```

Server-Side Managed Code

Now you've implemented the managed code for the client, but you still need to add the code for the server-side. Take these steps to implement the server-side managed code:

1. Implement a custom module service.
2. Implement a custom configuration module provider.

I discuss these two steps in the following sections.

Implementing a Custom Module Service

Next, you need to implement a custom module service named `RssModuleService` as shown in Listing 10-49. The `RssModuleService` module service exposes the methods that the client-side managed code can invoke to interact with the underlying configuration file. Next add a new source file named `RssModuleService.cs` to the `GraphicalManagement/Server` directory of the `Rss` project and add the code shown in Listing 10-49 to this source file.

Listing 10-49: The RssModuleService Class

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using System.Web.Configuration;
using Rss.ImperativeManagement;

namespace Rss.GraphicalManagement.Server
{
    public class RssModuleService : ModuleService
    {
        public RssModuleService() { }
        private RssSection GetSection()
        {
            if (base.ManagementUnit.Configuration == null)
                base.RaiseException("Configuration error");

            RssSection section1 =
                (RssSection)base.ManagementUnit.Configuration.GetSection(
                    "system.webServer/rss", typeof(RssSection));

            if (section1 == null)
                base.RaiseException("Configuration error");

            return section1;
        }

        [ModuleServiceMethod(PassThrough = true)]
        public void EnableRss()
        {
            RssSection section1 = this.GetSection();
            if (!section1.Enabled)
            {
                section1.Enabled = true;
                base.ManagementUnit.Update();
            }
        }

        [ModuleServiceMethod(PassThrough = true)]
        public void DisableRss()
        {
            RssSection section1 = this.GetSection();
            if (section1.Enabled)
            {
                section1.Enabled = false;
                base.ManagementUnit.Update();
            }
        }

        [ModuleServiceMethod(PassThrough = true)]
        public PropertyBag GetProviders()
        {
            PropertyBag bag1 = new PropertyBag();
        }
    }
}
```

(Continued)

Listing 10-49: (continued)

```
RssSection section1 = this.GetSection();
Rss.ImperativeManagement.ProviderSettingsCollection
    collection1 = section1.Providers;

if (collection1 == null)
    base.RaiseException("Configuration error");

string[] textArray1 = new string[collection1.Count];
for (int num1 = 0; num1 < textArray1.Length; num1++)
{
    textArray1[num1] = collection1[num1].Name;
}

bag1[0] = section1.IsLocked;
bag1[1] = section1.DefaultProvider;
bag1[2] = textArray1;
return bag1;
}

[ModuleServiceMethod(PassThrough = true)]
public PropertyBag GetSettings()
{
    PropertyBag bag1 = new PropertyBag();
    RssSection section1 = this.GetSection();
    bag1[0] = section1.ChannelTitle;
    bag1[1] = section1.ChannelDescription;
    bag1[2] = section1.ChannelLink;
    bag1[3] = section1.IsLocked;
    bag1[4] = section1.Enabled;
    return bag1;
}

[ModuleServiceMethod(PassThrough = true)]
public void UpdateChannelSettings(PropertyBag updatedChannelSettings)
{
    RssSection section1 = this.GetSection();
    section1.ChannelTitle = (string)updatedChannelSettings[0];
    section1.ChannelDescription = (string)updatedChannelSettings[1];
    section1.ChannelLink = (string)updatedChannelSettings[2];
    base.ManagementUnit.Update();
}
}

}
```

Enabling and Disabling the RSS Service

As Listing 10-49 shows, the `RssModuleService` module service exposes a method named `EnableRss`. This method first accesses the `RssSection` object that provides imperative access to the `<rss>` configuration section:

```
RssSection section1 = this.GetSection();
```

Then, it checks whether the `Enabled` property of this `RssSection` object is set to `true`. Keep in mind that this property maps to the `enabled` attribute on the `<rss>` configuration section. If the `Enabled` property is not set to `true`, it sets the value of this property to `true` and calls the `Update` method to commit the changes to the underlying configuration file:

```
section1.Enabled = true;
base.ManagementUnit.Update();
```

As you can see from Listing 10-49, the `RssModuleService` module service also exposes a method named `DisableRss` whose implementation is very similar to `EnableRss`.

GetProviders

As Listing 10-49 shows, the `RssModuleService` module service exposes a method named `GetProviders`. First, it instantiates a `PropertyBag` collection:

```
PropertyBag bag1 = new PropertyBag();
```

Next, it accesses the `RssSection` object that provides imperative access to the `<rss>` configuration section:

```
RssSection section1 = this.GetSection();
```

Then, it accesses the `ProviderSettingsCollection` object that represents the `<providers>` subelement of the `<rss>` configuration section:

```
Rss.ImperativeManagement.ProviderSettingsCollection
                                collection1 = section1.Providers;
```

Next, it instantiates a string array:

```
string[] textArray1 = new string[collection1.Count];
```

Then, it iterates through the `ProviderSettings` objects in the `ProviderSettingsCollection` collection, and stores the value of the `Name` property of each `ProviderSettings` object in the string array. Keep in mind that each `ProviderSettings` object provides imperative access to a particular `<add>` element in the `<providers>` subelement. The `Name` property of a `ProviderSettings` object maps to the `name` attribute on the `<add>` element that the object represents:

```
for (int num1 = 0; num1 < textArray1.Length; num1++)
{
    textArray1[num1] = collection1[num1].Name;
}
```

Next, it stores the value of the `IsLocked` property of the `RssSection` object in the `PropertyBag` collection. Recall that the `IsLocked` property maps to the `isLocked` attribute on the `<rss>` configuration section:

```
bag1[0] = section1.IsLocked;
```

Chapter 10: Extending the Integrated Providers Model

Then, it stores the value of the `DefaultProvider` property of the `RssSection` object in the `PropertyBag` collection. Recall that the `DefaultProvider` property maps to the `defaultProvider` attribute on the `<rss>` configuration section.

```
bag1[1] = section1.DefaultProvider;
```

Next, it stores the string array containing the friendly names of the available providers in the `PropertyBag` collection:

```
bag1[2] = textArray1;
```

Finally, it returns the `PropertyBag` collection to its caller:

```
return bag1;
```

GetSettings

As Listing 10-49 shows, the `GetSettings` method of the `RssModuleService` module service first instantiates a `PropertyBag` collection:

```
PropertyBag bag1 = new PropertyBag();
```

Then, it accesses the `RssSection` object that represents the `<rss>` configuration section:

```
RssSection section1 = this.GetSection();
```

Next, it stores the values of the `ChannelTitle`, `ChannelDescription`, `ChannelLink`, `IsLocked`, and `Enabled` properties of the `RssSection` object in the `PropertyBag` collection. Note that these properties map to the `channelTitle`, `channelDescription`, `channelLink`, `isLocked`, and `enabled` attributes on the `<rss>` configuration section:

```
bag1[0] = section1.ChannelTitle;  
bag1[1] = section1.ChannelDescription;  
bag1[2] = section1.ChannelLink;  
bag1[3] = section1.IsLocked;  
bag1[4] = section1.Enabled;
```

Finally, it returns the `PropertyBag` collection containing the preceding configuration settings to its caller:

```
return bag1;
```

UpdateChannelSettings

As Listing 10-49 shows, the `UpdateChannelSettings` method of the `RssModuleService` module service takes a `PropertyBag` collection containing channel settings as its argument. First, `UpdateChannelSettings` accesses the `RssSection` object that represents the `<rss>` configuration section:

```
RssSection section1 = this.GetSection();
```

Next, it retrieves the first, second, and third items in the `PropertyBag` collection and respectively assigns them to the `ChannelTitle`, `ChannelDescription`, and `ChannelLink` properties of the `RssSection` object. Keep in mind that these properties map to the `channelTitle`, `channelDescription`, and `channelLink` attributes on the `<rss>` configuration section, respectively.

```
section1.ChannelTitle = (string)updatedChannelSettings[0];
section1.ChannelDescription = (string)updatedChannelSettings[1];
section1.ChannelLink = (string)updatedChannelSettings[2];
```

Finally, it invokes the `Update` method to commit the changes made to the properties of the `RssSection` object to the underlying configuration file:

```
base.ManagementUnit.Update();
```

Implementing a Custom Configuration Module Provider

Following the recipe, Listing 10-50 implements a custom configuration module provider named `RssModuleProvider` to register your `RssModule` module and `RssModuleService` module service with the IIS 7 and ASP.NET integrated infrastructure. Add a new source file named `RssModuleProvider.cs` to the `GraphicalMangement/Server` directory of the `Rss` project and add the code shown in Listing 10-50 to this source file.

Listing 10-50: The `RssModuleProvider` Module Provider

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using System.Web.Configuration;
using System.Reflection;

namespace Rss.GraphicalManagement.Server
{
    public class RssModuleProvider : ConfigurationModuleProvider
    {
        private static string ConfigurationReadOnlyDelegationMode;
        internal static readonly DelegationState ConfigurationReadOnlyDelegationState;
        private static string ConfigurationReadWriteDelegationMode;
        internal static readonly DelegationState ConfigurationReadWriteDelegationState;

        static RssModuleProvider()
        {
            ConfigurationReadOnlyDelegationMode = "ConfigurationReadOnly";
            ConfigurationReadWriteDelegationMode = "ConfigurationReadWrite";
            ConfigurationReadOnlyDelegationState =
                new DelegationState(ConfigurationReadOnlyDelegationMode,
                                    "Resources.ConfigurationReadOnlyDelegationStateText",
                                    "Resources.ConfigurationReadOnlyDelegationStateToolTip");
            ConfigurationReadWriteDelegationState =
                new DelegationState(ConfigurationReadWriteDelegationMode,
                                    "Resources.ConfigurationReadWriteDelegationStateText",
                                    "Resources.ConfigurationReadWriteDelegationStateToolTip");
        }
    }
}
```

(Continued)

Listing 10-50: *(continued)*

```
public RssModuleProvider() { }

public override DelegationState GetChildDelegationState(string path)
{
    DelegationState childDelegationState = base.GetChildDelegationState(path);
    if (childDelegationState ==
        SimpleDelegatedModuleProvider.ReadWriteDelegationState)
        return ConfigurationReadWriteDelegationState;

    if (childDelegationState ==
        SimpleDelegatedModuleProvider.ReadOnlyDelegationState)
        return ConfigurationReadOnlyDelegationState;

    return childDelegationState;
}

public override ModuleDefinition GetModuleDefinition(
    IManagementContext context)
{
    AssemblyName assemblyName =
        typeof(Rss.GraphicalManagement.Client.RssModule).Assembly.GetName();

    return new ModuleDefinition(base.Name,
        "Rss.GraphicalManagement.Client.RssModule, " + assemblyName.FullName);
}

public override DelegationState[] GetSupportedChildDelegationStates(
    string path)
{
    DelegationState[] supportedChildDelegationStates =
        base.GetSupportedChildDelegationStates(path);
    for (int i = 0; i < supportedChildDelegationStates.Length; i++)
    {
        if (supportedChildDelegationStates[i] ==
            SimpleDelegatedModuleProvider.ReadOnlyDelegationState)
            supportedChildDelegationStates[i] = ConfigurationReadOnlyDelegationState;

        else if (supportedChildDelegationStates[i] ==
            SimpleDelegatedModuleProvider.ReadWriteDelegationState)
            supportedChildDelegationStates[i] =
                ConfigurationReadWriteDelegationState;

    }
    return supportedChildDelegationStates;
}

public override void SetChildDelegationState(string path,
    DelegationState state)
{

```

```
        if (state == ConfigurationReadOnlyDelegationState)
            base.SetChildDelegationState(path,
                SimpleDelegatedModuleProvider.ReadOnlyDelegationState);

        else if (state == ConfigurationReadWriteDelegationState)
            base.SetChildDelegationState(path,
                SimpleDelegatedModuleProvider.ReadWriteDelegationState);

        else
            base.SetChildDelegationState(path, state);
    }

    public override bool SupportsScope(ManagementScope scope)
    {
        if ((scope != ManagementScope.Application) &&
            (scope != ManagementScope.Site))
            return (scope == ManagementScope.Server);

        return true;
    }

    protected sealed override string ConfigurationSectionName
    {
        get { return "system.webServer/rss"; }
    }

    public override string FriendlyName
    {
        get { return "RSS"; }
    }

    public override Type ServiceType
    {
        get { return typeof(RssModuleService); }
    }
}
}
```

As Listing 10-50 shows, the `RssModuleProvider` module provider overrides the `GetModuleDefinition` method, where it instantiates and returns a `ModuleDefinition` object, passing in the assembly-qualified name of the `RssModule` module to register this module with the integrated infrastructure:

```
public override ModuleDefinition GetModuleDefinition(
    IManagementContext context)
{
    AssemblyName assemblyName =
        typeof(Rss.GraphicalManagement.Client.RssModule).Assembly.GetName();

    return new ModuleDefinition(base.Name,
        "Rss.GraphicalManagement.Client.RssModule, " + assemblyName.FullName);
}
```


Chapter 10: Extending the Integrated Providers Model

As Listing 10-50 shows, the `RssModuleProvider` module provider overrides the `ServiceType` property to return the `Type` object that represents the `RssModuleService` module service to register this module service with the integrated infrastructure:

```
public override Type ServiceType
{
    get { return typeof(RssModuleService); }
}
```

The `RssModuleProvider` module provider overrides the `ConfigurationSectionName` property to specify "system.webServer/rss" as the fully qualified name of the configuration section associated with the RSS service:

```
protected sealed override string ConfigurationSectionName
{
    get { return "system.webServer/rss"; }
}
```

The `RssModuleProvider` module provider overrides the `FriendlyName` property to specify "RSS" as the friendly name of the RSS service. This friendly name appears in the Feature combo box that displays the list of available provider-based services in the `ProviderConfigurationConsolidatedPage` module list page discussed earlier.

```
public override string FriendlyName
{
    get { return "RSS"; }
}
```

As Listing 10-50 shows, the `RssModuleProvider` module provider also exposes a few delegation-related members. The IIS 7 and ASP.NET integrated infrastructure comes with a class named `DelegationState`, shown in Listing 10-51. As the name implies, this class specifies the delegation state, which includes three pieces of information: delegation mode, text, and description. Note that the `DelegationState` class exposes these pieces of information as three read-only properties named `Mode`, `Text`, and `Description`.

Listing 10-51: The `DelegationState` Class

```
public class DelegationState
{
    private string description;
    private string mode;
    private string text;

    public DelegationState(string mode, string text, string description)
    {
        if (string.IsNullOrEmpty(mode))
            throw new ArgumentNullException("mode");

        if (string.IsNullOrEmpty(text))
            throw new ArgumentNullException("text");
    }
}
```

Listing 10-51: (continued)

```
        if (string.IsNullOrEmpty(description))
            throw new ArgumentNullException("description");

        this.mode = mode;
        this.text = text;
        this.description = description;
    }

    public string Description
    {
        get { return this.description; }
    }

    public string Mode
    {
        get { return this.mode; }
    }

    public string Text
    {
        get { return this.text; }
    }
}
```

As you can see from Listing 10-51, the static constructor of `RssModuleProvider` instantiates two instances of the `DelegationState` class and stores them in the `ConfigurationReadOnlyDelegationState` and `ConfigurationReadWriteDelegationState` static fields:

```
static RssModuleProvider()
{
    ConfigurationReadOnlyDelegationMode = "ConfigurationReadOnly";
    ConfigurationReadWriteDelegationMode = "ConfigurationReadWrite";

    ConfigurationReadOnlyDelegationState =
        new DelegationState(ConfigurationReadOnlyDelegationMode,
                           "Resources.ConfigurationReadOnlyDelegationStateText",
                           "Resources.ConfigurationReadOnlyDelegationStateToolTip");

    ConfigurationReadWriteDelegationState =
        new DelegationState(ConfigurationReadWriteDelegationMode,
                           "Resources.ConfigurationReadWriteDelegationStateText",
                           "Resources.ConfigurationReadWriteDelegationStateToolTip");
}
```

Finally, you need to add the boldfaced portions of Listing 10-52 to the `administration.config` file to register the `RssModuleProvider` module provider with the integrated infrastructure.

Listing 10-52: The administration.config Configuration File

```
<configuration>
  <moduleProviders>
    . . .
    <add name="RssModuleProvider"
type="Rss.GraphicalManagement.Server.RssModuleProvider, Rss,
    Version=2.0.0.0, Culture=Neutral, PublicKeyToken=a31626cc5fbb47c3" />
    . . .
  </moduleProviders>
  . . .
  <location path=".">
    <modules>
      . . .
      <add name="RssModuleProvider"/>
      . . .
    </modules>
  </location>
</configuration>
```

Summary

This chapter presented you with a detailed step-by-step recipe for extending the IIS 7 and ASP.NET integrated providers model and used this recipe to extend this model to implement a fully configurable RSS provider-based service that allows you to generate RSS data from any type of data store.

The next chapter moves on to the IIS 7 and ASP.NET integrated tracing and diagnostics infrastructure where you learn how to use this infrastructure to instrument your managed code with tracing.

11

Integrated Tracing and Diagnostics

As the previous chapters of this book show, the IIS 7 and ASP.NET integrated programming environment enables you to implement many of your application requirements in a .NET-compliant language such as C# or Visual Basic. For example, you learned how to use managed code to extend the integrated request processing pipeline to plug in a new managed module, handler, or handler factory, how to extend the integrated configuration system to add a new configuration section, how to extend the integrated imperative management system to add new imperative management classes, how to extend the integrated graphical management system to add new graphical management components such as module pages, task forms, and so on, and how to extend the integrated providers model to add support for new configurable provider-based services.

This chapter shows you how to use the IIS 7 and ASP.NET integrated tracing and diagnostics infrastructure to instrument your managed code with tracing to enable tracking of the execution of your managed code. You learn how to emit trace events from within your managed code, how to route these trace events to the IIS 7 tracing infrastructure, and how to configure the Failed Request Tracing module to consume these trace events.

Integrated Tracing Components

Tracing is a diagnostic system that enables you to trace the execution of your managed code at runtime. The IIS 7 and ASP.NET integrated tracing infrastructure is based on the .NET Framework's tracing system, which uses the best software design practices to ensure design modularity, extensibility, and configurability. One of the main characteristics of such a modular approach to tracing is that different tasks are assigned to different components, where each component is specifically designed to perform a specific task.

Chapter 11: Integrated Tracing and Diagnostics

A careful analysis of tracing reveals that it involves the following tasks:

- ❑ The task of emitting trace events. The component assigned to this task is known as a *trace source*. This component is called a trace source because it is the source of trace event emission. The .NET Framework comes with a trace source named `TraceSource`, which exposes tracing methods that you can use to emit trace events from within your managed code. It is highly recommended that you use the new `TraceSource` class instead of the old `Trace` class.
- ❑ The task of determining whether a specified trace event should be emitted. The component assigned to this task is known as a *switch*. All switches directly or indirectly inherit from the `Switch` base class. This base class defines the API through which a trace source interacts with its attached switch. The `TraceSource` trace source internally uses an instance of a switch named `SourceSwitch` to determine whether to emit a specified trace event.
- ❑ The task of listening for trace events and outputting them to the appropriate medium. The component assigned to this task is known as a *trace listener*. All trace listeners directly or indirectly inherit from the `TraceListener` base class. This base class defines the API through which a trace source interacts with its attached trace listeners in a generic fashion without knowing their real types. When you're programming in the IIS 7 and ASP.NET integrated environment, you should use a trace listener named `IisTraceListener`. This trace listener routes trace events to the IIS 7 tracing infrastructure where they can be consumed by the Failed Request Tracing module.
- ❑ The task of determining whether to output a specified trace event. The component assigned to this task is known as a *trace filter*. All trace filters directly or indirectly inherit from the `TraceFilter` base class. This base class defines the API through which a trace listener interacts with its attached trace filter in a generic fashion without knowing its real type. There are two standard trace filters named `EventTypeFilter` and `SourceFilter` that you can use to filter the trace events that the `IisTraceListener` routes to the IIS 7 tracing infrastructure. You can also implement your own custom trace filters and plug them into the IIS 7 and ASP.NET integrated tracing infrastructure.

You must perform two sets of tasks to instrument your managed code if you want to route your traces to the IIS 7 tracing infrastructure and to consume them in the Failed Request Tracing module. The first set of tasks must be performed from within your code, whereas the second set of tasks could be performed from the configuration file or your code.

Here is the first set of tasks, which must be performed from within your code:

1. Instantiate a `TraceSource` instance with a unique name. Large applications made up of many different components normally instantiate a separate `TraceSource` instance for each component to differentiate trace events emitted from different components. Keep in mind that each trace event contains the name of the trace source that emitted the trace event. Having a separate `TraceSource` instance for each component makes it a whole lot easier to read the application's trace output files and to make sense of their contents because you can easily tell which trace events come from which components of the application.

You should store the `TraceSource` instance in a static field or property to make it available to the rest of the code.

In large applications made up of many different components, each with its own trace source, storing the trace source of a component in a static field or property allows different subsystems

of the same component to use the same trace source to emit trace events to help differentiate trace events emitted from different subsystems of a component from those emitted from other components of the application.

2. Use the tracing methods of the trace source to add as many trace events as you need at as many different places in the code as you need. A large application could contain numerous trace events.
3. Ensure that the conditional compilation symbol "TRACE" is defined before the code is compiled. This conditional compilation symbol instructs the compiler to include tracing method calls in the compiled assembly. If this symbol is not defined, the compiler will simply ignore these tracing method calls, which means that your code will not trace any events.

These three steps are the only required steps that you must take from within your code to instrument your code with tracing. The rest of the necessary steps can be taken from within your code or from the configuration file. I recommend that you do the remaining steps in the configuration file to avoid hard-coding these steps and to ensure the configurability and customizability of the tracing capabilities of your code.

Here are the remaining steps that you must take to instrument your code with tracing:

1. Declaratively (from within the configuration file) or imperatively (from within your code) instantiate, initialize, and attach a switch to the trace source. At runtime, when a tracing method of the trace source is finally invoked to trace a specified event, the method internally consults with the attached switch to determine whether the specified event should be traced.

In larger applications where more than one trace source is used, two or more of these trace sources could share the same switch. Such a switch is known as a shared switch. Changing the settings of a shared switch will affect all the trace sources that use that switch.

As discussed earlier, you call the tracing methods of a trace source to add as many trace events as necessary at as many points in the code as necessary. In other words, you end up adding numerous trace events to your application. By attaching different switches to a trace source, you can control which of these numerous trace events should be traced. Just because you've added all those trace events to your code does not mean that you have to output all of them. Imagine how complex the output trace file of a large application would be and how hard it would be to make sense of the contents of the file if the application emitted all the trace events that the developers have added to the code.

2. Declaratively (from within the configuration file) or imperatively (from within your code) instantiate, initialize, and attach an `IisTraceListener` trace listener to the trace source. At runtime, when a tracing method of the trace source is invoked to trace a specified event, the method internally consults with the attached switch to determine whether the specified event should be traced. If the attached switch approves the emission of the event, the trace source sends the event to the attached trace listener.

In larger applications where more than one trace source is used, two or more trace sources could share the same trace listener. Such a trace listener is known as a shared trace listener. Changing the settings of a shared trace listener will affect the outputted traces of all the trace sources that share that listener.

- 3. Declaratively (from within the configuration file) or imperatively (from within your code) instantiate, initialize, and attach a trace filter to the trace listener. At runtime, when the trace listener receives a trace event, it consults with the attached trace filter to determine whether to route the event to the IIS 7 tracing infrastructure.

You use this recipe to instrument your provider-based RSS service with tracing to track the flow of its execution.

Tasks Performed from within Your Code

As just discussed, the recipe consists of two sets of tasks, and the first set must be performed from within your code. This section discusses and uses the tasks that must be performed from within your code.

Instantiating a Trace Source

TraceSource is a .NET tracing class that exposes methods that you can use within your code to add trace events. The following table describes the constructors of this class:

Constructor	Description
TraceSource (string name)	Instantiates a trace source with the specified name.
TraceSource(string name, SourceLevels defaultLevel)	Instantiates a trace source with the specified name and specified source level.

The name of a trace source uniquely identifies the trace source among other trace sources. If your application uses a single trace source, you should use the name of your application as the name of the trace source. Otherwise, use the name of the component that uses the trace source as the name of the trace source.

Because the name of a trace source appears as part of a tracing message, you can easily differentiate the traces of your application or a specific component of your application from the traces of other applications or other components of your application.

The TraceSource class exposes a public read-only property of type string named Name that returns the name of the trace source.

Note that the second constructor of the TraceSource class takes an enumeration parameter of type SourceLevels. To understand what the SourceLevels parameter does, you need to study the TraceEventType enumeration as defined in Listing 11-1.

Listing 11-1: The TraceEventType Enumeration

```

public enum TraceEventType
{
    Critical = 1,
    Error = 2,
    Information = 8,
    Resume = 0x800,
    Start = 0x100,
    Stop = 0x200,
    Suspend = 0x400,
    Transfer = 0x1000,
    Verbose = 0x10,
    Warning = 4
}

```

As the name suggests, the members of the `TraceEventType` enumeration represent different trace event types that your code can emit. The following table describes these members:

Member	When to Emit
Critical	Emit a <code>Critical</code> trace event type when an irrecoverable error occurs.
Error	Emit an <code>Error</code> trace event type when a recoverable error occurs.
Warning	Emit a <code>Warning</code> trace event type when something unusual but not necessarily an error occurs.
Information	Emit an <code>Information</code> trace event type when something right but of particular interest occurs.
Verbose	Emit a <code>Verbose</code> trace event type when something right but of particular interest occurs where the code emits a big chunk of data.
Resume	Emit a <code>Resume</code> trace event type when your code is about to resume the execution of a specified logical operation.
Start	Emit a <code>Start</code> trace event type when your code is about to start the execution of a specified logical operation.
Stop	Emit a <code>Stop</code> trace event type when your code is about to end the execution of a specified logical operation.
Suspend	Emit a <code>Suspend</code> trace event type when your code is about to suspend the execution of a specified logical operation.
Transfer	Emit a <code>Transfer</code> trace event type when your code is about to transfer control from a specified logical operation to another logical operation.

Chapter 11: Integrated Tracing and Diagnostics

The `Critical`, `Error`, `Warning`, `Information`, and `Verbose` event types are known as *severity event types* because they represent the severity of the trace event. Obviously the `Critical` event type has the highest level of severity because an irrecoverable error has occurred. The `Verbose` event type, on the other hand, has the lowest level of severity because something right has occurred and we want to provide more information about it.

The `Resume`, `Start`, `Stop`, `Suspend`, and `Transfer` event types are known as *activity event types*. These event types are based on the idea that you can divide your application into a set of logical operations or activities. The activity event types let you know when the application resumes, starts, stops, and suspends a specified logical operation or transfers control from one logical operation to another. This is yet another way to differentiate traces coming from different parts of your application.

Listing 11-2 presents the definition of the `SourceLevels` enumeration. Note that this enumeration type is annotated with the `Flags` metadata attribute. This means that you can use bitwise operations between the members of this enumeration.

Listing 11-2: The `SourceLevels` Enumeration

```
[Flags]
public enum SourceLevels
{
    ActivityTracing = 0xff00,
    All = -1,
    Critical = 1,
    Error = 3,
    Information = 15,
    Off = 0,
    Verbose = 0x1f,
    Warning = 7
}
```

The following table describes the members of the `SourceLevels` enumeration.

Member	Description
ActivityTracing	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of types <code>Stop</code> , <code>Start</code> , <code>Suspend</code> , <code>Transfer</code> , and <code>Resume</code> . Recall that these event types are known as activity event types. In other words, passing the <code>ActivityTracing</code> enumeration value as the second argument into the constructor of the <code>TraceSource</code> class tells this class that you’re interested only in tracing activity event types.
All	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace all types of events.
Off	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to disable tracing any type of event.

Member	Description
Critical	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of type <code>Critical</code> .
Error	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of types <code>Critical</code> and <code>Error</code> .
Warning	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of types <code>Critical</code> , <code>Error</code> , and <code>Warning</code> .
Information	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of types <code>Critical</code> , <code>Error</code> , <code>Warning</code> , and <code>Information</code> .
Verbose	Pass this <code>SourceLevels</code> enumeration value as the second argument to the constructor of the <code>TraceSource</code> class to trace only events of types <code>Critical</code> , <code>Error</code> , <code>Warning</code> , <code>Information</code> , and <code>Verbose</code> . Recall that these types of events are known as severity event types. Therefore, passing the <code>Verbose</code> enumeration value as the second argument into the constructor of the <code>TraceSource</code> class tells this class that you're interested only in tracing severity event types.

As discussed earlier, the second constructor of the `TraceSource` class takes a `SourceLevels` enumeration value as its second argument. This enumeration value determines which types of events to trace. In other words, you're telling the trace source which events to trace. This seems to contradict what I said earlier in this chapter. As discussed earlier, it is the responsibility of the switch attached to the trace source — not the trace source itself — to determine which events to trace.

There is no contradiction. If you do instantiate, initialize, and attach a switch to the trace source, the trace source simply ignores the `SourceLevels` enumeration value passed into its constructor. However, if you do not attach a switch to the trace source, the trace source internally instantiates a switch and passes this `SourceLevels` enumeration value to the switch. In other words, by passing this `SourceLevels` enumeration value into the constructor of the trace source, you're in effect setting the source level of the underlying switch. As you'll see later, the switch uses this source level to determine which events should be traced.

Following the recipe, modify the implementation of the `RssService` provider-based service to instantiate a `TraceSource` trace source and store it in a static field named `traceSource` to make it available to other components. Listing 11-3 presents a new version of the `RssService`.

Listing 11-3: The `RssService` Class

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
```

(Continued)

Listing 11-3: *(continued)*

```
using Microsoft.Web.Administration;
using Rss.ImperativeManagement;
using System.Web.Hosting;
using System.Diagnostics;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
        private static RssProviderCollection providers = null;
        private static bool IsInitialized = false;

        private static TraceSource traceSource = null;
        public static TraceSource TraceSource
        {
            get
            {
                Initialize();
                return traceSource;
            }
        }

        public RssProvider Provider
        {
            get { Initialize(); return provider; }
        }

        public RssProviderCollection Providers
        {
            get { Initialize(); return providers; }
        }

        public static void LoadRss(Stream stream)
        {
            Initialize();
            Channel channel = new Channel();
            channel.Title = channelTitle;
            channel.Link = channelLink;
            channel.Description = channelDescription;

            provider.LoadRss(channel, stream);
        }

        private static string channelTitle;
        private static string channelDescription;
        private static string channelLink;

        private static void Initialize()
        {
            if (!IsInitialized)
            {
```

Listing 11-3: *(continued)*

```

traceSource = new TraceSource("myTraceSource");
traceSource.TraceEvent(TraceEventType.Start, 0,
    "[RSS SERVICE] START Initialize");

ServerManager mgr = new ServerManager();
Configuration config =
    mgr.GetWebConfiguration(HostingEnvironment.SiteName,
        HttpContext.Current.Request.ApplicationPath);

RssSection section =
    (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
channelDescription = section.ChannelDescription;
channelLink = section.ChannelLink;
channelTitle = section.ChannelTitle;
traceSource.TraceInformation(
    "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
    channelDescription, channelLink, channelTitle);

providers = new RssProviderCollection();
ProvidersHelper.InstantiateProviders
    (section.Providers, providers, typeof(RssProvider));
provider = providers[section.DefaultProvider];

if (provider == null)
{
    ProviderException ex =
        new ProviderException("Unable to load default RssProvider");
    traceSource.TraceData(TraceEventType.Critical, 0, ex);
    throw ex;
}

IsInitialized = true;
traceSource.TraceEvent(TraceEventType.Stop, 0,
    "[RSS SERVICE] END Initialize");
    }
}
}
}

```

As the boldfaced portion of this code listing shows, `RssService` exposes a property of type `TraceSource` named `TraceSource`. As you'll see throughout this chapter, the rest of the components of the provider-based RSS service will use the trace source that this property returns to trace events. Note that the `TraceSource` class is instantiated only once, that is, when the `Initialize` method of the `RssService` class is invoked:

```

traceSource = new TraceSource("myTraceSource");

```

As discussed earlier, the `TraceSource` constructor takes a string argument that specifies the name of the trace source, which is `"myTraceSource"` in this case. Keep in mind that the trace source name is case sensitive.

Adding Trace Events

Following the recipe, the next order of business is to use the tracing methods of the trace source that you instantiated in the previous section to add trace events to your provider-based RSS service and its related components. However, first you need to gain a good understanding of the tracing methods of the `TraceSource` class, as discussed in the following sections.

TraceEvent

The `TraceSource` class comes with three overloads of the `TraceEvent` method as shown in Listing 11-4. Note that all three methods are annotated with the `ConditionalAttribute` metadata attribute, which is discussed later in this chapter.

Listing 11-4: The `TraceEvent` Overloads

```
[ConditionalAttribute("TRACE")]
public void TraceEvent(TraceEventType eventType, int id);

[ConditionalAttribute("TRACE")]
public void TraceEvent(TraceEventType eventType, int id, string message);

[ConditionalAttribute("TRACE")]
public void TraceEvent(TraceEventType eventType, int id, string format,
                      params object[] args);
```

The first overload of the `TraceEvent` method takes two parameters. The first parameter is of type `TraceEventType`, and specifies the type of the trace event being added. The second parameter is an integer value that specifies the numeric identifier of the trace event being added. This numeric identifier means what you want it to mean. For example, you may decide that all trace events that mark the beginning of the execution of methods in your application should have the numeric identifier of 4. The numeric identifier is just one way to differentiate one group of traces from other groups in the trace output file.

The second overload of the `TraceEvent` method takes the same two parameters as the first overload plus a third parameter that contains a message. As a matter of fact, the first overload under the hood uses an empty string as the message.

The third overload takes the same two parameters as the first overload plus two more parameters. The third parameter contains a format string with zero or more format items, and the fourth parameter contains an array of objects to be formatted. This overload allows you to use the objects of your application to provide more information about its flow of execution. Since the fourth parameter is preceded by the keyword “params”, you can pass the objects to format as separate arguments into the method. You’ll see an example of this later in this chapter.

TraceData

The `TraceData` method is just like the `TraceEvent` method with one notable difference. It allows you to attach an extra object to the trace event being emitted. For example, you can use this method to attach an exception object to a trace event. The `TraceSource` class comes with two overloads of the `TraceData` method, as shown in Listing 11-5. Again notice that both overloads are annotated with the `ConditionalAttribute("TRACE")` metadata attribute, which is discussed later in this chapter.

Listing 11-5: The TraceData Method

```
[Conditional("TRACE")]
public void TraceData(TraceEventType eventType, int id, object data);

[Conditional("TRACE")]
public void TraceData(TraceEventType eventType, int id, params object[] data);
```

Note that the first two parameters of these two overloads are the same as the first two parameters of the `TraceEvent` method. The third parameter references the object being attached in the case of the first overload and an array of objects being attached in the case of the second overload. This means that the second overload allows you to attach multiple objects. Since the fourth parameter is preceded by the keyword “params”, you can pass the objects to format as separate arguments into the method. You’ll see an example of this later in this chapter.

TraceInformation

As Listing 11-6 shows, the `TraceSource` class comes with two overloads of the `TraceInformation` method.

Listing 11-6: The TraceInformation Method

```
[Conditional("TRACE")]
public void TraceInformation(string message)
{
    this.TraceEvent(TraceEventType.Information, 0, message, null);
}

[Conditional("TRACE")]
public void TraceInformation(string format, params object[] args)
{
    this.TraceEvent(TraceEventType.Information, 0, format, args);
}
```

As you can see, the first overload under the hood invokes the following overload of the `TraceEvent` method, passing in the `TraceEventType.Information` as the trace event type, 0 as the trace event numeric identifier, and the specified event message as the event message:

```
[ConditionalAttribute("TRACE")]
public void TraceEvent(TraceEventType eventType, int id, string message);
```

In other words, the first overload of the `TraceInformation` method allows you to emit a trace event of type `Information` with the specified event message.

As Listing 11-6 shows, the second overload of the `TraceInformation` method under the hood invokes the following overload of the `TraceEvent` method, passing in the `TraceEventType.Information` as the trace event type, 0 as the trace event numeric identifier, the specified format string as the format string, and the specified array of objects as the array of objects being formatted:

```
[ConditionalAttribute("TRACE")]
public void TraceEvent(TraceEventType eventType, int id, string format,
    params object[] args);
```

Chapter 11: Integrated Tracing and Diagnostics

Therefore, you can think of these two overloads of the `TraceInformation` method as shortcut methods saving you from having to invoke the underlying `TraceEvent` overloads.

TraceTransfer

The `TraceSource` class comes with a method named `TraceTransfer` as follows:

```
[Conditional("TRACE")]
public void TraceTransfer(int id, string message, Guid relatedActivityId);
```

This method emits a trace event of type `TraceEventType.Transfer` with the specified trace event numeric identifier and specified message. Note that this method also takes a third argument of type `Guid` that contains what is known as a *related activity identifier*. As discussed earlier, you can assign different identifiers to different logical operations or activities of your application. This is yet another way to differentiate one group of traces from other groups in the trace output file.

Next, you use the tracing methods of the trace source that you instantiated for your provider-based RSS service to add trace events to your service as shown in the highlighted portions of Listing 11-7.

Listing 11-7: The `RssService` Class

```
. . .
namespace Rss.Base
{
    public class RssService
    {
        . . .

        private static TraceSource traceSource = null;
        public static TraceSource TraceSource
        {
            get
            {
                Initialize();
                return traceSource;
            }
        }

        . . .

        private static void Initialize()
        {
            if (!IsInitialized)
            {
                traceSource = new TraceSource("myTraceSource");
                traceSource.TraceEvent(TraceEventType.Start, 0,
                    "[RSS SERVICE] START Initialize");
            }
        }

        . . .

        RssSection section =
            (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
        channelDescription = section.ChannelDescription;
```

Listing 11-7: (continued)

```
channelLink = section.ChannelLink;
channelTitle = section.ChannelTitle;
traceSource.TraceInformation(
    "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
    channelDescription, channelLink, channelTitle);

. . .

if (provider == null)
{
    ProviderException ex =
        new ProviderException("Unable to load default RssProvider");
    traceSource.TraceData(TraceEventType.Critical, 0, ex);
    throw ex;
}

IsInitialized = true;
traceSource.TraceEvent(TraceEventType.Stop, 0,
    "[RSS SERVICE] END Initialize");
}
}
}
```

Listing 11-7 calls the `TraceEvent` method on the trace source to add a trace event of type `Start` with a numeric identifier of 0 and a descriptive message to mark the beginning of the execution of the `Initialize` method:

```
traceSource.TraceEvent(TraceEventType.Start, 0, "[RSS SERVICE] START Initialize");
```

Next, it invokes the `TraceInformation` method on the trace source to add a trace event that contains the channel information. Note that it passes four parameters into the `TraceInformation` method: the first parameter is a format string with three format items, and the remaining three parameters specify the objects being formatted, which are the channel information, link, and title in this case.

```
traceSource.TraceInformation(
    "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
    channelDescription, channelLink, channelTitle);
```

If the underlying configuration section does not specify a default provider, Listing 11-7 invokes the `TraceData` method on the trace source to add a trace event. Note that this code listing passes three parameters into this tracing method. The first parameter specifies that the trace event is of type `Critical` because the provider-based RSS service did not recover from the associated error. You've passed a numeric identifier of 0 as the second parameter. The third parameter is a reference to the actual `ProviderException` object.

```
traceSource.TraceData(TraceEventType.Critical, 0, ex);
```


Chapter 11: Integrated Tracing and Diagnostics

Finally, Listing 11-7 invokes the `TraceEvent` method on the trace source to add a trace event of type `Stop` and a message to mark the end of the execution of the `Initialize` method:

```
traceSource.TraceEvent(TraceEventType.Stop, 0,
    "[RSS SERVICE] END Initialize");
```

Defining the Conditional Compilation Symbol “TRACE”

As you saw earlier, all tracing methods are annotated with a metadata attribute named `ConditionalAttribute("TRACE")`. When a method is annotated with the `ConditionalAttribute("ConditionalCompilationSymbol")` metadata attribute, the C#, J#, and VB compilers do not compile the calls into this method unless the conditional compilation symbol passed in the `ConditionalAttribute` metadata attribute is defined. In this case, the conditional compilation symbol is `"TRACE"`. Therefore, you must make sure that the `"TRACE"` conditional compilation symbol is defined. Otherwise none of the trace events that you added in the previous section would make it to the compiled assembly.

There are four different ways to define this symbol:

- ☐ Use the `/define:TRACE` switch if you’re compiling your code from the command line
- ☐ Take these steps if you’re compiling your code from Visual Studio:
 1. In the Solution Explorer, right-click the project node that contains your code.
 2. Select Properties from the menu to launch the Properties page.
 3. Select the Build tab.
 4. Check the Define TRACE constant checkbox.
- ☐ Set the TRACE environment variable, that is, set `TRACE=1`.
- ☐ Add the `#define TRACE` statement to the source code.

Tasks Performed from the Configuration File

As discussed earlier, you must perform two sets of tasks to instrument your code with tracing. The first set of tasks must be performed from within your code; these were discussed in the previous section. The second set contains those tasks that can be performed either from the code or the configuration file, and are discussed in this section. Even though the tasks in the second set can be performed both in the code and the configuration file, I highly recommend that you perform these tasks from the configuration file.

Instantiating and Attaching a Switch

As discussed earlier, a switch is a component that contains the logic that determines whether a specified trace source should trace a specified event. You may be wondering why this logic is not directly included in the trace source itself. The answer to this question is twofold. First, it is not a good design

practice to have the same component perform two different tasks. Actually emitting a trace event is a different task from determining whether the trace event should be emitted. Second, a switch is a configurable component, that is, it can be configured from the configuration file. This allows you to attach a different switch to the same trace source to change the logic that determines which events should be traced without making any code changes. It's all done through the configuration file. We're interested in a special type of switch named `SourceSwitch`.

Before diving into the internals of this switch, let's see how you can attach a switch to a trace source in the configuration file. All tracing-related items in the configuration file go under a section named `<system.diagnostics>`. This section contains a subelement named `<sources>` where you must specify the trace sources that you want to work with. Recall that trace sources are added imperatively from within the code. In other words, you cannot add a new trace source from the configuration file. Your application code must already contain every trace source that you specify in the `<sources>` section.

The `<sources>` section contains zero or more `<source>` elements, where each `<source>` element specifies a particular trace source that you want to work with in the configuration file. You must set the name attribute on a `<source>` element to the name of the trace source that the element represents. For example, the `<sources>` element in the following configuration fragment contains a `<source>` subelement whose name attribute is set to the value `"myTraceSource"` (shown in boldface), which is the name of the trace source that the bottom boldfaced portion of Listing 11-3 instantiates:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" . . .>
        . . .
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

In general, there are two types of switches: local and shared. A shared switch is a switch that two or more trace sources can share. This means that if you change the settings on a shared switch, it will affect all trace sources that use that switch. A local switch is a switch that attaches to a single trace source. As a result, any change in the settings of a local switch affects only the trace source to which it is attached.

The `<system.diagnostics>` section comes with a subelement named `<switches>` where you can add shared switches. To add a shared switch, simply add a new `<add>` subelement to the `<switches>` element and set its name and value attributes. These two attributes specify the name and value of the switch that the `<add>` element represents. The name of a switch is a string that uniquely identifies that switch among other switches. A switch uses its value to determine which trace events should be traced. Because we're only interested in switches of type `SourceSwitch`, this value is a member of the `SourceLevels` enumeration.

For example, the `<switches>` element in the following configuration fragment contains an `<add>` element that represents a switch named `"mySwitch"` with the value `"Error"`. This switch will only let trace events of type `Critical` and `Error` pass through and will block all other types of trace events:

```
<configuration>
  <system.diagnostics>
```

```
<sources>
  <source name="myTraceSource" . . .>
    ...
  </source>
</sources>
<switches>
  <add name="mySwitch" value="Error"/>
</switches>
</system.diagnostics>
</configuration>
```

Adding an `<add>` subelement that represents a shared switch with a specified name and value to the `<switches>` element does not mean that the trace sources will automatically use that switch. To have a trace source use a shared switch defined in the `<switches>` section, you must attach the shared switch to the trace source. To attach a shared switch to a trace source, simply set the `switchName` attribute on the `<source>` element to the value of the `name` attribute of the `<add>` element that represents the shared switch, as shown in the boldfaced portions of the following configuration fragment:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        ...
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="Error"/>
    </switches>
  </system.diagnostics>
</configuration>
```

Next, I show you how to define a local switch for a trace source. The `<source>` element that represents a trace source with a specified name supports two attributes named `switchType` and `switchValue`. Because we're only interested in switches of type `SourceSwitch`, and because this switch type is the default switch type, we don't need to worry about setting the value of the `switchType` attribute. You must set the value of the `switchValue` attribute to the appropriate `SourceLevels` enumeration value. The local switch uses this enumeration value to determine which events should be traced. The boldfaced portion of the following configuration fragment defines a local switch that only lets traces of type `Critical` and `Error` pass through and blocks all other traces:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchValue="Error">
        ...
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

Under the Hood of SourceSwitch

The `SourceSwitch` class, like any other switch, inherits from the `Switch` base class. This base class exposes the following three important properties:

- ❑ `DisplayName`: A public read-only property that maps to the `name` attribute on the `<add>` subelement of the `<switches>` element that represents the switch in the configuration file.
- ❑ `Value`: A protected read-only property that maps to the `value` attribute on the `<add>` subelement of the `<switches>` element that represents the switch in the configuration file.
- ❑ `SwitchSetting`: A protected virtual read/write integer property that gets or sets the current setting of the switch. This integer value reflects the value of the `Value` property.

The `Switch` base class exposes a protected virtual method named `OnValueChanged`, which is invoked every time the value of the switch changes. For example, when you change the value of the `value` attribute on the `<add>` element that represents the switch in the configuration file, the `OnValueChanged` method is automatically invoked under the hood. Listing 11-8 presents the `Switch` class's implementation of this method.

Listing 11-8: The `OnValueChanged` method of the `Switch` Base Class

```
protected virtual void OnValueChanged()
{
    this.SwitchSetting = int.Parse(this.Value, CultureInfo.InvariantCulture);
}
```

As you can see, the base class's implementation simply parses the value of the `Value` property into an integer and assigns this integer to the `SwitchSetting` property. In other words, the base class's implementation assumes that the value assigned to the `value` attribute on the `<add>` element that represents the switch in the configuration file or the value assigned to the `switchValue` attribute on the `<source>` element that represents the associated trace source is an integer.

A subclass of the `Switch` base class such as `SourceSwitch` can override the `OnValueChanged` method to use a different logic to map the value of the `Value` property to the `SwitchSetting` property. Listing 11-9 presents the `SourceSwitch` class's implementation of the `OnValueChanged` method.

Listing 11-9: The `OnValueChanged` Method of the `SourceSwitch` Class

```
protected override void OnValueChanged()
{
    base.SwitchSetting = (int) Enum.Parse(typeof(SourceLevels), base.Value, true);
}
```

As you can see, the `SourceSwitch` class's implementation does not directly map the `Value` property to the `SwitchSetting` property. Instead, it first parses the value of the `Value` property to a `SourceLevels` enumeration value, casts this enumeration value to an integer type, and finally assigns this integer to the `SwitchSetting` property. In other words, the `SourceSwitch` class's implementation of the `OnValueChanged` method assumes that the value assigned to the `value` attribute on the `<add>` element that represents the `SourceSwitch` in the configuration file or the value of the `switchValue` attribute on the `<source>` element that represents the associated trace source is a `SourceLevels` enumeration value.

Chapter 11: Integrated Tracing and Diagnostics

Recall that each member of the `SourceLevels` enumeration type specifies which type of trace events to emit. Therefore, you can use the `value` attribute on the `<add>` element that represents a shared `SourceSwitch` switch or the `switchValue` attribute on the `<source>` element that represents the associated trace source to instruct this switch which events to trace.

The `SourceSwitch` class exposes a method named `ShouldTrace` as shown in Listing 11-10.

Listing 11-10: The `ShouldTrace` Method of `SourceSwitch`

```
public bool ShouldTrace(TraceEventType eventType)
{
    return ((base.SwitchSetting & eventType) != 0);
}
```

As you can see, this method takes a trace event type as its argument and performs a bitwise AND operation between its argument and the `SwitchSettings` property value to determine whether the current setting supports the specified trace event type. If so, it returns true to signal its associated trace source to trace the specified event. As you'll see later in this chapter, the tracing methods of the `TraceSource` class internally invoke the `ShouldTrace` method on the attached `SourceSwitch` to determine whether to trace the specified event. If the `ShouldTrace` method returns false, these tracing methods simply return without tracing the event.

The `SourceSwitch` class exposes a public property named `Level` that allows you to get or set the value of the `SwitchSetting` property as a strongly-typed property as shown in Listing 11-11. Therefore, you have two ways to specify the level of a `SourceSwitch`:

- ❑ Declaratively via the configuration file by setting the value of the `value` attribute on the `<add>` element that represents the `SourceSwitch` if the switch is a shared switch and by setting the value of the `switchValue` attribute on the `<source>` element that represents the associated trace source if the switch is a local switch.
- ❑ Imperatively via setting the value of the `Level` property from within your code.

Listing 11-11: The `Level` Property of `SourceSwitch`

```
public SourceLevels Level
{
    get { return (SourceLevels) base.SwitchSetting; }
    set { base.SwitchSetting = (int) value; }
}
```

Imperatively Instantiating and Attaching a Switch

The previous sections showed you how to declaratively instantiate and attach a shared or local switch to a trace source in the configuration file without writing a single line of code. This section shows you how to do the same thing in code.

Listing 11-12 presents a version of the provider-based `RssService` that imperatively instantiates and attaches a local switch to your trace source.

Listing 11-12: Imperative Instantiation and Attaching of a Local Switch

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
using Microsoft.Web.Administration;
using Rss.ImperativeManagement;
using System.Web.Hosting;
using System.Diagnostics;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
        private static RssProviderCollection providers = null;
        private static bool IsInitialized = false;

        private static TraceSource traceSource = null;
        public static TraceSource TraceSource
        {
            get
            {
                Initialize();
                return traceSource;
            }
        }

        public RssProvider Provider
        {
            get { Initialize(); return provider; }
        }

        public RssProviderCollection Providers
        {
            get { Initialize(); return providers; }
        }

        public static void LoadRss(Stream stream)
        {
            Initialize();
            Channel channel = new Channel();
            channel.Title = channelTitle;
            channel.Link = channelLink;
            channel.Description = channelDescription;

            provider.LoadRss(channel, stream);
        }

        private static string channelTitle;
        private static string channelDescription;
        private static string channelLink;
    }
}
```

(Continued)

Listing 11-12: (continued)

```
private static void Initialize()
{
    if (!IsInitialized)
    {
        traceSource = new TraceSource("myTraceSource");
        SourceSwitch mySwitch = new SourceSwitch("mySwitch");
        mySwitch.Level = SourceLevels.Error;
        traceSource.Switch = mySwitch;

        traceSource.TraceEvent(TraceEventType.Start, 0,
                               "[RSS SERVICE] START Initialize");

        ServerManager mgr = new ServerManager();
        Configuration config =
            mgr.GetWebConfiguration(HostingEnvironment.SiteName,
                                    HttpContext.Current.Request.ApplicationPath);

        RssSection section =
            (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
        channelDescription = section.ChannelDescription;
        channelLink = section.ChannelLink;
        channelTitle = section.ChannelTitle;
        traceSource.TraceInformation(
            "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
            channelDescription, channelLink, channelTitle);

        providers = new RssProviderCollection();
        ProvidersHelper.InstantiateProviders
            (section.Providers, providers, typeof(RssProvider));
        provider = providers[section.DefaultProvider];

        if (provider == null)
        {
            ProviderException ex =
                new ProviderException("Unable to load default RssProvider");
            traceSource.TraceData(TraceEventType.Critical, 0, ex);
            throw ex;
        }

        IsInitialized = true;
        traceSource.TraceEvent(TraceEventType.Stop, 0,
                               "[RSS SERVICE] END Initialize");
    }
}
}
```

As you can see from the boldfaced portion of Listing 11-12, the `Initialize` method first instantiates the trace source as usual:

```
traceSource = new TraceSource("myTraceSource");
```

Next, it instantiates a `SourceSwitch` named `mySwitch`:

```
SourceSwitch mySwitch = new SourceSwitch("mySwitch");
```

Then, it sets the switch level to `SourceLevels.Error` to instruct the switch that you're only interested in tracing events of type `Critical` and `Error`:

```
mySwitch.Level = SourceLevels.Error;
```

Finally, it assigns the newly instantiated switch to the `Switch` property of the trace source:

```
traceSource.Switch = mySwitch;
```

Imperative instantiation and attaching of a shared switch is very similar to a local switch. You still have to instantiate your shared switch and set its `Level` property. The only difference is that you assign the same switch to the `Switch` property of two or more trace sources. In other words, two or more trace sources share that same switch.

Instantiating and Attaching an `IisTraceListener`

As discussed in the previous sections, you use the `TraceEvent`, `TraceData`, and `TraceTransfer` tracing methods of the `TraceSource` class to emit trace events from within your code. As the name implies, the `TraceSource` class is the trace event source, that is, it is the emission source of trace events. This raises the following question: Where does an emitted trace event go? It depends on the configured trace listener. As the name suggests, a trace listener is an object that listens for trace events. In other words, the `TraceSource` object emits the trace events and the configured trace listener catches these trace events. What the configured trace listener does with a trace event that it catches depends entirely on the implementation of the trace listener. For example, as you'll see later, the `IisTraceListener` trace listener routes a captured trace event to the IIS 7 tracing infrastructure.

All trace listeners directly or indirectly inherit from the `TraceListener` base class. As Listing 11-13 shows, the `TraceListener` base class exposes the same tracing methods that a trace source exposes. This is because the tracing methods of a trace source under the hood delegate to the associated methods of the configured trace listener.

Listing 11-13: The Tracing Methods of the `TraceListener` Base Class

```
public virtual void TraceData(TraceEventCache eventCache, string source,
                             TraceEventType eventType, int id, params object[] data);

public virtual void TraceData(TraceEventCache eventCache, string source,
                             TraceEventType eventType, int id, object data);

public virtual void TraceEvent(TraceEventCache eventCache, string source,
                              TraceEventType eventType, int id);

public virtual void TraceEvent(TraceEventCache eventCache, string source,
                              TraceEventType eventType, int id, string message);

public virtual void TraceEvent(TraceEventCache eventCache, string source,
```

(Continued)

Listing 11-13: (continued)

```
TraceEventType eventType, int id, string format, params object[] args);

public virtual void TraceTransfer(TraceEventCache eventCache, string source,
                                int id, string message, Guid relatedActivityId);
```

Note that the tracing methods of the `TraceListener` base class have the same signature as the corresponding tracing methods of the trace source, except for one notable difference. All tracing methods of the `TraceListener` base class take an instance of a class named `TraceEventCache` as their first argument. The associated trace source creates this instance and passes it into these methods as their first argument. These methods internally use this `TraceEventCache` object for performance optimization. In other words, all trace events are cached in this `TraceEventCache` object.

The tracing methods of a trace listener write the specified trace event to a specified output. The type of output depends on the type of trace listener. For example, the tracing methods of the `FileLogTraceListener` trace event listener write trace events to a specified file. We're interested in a trace listener named `IisTraceListener`. The tracing methods of this trace listener route the trace events to the IIS tracing infrastructure where it can be captured by trace event consumers, such as the Failed Request Tracing module.

In general, there are two types of trace listeners: local and shared. A shared trace listener attaches to more than one trace source. As such, a shared trace listener outputs traces coming from all the trace sources to which it is attached. Any changes in the settings of a shared trace listener will affect all the trace sources that use that trace listener. A local trace listener attaches to a single trace source and outputs only traces coming from that trace source.

There are two ways to instantiate and to attach trace listeners to trace sources: declaratively via the configuration file and imperatively via code. As discussed earlier, it is highly recommended that you do this declaratively from the configuration file.

The `<system.diagnostics>` section contains a subelement named `<sharedListeners>`. To add a new shared trace listener you must add a new `<add>` subelement to the `<sharedListeners>` element and sets its `name` and `type` attributes. The `name` attribute specifies the name of the shared trace listener and the `type` attribute specifies the complete information about the trace listener type. For example, the boldfaced portion of the following configuration fragment adds an `IisTraceListener` shared trace listener named `myListener`:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        ...
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="Error"/>
    </switches>
    <sharedListeners>
      <add name="myListener" type="System.Web.IisTraceListener">
```

```

    . . .
  </add>
</sharedListeners>
</system.diagnostics>
</configuration>

```

Adding a shared trace listener to the `<sharedListeners>` section does not mean that the trace sources specified in the configuration file will automatically use this shared trace listener. You must explicitly attach the shared trace listener to each trace source that you want to use the listener. The `<source>` element that represents a trace source in the configuration file contains a child element named `<listeners>`. To attach a shared listener to a trace source you must add an `<add>` child element to the `<listeners>` section of the `<source>` element that represents the trace source in the configuration file and set its name attribute to the value of the name attribute of the `<add>` element that adds the trace listener to the `<sharedListeners>` section. For example, the top boldfaced portion of the following configuration fragment attaches the shared listener defined in the bottom boldfaced portion:

```

<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        <listeners>
          <add name="myListener" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="Error" />
    </switches>
    <sharedListeners>
      <add name="myListener" type="System.Web.IisTraceListener">
      . . .
      </add>
    </sharedListeners>
  </system.diagnostics>
</configuration>

```

To attach a local trace listener to a trace source, you must add an `<add>` child element to the `<listeners>` child element of the `<source>` element that represents the trace source in the configuration file and set its name and type attributes. For example, the boldfaced portion of the following configuration fragment instantiates and attaches an `IisTraceListener` local trace listener named `myListener` to the trace source named `myTraceSource`:

```

<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        <listeners>
          <add name="myListener" type="System.Web.IisTraceListener">
          . . .
          </add>
        </listeners>
      </source>

```

```
</sources>
<switches>
  <add name="mySwitch" value="Error"/>
</switches>
</system.diagnostics>
</configuration>
```

So far, you've learned how to declaratively instantiate and attach a shared or local trace listener to a trace source from the configuration file. Next, I show you how to do this from code. Listing 11-14 presents a new version of the `RssService` that imperatively instantiates and attaches a local trace listener to your trace source.

Listing 11-14: Imperative Instantiation and Attaching of a Trace Listener

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
using Microsoft.Web.Administration;
using Rss.ImperativeManagement;
using System.Web.Hosting;
using System.Diagnostics;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
        private static RssProviderCollection providers = null;
        private static bool IsInitialized = false;

        private static TraceSource traceSource = null;
        public static TraceSource TraceSource
        {
            get
            {
                Initialize();
                return traceSource;
            }
        }

        public RssProvider Provider
        {
            get { Initialize(); return provider; }
        }

        public RssProviderCollection Providers
        {
            get { Initialize(); return providers; }
        }

        public static void LoadRss(Stream stream)
        {

```

Listing 11-14: *(continued)*

```

        Initialize();
        Channel channel = new Channel();
        channel.Title = channelTitle;
        channel.Link = channelLink;
        channel.Description = channelDescription;

        provider.LoadRss(channel, stream);
    }

    private static string channelTitle;
    private static string channelDescription;
    private static string channelLink;

    private static void Initialize()
    {
        if (!IsInitialized)
        {
            traceSource = new TraceSource("myTraceSource");
            SourceSwitch mySwitch = new SourceSwitch("mySwitch");
            mySwitch.Level = SourceLevels.Error;
            traceSource.Switch = mySwitch;

            IisTraceListener myListener = new IisTraceListener();
            myListener.Name = "myListener";
            traceSource.Listeners.Add(myListener);

            traceSource.TraceEvent(TraceEventType.Start, 0,
                                   "[RSS SERVICE] START Initialize");

            ServerManager mgr = new ServerManager();
            Configuration config =
                mgr.GetWebConfiguration(HostingEnvironment.SiteName,
                                         HttpContext.Current.Request.ApplicationPath);

            RssSection section =
                (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
            channelDescription = section.ChannelDescription;
            channelLink = section.ChannelLink;
            channelTitle = section.ChannelTitle;
            traceSource.TraceInformation(
                "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
                channelDescription, channelLink, channelTitle);

            providers = new RssProviderCollection();
            ProvidersHelper.InstantiateProviders
                (section.Providers, providers, typeof(RssProvider));
            provider = providers[section.DefaultProvider];

            if (provider == null)
            {
                ProviderException ex =
                    new ProviderException("Unable to load default RssProvider");
            }
        }
    }

```

(Continued)

Listing 11-14: *(continued)*

```
        traceSource.TraceData(TraceEventType.Critical, 0, ex);
        throw ex;
    }

    IsInitialized = true;
    traceSource.TraceEvent(TraceEventType.Stop, 0,
                          "[RSS SERVICE] END Initialize");
    }
}
}
```

As you can see from the boldfaced portion of Listing 11-14, `Initialize` first instantiates an `IisTraceListener`:

```
IisTraceListener myListener = new IisTraceListener();
```

It names the trace listener "myListener":

```
myListener.Name = "myListener";
```

It adds the trace listener to the `Listeners` collection of the trace source to attach the trace listener to the trace source:

```
traceSource.Listeners.Add(myListener);
```

Instantiating and attaching a shared trace listener to a trace source is very similar to a local trace listener. You still have to instantiate the trace listener and set its `Name` property. The only difference is that you add this trace listener to the `Listeners` collection property of more than one trace source.

Instantiating and Attaching a Trace Filter

Following the recipe, the next order of business is to instantiate and attach a trace filter to the trace listener. As discussed earlier, a trace filter filters the events that its associated trace listener traces. All trace filters directly or indirectly inherit from a base class named `TraceFilter`. Listing 11-15 presents the implementation of this base class. This base class defines the API that the `TraceListener` base class uses to interact with the configured trace filter in a generic fashion without knowing its real type. This allows the same trace listener to interact with different types of trace filters.

Listing 11-15: The `TraceFilter` Base Class

```
public abstract class TraceFilter
{
    internal string initializeData;

    internal bool ShouldTrace(TraceEventCache cache, string source,
                             TraceEventType eventType, int id,
                             string formatOrMessage)
```

Listing 11-15: *(continued)*

```
{
    return this.ShouldTrace(cache, source, eventType, id, formatOrMessage,
                           null, null, null);
}

internal bool ShouldTrace(TraceEventCache cache, string source,
                          TraceEventType eventType, int id,
                          string formatOrMessage, object[] args)
{
    return this.ShouldTrace(cache, source, eventType, id, formatOrMessage,
                           args, null, null);
}

internal bool ShouldTrace(TraceEventCache cache, string source,
                          TraceEventType eventType, int id,
                          string formatOrMessage, object[] args, object data1)
{
    return this.ShouldTrace(cache, source, eventType, id, formatOrMessage,
                           args, data1, null);
}

public abstract bool ShouldTrace(TraceEventCache cache, string source,
                                TraceEventType eventType, int id,
                                string formatOrMessage, object[] args,
                                object data1, object[] data);
}
```

As you can see, the `TraceFilter` base class exposes four overloads of the `ShouldTrace` method. Note that the first three overloads are marked as internal, whereas the last overload is marked as public. As you'll see later in this chapter, the tracing methods of the `TraceListener` base class use the internal overloads. Also note that the internal overloads of the `ShouldTrace` method delegate to the public overload of this method, which is marked as abstract. Every trace filter must implement this abstract method to include the necessary logic to determine whether the trace event with the specified event type, numeric identifier, format string or message, array of objects to be formatted, and attached data objects should be traced.

The .NET Framework comes with two standard implementations of the `TraceFilter` base class: `EventFilter` and `SourceFilter`, as discussed in the following sections.

EventFilter

The `EventFilter` filters trace events based on their event types. Listing 11-16 presents the implementation of the `EventFilter` class.

Listing 11-16: The EventFilter Class

```
public class EventFilter : TraceFilter
{
    private SourceLevels level;
```

(Continued)

Listing 11-16: (continued)

```
public EventTypeFilter(SourceLevels level)
{
    this.level = level;
}

public override bool ShouldTrace(TraceEventCache cache, string source,
                                TraceEventType eventType, int id,
                                string formatOrMessage, object[] args,
                                object data1, object[] data)
{
    return ((eventType & ((TraceEventType) ((int) this.level))) !=
            ((TraceEventType) 0));
}

public SourceLevels EventType
{
    get { return this.level; }
    set { this.level = value; }
}
}
```

The constructor of the `EventTypeFilter` takes an argument of type `SourceLevels` enumeration and stores it in a private field. Note that the `EventType` read/write property gets or sets the value of this private field. In other words, you can use the `EventType` property to imperatively change the source level of an `EventTypeFilter` filter after you instantiate the filter.

The `EventTypeFilter` class, like any other trace filter, inherits from the `TraceFilter` base class and implements its `ShouldTrace` abstract method. As Listing 11-16 shows, the `EventTypeFilter` class's implementation of the `ShouldTrace` method simply performs a bitwise AND operation between the event type of the specified trace event and the source level of the filter. If the current source level supports the specified event type, the `ShouldTrace` method returns true to signal its associated trace listener that the specified trace event should be traced.

Next, I show you how to instantiate and attach an `EventTypeFilter` filter to a trace listener. In general, you can do this either from code or the configuration file. As discussed earlier, it is highly recommended that you do this from the configuration file.

To instantiate and attach an `EventTypeFilter` filter to a local trace listener, you must add a `<filter>` child element to the `<add>` element that represents the local trace listener in the `<listeners>` subelement of the `<source>` element that represents the associated trace source. You must also set the `type` attribute on this `<filter>` child element to `"System.Diagnostics.EventTypeFilter"` to specify that you want the trace listener to use an `EventTypeFilter` to filter which events to trace. You must also set the `initializeData` attribute on this `<filter>` child element to a `SourceLevels` enumeration value to specify which events the trace listener should trace.

For example, the boldfaced portion of the following configuration fragment attaches an `EventTypeFilter` trace filter to the `IisTraceListener` and sets its `initializeData` attribute to

Error to specify that `IisTraceListener` should only trace events of type `Critical` and `Error` and should ignore all other types of events:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        <listeners>
          <add name="myListener" type="System.Web.IisTraceListener">
            <filter type="System.Diagnostics.EventTypeFilter"
              initializeData="Error"/>
          </add>
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="All"/>
    </switches>
  </system.diagnostics>
</configuration>
```

Note that this configuration fragment attaches a switch with a source level value of `All` to the trace source. This means that the trace source traces all types of events. However, thanks to the `EventTypeFilter` trace filter, the `IisTraceListener` only routes events of type `Critical` and `Error` to the IIS 7 tracing infrastructure. In other words, IIS 7 modules such as Failed Request Tracing will only see `Critical` and `Error` type events.

To attach an `EventTypeFilter` filter to a shared trace listener, add a `<filter>` child element to the `<add>` child element that represents the shared trace listener in the `<sharedListeners>` section. You must also set the `type` and `initializeData` attributes of the `<filter>` element as discussed earlier. The bold faced portion of the following configuration fragment attaches an `EventTypeFilter` filter to the `IisTraceListener` shared listener:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        <listeners>
          <add name="myListener" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="All"/>
    </switches>
    <sharedListeners>
      <add name="myListener" type="System.Web.IisTraceListener">
        <filter type="System.Diagnostics.EventTypeFilter" initializeData="Error"/>
      </add>
    </sharedListeners>
  </system.diagnostics>
</configuration>
```


Chapter 11: Integrated Tracing and Diagnostics

Next, I show you how to instantiate and attach an `EventFilter` filter to a trace listener from code. Listing 11-17 presents a new version of `RssService` that instantiates and attaches an `EventFilter` filter to the `IisTraceListener`.

Listing 11-17: Imperative Instantiation and Attachment of an `EventFilter` filter

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
using Microsoft.Web.Administration;
using Rss.ImperativeManagement;
using System.Web.Hosting;
using System.Diagnostics;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
        private static RssProviderCollection providers = null;
        private static bool IsInitialized = false;

        private static TraceSource traceSource = null;
        public static TraceSource TraceSource
        {
            get
            {
                Initialize();
                return traceSource;
            }
        }

        public RssProvider Provider
        {
            get { Initialize(); return provider; }
        }

        public RssProviderCollection Providers
        {
            get { Initialize(); return providers; }
        }

        public static void LoadRss(Stream stream)
        {
            Initialize();
            Channel channel = new Channel();
            channel.Title = channelTitle;
            channel.Link = channelLink;
            channel.Description = channelDescription;

            provider.LoadRss(channel, stream);
        }
    }
}
```

Listing 11-17: *(continued)*

```

private static string channelTitle;
private static string channelDescription;
private static string channelLink;

private static void Initialize()
{
    if (!IsInitialized)
    {
        traceSource = new TraceSource("myTraceSource");
        SourceSwitch mySwitch = new SourceSwitch("mySwitch");
        mySwitch.Level = SourceLevels.Error;
        traceSource.Switch = mySwitch;

        IisTraceListener myListener = new IisTraceListener();
myListener.Name = "myListener";
EventTypeFilter myFilter = new EventTypeFilter(SourceLevels.Error);
myListener.Filter = myFilter;
traceSource.Listeners.Add(myListener);

        traceSource.TraceEvent(TraceEventType.Start, 0,
                                "[RSS SERVICE] START Initialize");

        ServerManager mgr = new ServerManager();
        Configuration config =
            mgr.GetWebConfiguration(HostingEnvironment.SiteName,
                                    HttpContext.Current.Request.ApplicationPath);

        RssSection section =
            (RssSection)config.GetSection("system.webServer/rss", typeof(RssSection));
        channelDescription = section.ChannelDescription;
        channelLink = section.ChannelLink;
        channelTitle = section.ChannelTitle;
        traceSource.TraceInformation(
            "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
            channelDescription, channelLink, channelTitle);

        providers = new RssProviderCollection();
        ProvidersHelper.InstantiateProviders
            (section.Providers, providers, typeof(RssProvider));
        provider = providers[section.DefaultProvider];

        if (provider == null)
        {
            ProviderException ex =
                new ProviderException("Unable to load default RssProvider");
            traceSource.TraceData(TraceEventType.Critical, 0, ex);
            throw ex;
        }

        IsInitialized = true;
        traceSource.TraceEvent(TraceEventType.Stop, 0,
                                "[RSS SERVICE] END Initialize");
    }
}

```

(Continued)

Listing 11-17: *(continued)*

```
    }  
  }  
}
```

As the boldfaced portion of this code listing shows, the `Initialize` method first instantiates an `EventTypeFilter`, passing in `SourceLevels.Error` as the source level:

```
EventTypeFilter myFilter = new EventTypeFilter(SourceLevels.Error);
```

Then, it assigns this `EventTypeFilter` filter to the `Filter` property of the trace listener:

```
myListener.Filter = myFilter;
```

SourceFilter

The `SourceFilter` filter uses the source of the specified trace event to determine whether the trace event should be traced. Listing 11-18 presents the implementation of the `SourceFilter` filter.

Listing 11-18: The SourceFilter Filter

```
public class SourceFilter : TraceFilter  
{  
    private string src;  
  
    public SourceFilter(string source)  
    {  
        this.Source = source;  
    }  
  
    public override bool ShouldTrace(TraceEventCache cache, string source,  
                                     TraceEventType eventType, int id,  
                                     string formatOrMessage, object[] args,  
                                     object data1, object[] data)  
    {  
        if (source == null)  
            throw new ArgumentNullException("source");  
  
        return string.Equals(this.src, source);  
    }  
  
    public string Source  
    {  
        get { return this.src; }  
  
        set  
        {  
            if (value == null)  
                throw new ArgumentNullException("source");  
            this.src = value;  
        }  
    }  
}
```

Note that the constructor of the `SourceFilter` filter takes a string argument that contains the name of a trace source and stores this argument in a private field. The `Source` read/write string property of the `SourceFilter` filter simply gets and sets the value of this field. This means that you can use the `Source` property to imperatively change the associated trace source of a `SourceFilter` filter after you instantiate the filter. Notice that the setter of the `Source` property raises an exception if the value being assigned to the property is null.

The `SourceFilter` filter, like any other trace event filter, inherits from the `TraceFilter` base class and implements its `ShouldTrace` abstract method. As Listing 11-18 shows, this method simply checks whether the source of the specified trace event is the same source associated with the filter. If so, it returns true. Otherwise it returns false. Note that the method raises an exception if the source name passed to the constructor was null.

One common scenario where the `SourceFilter` filter comes in handy is when several trace sources share the same trace listener. In this case this trace listener is an `IisTraceListener`. This means that `IisTraceListener` routes events coming from several trace sources to the IIS 7 tracing infrastructure. This ends up cluttering the output trace file, making it hard to make sense of the content of the file. In these situations you can temporarily attach a `SourceFilter` filter to `IisTraceListener` to have the trace listener only route events coming from a particular trace source. The following listing shows how you can do this from the configuration file:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource1" switchName="mySwitch">
        <listeners>
          <add name="myListener" />
        </listeners>
      </source>
      <source name="myTraceSource2" switchName="mySwitch">
        <listeners>
          <add name="myListener" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="mySwitch" value="All"/>
    </switches>
    <sharedListeners>
      <add name="myListener" type="System.Web.IisTraceListener">
        <filter type="System.Diagnostics.SourceFilter"
          initializeData="myTraceSource2"/>
      </add>
    </sharedListeners>
  </system.diagnostics>
</configuration>
```

Note that the `<sources>` element contains two `<source>` elements that represent two trace sources named `myTraceSource1` and `myTraceSource2`, which share an `IisTraceListener` trace listener named `myListener`. Also note that you have attached a `SourceFilter` trace filter to this shared trace listener and set its `initializeData` attribute to `myTraceSource2`, which is the name of the second trace source. This means that your `SourceFilter` trace filter will instruct the shared trace listener to route

Chapter 11: Integrated Tracing and Diagnostics

only traces originating from the `myTraceSource2` trace source to the IIS 7 tracing infrastructure ignoring the traces coming from the `myTraceSource1` trace source.

Putting It All Together

Open the RSS solution that you created in Chapter 10 in Visual Studio. Recall that this solution contains a Class Library project named `Rss`. Now take these steps:

1. Replace the content of the `RssService.cs` file in the `Base` directory of the `Rss` project with the code shown in Listing 11-3.
2. Add a new Web application named `RssWebApp` to this solution.
3. Add an empty text file named `MyFile.rss` to this Web application.
4. Add the boldfaced portion of Listing 11-19 to the `web.config` file of this Web application.
5. Add the boldfaced portions of Listing 11-20 to the `applicationHost.config` file. Don't forget to replace the value of the `PublicKeyToken` attribute with the actual public key token of the assembly. Chapter 7 showed you how to access the public key token of an assembly.
6. Add the boldfaced portion of the following XML fragment to the `<connectionStrings>` section of the `machine.config` file:

```
<connectionStrings>
  <add name="MyXmlFile" connectionString="App_Data/Articles.xml" />
</connectionStrings>
```

7. Add an empty XML file named `Articles.xml` to this Web application and add the content of Listing 11-21 to this file.
8. Right-click the `Rss` project node in the Solution Explorer and select Properties.
9. Select the Build tab from the Properties page and make sure the `Define TRACE` constant is selected.

Listing 11-19: The web.config File

```
<configuration>
  <system.web>
    <compilation debug="true"/>
  </system.web>
  <system.diagnostics>
    <sources>
      <source name="myTraceSource" switchName="mySwitch">
        <listeners>
          <add name="myListener" type="System.Web.IisTraceListener, System.Web,
            Version=2.0.0.0, Culture=Neutral, PublicKeyToken=b03f5f7f11d50a3a">
            <filter type="System.Diagnostics.EventTypeFilter"
              initializeData="All"/>
          </add>
        </listeners>
      </source>
    </sources>
```

Listing 11-19: *(continued)*

```

    <switches>
      <add name="mySwitch" value="All"/>
    </switches>
  </system.diagnostics>
  <system.webServer>
    <defaultDocument enabled="true">
      <files>
        <add value="MyFile.rss"/>
      </files>
    </defaultDocument>
  </system.webServer>
</configuration>

```

Listing 11-20: The applicationHost.config File

```

<configuration>
  . . .
  <location path="" overrideMode="Allow">
    <system.webServer>
      <handlers accessPolicy="Read, Script">

        <add name="RssHandler" path="*.rss" verb="*" preCondition="integratedMode"
          type="Rss.Base.RssHandler, Rss, Version=1.0.0.0, Culture=Neutral,
            PublicKeyToken=3c8f2b8aeeec5395"/>

        . . .
      </handlers>
      . . .
      <rss enabled="true" defaultProvider="XmlRssProvider"
        channelTitle="Free articles from Articles.com site"
        channelDescription="This site is dedicated to ASP.NET technologies"
        channelLink="http://articles.com">
        <providers>
          <add name="XmlRssProvider" connectionStringName="MyXmlFile"
            item="//Article"
            itemLinkFormatString="http://articles.com/{0}" itemTitle="@title"
            itemDescription="Abstract/text()" itemLink="@link"
            type="Rss.Base.XmlRssProvider, Rss, Version=1.0.0.0, Culture=Neutral,
              PublicKeyToken=3c8f2b8aeeec5395"/>
        </providers>
      </rss>
      . . .
    </system.webServer>
  </location>
  . . .
</configuration>

```

Listing 11-21: The Articles.xml File

```

<?xml version="1.0" encoding="utf-8" ?>
<Articles>

```

(Continued)

Chapter 11: Integrated Tracing and Diagnostics

Listing 11-21: (continued)

```
<Article title="What's new in ASP.NET?" link="Smith1.aspx">
  <Abstract>Describes the new ASP.NET features</Abstract>
</Article>
<Article title="XSLT in ASP.NET Applications" link="Carey.aspx">
  <Abstract>Shows to use XSLT in your ASP.NET applications</Abstract>
</Article>
<Article title="XML programming" link="Smith2.aspx">
  <Abstract>Reviews .NET 2.0 XML programming features</Abstract>
</Article>
</Articles>
```

Build the Rss project. Then access the `MyFile.rss` page of the `RssWebApp` Web application from your browser. You should see the RSS document that the provider-based RSS service generates. You may be wondering what happened to the traces you were trying to route to the IIS 7 tracing infrastructure. You're not done yet. There are still a few more steps that you have to take, as follows. Routing trace events to the IIS 7 infrastructure is just half the story. Next, you need to configure an IIS 7 module named Failed Request Tracing to capture these trace events and log them in a log file. This configuration involves two important steps. Here is the first step: Launch the IIS 7 Manager and select the Default Web Site node from the Connections pane, as shown in Figure 11-1.



Figure 11-1

Now click the Failed Request Tracing link button in the Actions pane to launch the Edit Web Site Failed Request Tracing Settings dialog shown in Figure 11-2.



Figure 11-2

Check the Enable checkbox shown in Figure 11-2. Note that this dialog allows you to specify the location of the trace output file and the maximum number of trace files that can be stored in this location. Click OK to enable the Failed Request Tracing. As mentioned earlier, it takes two steps to configure the Failed Request Tracing module. So far we've covered the first step. The second step requires you to specify a failed request tracing rule. As the name implies, a failed request tracing rule instructs the Failed Request Tracing module to log the trace events of those requests that meet the criteria specified by the failed request tracing rule. The Failed Request Tracing module does not log trace events of those requests that meet none of the criteria specified by the failed request tracing rules. Keep in mind that you can define multiple failed request tracing rules. Next, I show you how to define a failed request tracing rule. Now go back to the IIS 7 Manager and select the RssWebSite node in the Connections pane as shown in Figure 11-3.



Figure 11-3

Double-click the Failed Request Tracing Rules icon in the workspace in Figure 11-3 to navigate to the Failed Request Tracing Rules page shown in Figure 11-4.



Figure 11-4

Click the Add link button in the Actions pane in Figure 11-4 to launch the Add Failed Request Tracing Rule Wizard shown in Figure 11-5. This wizard allows you to add a new failed request tracing rule.

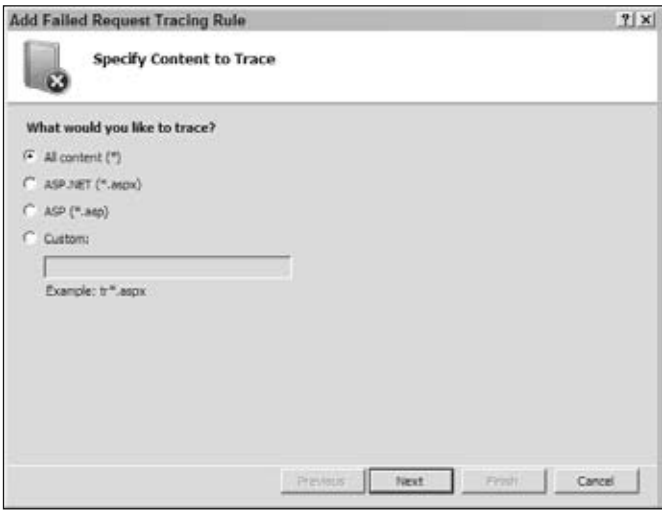


Figure 11-5

Select the All content (*) radio button, as shown in Figure 11-5, and click the Next button to move to the Define Trace Conditions step shown in Figure 11-6.

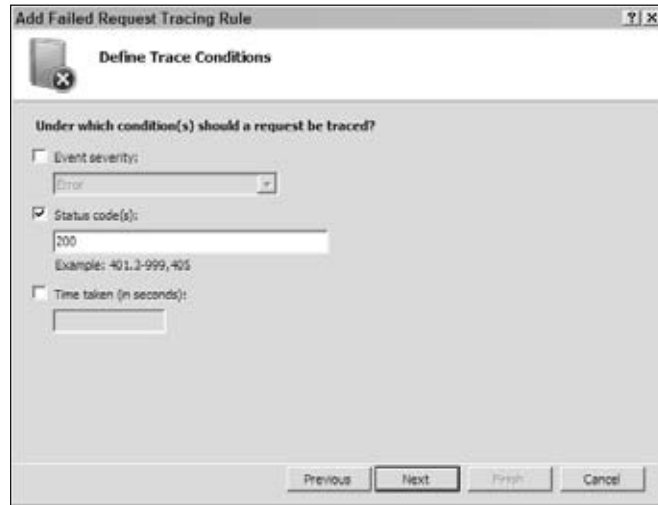


Figure 11-6

Select the Status code(s) checkbox, and enter **200** for the status code. Then click the Next button to move on to the Select Trace Providers step shown in Figure 11-7.

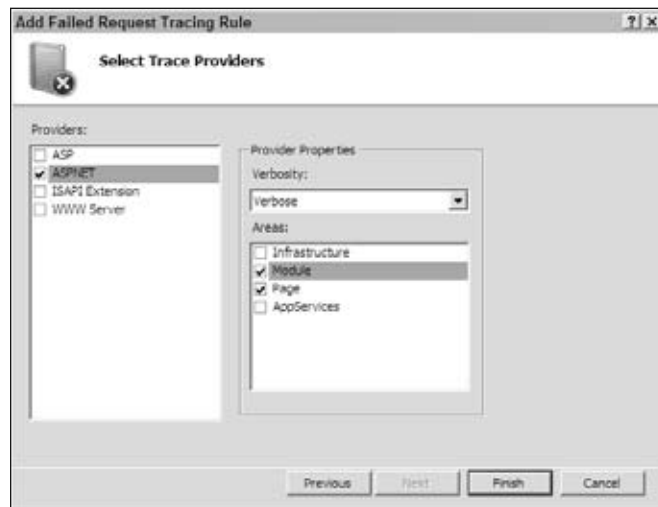


Figure 11-7

Uncheck all the providers in the left pane except the ASP.NET provider. Select the Verbose option from the Verbosity combo box. Select the Module and Page toggles in the Areas section. Click the Finish button to exit the wizard.

Chapter 11: Integrated Tracing and Diagnostics

Now access the `MyFile.rss` page of the `RssWebSite` from your browser. Go to the directory where the trace output file is added. Open the file. You should see the result shown in Listing 11-22. As you can see, this file contains the three trace events emitted by the `Initialize` method of the `RssService`.

Listing 11-22: The Trace Output File

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type='text/xsl' href='freb.xsl'?>
<failedRequest url="http://localhost:80/RssWebSite2/"
  siteId="1"
  appPoolId="DefaultAppPool"
  processId="248"
  verb="GET"
  remoteUserName=" "
  userName=" "
  tokenUserName="NT AUTHORITY\IUSR"
  authenticationType="anonymous"
  activityId="{00000000-0000-0000-2D00-0080000000FA}"
  failureReason="STATUS_CODE"
  statusCode="200"
  triggerStatusCode="200"
  timeTaken="202"
  xmlns:freb="http://schemas.microsoft.com/win/2006/06/iis/freb">

<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="ASPNET" Guid="{AFF081FE-0247-4275-9C4E-021F3DC1DA35}"/>
    <EventID>0</EventID>
    <Version>0</Version>
    <Level>0</Level>
    <Opcode>66</Opcode>
    <Keywords>0x2</Keywords>
    <TimeCreated SystemTime="2007-06-13T08:31:10.589Z"/>
    <Correlation ActivityID="{00000000-0000-0000-2D00-0080000000FA}"/>
    <Execution ProcessID="248" ThreadID="3920"/>
    <Computer>LH-WGPSJIEKZFID</Computer>
  </System>
  <EventData>
    <Data Name="ContextId">{00000000-0000-0000-2D00-0080000000FA}</Data>
    <Data Name="Uri">/RssWebSite/MyFile.rss</Data>
    <Data Name="eventData">[RSS SERVICE] START Initialize</Data>
  </EventData>
  <RenderingInfo Culture="en-US">
    <Opcode>AspNetModuleDiagStartEvent</Opcode>
    <Keywords>
      <Keyword>Module</Keyword>
    </Keywords>
  </RenderingInfo>
  <ExtendedTracingInfo
    xmlns="http://schemas.microsoft.com/win/2004/08/events/trace">
    <EventGuid>{06A01367-79D3-4594-8EB3-C721603C4679}</EventGuid>
  </ExtendedTracingInfo>
</Event>
```

Listing 11-22: (continued)

```

<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="ASPNET" Guid="{AFF081FE-0247-4275-9C4E-021F3DC1DA35}" />
    <EventID>0</EventID>
    <Version>0</Version>
    <Level>4</Level>
    <Opcode>64</Opcode>
    <Keywords>0x2</Keywords>
    <TimeCreated SystemTime="2007-06-13T08:31:10.621Z" />
    <Correlation ActivityID="{00000000-0000-0000-2D00-0080000000FA}" />
    <Execution ProcessID="248" ThreadID="3920" />
    <Computer>LH-WGPSJIEKZFID</Computer>
  </System>
  <EventData>
    <Data Name="ContextId">{00000000-0000-0000-2D00-0080000000FA}</Data>
    <Data Name="Uri">/RssWebSite2/MyFile.rss</Data>
    <Data Name="eventData">
      Channel Description: This site is dedicated to ASP.NET
      Channel Link: http://articles.com
      Channel Title: Free articles from Articles.com site
    </Data>
  </EventData>
  <RenderingInfo Culture="en-US">
    <Opcode>AspNetModuleDiagInfoEvent</Opcode>
    <Keywords>
      <Keyword>Module</Keyword>
    </Keywords>
  </RenderingInfo>
  <ExtendedTracingInfo
    xmlns="http://schemas.microsoft.com/win/2004/08/events/trace">
    <EventGuid>{06A01367-79D3-4594-8EB3-C721603C4679}</EventGuid>
  </ExtendedTracingInfo>
</Event>

<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="ASPNET" Guid="{AFF081FE-0247-4275-9C4E-021F3DC1DA35}" />
    <EventID>0</EventID>
    <Version>0</Version>
    <Level>0</Level>
    <Opcode>67</Opcode>
    <Keywords>0x2</Keywords>
    <TimeCreated SystemTime="2007-06-13T08:31:10.636Z" />
    <Correlation ActivityID="{00000000-0000-0000-2D00-0080000000FA}" />
    <Execution ProcessID="248" ThreadID="3920" />
    <Computer>LH-WGPSJIEKZFID</Computer>
  </System>
  <EventData>
    <Data Name="ContextId">{00000000-0000-0000-2D00-0080000000FA}</Data>
    <Data Name="Uri">/RssWebSite2/MyFile.rss</Data>
    <Data Name="eventData">[RSS SERVICE] END Initialize</Data>
  </EventData>

```

(Continued)

Listing 11-22: (continued)

```
<RenderingInfo Culture="en-US">
  <Opcode>AspNetModuleDiagStopEvent</Opcode>
  <Keywords>
    <Keyword>Module</Keyword>
  </Keywords>
</RenderingInfo>
<ExtendedTracingInfo
  xmlns="http://schemas.microsoft.com/win/2004/08/events/trace">
  <EventGuid>{06A01367-79D3-4594-8EB3-C721603C4679}</EventGuid>
</ExtendedTracingInfo>
</Event>
</failedRequest>
```

Configurable Tracing

The current implementation of the `RssService` class hard-codes the trace source's name, as you can see from the following excerpt from Listing 11-7:

```
traceSource = new TraceSource("myTraceSource");
```

This introduces two problems. First of all, the trace source name is hard-coded. Recall that the page developer must use the same trace name in the configuration file as it is used in the code. This means that the page developer cannot configure the trace source without knowing its hard-coded name. In addition, the page developer won't be able to use a name of her choosing. Second of all, it does not allow the page developer to use an external trace source to trace the `RssService`'s trace events. To fix these problems, extend your `<rss>` configuration system to add support for two attributes named `traceSource` and `isExternalTraceSource`. The page developer can then set the `isExternalTraceSource` attribute to `true` and assign the fully qualified name of a type to the `traceSource` attribute. This type must expose a static property of type `TraceSource` named `TraceSource` that returns a reference to the desired trace source. If the page developer wants the `RssService` to use its own internal trace source, she must set the `isExternalTraceSource` attribute to `false` and assign the desired trace source name to `traceSource` attribute.

Therefore, the first order of business is to modify the `RSS_Schema.xml` file to add support for the `traceSource` and `isExternalTraceSource` attributes on the `<rss>` configuration section, as shown in boldfaced portion of Listing 11-23.

Listing 11-23: The `RSS_Schema.xml` File

```
<configSchema>
  <sectionSchema name="system.webServer/rss">
    <attribute name="enabled" type="bool" defaultValue="true"/>
    <attribute name="channelTitle" type="string" defaultValue="Unknown"/>
    <attribute name="channelDescription" type="string" defaultValue="Unknown"/>
    <attribute name="channelLink" type="string" defaultValue="Unknown"/>
    <attribute name="traceSource" type="string" defaultValue="Unknown"/>
    <attribute name="isExternalTraceSource" type="bool" defaultValue="false"/>
  </sectionSchema>
</configSchema>
```

Listing 11-23: (continued)

```
<attribute name="defaultProvider" type="string"
validationType="requireTrimmedString" defaultValue="SqlRssProvider" />
<element name="providers">
  <collection addElement="add" removeElement="remove" clearElement="clear"
allowUnrecognizedAttributes="true">
    <attribute name="name" required="true" isUniqueKey="true" type="string" />
    <attribute name="type" required="true" type="string" />
  </collection>
</element>
</sectionSchema>
</configSchema>
```

Next, you need to edit the `RssSection` imperative management class to add support for two new properties named `TraceSource` and `IsExternalTraceSource` that provide imperative access to the `traceSource` and `isExternalTraceSource` attributes on the `<rss>` configuration section as shown in Listing 11-24.

Listing 11-24: The `RssSection` Class

```
using System;
using Microsoft.Web.Administration;
using IIS7AndAspNet2IntegratedProvidersModel.ImperativeManagement;

namespace Rss.Base
{
    public class RssSection : ConfigurationSection
    {
        static RssSection()
        {
            RssSection.ProvidersAttribute = "providers";
            RssSection.DefaultProviderAttribute = "defaultProvider";
            RssSection.EnabledAttribute = "enabled";
            RssSection.ChannelTitleAttribute = "channelTitle";
            RssSection.ChannelDescriptionAttribute = "channelDescription";
            RssSection.ChannelLinkAttribute = "channelLink";
            RssSection.TraceSourceAttribute = "traceSource";
            RssSection.IsExternalTraceSourceAttribute = "isExternalTraceSource";
        }

        . . .

        public string TraceSource
        {
            get
            {
                return (string)base[RssSection.TraceSourceAttribute];
            }
            set
            {
                base[RssSection.TraceSourceAttribute] = value;
            }
        }
    }
}
```

(Continued)

Listing 11-24: (continued)

```
    }

    public bool IsExternalTraceSource
    {
        get
        {
            return (bool)base[RssSection.IsExternalTraceSourceAttribute];
        }
        set
        {
            base[RssSection.IsExternalTraceSourceAttribute] = value;
        }
    }

    private ProviderSettingsCollection _providers;
    private static readonly string DefaultProviderAttribute;
    private static readonly string EnabledAttribute;
    private static readonly string ProvidersAttribute;
    private static readonly string ChannelTitleAttribute;
    private static readonly string ChannelDescriptionAttribute;
    private static readonly string ChannelLinkAttribute;
    private static readonly string TraceSourceAttribute;
    private static readonly string IsExternalTraceSourceAttribute;
}
}
```

Next, you need to modify the implementation of the `RssService` class as shown in the boldfaced portion of Listing 11-25.

Listing 11-25: The `RssService` Class

```
using System;
using System.Configuration.Provider;
using System.Web;
using System.IO;
using Microsoft.Web.Administration;
using IIS7AndAspNet2IntegratedProvidersModel.ImperativeManagement;
using System.Diagnostics;
using System.Text;
using System.Web.Compilation;
using System.Reflection;

namespace Rss.Base
{
    public class RssService
    {
        private static RssProvider provider = null;
        private static RssProviderCollection providers = null;
        private static bool IsInitialized = false;

        private static TraceSource traceSource = null;
    }
}
```

Listing 11-25: *(continued)*

```

public static TraceSource TraceSource
{
    get
    {
        Initialize();
        return traceSource;
    }
}

private static string channelTitle;
private static string channelDescription;
private static string channelLink;

private static void Initialize()
{
    if (!IsInitialized)
    {
        ServerManager mgr = new ServerManager();
        Configuration config = mgr.GetApplicationHostConfiguration();
        RssSection section = (RssSection)config.GetSection(
            "system.webServer/rss", typeof(RssSection));
        channelDescription = section.ChannelDescription;
        channelLink = section.ChannelLink;
        channelTitle = section.ChannelTitle;

        if (section.IsExternalTraceSource)
        {
            Type c = BuildManager.GetType(section.TraceSource, true, true);
            PropertyInfo pi = c.GetProperty("TraceSource");
            traceSource = (TraceSource)pi.GetValue(null, null);
        }
        else
            traceSource = new TraceSource(section.TraceSource);

        traceSource.TraceInformation(
            "Channel Description: {0}\nChannel Link: {1}\nChannel Title: {2}\n",
            channelDescription, channelLink, channelTitle);

        providers = new RssProviderCollection();
        ProvidersHelper.InstantiateProviders
            (section.Providers, providers, typeof(RssProvider));
        provider = providers[section.DefaultProvider];

        if (provider == null)
        {
            ProviderException ex =
                new ProviderException("Unable to load default RssProvider");
            traceSource.TraceData(TraceEventType.Critical, 0, ex);
            throw ex;
        }
        IsInitialized = true;
        traceSource.TraceEvent(TraceEventType.Stop, 0,

```

(Continued)

Listing 11-25: (continued)

```
        "[RSS SERVICE] END Initialize");
    }
}
}
```

As you can see, the `Initialize` method first checks whether the `isExternalTraceSource` attribute on the `<rss>` configuration section is set to `true`. If so, it takes these steps to access a reference to the external trace source:

1. Invokes the `GetType` static method on the `BuildManager` class, passing in the value of the `traceSource` attribute on the `<rss>` configuration section to return a reference to the `Type` object that represents the type that contains the external trace source:

```
Type c = BuildManager.GetType(section.TraceSource, true, true);
```

2. Calls the `GetProperty` method on this `Type` object to return a reference to a `PropertyInfo` object that represents the `TraceSource` property:

```
PropertyInfo pi = c.GetProperty("TraceSource");
```

3. Calls the `GetValue` method on this `PropertyInfo` object to return a reference to the external trace source and assigns this reference to the `traceSource` static field:

```
traceSource = (TraceSource)pi.GetValue(null, null);
```

If the `isExternalTraceSource` attribute on the `<rss>` configuration section is set to `false`, the `Initialize` method calls the constructor of the `TraceSource` class, passing in the value of the `traceSource` attribute to instantiate a new trace source with the specified name:

```
traceSource = new TraceSource(section.TraceSource);
```

Next, you need to modify the implementation of the `RssPage` module page in order to add UI support for configuring your trace settings as shown in Listing 11-26.

Listing 11-26: The `RssPage` Module Page

```
namespace Rss.Client
{
    class RssPage : ModuleDialogPage
    {
        . . .

        private Label channelTitleLabel;
        private TextBox channelTitleTextBox;
        private Label channelDescriptionLabel;
        private TextBox channelDescriptionTextBox;
        private Label channelLinkLabel;
        private TextBox channelLinkTextBox;
        private Label traceSourceLabel;
```

Listing 11-26: *(continued)*

```

private TextBox traceSourceTextBox;
private CheckBox isExternalTraceSourceCheckBox;

. . .

private void OnTraceSourceTextBoxTextChanged(object sender, EventArgs e)
{
    this.UpdateUIState();
}

private void OnIsExternalTraceSourceCheckBoxCheckedChanged(object sender,
    EventArgs e)
{
    this.UpdateUIState();
}

private void InitializeUI()
{
    if (localInfo == null)
        return;

    ClearChannelSettings();
    this.channelTitleTextBox.Text = localInfo.ChannelTitle;
    this.channelDescriptionTextBox.Text = localInfo.ChannelDescription;
    this.channelLinkTextBox.Text = localInfo.ChannelLink;
    this.traceSourceTextBox.Text = localInfo.TraceSource;
    this.isExternalTraceSourceCheckBox.Checked = localInfo.IsExternalTraceSource;
}

private void ClearChannelSettings()
{
    this.channelTitleTextBox.Clear();
    this.channelDescriptionTextBox.Clear();
    this.channelLinkTextBox.Clear();
    this.traceSourceTextBox.Clear();
}

private void GetChannelValues()
{
    this.clone = this.bag.Clone();
    this.clone[0] = this.channelTitleTextBox.Text;
    this.clone[1] = this.channelDescriptionTextBox.Text;
    this.clone[2] = this.channelLinkTextBox.Text;
    this.clone[3] = this.traceSourceTextBox.Text;
    this.clone[4] = this.isExternalTraceSourceCheckBox.Checked;
}

private void InitializeComponent()
{
    traceSourceLabel = new Label();
    traceSourceTextBox = new TextBox();
    isExternalTraceSourceCheckBox = new CheckBox();

```

(Continued)

Listing 11-26: (continued)

```
base.SuspendLayout();

. . .

traceSourceLabel.Location = new Point(0, 240);
traceSourceLabel.Name = "channelLinkLabel";
traceSourceLabel.AutoSize = true;
traceSourceLabel.TabIndex = 4;
traceSourceLabel.Text = "Trace Source:";
traceSourceLabel.TextAlign = ContentAlignment.MiddleLeft;

traceSourceTextBox.Location = new Point(110, 240);
traceSourceTextBox.Name = "channelLinkTextBox";
traceSourceTextBox.Width = 250;
traceSourceTextBox.TabIndex = 5;
traceSourceTextBox.TextChanged +=
    new EventHandler(OnTraceSourceTextBoxTextChanged);

isExternalTraceSourceCheckBox.Location = new Point(0, 280);
isExternalTraceSourceCheckBox.Name = "channelLinkLabel";
isExternalTraceSourceCheckBox.AutoSize = true;
isExternalTraceSourceCheckBox.TabIndex = 4;
isExternalTraceSourceCheckBox.Text = "Is external trace source";
isExternalTraceSourceCheckBox.TextAlign = ContentAlignment.MiddleLeft;
isExternalTraceSourceCheckBox.CheckedChanged +=
    new EventHandler(OnIsExternalTraceSourceCheckBoxCheckedChanged);

. . .

base.Controls.Add(traceSourceLabel);
base.Controls.Add(traceSourceTextBox);
base.Controls.Add(isExternalTraceSourceCheckBox);

base.ResumeLayout(false);
}
}
```

As you can see, the new version of `RssPage` contains a new label that displays the text “Trace Source:”, a new textbox that allows the user to enter the trace source name or the fully qualified name of the type that exposes a `TraceSource` property that returns a reference to an external trace source, and a checkbox that allows the page developer to specify whether the textbox contains a trace source name or the required type information. Note that the `InitializeComponents` method registers a method named `OnTraceSourceNameTextBoxTextChanged` as an event handler for the `TextChanged` event of this new textbox and a method named `OnIsExternalTraceSourceCheckBoxCheckedChanged` as an event handler for the `CheckedChanged` event of the checkbox. Figure 11-8 shows what the new `RssPage` module page looks like.



Figure 11-8

You also need to add two new properties named `TraceSource` and `IsExternalTraceSource` to the `RssSectionInfo` class as shown in Listing 11-27. Recall that this class exposes the contents of the `PropertyBag` collection that contains the settings on the `<rss>` configuration section as strongly-typed properties.

Listing 11-27: The `RssSectionInfo` Class

```
using Microsoft.Web.Management.Server;

namespace Rss.Client
{
    public sealed class RssSectionInfo
    {
        private PropertyBag bag;

        public RssSectionInfo(PropertyBag bag)
        {
            this.bag = bag.Clone();
        }

        . . .

        public string TraceSource
        {
            get { return (string)this.bag[5]; }
        }
    }
}
```

(Continued)

Listing 11-27: *(continued)*

```
    public bool IsExternalTraceSource
    {
        get { return (bool)this.bag[6]; }
    }
}
```

Finally, you need to modify the implementation of the `RequestLoggingModuleService` module service class as shown in Listing 11-28. Recall that this module service is responsible for reading the configuration settings from and writing the configuration settings into the underlying configuration file. There are two particular methods that you have to edit. The first method is the `GetChannelSettings` method, because you also want to get the values of the `traceSource` and `isExternalTraceSource` attributes. The second method is the `UpdateChannelSettings` method, because you also want to store the new values of these two attributes in the underlying configuration file.

Listing 11-28: The RequestLoggingModuleService Class

```
using System;
using Rss.Base;
using Microsoft.Web.Administration;
using Microsoft.Web.Management.Server;
using System.Web.Configuration;
using Rss.ImperativeManagement;

namespace Rss.GraphicalManagement.Server
{
    public class RssModuleService : ModuleService
    {
        . . .

        [ModuleServiceMethod(PassThrough = true)]
        public PropertyBag GetSettings()
        {
            PropertyBag bag1 = new PropertyBag();
            RssSection section1 = this.GetSection();
            bag1[0] = section1.ChannelTitle;
            bag1[1] = section1.ChannelDescription;
            bag1[2] = section1.ChannelLink;
            bag1[3] = section1.IsLocked;
            bag1[4] = section1.Enabled;
            bag1[5] = section1.TraceSource;
            bag1[6] = section1.IsExternalTraceSource;
            return bag1;
        }

        [ModuleServiceMethod(PassThrough = true)]
        public void UpdateChannelSettings(PropertyBag updatedChannelSettings)
        {
            RssSection section1 = this.GetSection();
            section1.ChannelTitle = (string)updatedChannelSettings[0];
        }
    }
}
```

Listing 11-28: *(continued)*

```

        section1.ChannelDescription = (string)updatedChannelSettings[1];
        section1.ChannelLink = (string)updatedChannelSettings[2];
        section1.TraceSource = (string)updatedChannelSettings[3];
        section1.IsExternalTraceSource = (bool)updatedChannelSettings[4];
        base.ManagementUnit.Update();
    }
}
}

```

Runtime Status and Control API

As mentioned in Chapter 4, there are two categories of the IIS 7 and ASP.NET integrated imperative management types. The types in the first category expose members (methods and properties), which allow you to imperatively access and manipulate the associated XML constructs of the IIS 7 and ASP.NET integrated configuration system. I discussed these types in great detail in Chapter 4.

The types in the second category expose members (methods and properties), which allow you to imperatively access and manipulate the runtime state of the associated IIS 7 runtime objects. These types include `Request`, `RequestCollection`, `ApplicationDomain`, `ApplicationDomainCollection`, `WorkerProcess`, and `WorkerProcessCollection`, which I discuss in this section.

There are three IIS 7 and ASP.NET integrated imperative management types that fall in both categories because they expose both members (methods and properties) that allow you to imperatively access and manipulate the associated XML constructs of the IIS 7 and ASP.NET integrated configuration system, and members that allow you to imperatively access and manipulate the runtime state of the associated IIS 7 runtime objects. These three types are `ServerManager`, `ApplicationPool`, and `Site`.

To understand how the types in the second category manage to provide imperative access and control over the runtime states of the IIS 7 runtime objects, first you need to familiarize yourself with an important unmanaged API known as Runtime Status and Control API (RSCA). This unmanaged API consists of the following unmanaged types:

- ❑ `IRSCA_AppPool`: This unmanaged RSCA class allows unmanaged code to imperatively access detailed up-to-date runtime data for a specified application pool and to imperatively start, recycle, and stop the application pool.
- ❑ `IRSCA_RequestData`: This unmanaged RSCA class allows unmanaged code to imperatively access detailed up-to-date runtime data for a specified request.
- ❑ `IRSCA_AppDomain`: This unmanaged RSCA class allows unmanaged code to imperatively access detailed up-to-date runtime data for a specified application domain and to imperatively unload the application domain and its assemblies.
- ❑ `IRSCA_RequestReader`: This unmanaged RSCA class allows unmanaged code to iterate through a list of worker process requests and to use the `IRSCA_RequestData` objects associated with these requests to imperatively access detailed up-to-date runtime data for each worker process request.

Chapter 11: Integrated Tracing and Diagnostics

- ❑ `IRSCA_WorkerProcess`: This unmanaged RSCA class allows unmanaged code to imperatively access detailed up-to-date runtime data for a specified worker process.
- ❑ `IRSCA_VirtualSite`: This unmanaged RSCA class allows unmanaged code to imperatively access detailed up-to-date runtime data for a specified virtual Web site and to imperatively start and stop the Web site.
- ❑ `IRSCA_W3SVC`: This unmanaged RSCA class allows unmanaged code to iterate through a list of virtual Web sites to access the `IRSCA_VirtualSite` objects that allow unmanaged code to access detailed up-to-date runtime data for these virtual Web sites and to imperatively start and stop these virtual Web sites.
- ❑ `IRSCA_WAS`: This unmanaged RSCA class allows unmanaged code to:
 - ❑ Iterate through a list of application pools to access the `IRSCA_AppPool` objects that allow unmanaged code to imperatively access detailed up-to-date runtime data for these application pools and to imperatively start, recycle, and stop these application pools.
 - ❑ Iterate through a list of worker processes to access the `IRSCA_WorkerProcess` objects that allow unmanaged code to imperatively access detailed up-to-date runtime data for these worker processes.

As you can see, the detailed up-to-date runtime data for the IIS 7 runtime objects from RSCA provides you with a powerful diagnostic tool to study the IIS 7 runtime state to pinpoint runtime problems and issues.

As mentioned, RSCA is an unmanaged API. As such, only COM/C++ developers can directly program against the unmanaged classes that make up this API. The IIS 7 and ASP.NET integrated imperative management system comes with an internal sealed class named `RscInterop`, which contains a bunch of managed interfaces. As the name suggests, the `RscInterop` class enables the interoperation between managed code and RSCA unmanaged classes. As such, each interface defined within the `RscInterop` class facilitates the interoperation between managed code and an RSCA unmanaged class with the same name as the interface. As Listing 11-29 shows, the `RscInterop` class exposes eight interfaces named `IRSCA_AppDomain`, `IRSCA_AppPool`, `IRSCA_RequestData`, `IRSCA_RequestReader`, `IRSCA_VirtualSite`, `IRSCA_W3SVC`, `IRSCA_WAS`, and `IRSCA_WorkerProcess` that enable interoperation between managed code and the RSCA unmanaged classes with the same names and an enumeration type named `RSCA_OBJECT_STATE_ENUM` that maps into an RSCA unmanaged enumeration type with the same name.

Listing 11-29: The `RscInterop` Internal Class

```
internal sealed class RscInterop
{
    private RscInterop() { }

    [ComImport, Guid("35D651CB-0787-46b2-9B92-667D15E5591B"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    public interface IRSCA_AppDomain { . . . }

    [ComImport, Guid("6b49610c-060b-4d63-b524-cf56c2f890b5"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    public interface IRSCA_AppPool { . . . }
```

Listing 11-29: *(continued)*

```

[ComImport, InterfaceType(ComInterfaceType.InterfaceIsIUnknown),
Guid("6B5D2FE4-0093-46df-B2BC-2D0CDAB2A748")]
public interface IRSCA_RequestData { . . . }

[ComImport, InterfaceType(ComInterfaceType.InterfaceIsIUnknown),
Guid("DAD4B26E-185B-452d-A33B-9548A7D0959A")]
public interface IRSCA_RequestReader { . . . }

[ComImport, InterfaceType(ComInterfaceType.InterfaceIsIUnknown),
Guid("375F4C11-A2EA-4453-ABE4-AE040B64F597")]
public interface IRSCA_VirtualSite { . . . }

[ComImport, Guid("c9041162-3c4f-417e-8b7f-ba2731d585bb"),
InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IRSCA_W3SVC { . . . }

[ComImport, InterfaceType(ComInterfaceType.InterfaceIsIUnknown),
Guid("77DE72A3-C0F1-4820-BFB7-057A21A5A4B2")]
public interface IRSCA_WAS { . . . }

[ComImport, Guid("9D9DE1BD-0D37-4352-B959-A6F2E7BF95DC"),
InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IRSCA_WorkerProcess { . . . }

public enum RSCA_OBJECT_STATE_ENUM { . . . }
}

```

I mentioned that the nested interfaces of the `RscInterop` class enable interoperability between managed code and the underlying RSCA unmanaged classes. The managed code in this case is an internal sealed class named `RscWrapper`. In other words, the nested interfaces of the `RscInterop` class enable the `RscWrapper` managed class to interoperate with the underlying RSCA unmanaged classes to access and to control the IIS 7 runtime state.

The IIS 7 and ASP.NET integrated imperative management types in the second category of types discussed earlier expose the functionality of the `RscWrapper` managed class in the form of a convenient set of methods and properties that you use in your own C# or Visual Basic code to programmatically access and control the IIS 7 runtime state. In other words, you use the IIS 7 and ASP.NET integrated imperative management types in the second category to indirectly interact with the underlying RSCA unmanaged classes to programmatically access detailed up-to-date runtime data for various IIS 7 runtime objects and to programmatically control these IIS 7 runtime objects.

ServerManager

Chapter 4 provided in-depth coverage of those members of the `ServerManager` class that allow your managed code to access configuration data from the IIS 7 and ASP.NET integrated configuration system. The `ServerManager` class also exposes a property of type `WorkerProcessCollection` named

Chapter 11: Integrated Tracing and Diagnostics

`WorkerProcesses` as shown in the following code fragment. As you'll see shortly, this property provides up-to-date runtime data for the current worker processes.

```
public sealed class ServerManager : IDisposable
{
    public WorkerProcessCollection WorkerProcesses { get; }
    . . .
}
```

WorkerProcessCollection

The `WorkerProcessCollection` class acts as a container for `WorkerProcess` objects as demonstrated in Listing 11-30. I discuss `WorkerProcess` shortly.

Listing 11-30: The WorkerProcessCollection Class

```
public sealed class WorkerProcessCollection : ICollection,
                                             IEnumerable<WorkerProcess>, IEnumerable
{
    public IEnumerator<WorkerProcess> GetEnumerator();
    public WorkerProcess GetWorkerProcess(int processId);

    public int IndexOf(WorkerProcess workerProcess);
    public int Count { get; }
    public WorkerProcess this[int index] { get; }
}
```

The `WorkerProcessCollection` class exposes the following members:

- ❑ `GetEnumerator`: This method returns an `IEnumerator<WorkerProcess>` enumerator that you can use to iterate through the `WorkerProcess` objects in the `WorkerProcessCollection` collection.
- ❑ `GetWorkerProcess`: This method returns a reference to the `WorkerProcess` object with a specified process id.
- ❑ `IndexOf`: This method returns the index of a specified `WorkerProcess` object in the `WorkerProcessCollection` collection.
- ❑ `Count`: This property returns the number of the `WorkerProcess` objects in the `WorkerProcessCollection` collection.
- ❑ `Item`: This indexer returns a reference to the `WorkerProcess` object in the `WorkerProcessCollection` with a specified index.

WorkerProcess

The IIS 7 and ASP.NET integrated imperative management API comes with a class named `WorkerProcess`, which allows your C# or Visual Basic code to indirectly interact with the underlying `IRSCA_WorkerProcess` RSCA unmanaged object to imperatively retrieve detailed up-to-date runtime data for the worker process responsible for processing the requests for a particular application pool.

Listing 11-31 presents the declaration of the members of this class.

Notice that this class exposes a method named `GetRequests` that returns a `RequestCollection` collection. I discuss `RequestCollection` shortly. The `GetRequests` method takes an integer value that allows you to select requests with elapsed time less than the specified value. The elapsed time of a request specifies how long the worker process has been processing the request.

Listing 11-31: The WorkerProcess Class

```
public sealed class WorkerProcess
{
    public RequestCollection GetRequests(int timeElapsedFilter);

    public ApplicationDomainCollection ApplicationDomains { get; }
    public string AppPoolName { get; }
    public int ProcessId { get; }
    public Guid ProcessGuid { get; }
    public WorkerProcessState State { get; }
}
```

Here are the descriptions of important properties of the `WorkerProcess` class:

- ❑ `ApplicationDomains`: Gets the `ApplicationDomainCollection` collection that contains `ApplicationDomain` objects. I discuss `ApplicationDomain` shortly.
- ❑ `AppPoolName`: Gets the name of the application pool to which the worker process is assigned. The same worker process cannot be assigned to two different application pools simultaneously. This ensures that the application domains are isolated by process boundaries. Such isolation ensures that the application misbehavior in one application pool does not affect the applications running in another application pool.
- ❑ `ProcessId`: Gets the process id of the worker process.
- ❑ `ProcessGuid`: Gets the Guid for the worker process.
- ❑ `State`: Gets the running state of the worker process. Note that this property is of enumeration type `WorkerProcessState` as defined in Listing 11-32.

Listing 11-32: The WorkerProcessState Enumeration

```
public enum WorkerProcessState
{
    Starting,
    Running,
    Stopping,
    Unknown
}
```

RequestCollection

The IIS 7 and ASP.NET integrated imperative management API comes with a class named `RequestCollection` that acts as a container for `Request` objects as shown in Listing 11-33. I discuss `Request` shortly.

Listing 11-33: The RequestCollection Class

```
public sealed class RequestCollection : ICollection, IEnumerable<Request>,
                                     IEnumerable
{
    public IEnumerator<Request> GetEnumerator();
    public int IndexOf(Request element);

    public int Count { get; }
    public Request this[int index] { get; }
}
```

Here are the descriptions of the methods and properties of the RequestCollection class:

- ❑ GetEnumerator: Returns the `IEnumerator<Request>` enumerator that you can use to iterate through the `Request` objects in the collection.
- ❑ IndexOf: Returns the index of the specified `Request` object in the collection.
- ❑ Count: Gets the total number of the `Request` objects in the collection.
- ❑ Item: Gets the `Request` object with the specified index in the collection.

Request

The IIS 7 and ASP.NET integrated imperative management API comes with a managed class named `Request`, which allows your managed code to indirectly interact with the underlying `IRSCA_RequestData` RSCA unmanaged object to retrieve detailed up-to-date runtime data for a specified client request and expose this data through strongly-typed properties shown in Listing 11-34.

Listing 11-34: The Request Class

```
public sealed class Request
{
    public string ClientIPAddr { get; }
    public string ConnectionId { get; }
    public string CurrentModule { get; }
    public string HostName { get; }
    public string LocalIPAddress { get; }
    public int LocalPort { get; }
    public PipelineState PipelineState { get; }
    public int ProcessId { get; }
    public string RequestId { get; }
    public int SiteId { get; }
    public int TimeElapsed { get; }
    public int TimeInModule { get; }
    public int TimeInState { get; }
    public string Url { get; }
    public string Verb { get; }
}
```

Here are the descriptions of some of these properties:

- ❑ `ClientIPAddr`: Gets the IP address of the client that made the request.
- ❑ `CurrentModule`: Gets the name of the IIS 7 module currently processing the request.
- ❑ `TimeInModule`: Specifies how long the current IIS 7 module has been processing the request.
- ❑ `ProcessId`: Gets the process ID of the worker process processing the request.
- ❑ `SiteId`: Gets the identifier of the Web site to which the request was made.
- ❑ `TimeElapsed`: Specifies how long the worker process has been processing the request.
- ❑ `Url`: Gets the request URL.
- ❑ `Verb`: Gets the HTTP verb that the client used to make the request.
- ❑ `PipelineState`: Gets the `PipelineState` enumeration value that specifies the current pipeline state of the request. Listing 11-35 presents the definition of this enumeration type. These pipeline states basically map to the states at which the `HttpApplication` object fires its corresponding events.
- ❑ `TimeInState`: Specifies how long the request has been in the current pipeline state.

Listing 11-35: The `PipelineState` Enumeration Type

```
public enum PipelineState
{
    AcquireRequestState = 0x20,
    AuthenticateRequest = 2,
    AuthorizeRequest = 4,
    BeginRequest = 1,
    EndRequest = 0x800,
    ExecuteRequestHandler = 0x80,
    LogRequest = 0x400,
    MapRequestHandler = 0x10,
    PreExecuteRequestHandler = 0x40,
    ReleaseRequestState = 0x100,
    ResolveRequestCache = 8,
    SendResponse = 0x20000000,
    Unknown = 0,
    UpdateRequestCache = 0x200
}
```

ApplicationDomain

A .NET application domain acts as a container for one or more assemblies, and isolates these assemblies from the assemblies in other application domains. Application domains provide a level of isolation similar to OS processes, without the expensive context switch overhead. Application domains allow you to load more than one application into the same OS process, where each application runs in a separate application domain.

You cannot run any managed application — be it desktop or Web — without creating an application domain and loading the application into the domain. The `ApplicationDomain` class is the IIS 7 and

Chapter 11: Integrated Tracing and Diagnostics

ASP.NET integrated imperative management API's representation of an application domain as shown in Listing 11-36 and provides imperative access to the detailed up-to-date runtime data for an application domain and imperative means to unload the application domain.

Listing 11-36: The ApplicationDomain Class

```
public class ApplicationDomain
{
    // Methods
    public void Unload();

    // Properties
    public string Id { get; }
    public string PhysicalPath { get; }
    public string VirtualPath { get; }
    public WorkerProcess WorkerProcess { get; }
}
```

The following describes the members of the ApplicationDomain class:

- ❑ **Unload:** This method allows your managed code to indirectly invoke the `Unload` method on the underlying `IRSCA_AppDomain` RSCA unmanaged object to unload an application domain.
- ❑ **Id:** Gets the identifier of the application domain.
- ❑ **PhysicalPath:** Gets the physical path of the application loaded into the application domain.
- ❑ **VirtualPath:** Gets the virtual path of the application loaded into the application domain.
- ❑ **WorkerProcess:** Gets the `WorkerProcess` that represents the worker process where the application domain resides.

ApplicationDomainCollection

The IIS 7 and ASP.NET integrated imperative management API comes with a collection class named `ApplicationDomainCollection`, which acts as a container for `ApplicationDomain` objects. Listing 11-37 presents the members of this class.

Listing 11-37: The ApplicationDomainCollection Class

```
public sealed class ApplicationDomainCollection : ICollection,
                                                IEnumerable<ApplicationDomain>, IEnumerable
{
    // Methods
    public IEnumerator<ApplicationDomain> GetEnumerator();
    public int IndexOf(ApplicationDomain element);

    // Properties
    public int Count { get; }
    public ApplicationDomain this[int index] { get; }
}
```

Here are the descriptions of the members of this class:

- ❑ **GetEnumerator:** Returns the `IEnumerator<ApplicationDomain>` object that can be used to iterate through the `ApplicationDomain` objects in the collection.
- ❑ **IndexOf:** Returns the index of the specified `ApplicationDomain` object in the collection.
- ❑ **Count:** Gets the total number of the `ApplicationDomain` objects in the collection, which is basically the total number of application domains that reside in the associated worker process.
- ❑ **Item:** Gets the `ApplicationDomain` object with the specified index.

ApplicationPool

Chapter 4 discussed those members of the `ApplicationPool` class that allow you to imperatively access and manipulate the XML constructs that represent an application pool in the IIS 7 and ASP.NET integrated configuration system. This section discusses those members of the `ApplicationPool` class that allow you to imperatively access and manipulate the runtime state of an application pool as shown in Listing 11-38.

Listing 11-38: The ApplicationPool Class

```
public sealed class ApplicationPool : ConfigurationElement
{
    public ObjectState Recycle();
    public ObjectState Start();
    public ObjectState Stop();

    public ObjectState State { get; }
    public WorkerProcessCollection WorkerProcesses { get; }
}
```

Here are the descriptions of these members:

- ❑ **State:** This read-only property returns an object of type `ObjectState` enumeration. Listing 11-39 presents the definition of this enumeration type. As the name suggests, the `State` property specifies the running state of an application pool.
- ❑ **WorkerProcesses:** This read-only property returns a reference to a `WorkerProcessCollection` collection of `WorkerProcess` objects, each allowing your managed code to imperatively access detailed up-to-date runtime data for a particular worker process assigned to the application pool.
- ❑ **Recycle:** This method allows your managed code to indirectly invoke the `Recycle` method on the underlying `IRSCA_AppPool` RSCA unmanaged object to recycle an application pool.
- ❑ **Start:** This method allows your managed code to indirectly invoke the `Start` method on the underlying `IRSCA_AppPool` RSCA unmanaged object to start an application pool.
- ❑ **Stop:** This method allows your managed code to indirectly invoke the `Stop` method on the underlying `IRSCA_AppPool` RSCA unmanaged object to stop an application pool.

Listing 11-39: The ObjectState Enumeration Type

```
public enum ObjectState
{
    Starting,
    Started,
    Stopping,
    Stopped,
    Unknown
}
```

Site

Chapter 4 provides in-depth coverage of those members of the `Site` class that allow your managed code to access and to manipulate the XML representation of a virtual Web site in the underlying configuration file. This section covers those members of the `Site` class that allow your managed code to access and to manipulate the runtime state of a virtual Web site as shown in Listing 11-40.

Listing 11-40: The Site Class

```
public sealed class Site : ConfigurationElement
{
    public ObjectState Start();
    public ObjectState Stop();

    public ObjectState State { get; }
}
```

The following list describes these members:

- ❑ **Start:** Call this method from your managed code to indirectly invoke the `Start` method on the underlying `IRSCA_VirtualSite` RSCA unmanaged object to start a virtual Web site.
- ❑ **Stop:** Call this method from your managed code to indirectly invoke the `Stop` method on the underlying `IRSCA_VirtualSite` RSCA unmanaged object to stop a virtual Web site.
- ❑ **State:** Call this property to return an `ObjectState` enumeration value that specifies the current runtime state of a virtual Web site.

Putting It All Together

In this section, I present a few examples that use the `Request`, `RequestCollection`, `ApplicationDomain`, `ApplicationDomainCollection`, `WorkerProcess`, `WorkerProcessCollection`, `ServerManager`, `ApplicationPool`, and `Site` IIS 7 and ASP.NET integrated imperative management types to access and to control the runtime states of the IIS 7 runtime objects.

To set up the first example, launch Visual Studio and add a blank solution. Next add a new Console Application to this solution and replace the content of the `Program.cs` file with the code shown in Listing 11-41. Then add a new Web application to this solution and replace the content of the `Default.aspx` file with the code shown in Listing 11-42. Note that the `Page_Load` method in this code listing

uses the `Sleep` static method of the `.NET Thread` class. This will allow the console application to retrieve up-to-date runtime data for the HTTP request made for the `Default.aspx` file. Now first access the `Default.aspx` file from your browser and then run the console application. You should get the result shown in Figure 11-9.

Listing 11-41: Displaying Up-to-Date Runtime Data for IIS 7 Runtime Objects

```
using System;
using Microsoft.Web.Administration;

class Program
{
    static void Main(string[] args)
    {
        ServerManager mgr = new ServerManager();
        foreach (WorkerProcess wp in mgr.WorkerProcesses)
        {
            Console.WriteLine("Worker Process Info:");
            Console.WriteLine("    AppPoolName: " + wp.AppPoolName);
            Console.WriteLine("    ProcessId: " + wp.ProcessId.ToString());
            Console.WriteLine("    ProcessGuid: " + wp.ProcessGuid.ToString());
            Console.WriteLine("    State: " + wp.State.ToString());
            Console.WriteLine();

            foreach (Request request in wp.GetRequests(10))
            {
                Console.WriteLine("Request Info: ");
                Console.WriteLine("    ClientIPAddr: " + request.ClientIPAddr);
                Console.WriteLine("    ConnectionId: " + request.ConnectionId);
                Console.WriteLine("    CurrentModule: " + request.CurrentModule);
                Console.WriteLine("    HostName: " + request.HostName);
                Console.WriteLine("    LocalIPAddress: " + request.LocalIPAddress);
                Console.WriteLine("    LocalPort: " + request.LocalPort.ToString());
                Console.WriteLine("    PipelineState: " +
                    request.PipelineState.ToString());
                Console.WriteLine("    ProcessId: " + request.ProcessId.ToString());
                Console.WriteLine("    RequestId: " + request.RequestId);
                Console.WriteLine("    SiteId: " + request.SiteId.ToString());
                Console.WriteLine("    TimeElapsed: " + request.TimeElapsed.ToString());
                Console.WriteLine("    TimeInModule: " +
                    request.TimeInModule.ToString());
                Console.WriteLine("    TimeInState: " + request.TimeInState.ToString());
                Console.WriteLine("    Url: " + request.Url);
                Console.WriteLine("    Verb: " + request.Verb);
            }
            Console.WriteLine();

            foreach (ApplicationDomain appDomain in wp.ApplicationDomains)
            {
                Console.WriteLine("Application Domain Info: ");
                Console.WriteLine("    Id: " + appDomain.Id);
                Console.WriteLine("    PhysicalPath: " + appDomain.PhysicalPath);
                Console.WriteLine("    VirtualPath: " + appDomain.VirtualPath);
            }
        }
    }
}
```

(Continued)

Listing 11-41: (continued)

```
        Console.WriteLine();  
        Console.WriteLine();  
    }  
    Console.ReadKey();  
}  
}
```

Listing 11-42: The Default.aspx Page

```
<%@ Page Language="C#" %>  
  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<script runat="server">  
    void Page_Load(object sender, EventArgs e)  
    {  
        System.Threading.Thread.Sleep(60000);  
    }  
</script>  
<html xmlns="http://www.w3.org/1999/xhtml">  
<head runat="server">  
    <title>Untitled Page</title>  
</head>  
<body>  
    <form id="form1" runat="server">  
    </form>  
</body>  
</html>
```



Figure 11-9

Next, I walk you through the implementation of the `Main` method shown in Listing 11-41. This method begins by instantiating a `ServerManager` instance:

```
ServerManager mgr = new ServerManager();
```

Then, it invokes the `WorkerProcesses` property on this `ServerManager` instance to return a reference to the `WorkerProcessCollection` collection of `WorkerProcess` objects. Recall that each `WorkerProcess` object allows your managed code to indirectly use the underlying `IRSCA_WorkerProcess` RSCA unmanaged object to retrieve detailed up-to-date runtime data for the associated worker process. Next, the `Main` method iterates through the `WorkerProcess` objects in this `WorkerProcessCollection` collection for each enumerated `WorkerProcess` object. First, it writes out the values of the properties of the enumerated `WorkerProcess`:

```
Console.WriteLine("Worker Process Info:");
Console.WriteLine("    AppPoolName: " + wp.AppPoolName);
Console.WriteLine("    ProcessId: " + wp.ProcessId.ToString());
Console.WriteLine("    ProcessGuid: " + wp.ProcessGuid.ToString());
Console.WriteLine("    State: " + wp.State.ToString());
Console.WriteLine();
```

Then, it calls the `GetRequest` method on the enumerated `WorkerProcess` object to return a reference to the `RequestCollection` collection of `Request` objects. Recall that each `Request` object in this collection allows your managed code to indirectly use the underlying `IRSCA_RequestData` RSCA unmanaged object to retrieve detailed up-to-date runtime data for the associated client request. Next, the `Main` method iterates through the `Request` objects in this `RequestCollection` collection and writes out the values of the properties of each `Request` object:

```
foreach (Request request in wp.GetRequests(10))
{
    Console.WriteLine("Request Info: ");
    Console.WriteLine("    ClientIPAddr: " + request.ClientIPAddr);
    Console.WriteLine("    ConnectionId: " + request.ConnectionId);
    Console.WriteLine("    CurrentModule: " + request.CurrentModule);
    Console.WriteLine("    HostName: " + request.HostName);
    Console.WriteLine("    LocalIPAddress: " + request.LocalIPAddress);
    Console.WriteLine("    LocalPort: " + request.LocalPort.ToString());
    Console.WriteLine("    PipelineState: " +
        request.PipelineState.ToString());
    Console.WriteLine("    ProcessId: " + request.ProcessId.ToString());
    Console.WriteLine("    RequestId: " + request.RequestId);
    Console.WriteLine("    SiteId: " + request.SiteId.ToString());
    Console.WriteLine("    TimeElapsed: " + request.TimeElapsed.ToString());
    Console.WriteLine("    TimeInModule: " +
        request.TimeInModule.ToString());
    Console.WriteLine("    TimeInState: " + request.TimeInState.ToString());
    Console.WriteLine("    Url: " + request.Url);
    Console.WriteLine("    Verb: " + request.Verb);
}
Console.WriteLine();
```

Finally, the `Main` method calls the `ApplicationDomains` property on the enumerated `WorkerProcess` to return a reference to the `ApplicationDomainCollection` collection of `ApplicationDomain` objects.

Chapter 11: Integrated Tracing and Diagnostics

Recall that each `ApplicationDomain` object allows your managed code to use the underlying `IRSCA_AppDomain` RSCA unmanaged object to retrieve detailed up-to-date runtime data for the associated application domain. Next, the `Main` method iterates through the `ApplicationDomain` objects in this `ApplicationDomainCollection` collection and writes out the values of the properties of each `ApplicationDomain` object:

```
foreach (ApplicationDomain appDomain in wp.ApplicationDomains)
{
    Console.WriteLine("Application Domain Info: ");
    Console.WriteLine("    Id: " + appDomain.Id);
    Console.WriteLine("    PhysicalPath: " + appDomain.PhysicalPath);
    Console.WriteLine("    VirtualPath: " + appDomain.VirtualPath);
}
Console.WriteLine();
Console.WriteLine();
}
```

LogRequest

The `HttpApplication` object responsible for processing requests for a given Web application fires an event named `LogRequest` if the application pool containing the Web application is running in integrated mode. The `HttpApplication` object fires this event to allow interested HTTP modules and application code to log request data to the desired data store. Such logged request data provides a powerful diagnostic tool that you can use for troubleshooting your Web applications.

In this section you implement an HTTP module named `MyModule` that registers an event handler for the `LogRequest` event of the `HttpApplication` object to log request data to an XML file. The first order of business is to decide on the XML schema of this XML file. Listing 11-43 presents an example of an XML document that the `MyModule` HTTP module supports.

Listing 11-43: An Example of an XML File That the `MyModule` HTTP Module Supports

```
<?xml version="1.0" encoding="utf-8"?>
<requests>
  <request applicationPath="/WebSite2" date="7/12/2007 1:35:47 AM"
    hostAddress="::1" httpMethod="GET" siteName="Default Web Site"
    url="http://localhost/WebSite2/Default.aspx" />
</requests>
```

As you can see, this XML file consists of a document element named `<requests>`, which contains zero or more child elements named `<request>`. Each `<request>` child element exposes six attributes named `applicationPath`, `date`, `hostAddress`, `httpMethod`, `siteName`, and `url`, which store request data. You're not limited to storing these six pieces of information about a given request. However, to keep our discussions focused, you'll store only these six pieces of information for each request. You can also assume that the `<connectionStrings>` section of the `web.config` file of the Web application for which the request is made contains an `<add>` element whose `name` attribute value is set to "MyXmlFile" and whose `connectionString` attribute value is set to the virtual path of the XML file where the request data will be stored. Listing 11-44 presents an example of such a `web.config` file.

Listing 11-44: The web.config File

```
<configuration>
  <connectionStrings>
    <add name="MyXmlFile" connectionString="App_Data/data.xml" />
  </connectionStrings>
</configuration>
```

Launch Visual Studio. Add a blank solution named `MyModuleSol` and add a new Class Library project named `MyModuleProj` to this solution. Right-click this project in Solution Explorer and select the Properties option to launch the Properties dialog. Switch to the Application tab and enter **MyAssembly** and **MyNamespace** into the Assembly name and Default namespace textboxes, respectively. Then follow the steps discussed in Chapter 7 to configure Visual Studio to:

- ☐ Compile `MyModuleProj` project into a strongly-named assembly.
- ☐ Deploy the assembly to the Global Assembly Cache (GAC).

Next, add a new Web application named `MyWebApplication` to the `MyModuleSol` solution. Add a new XML file named `data.xml` to the `App_Data` directory of this Web application and add the following listing to this file:

```
<?xml version="1.0" encoding="utf-8"?>
<requests>
</requests>
```

Then add the `<add>` element shown in Listing 11-44 to the `<connectionStrings>` section of the `web.config` file of `MyWebApplication` application.

Next, we'll get down to the implementation of the `MyModule` HTTP module as shown in Listing 11-45. As you can see, this HTTP module, like any other HTTP module, inherits the `IHttpModule` interface and implements the `Dispose` and `Init` methods of this interface. Note that the `MyModule` HTTP module's implementation of the `Init` method registers a method named `OnLogRequest` as an event handler for the `LogRequest` event of the `HttpApplication` object. Now add a new source file named `MyModule.cs` to the `MyModuleProj` project and add the code shown in Listing 11-45 to this file. You also need to add references to the `System.Web.dll` and `System.Configuration.dll` assemblies to the `MyModuleProj` project.

Listing 11-45: The MyModule HTTP Module

```
using System;
using System.Xml;
using System.Web;
using System.Xml.XPath;
using System.Web.Hosting;
using System.Configuration;

namespace MyNamespace
{
    public class MyModule : IHttpModule
    {
```

(Continued)

Listing 11-45: *(continued)*

```
void IHttpModule.Dispose() { }

void IHttpModule.Init(HttpApplication app)
{
    app.LogRequest += new EventHandler(OnLogRequest);
}

void OnLogRequest(object sender, EventArgs e)
{
    string dataFile =
        ConfigurationManager.ConnectionStrings["MyXmlFile"].ConnectionString;
    HttpApplication app = sender as HttpApplication;
    HttpContext context = app.Context;

    XmlDocument doc = new XmlDocument();
    doc.Load(context.Server.MapPath(dataFile));

    XPathNavigator nav = doc.CreateNavigator();
    nav.MoveToChild(XPathNodeType.Element);

    using (XmlWriter writer = nav.AppendChild())
    {
        writer.WriteStartElement("request");
        writer.WriteAttributeString("applicationPath",
                                    context.Request.ApplicationPath);
        writer.WriteAttributeString("date", DateTime.Now.ToString());
        writer.WriteAttributeString("hostAddress",
                                    context.Request.UserHostAddress);
        writer.WriteAttributeString("httpMethod", context.Request.HttpMethod);
        writer.WriteAttributeString("siteName", HostingEnvironment.SiteName);
        writer.WriteAttributeString("url", context.Request.Url.ToString());
        writer.WriteEndElement();
    }

    doc.Save(HttpContext.Current.Server.MapPath(dataFile));
}
}
```

Next, I walk you through the implementation of the `OnLogRequest` method. `OnLogRequest` begins by accessing the virtual path of the XML file where the request data will be stored:

```
string dataFile =
    ConfigurationManager.ConnectionStrings["MyXmlFile"].ConnectionString;
```

Next, `OnLogRequest` accesses a reference to the current `HttpContext` object:

```
HttpApplication app = sender as HttpApplication;
HttpContext context = app.Context;
```

Then, it instantiates an `XmlDocument` and populates it with the content of the XML file with the specified virtual path:

```
XmlDocument doc = new XmlDocument();
doc.Load(context.Server.MapPath(dataFile));
```

Next, it calls the `CreateNavigator` method on the `XmlDocument` to return the `XPathNavigator` object that knows how to navigate this document:

```
XPathNavigator nav = doc.CreateNavigator();
```

Then, it invokes the `MoveToChild` method to move the document element of the XML document:

```
nav.MoveToChild(XPathNodeType.Element);
```

As mentioned earlier, in this case, the document element is an element named `<requests>`.

Next, `OnLogRequest` invokes the `AppendChild` method on the `XPathNavigator` to return a reference to an `XmlWriter` instance that allows you to add a new child `<request>` element to the `<requests>` document element:

```
XmlWriter writer = nav.AppendChild()
```

Next, it stores the request data in the associated attributes of this `<request>` child element:

```
writer.WriteStartElement("request");
writer.WriteAttributeString("applicationPath",
                           context.Request.ApplicationPath);
writer.WriteAttributeString("date", DateTime.Now.ToString());
writer.WriteAttributeString("hostAddress", context.Request.UserHostAddress);
writer.WriteAttributeString("httpMethod", context.Request.HttpMethod);
writer.WriteAttributeString("siteName", HostingEnvironment.SiteName);
writer.WriteAttributeString("url", context.Request.Url.ToString());
writer.WriteEndElement();
```

Finally, `OnLogRequest` commits the changes to the underlying XML file:

```
doc.Save(HttpContext.Current.Server.MapPath(dataFile));
```

Next, follow the steps discussed in Chapter 8 to plug the `MyModule` HTTP module into the IIS 7 and ASP.NET integrated request processing pipeline.

Now access the `Default.aspx` page of the `MyWebApplication` project. The content of this page is irrelevant to our current discussions. Next open the `data.xml` file in your favorite editor. You should see the result shown in the following code listing:

```
<?xml version="1.0" encoding="utf-8"?>
<requests>
  <request applicationPath="/WebSite2" date="7/12/2007 1:35:47 AM"
    hostAddress="::1" httpMethod="GET" siteName="Default Web Site"
    url="http://localhost/WebSite2/Default.aspx" />
</requests>
```

Chapter 11: Integrated Tracing and Diagnostics

When you access the `Default.aspx` page, the current `HttpApplication` object first invokes the `Init` method of the `MyModule` HTTP module to allow the module to register the `OnLogRequest` method as an event handler for the `LogRequest` event. The `HttpApplication` object then at some point fires its `LogRequest` event and consequently invokes the `OnLogRequest` method of the `MyModule` HTTP module, which in turn stores the request data into the `data.xml` file.

Summary

This chapter provided in-depth coverage of the IIS 7 and ASP.NET integrated tracing infrastructure and its main components. You learned how to use this integrated infrastructure to instrument your managed code with tracing, to route traces to the IIS 7 tracing infrastructure, and to configure IIS 7 modules such as Failed Request Tracing to consume these traces. The next chapter moves on to the deep integration of ASP.NET and Windows Communication Foundation in IIS 7, where you learn how to take advantage of this deep integration in your own Web applications.

12

ASP.NET and Windows Communication Foundation Integration in IIS 7

Windows Communication Foundation (WCF) is a comprehensive layered framework for service-oriented programming. The top layer of this framework, known as the *Service Model* layer, allows you to model the communications of your software product with the outside world with minimal time and effort. The framework then extracts all the required information from your model and uses this information to create and to configure the runtime components needed to implement your model from a lower layer of the framework (known as the *Channel* layer). In other words, you're only responsible for modeling the communications of your products with the outside world, and you don't have to worry about actually implementing this model. This saves you from having to deal with dirty little details of the underlying runtime components that implement your model, and consequently allows you to focus on what matters to your application, that is, the domain-specific requirements of your application.

This chapter presents and discusses the implementation of an example that will show you how to use the Windows Communication Foundation Service Model to model the communications of your own software products. During the course of this chapter you'll see how you can take advantage of the deep integration of ASP.NET and WCF services in the IIS 7 environment in your own Web applications.

Installing the Required Software

Because Windows Vista and Windows Server 2008 automatically install the required .NET Framework components, you should already have everything you need to use WCF. However, if

you want to take full advantage of the programming convenience of Visual Studio in your own service-oriented programming activities, you'll need to install one of the following:

- ☐ Visual Studio 2008.
- ☐ Visual Studio 2005 plus the required WCF Visual Studio Extensions. You can download a free copy of these extensions from the Microsoft Web site.

Bug Report Manager

As mentioned earlier, all discussions of this chapter are presented in the context of an example. This example implements a bug report manager system. Every piece of software ships with bugs that don't get caught and fixed during the development process of the software. As a result, every organization needs to have a bug report management system in place to manage the reported bugs such as:

- ☐ Adding a new bug report: Bugs are reported via different mechanisms. For example, the software itself may contain the necessary logic for automatically reporting bugs when and where they occur. Such automatic bug reporting does not involve human interaction other than possibly showing a popup to the user asking for her permission to report the bug. The users, administrators, support staff, and the like should also be able to report bugs. Such bug reports are normally done through some elaborate user interface where people can enter information about the bug.
- ☐ Updating a bug report. For example, the engineer assigned to a bug may not be able to reproduce the bug and may want to update the bug report accordingly.
- ☐ Providing statistical analyses of the bug reports such as the number of high-priority bugs, how long a typical bug report sits in the system before it is finally resolved and closed, the number of closed bugs, the number of bugs that haven't yet been assigned to any engineer, and so on.

To keep our discussions focused, the bug report manager system will support only two features:

- ☐ Adding new bug reports
- ☐ Retrieving the list of all bug reports submitted to the system

The bug report manager system will also maintain all bug reports in memory. As you'll see later, keeping bug reports in memory will also help with our discussions of the deep integration of ASP.NET and WCF in the IIS 7 environment.

Next, we'll get down to the implementation of the bug report manager system. First, go ahead and create a blank solution named `ProIIS7AspNetIntegProgCh12` in Visual Studio. You'll add all of the projects used in this chapter to this solution. Next, add a new Class Library project named `BugReportManager` to this solution. Right-click this project in Solution Explorer and select Properties from the popup menu to launch the Properties dialog. Switch to the Application tab in the Properties dialog and enter `ProIIS7AspNetIntegProgCh12` into the "Assembly name" and "Default namespace" textboxes. Don't forget to save these changes.

Next, add an empty source file named `BugReportManager.cs` to the `BugReportManager` project and add the code shown in Listing 12-1 to this source file. Now go ahead and build the `BugReportManager`

Class Library project.

Listing 12-1: The BugReportManager Class

```
using System.Collections;
using System.Data;

namespace ProIIS7AspNetIntegProgCh12
{
    public static class BugReportManager
    {
        private static DataTable bugReports;
        static BugReportManager()
        {
            bugReports = new DataTable();
            bugReports.Columns.Add(new DataColumn("Sender"));
            bugReports.Columns.Add(new DataColumn("Subject"));
            bugReports.Columns.Add(new DataColumn("Body"));
        }

        public static void AddBugReport(string sender, string subject, string body)
        {
            DataRow row = bugReports.NewRow();
            row["Sender"] = sender;
            row["Subject"] = subject;
            row["Body"] = body;
            bugReports.Rows.Add(row);
        }

        public static IEnumerable GetBugReports()
        {
            return bugReports.DefaultView;
        }
    }
}
```

As you can see, Listing 12-1 implements a static class named `BugReportManager` that exposes two static methods named `AddBugReport` and `GetBugReports`. The `AddBugReport` method takes three strings as its arguments, creates a `DataRow` object and populates it with these three strings, and adds the object to an internal `DataTable` object named `bugReports`. As you can see, each `DataRow` object in this internal `DataTable` contains three pieces of information about a bug report: its sender, subject, and body. The `GetBugReports` method simply returns the `DataRowView` object that represents the default view of this `DataTable` object. Note that the `GetBugReports` method returns this `DataRowView` object as an object of type `IEnumerable`. This will allow you to change the internal storage mechanism of your bug report manager system without affecting its clients.

The following sections show you how to use WCF to enable the bug report manager system to communicate with its clients, such as engineers responsible for fixing bugs, support staff, and the like.

Windows Communication Foundation Service

As you can imagine, the bug report manager system relies heavily on its communications with the outside world. After all, it is through these communications that new bug reports are added to the system, existing bug reports are updated, and statistical information is retrieved.

One of the important software design criteria is that each component of the software must be specifically designed and optimized for a particular task. Therefore, you must not incorporate the logic that handles the communications of the bug report manager system with the outside world into the bug report manager system itself. Instead, you must delegate the responsibility of the communications with the outside world to a different component. This component is known as a WCF *service* in the WCF jargon. More formally, a WCF service is a piece of software that responds to communications from the outside world. You can think of a WCF service as a wrapper around a piece of software such as the bug report manager system to enable the software to communicate with the outside world.

Windows Communication Foundation Endpoint

A WCF service must be able to provide different access points for different communication scenarios. For example, engineers responsible for fixing bugs should communicate with the service through a different access point than the end users. These access points are known as WCF *endpoints* in the WCF jargon, as shown in Figure 12-1.

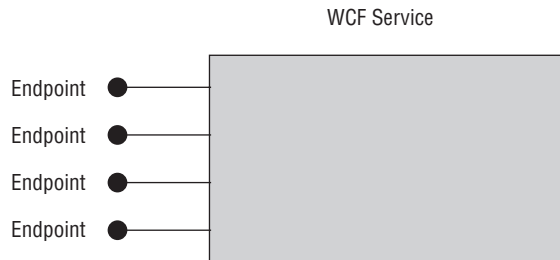


Figure 12-1

Every WCF endpoint has a *binding*, an *address*, and a *contract*:

1. **Binding:** The binding of an endpoint specifies *how* the endpoint communicates with the outside world, that is, it specifies the communication protocols for the endpoint. Here are a few examples:
 - ❑ The transport protocol, through which the endpoint and its clients communicate. HTTP and TCP are two examples of transport protocols.

- ❑ The encoding protocol that the endpoint and its clients use to encode their communications before they go over the wire. XML text and binary are two examples of encoding protocols.
 - ❑ The security protocol that the endpoint and its clients use to secure their communications. For example, some bug reports may contain sensitive information requiring a secure transmission channel. Or retrieving some sensitive statistical information from the bug report manager system may require secure channels. HTTPS and WS-Security are examples of security protocols.
2. *Address*: The address of an endpoint specifies *where* the endpoint is, that is, the network address to which the clients of an endpoint must direct their communications if they want to communicate with the WCF service through that endpoint.
 3. *Contract*: The contract of an endpoint specifies *what* operations of the WCF service the clients can access through that endpoint.

Windows Communication Foundation Service Model

As discussed earlier, a WCF service is a wrapper around a piece of software enabling the software to communicate with the outside world. Later in this chapter, you develop a WCF service, which will enable the bug report manager system to communicate with the outside world.

As mentioned earlier, Windows Communication Foundation is a layered framework. The top layer of this framework is known as *Service Model*. As the name suggests, you use this layer to *model* your WCF service. The keyword here is *modeling*. Modeling a piece of software is much less time-consuming and takes much less effort than implementing the software. Besides, modeling allows you to avoid distracting implementation details and focus on the actual design of your software.

The Windows Communication Foundation Service Model provides you with three important facilities to model your WCF service:

- ❑ **Attribute-based programming**: The Windows Communication Foundation Service Model comes with a convenient set of metadata attributes that you can use to annotate the appropriate entities in your program such as interfaces, classes, and properties. In other words, you get to model some aspects of your service in terms of the Windows Communication Foundation Service Model's metadata attributes.
- ❑ **Configuration-based programming**: The Windows Communication Foundation Service Model comes with a convenient configuration language that you can use to model some aspects of your service in the configuration file.
- ❑ **Regular imperative programming**: The Windows Communication Foundation Service Model comes with a convenient set of managed classes that you can use to model your service in code.

Windows Communication Foundation then uses your service model, which is modeled in terms of the Windows Communication Foundation Service Model's metadata attributes, configuration language,

and managed classes, to instantiate and configure the runtime components needed to implement your model. Your sole job is to define your service model in terms of the Windows Communication Foundation Service Model's metadata attributes, configuration language, and managed classes. You don't have to worry about instantiating and configuring the runtime components that implement your model because Windows Communication Foundation automatically takes care of this for you.

You may be wondering just exactly what a typical model of a WCF service looks like. The answer lies in the discussions of the previous section and Figure 12-1. As this figure shows, from the outside, a WCF service looks like a collection of endpoints. This outside view of the service is exactly what we care about when we're using the Windows Communication Foundation Service Model's metadata attributes, configuration language, and managed classes to model a service.

The inside view of a service is quite complex because that's where all the details of the runtime components that implement the service come into play.

As you can see, developing a WCF service model boils down to developing its endpoints. Recall that every endpoint has an address, a binding, and a contract. In general it takes two sets of tasks to develop a WCF service model: imperative (coding or development tasks) and administrative.

Imperative tasks involve designing those parts of the model that requires using the Windows Communication Foundation Service Model's metadata attributes and managed classes because using these attributes and classes requires coding. Administrative tasks, on the other hand, involve designing those parts of the service model that requires using the Windows Communication Foundation Service Model's configuration language because using this language does not require coding.

Windows Communication Foundation allows you to develop all aspects of your service in code if you choose to do so. However, this is not a recommended approach. As a rule of thumb, anything that can be done declaratively in the configuration files should be done in the configuration files to allow administrators to modify the service without any code changes.

Developing a WCF Service

Developing the endpoints of a WCF service involves the following tasks:

- ❑ Developing the service contracts. Recall that the contract of an endpoint specifies those operations of a WCF service that are available at that endpoint. In other words, the client communications directed at a particular endpoint of a WCF service will only be able to access those operations of the service that are part of the endpoint's contract. As you'll see later in this chapter, developing service contracts requires coding.
- ❑ Developing bindings. Recall that the binding of an endpoint specifies the communication protocols through which the endpoint communicates with the outside world. Windows Communication Foundation comes with predefined bindings that address common scenarios. As a result, there is no real need to develop custom bindings unless the existing bindings do not meet your application requirements. Developing custom bindings is beyond the scope of this book.

- ❑ Adding, updating, removing, and configuring endpoints. Can you do this in code? Sure, but you're highly discouraged from doing so. If you hard-code the service endpoints, every time there's a need to update or remove an endpoint, the code has to change. Therefore, adding and configuring the service endpoint should be done from the configuration file, unless there's a good reason to do it from the code.
- ❑ Adding behaviors. As the name suggests, a Windows Communication Foundation behavior is a component that enhances the runtime behavior of a service, an endpoint, or an operation. Some behaviors must be added in code, and some others must be added in the configuration file. You'll see an example of a behavior later in this chapter.

Every WCF service can expose one or more service contracts, each of which is a collection of operations. As mentioned earlier, every endpoint of a service exposes a particular service contract. A WCF service can expose the same service contract (the same set of operations) through more than one endpoint. This allows the clients of a service to access the same set of operations through different network addresses or different communication protocols.

Because a WCF service can expose more than one service contract, you need a way to uniquely identify a service contract among other service contracts of the service. The name and namespace of a service contract together uniquely identify the service contract among other service contracts. I show you later in this chapter how the name and namespace of a service contract are specified.

Developing a WCF Service Contract

A WCF service contract is a .NET interface or class decorated with the `ServiceContractAttribute` metadata attribute. Go back to your Visual Studio project and add a new WCF Service Library named `BugReportManagerService` to the `ProIIS7AspNetIntegProgCh12` solution. Take these steps to use the WCF Service Model to develop a service contract (see Listing 12-2):

1. Define a regular .NET interface named `IBugReportManagerServiceContract`, which exposes the following methods:
 - ❑ `AddBugReport`: Adds a new bug report. For example, a support staff may use this operation to report a new bug.
 - ❑ `GetBugReports`: Gets all the bug reports.
2. Annotate the `IBugReportManagerServiceContract` interface with the WCF Service Model's `ServiceContractAttribute` metadata attribute to designate the interface as a WCF service contract.
3. Annotate the `AddBugReport` and `GetBugReports` methods of the `IBugReportManagerServiceContract` interface with the `OperationContractAttribute` metadata attribute to designate these methods as the operations of the WCF service contract. Note that Windows Communication Foundation will *not* treat those methods of the interface that are not annotated with the `OperationContractAttribute` metadata attribute as part of the WCF service contract, and consequently those methods will not be available to the clients of the WCF service.

Listing 12-2: The `IBugReportManagerServiceContract` Service Contract

```
namespace ProIIS7AspNetIntegProgCh12
{
    using System.ServiceModel;

    [ServiceContract]
    public interface IBugReportManagerServiceContract
    {
        [OperationContract]
        void AddBugReport (BugReport bugReport);

        [OperationContract]
        BugReportCollection GetBugReports ();
    }
}
```

Now add a new source file named `IBugReportManagerServiceContract.cs` to the `BugReportManagerService` project and add the code shown in Listing 12-2 to this source file. As you can see, this source file contains the service contract.

Note that the `GetBugReports` operation of the `IBugReportManagerServiceContract` service contract returns a collection of type `BugReportCollection`, which is defined in Listing 12-3. Thanks to .NET generics, implementing a new collection type is as easy as implementing a new class that derives from one of the standard generic collections. Add a new source file named `BugReportCollection.cs` to the `BugReportManagerService` project and add the code shown in Listing 12-3 to this file.

Listing 12-3: The `BugReportCollection` Class

```
using System.Collections.Generic;

namespace ProIIS7AspNetIntegProgCh12
{
    public class BugReportCollection : List<BugReport> { }
}
```

As you can see from Listing 12-3, the `BugReportCollection` acts as a container for objects of type `BugReport`. Listing 12-4 presents the implementation of the `BugReport` class. A `BugReport` object contains the complete information about a given bug report. In this simple case, this information includes the sender, subject, and body of the bug report.

Listing 12-4: The `BugReport` Class

```
namespace ProIIS7AspNetIntegProgCh12
{
    public class BugReport
    {
        private string body;
        private string subject;
        private string sender;
    }
}
```

Listing 12-4: *(continued)*

```
public string Sender
{
    get { return sender; }
    set { sender = value; }
}

public string Subject
{
    get { return subject; }
    set { subject = value; }
}

public string Body
{
    get { return body; }
    set { body = value; }
}
}
```

WCF services and their clients exchange SOAP messages. Therefore, the data exchanged between a WCF service and its client must be serialized into XML on the sender side and deserialized from XML on the receiver side.

For example, those clients that access the `AddBugReport` operation of the bug report manager service at a particular service endpoint must pass a `BugReport` object that represents the bug report being added into this operation. In other words, the data exchanged between the WCF service and its clients in this case is a `BugReport` object.

Those clients that access the `GetBugReports` operation of the bug report manager service, on the other hand, receive a collection of `BugReport` objects. In other words, the data exchanged between the service and its clients in this case is a collection of `BugReport` objects.

Because the bug report manager service and its clients only exchange SOAP messages, the `BugReport` object that the client wants to pass into the `AddBugReport` operation must be serialized into XML before it is sent to the service. The service, on the other hand, must deserialize this XML back into a `BugReport` object before it is passed into the operation because the `AddBugReport` operation expects a `BugReport` object as its argument, not XML data.

In addition, the service must serialize the `BugReport` objects that the `GetBugReports` operation returns into XML before they're sent to the client that invoked this operation in the first place. The client, on the other hand, must deserialize this XML back into a collection of `BugReport` objects before they're passed into the client code that invoked the `GetBugReports` operation because the client code expects to receive a collection of `BugReport` objects from the `GetBugReports` operation, not XML data.

As you can see, serialization and deserialization play a significant role in Windows Communication Foundation. By default, Windows Communication Foundation uses an XML serializer named `DataContractSerializer` to serialize and deserialize the data exchanged between a WCF service and its clients.

Chapter 12: ASP.NET and WCF Integration in IIS 7

All you have to do as a developer is to:

1. Annotate the type of the data exchanged between a WCF service and its clients with the `DataContractAttribute` metadata attribute. In this case, the data is of type `BugReport`. As the highlighted portions of Listing 12-5 show, the `BugReport` class is annotated with the `DataContractAttribute` metadata attribute.
2. Annotate the desired properties of the type of the data exchanged between a WCF service and its clients with the `DataMemberAttribute` metadata attribute. As the highlighted portions of Listing 12-5 show, the `Sender`, `Subject`, and `Body` properties of the `BugReport` class are annotated with the `DataMemberAttribute` metadata attributes to instruct the `DataContractSerializer` that you want the serializer to serialize all these properties. Keep in mind that the serializer only serializes those properties that are annotated with the `DataMemberAttribute` metadata attribute.

Now add a new source file named `BugReport.cs` to the `BugReportManagerService` project and add the code shown in Listing 12-5 to this file.

Listing 12-5: The BugReport Class

```
using System.Runtime.Serialization;

namespace ProIIS7AspNetIntegProgCh12
{
    [DataContract]
    public class BugReport
    {
        private string body;
        private string subject;
        private string sender;

        [DataMember]
        public string Sender
        {
            get { return sender; }
            set { sender = value; }
        }

        [DataMember]
        public string Subject
        {
            get { return subject; }
            set { subject = value; }
        }

        [DataMember]
        public string Body
        {
            get { return body; }
            set { body = value; }
        }
    }
}
```

Implementing a WCF Service Contract

The previous section showed you how to develop a WCF service contract. As discussed earlier, the contract of an endpoint specifies the operations available to the clients of a WCF service at that endpoint. The next step is to provide the actual implementation for these operations. Thanks to the WCF Service Model, implementing a service contract is as easy as writing a .NET class that implements the interface that defines the service contract. Listing 12-6 presents the implementation of a .NET class named `BugReportManagerServiceImpl` that implements the `IBugReportManagerServiceContract` service contract.

Listing 12-6: The `BugReportManagerServiceImpl` Class

```
using System.ServiceModel;
using System.Collections;
using System.Data;
using System.ComponentModel;

namespace ProIIS7AspNetIntegProgCh12
{
    public class BugReportManagerServiceImpl :
        IBugReportManagerServiceContract
    {
        public void AddBugReport(BugReport bugReport)
        {
            BugReportManager.AddBugReport(bugReport.Sender, bugReport.Subject,
                                           bugReport.Body);
        }

        public BugReportCollection GetBugReports()
        {
            IEnumerable list = BugReportManager.GetBugReports();
            BugReportCollection bugReports = new BugReportCollection();
            BugReport bugReport;
            IEnumerator iter = list.GetEnumerator();
            bool firstIteration = true;
            PropertyDescriptorCollection pds = null;
            while (iter.MoveNext())
            {
                if (firstIteration)
                {
                    firstIteration = false;
                    pds = TypeDescriptor.GetProperties(iter.Current);
                }

                bugReport = new BugReport();
                bugReport.Sender = (string)pds["Sender"].GetValue(iter.Current);
                bugReport.Subject = (string)pds["Subject"].GetValue(iter.Current);
                bugReport.Body = (string)pds["Body"].GetValue(iter.Current);
                bugReports.Add(bugReport);
            }

            return bugReports;
        }
    }
}
```

Chapter 12: ASP.NET and WCF Integration in IIS 7

As Listing 12-6 shows, the `BugReportManagerServiceImpl` class's implementation of the `AddBugReport` and `GetBugReports` operations of the `IBugReportManagerServiceContract` service contract simply delegate to the `AddBugReport` and `GetBugReports` methods of the `BugReportManager` static class that Listing 12-1 implements. This should not come as a surprise because the WCF bug report manager service is a wrapper around the `BugReportManager` system to enable the system to communicate with the outside world.

Next, I walk you through the implementation of the `GetBugReports` method. This method begins by invoking the `GetBugReports` static method on the `BugReportManager` static class to return an `IEnumerable` collection that contains all the bug reports filed with the system:

```
IEnumerable list = BugReportManager.GetBugReports();
```

Next, it instantiates a `BugReportCollection`:

```
BugReportCollection bugReports = new BugReportCollection();
```

Then, it invokes the `GetEnumerator` method on the `IEnumerable` collection to return a reference to the `IEnumerator` object that knows how to enumerate the bug reports stored in this `IEnumerable` collection:

```
IEnumerator iter = list.GetEnumerator();
```

Then, it uses this `IEnumerator` object to iterate through the bug reports stored in the `IEnumerable` collection and takes these steps for each iteration. If it is the first iteration, it calls the `GetProperties` static method on a .NET class named `TypeDescriptor` to return a reference to a `PropertyDescriptorCollection` of `PropertyDescriptor` objects where each property descriptor object represents a piece of information about a bug report. Note that it stores this collection in a local variable named `pds` to make it available for the next iterations.

```
if (firstIteration)
{
    firstIteration = false;
    pds = TypeDescriptor.GetProperties(iter.Current);
}
```

Next, it creates a `BugReport` object:

```
bugReport = new BugReport();
```

Then, it uses the string "Sender" as an index into the previously discussed `PropertyDescriptorCollection` to return a reference to the `PropertyDescriptor` object that represents the sender part of a bug report. It invokes the `GetValue` method on this `PropertyDescriptor` object to return a string that contains the sender of the enumerated bug report and stores this information in the `Sender` property of the `BugReport` object:

```
bugReport.Sender = (string)pds["Sender"].GetValue(iter.Current);
```

Next, it repeats the same process to retrieve the subject and body of the enumerated bug report and stores them in `Subject` and `Body` properties of the `BugReport` object, respectively:

```
bugReport.Subject = (string)pds["Subject"].GetValue(iter.Current);
bugReport.Body = (string)pds["Body"].GetValue(iter.Current);
```

Then, it adds this `BugReport` object to the `BugReportCollection`:

```
bugReports.Add(bugReport);
```

Add a new source file named `BugReportManagerServiceImpl.cs` to the `BugReportManagerService` WCF Service Library project and add the code shown in Listing 12-6 to this file. Next, add a reference to the assembly into which the `BugReportManager` Class Library project was compiled in the previous section. As you would expect, this assembly is located in the `bin` directory of the `BugReportManager` Class Library project. Finally, go ahead and build the `BugReportManagerService` WCF Service Library project.

Hosting a WCF Service

You must host a WCF service in an application domain to make it available to the outside world. Therefore, one of the responsibilities of a WCF service developer is to take the necessary steps to provide support for loading the service in an application domain. These steps depend on where you want to host your WCF service. You can host your service in any .NET application, because any .NET application can load a service into an application domain. If you decide to load your service into a custom .NET application such as a Console Application, you must also develop this application.

In this case, you'll delegate the responsibility of loading the service into an application domain to IIS 7. This will allow you to take full advantage of the great features of IIS 7 that we've been talking about throughout this book. Recall that IIS 7 loads each Web application into a separate application domain to ensure that the execution of one application does not interfere with the execution of another application.

This means that all the assemblies of a given Web application including the assembly that contains your WCF service contracts and their implementations are loaded into the same application domain. This allows you to treat these WCF services as you would any other component of your Web applications, such as the ASP.NET pages. This also means that the same ASP.NET compilation system that compiles the rest of your Web application also compiles your WCF service contracts and their implementations. Such deep integration of your WCF services with the rest of your Web applications opens up great programming opportunities that would not be possible otherwise. We'll take a look at an example of this case later in this chapter.

Now go ahead and add a new WCF Service Web site named `BugReportManagerServiceHost` to the `ProIIS7AspNetIntegProgCh12` solution. This Web site will automatically include a `.svc` file and a couple of `.cs` files. Delete both `.cs` files, change the name of `.svc` file to `BugReportManagerService.svc`, remove all the content of this file, and add the following directive to this file:

```
<%@ ServiceHost
    Service="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl" %>
```

As you can see, this directive assigns a string containing the fully qualified name of the `BugReportManagerServiceImpl` class, which implements the WCF `IBugReportManagerServiceContract` service contract, to the `Service` attribute of the `ServiceHost` directive.

When the first request for the `BugReportManagerService.svc` resource arrives in IIS 7, IIS 7 checks with the `handlers` section of the configuration files in the appropriate configuration hierarchy to determine

Chapter 12: ASP.NET and WCF Integration in IIS 7

whether a handler has been registered for handling requests for resources with the file extension `.svc`. If you open the `applicationHost.config` file in your favorite editor, you should see the following section in this file:

```
<configuration>
  <location path="" overrideMode="Allow">
    <system.webServer>
      <handlers accessPolicy="Script, Read">
        <add path="*.svc" name="svc-Integrated" verb="*"
          precondition="integratedMode"
          type="System.ServiceModel.Activation.HttpHandler, System.ServiceModel,
            Version=3.0.0.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089" />
      </handlers>
    </system.webServer>
  </location>
</configuration>
```

This section registers the `System.ServiceModel.Activation.HttpHandler` HTTP handler for handling requests for resources with the file extension `.svc`.

IIS 7 then passes this first request to the `System.ServiceModel.Activation.HttpHandler` HTTP handler for processing. This HTTP handler under the hood invokes a method named `Open` on an object of type `ServiceHost`. The `Open` method in turn triggers the process through which Windows Communication Foundation examines your service model and automatically creates the runtime components needed to implement your service model.

If you visit the following standard directory on your machine:

```
%windir%\Microsoft.NET\Framework\v2.0.50727\CONFIG
```

and open the `web.config` file located in this directory in your favorite editor, you should see the following section in this file:

```
<configuration>
  <system.web>
    <compilation>
      <buildProviders>
        <add extension=".svc"
          type="System.ServiceModel.Activation.ServiceBuildProvider,
            System.ServiceModel, Version=3.0.0.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089" />
      </buildProviders>
    </compilation>
  </system.web>
</configuration>
```

As you can see, this section registers the `System.ServiceModel.Activation.ServiceBuildProvider` as the build provider for resources with the file extension `.svc`. The main responsibility of a build provider is to parse its associated file, generate the appropriate source code from the parsed information, and add this source code to the appropriate assembly builder. The assembly builder then builds this source code together with source code contributed by other build providers into a single assembly, which is then loaded into the current application domain.

In this case, the `System.ServiceModel.Activation.ServiceBuildProvider` build provider parses the following directive in the `BugReportManagerService.svc` file to retrieve the value of the `Service` attribute, which is nothing but the fully qualified name of the `BugReportManagerServiceImpl` class that implements the WCF `IBugReportManagerServiceContract` service contract:

```
<%@ ServiceHost
    Service="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl" %>
```

The `ServiceBuildProvider` then adds a reference to the assembly that contains the `BugReportManagerServiceImpl` class and passes it on to the assembly builder, causing this assembly to load into the current application domain.

Administrative Tasks

So far, you've learned how to define and how to implement a WCF service contract. As you saw, these two tasks require coding. This section covers the administrative tasks, which do not require coding and are performed from the configuration files.

The first administrative task involves adding a `<system.serviceModel>` element as the child element of the `<configuration>` document element of the `web.config` file of the `BugReportManagerServiceHost` WCF Service Web site if it hasn't already been added. Recall that `BugReportManagerServiceHost` hosts the bug report manager service.

```
<configuration>
  <system.serviceModel>
    . . .
  </system.serviceModel>
</configuration>
```

The `<system.serviceModel>` section of the `web.config` file contains all the WCF-related configuration settings.

The second administrative task involves adding a `<services>` element as the child element of the `<system.serviceModel>` element if it hasn't already been added. As the name suggests, the `<services>` element will contain the configuration settings specific to the WCF services.

```
<configuration>
  <system.serviceModel>
    <services>
      . . .
    </services>
  </system.serviceModel>
</configuration>
```

Note that more than one service can be configured in the same `web.config` file, which means that more than one service could run in the same application.

The third administrative task is to add one or more `<service>` elements as the child elements of the `<services>` element, and assign a string containing the fully qualified name of the class that implements the desired WCF service contract(s) to the `name` attribute of each `<service>` element.

Chapter 12: ASP.NET and WCF Integration in IIS 7

In this case, assign a string containing the fully qualified name of the `BugReportManagerServiceImpl` class as the value of the `name` attribute, because this class implements the `IBugReportManagerService` service contract:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl">
        . . .
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

In this example, the `BugReportManagerServiceImpl` class implements a single service contract, that is, the `IBugReportManagerServiceContract` service contract. In other words, the bug report manager service supports a single service contract. In general, a WCF service may support more than one WCF service contract. In other words, the class whose fully qualified name is given by the value of the `name` attribute of the `<service>` element may implement more than one WCF service contract.

The final administrative task is to add one or more `<endpoint>` elements as the child elements of each `<service>` element and set the values of the `address`, `binding`, and `contract` attributes of these `<endpoint>` elements:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl">
        <endpoint
          address=""
          binding="basicHttpBinding"
          contract="ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Each `<endpoint>` child element of a `<service>` element represents a particular endpoint of the WCF service. In this case, the `<endpoint>` child element represents an endpoint of the WCF bug report manager service.

Note that the `binding` attribute of this `<endpoint>` element is set to the string `"basicHttpBinding"`. As discussed earlier, the WCF Service Model comes with standard set of bindings that you can use in your own Web applications. Each standard binding is a class that directly or indirectly inherits from the `Binding` base class. Each class specifies a set of communication protocols that contains at least a transport protocol. If a binding does not explicitly specify a message-encoding protocol, the transport protocol will use its default transport-protocol specific message-encoding protocol to encode the SOAP messages exchanged between the endpoint and its clients.

The value of the `binding` attribute of an `<endpoint>` element is nothing but the name of a binding class in camel casing, in compliance with the camel casing convention used in configuration files. For example, in this case, the value of the `binding` attribute is the name of the `BasicHttpBinding` class in camel

casing. The `BasicHttpBinding` specifies a set of two protocols: an HTTP transport protocol and a text message-encoding protocol for encoding SOAP messages in text format. The `BasicHttpBinding` is configured to meet the WS-I Basic Profile Specification 1.1 to promote maximum interoperability.

Also note that the value of the `contract` attribute of the `<endpoint>` element, in this case, is set to the string `"ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract"`. This means that all the operations defined in the `IBugReportManagerServiceContract` service contract are available to the clients of the bug report manager service at this endpoint. Recall that this service contract defines two operations named `AddBugReport` and `GetBugReports`.

As you can see, the value of the `address` attribute on this `<endpoint>` element has not been set. This is because when a WCF service is hosted in IIS 7, Windows Communication Foundation picks up the address of the endpoint from the virtual directory of the `.svc` file associated with the service.

Now go ahead and add the following configuration fragment to the `web.config` file of the `BugReportManagerServiceHost` WCF Service Web site:

```
<system.serviceModel>
  <services>
    <service name="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl">
      <endpoint
        address=""
        binding="basicHttpBinding"
        contract="ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract" />
    </service>
  </services>
</system.serviceModel>
```

Add a reference to the assembly into which the `BugReportManagerService` WCF Service Library project was built in the previous section to the `BugReportManagerServiceHost` WCF Service Web site. This assembly is located in the `bin` directory of the `BugReportManagerService` WCF Service Library project. Now use the `http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc` URL to access the `BugReportManagerService.svc` file from a browser. You should get the result shown in Figure 12-2.



Figure 12-2

Chapter 12: ASP.NET and WCF Integration in IIS 7

Notice the message shown in bold in Figure 12-2. According to this message, the metadata publishing for this service is disabled. This is done for security purposes. Your WCF service is not available to its clients until you explicitly turn on the metadata publishing for your service. This is where WCF service behaviors come into play.

A service behavior is a component that attaches to a service and extends its capabilities. Windows Communication Foundation ships with a service behavior named `ServiceMetadataBehavior` that you can attach to your WCF service to enable your service to generate and return a WSDL document or metadata to a client that asks for this document. The WSDL document of your WCF service provides its clients with the complete recipe for generating SOAP messages that they need to send to your service and for consuming SOAP messages that they receive from your service.

Can you attach the `ServiceMetadataBehavior` service behavior to your service from within your code? Sure, but you're discouraged from doing so, because hard-coding this service behavior would require code changes every time there's a need to turn on or off this metadata publishing capability of your service.

The `ServiceMetadataBehavior` service behavior is one of those behaviors that should be attached to a WCF service through a configuration file. Take these steps to attach the `ServiceMetadataBehavior` service behavior to your WCF service from a configuration file:

1. Add a `<behaviors>` element as the child element of the `<system.serviceModel>` element as shown in the highlighted portion of the following listing if it hasn't already been added:

```
<system.serviceModel>
  <services>
    . . .
  </services>
  <behaviors>
    . . .
  </behaviors>
</system.serviceModel>
```

As the name suggests, the `<behaviors>` element will contain behaviors that will be attached to the WCF services, endpoints, and operations in the current application.

2. Add a `<serviceBehaviors>` element as the child element of the `<behaviors>` element as shown in highlighted portion of the following listing if it hasn't already been added:

```
<system.serviceModel>
  <services>
    . . .
  </services>
  <behaviors>
    <serviceBehaviors>
      . . .
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

As the name implies, the `<serviceBehaviors>` element will contain behaviors that will be attached to WCF services in the current application.

3. Add a `<behavior>` child element to the `<serviceBehaviors>` element and assign a value to its name attribute as shown in the highlighted portion of the following configuration fragment. You can assign any value you want to this attribute as long as it is unique and does not violate the typical XML attribute naming rules.

```
<system.serviceModel>
  <services>
    . . .
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="BugReportManagerService_ServiceMetadata">
        . . .
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

4. Add an element with the same name as the behavior as the child element of the `<behavior>` element and assign the appropriate values to the appropriate attributes of this element. The highlighted portion of the following configuration fragment adds an element named `<serviceMetadata>` as the child element of the `<behavior>` element and sets its `httpGetEnabled` attribute to `true` to turn on the metadata publishing capabilities of the service to which the behavior attaches. As the name suggests, setting the `httpGetEnabled` attribute to `true` enables the service to send a WSDL document or metadata in response to a GET HTTP request for this document.

Notice that the name of this child element follows the camel casing naming convention of the .NET configuration files:

```
<system.serviceModel>
  <services>
    . . .
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="BugReportManagerService_ServiceMetadata">
        <serviceMetadata httpGetEnabled="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

5. Add an attribute named `behaviorConfiguration` to the `<service>` element that represents the WCF service to which the specified behavior is to be attached. Assign the value of the name

Chapter 12: ASP.NET and WCF Integration in IIS 7

attribute of the <behavior> element to this attribute as shown in the boldfaced portions of the following configuration fragment:

```
<system.serviceModel>
  <services>
    <service name="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl"
      behaviorConfiguration="BugReportManagerService_ServiceMetadata">
      <endpoint address="" binding="basicHttpBinding"
        contract="ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract" />
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="BugReportManagerService_ServiceMetadata">
        <serviceMetadata httpGetEnabled="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Now add this configuration fragment to the web.config file of the BugReportManagerServiceHost WCF Service Web site. Now if you access the BugReportManagerService.svc file at <http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc>, you should get the page shown in Figure 12-3.

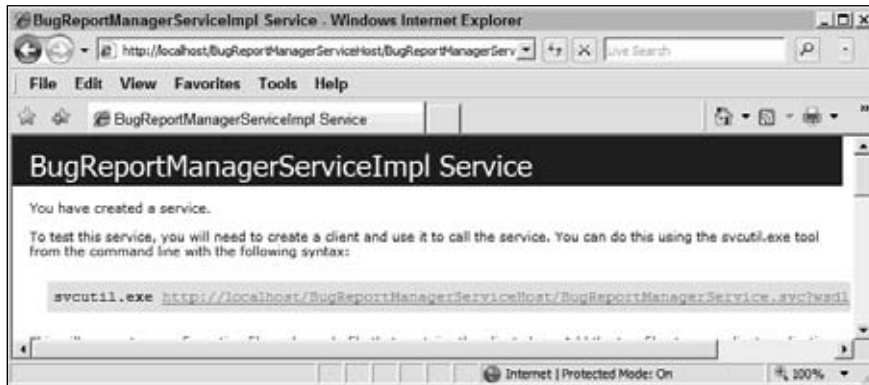


Figure 12-3

As you can see, this time around the page contains a link to the WSDL document or metadata that describes the bug report manager service. If you click this link, it will take you to the page shown in Figure 12-4, which displays the WSDL document.

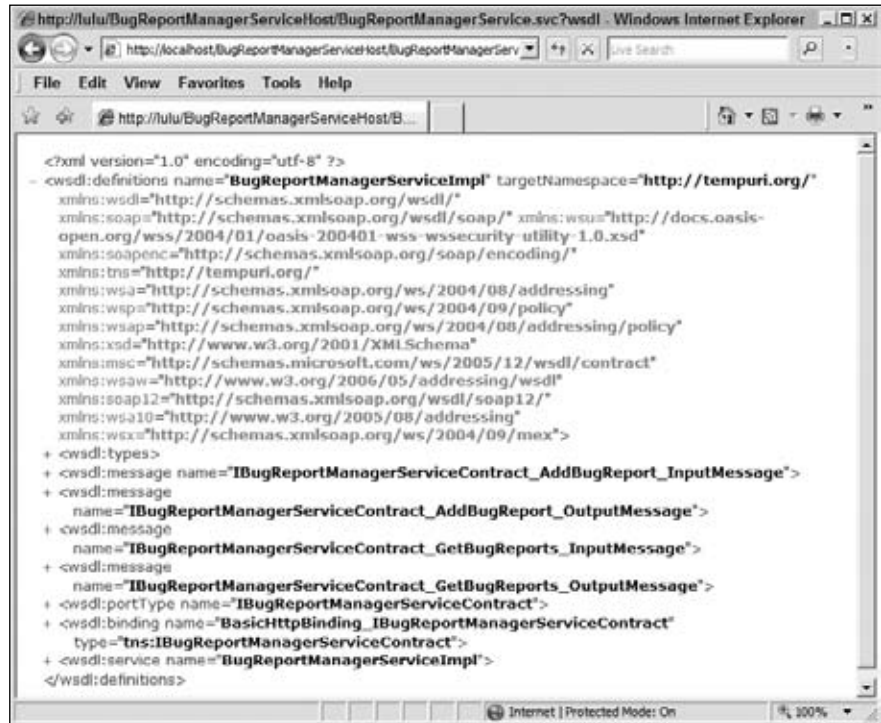


Figure 12-4

Developing a Windows Communication Foundation Client

In this section, you develop a Windows Communication Foundation client that will communicate with your bug report manager service through the endpoint you defined in the previous sections. When it comes to developing a WCF client, you have several options. I cover three of these options in the following sections:

- ☐ Adding a Web reference
- ☐ Using the `svcutil.exe` tool
- ☐ Imperative approach

You'll use each option to create a separate client application.

Adding a Web Reference

Add a new Web site named `BugReportManagerServiceClientWebRef` to the `ProIIS7AspNetIntegProgCh12` solution. Right-click the `BugReportManagerServiceClientWebRef`

Chapter 12: ASP.NET and WCF Integration in IIS 7

Web site in Solution Explorer and select the Add Web Reference option to launch the Add Web Reference dialog. Use the following URL to navigate to the page shown in Figure 12-3:

`http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc`

Enter **ProIIS7AspNetIntegProgCh12** in the “Web reference name” textbox. Now click the Add Reference button. This will automatically do the following:

- ☐ Download the WSDL document for the bug report manager service from the specified URL.
- ☐ Generate a class named `BugReportManagerServiceImpl`. This class is known as the *proxy class* because it acts as a proxy for the server-side `BugReportManagerServiceImpl` class. Recall that this server-side class implements the `IBugReportManagerServiceContract` service contract. Because the client-side `BugReportManagerServiceImpl` class acts as a proxy for the server-side `BugReportManagerServiceImpl` class, it exposes methods with the same signatures as the methods of the server-side class. As such, it exposes two methods named `AddBugReport` and `GetBugReports`, where the former takes an object of type `BugReport` as its argument just like its server-side counterpart, and the latter returns a collection of `BugReport` objects just like its server-side counterpart.
- ☐ Generate the appropriate configuration settings.

Now you’re ready to use the `BugReportManagerServiceImpl` proxy class to communicate with the bug report manager service. Now add a new Web Form named `Default.aspx` to the `BugReportManagerServiceClientWebRef` Web site and add the code shown in Listing 12-7 to `Default.aspx` file.

Listing 12-7: The Default.aspx File

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:GridView ID="GridView1" DataSourceID="MySource" runat="server"
            BackColor="LightGoldenrodYellow" BorderColor="Tan" BorderWidth="1px"
            CellPadding="2" ForeColor="Black" GridLines="None">
            <HeaderStyle BackColor="Tan" Font-Bold="True" />
            <AlternatingRowStyle BackColor="PaleGoldenrod" />
        </asp:GridView>
        <br />
        <asp:ObjectDataSource runat="server" ID="MySource"
            TypeName="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl"
            SelectMethod="GetBugReports" />
    </form>
</body>
</html>
```

The Web page shown in Listing 12-7 consists of a GridView server control, which is bound to an ObjectDataSource data source control. The ObjectDataSource data source control exposes two important properties named `TypeName` and `SelectMethod`. You must set the value of the `TypeName` property to the fully qualified name of the type whose method you want the data source control to invoke. You must set the value of the `SelectMethod` property to the name of the method being invoked.

In your case, you want to have the ObjectDataSource data source control invoke the `GetBugReports` method of the `BugReportManagerServiceImpl` proxy class. Therefore, you need to assign the fully qualified name of this proxy class (including its namespace) to the `TypeName` property of the data source control and the method name (`GetBugReports`) to the `SelectMethod` property.

The beauty of the ObjectDataSource data source control is that it allows you to invoke the `GetBugReports` method of the `BugReportManagerServiceImpl` proxy class without writing a single line of imperative code. It is all done declaratively in the `.aspx` file.

Using the svcutil.exe Tool

Windows Communication Foundation ships with a tool named `svcutil.exe` that automatically downloads the WSDL document from a specified URL, generates the code for the proxy class and stores it in a specified file, and generates the appropriate configuration file with the appropriate settings. When you run this command

```
svcutil.exe /out:BugReportManagerServiceClient.cs /config:Web.config  
http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc
```

the `svcutil.exe` tool will automatically do the following:

- ☐ Download the metadata document (WSDL document) for the bug report manager service from the `http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc` URL.
- ☐ Generate the code for a proxy class named `BugReportManagerServiceContractClient` and store the code in the specified `BugReportManagerServiceClient.cs` file in the current directory.
- ☐ Generate the required configuration settings and store them in the specified `web.config` file in the current directory.

If you check out the directory where you ran the `svcutil.exe` tool, you should see the `BugReportManagerServiceClient.cs` and `web.config` files. Add a new Web site named `BugReportManagerServiceClientSvcUtil` to the `ProIIS7AspNetIntegProgCh12` solution. Next, add the `App_Code` directory to this Web site. Right-click `App_Code` and select `Add Existing Item` to launch the `Add Existing Item` dialog. Navigate to the directory where you ran the `svcutil.exe` tool to add the `BugReportManagerServiceClient.cs` file to the `App_Code` directory. Now, right-click `BugReportManagerServiceClientSvcUtil` and select `Add Existing Item` to launch the `Add Existing Item` dialog. Navigate to the directory where you ran the `svcutil.exe` tool to add the `web.config` file to the `BugReportManagerServiceClientSvcUtil` Web site. Now go back to the Solution Explorer panel of Visual Studio. Drag the `Default.aspx` file from the `BugReportManagerServiceClientWebRef` Web site to the `BugReportManagerServiceClientSvcUtil` Web site. This will automatically copy this file into the `BugReportManagerServiceClientSvcUtil` Web site. Next, change the value of the `TypeName`

Chapter 12: ASP.NET and WCF Integration in IIS 7

attribute on the `<asp:ObjectDataSource>` in the `Default.aspx` file to `BugReportManagerServiceContractClient`, which is the name of the type of the proxy class that `svcutil.exe` generates, as shown in the following code fragment:

```
<asp:ObjectDataSource runat="server" ID="MySource"
  TypeName="BugReportManagerServiceContractClient"
  SelectMethod="GetBugReports" />
```

Next, I take a look at the contents of the `BugReportManagerServiceClient.cs` and `web.config` files. Listing 12-8 presents the content of the `BugReportManagerServiceClient.cs` file.

Listing 12-8: The Content of the `BugReportManagerServiceClient.cs` File

```
namespace ProIIS7AspNetIntegProgCh12
{
    using System.Runtime.Serialization;

    [DataContract]
    public partial class BugReport : object, IExtensibleDataObject
    {
        private ExtensionDataObject extensionDataField;

        private string BodyField;
        private string SenderField;
        private string SubjectField;

        public ExtensionDataObject ExtensionData
        {
            get { return this.extensionDataField; }
            set { this.extensionDataField = value; }
        }

        [DataMember]
        public string Sender
        {
            get { return this.SenderField; }
            set { this.SenderField = value; }
        }

        [DataMember]
        public string Subject
        {
            get { return this.SubjectField; }
            set { this.SubjectField = value; }
        }

        [DataMember]
        public string Body
        {
            get { return this.BodyField; }
            set { this.BodyField = value; }
        }
    }
}
```

Listing 12-8: *(continued)*

```

    }

    [ServiceContract(ConfigurationName = "IBugReportManagerServiceContract")]
    public interface IBugReportManagerServiceContract
    {
        [OperationContract]
        Action = "http://tempuri.org/IBugReportManagerServiceContract/AddBugReport",
        ReplyAction =
            "http://tempuri.org/IBugReportManagerServiceContract/AddBugReportResponse")
        void AddBugReport(BugReport bugReport);

        [OperationContract]
        Action = "http://tempuri.org/IBugReportManagerServiceContract/GetBugReports",
        ReplyAction =
            "http://tempuri.org/IBugReportManagerServiceContract/GetBugReportsResponse")
        BugReport[] GetBugReports();
    }

    public partial class BugReportManagerServiceContractClient :
        ClientBase<IBugReportManagerServiceContract>,
        IBugReportManagerServiceContract
    {

        public BugReportManagerServiceContractClient() { }

        public BugReportManagerServiceContractClient(string endpointConfigurationName)
            : base(endpointConfigurationName)
        {
        }

        public BugReportManagerServiceContractClient(string endpointConfigurationName,
            string remoteAddress)
            : base(endpointConfigurationName, remoteAddress)
        {
        }

        public BugReportManagerServiceContractClient(string endpointConfigurationName,
            EndpointAddress remoteAddress)
            : base(endpointConfigurationName, remoteAddress)
        {
        }

        public BugReportManagerServiceContractClient(Binding binding,
            EndpointAddress remoteAddress)
            : base(binding, remoteAddress)
        {
        }

        public void AddBugReport(BugReport bugReport)
        {
            base.Channel.AddBugReport(bugReport);
        }
    }

```

(Continued)

Listing 12-8: *(continued)*

```
public BugReport[] GetBugReports()
{
    return base.Channel.GetBugReports();
}
```

As Listing 12-8 shows, the `svcutil.exe` tool parses the WSDL document and generates the code for the following types:

- ❑ **BugReport:** This is the same `BugReport` type defined on the server side. Note that this client-side `BugReport` type and its properties are annotated with the same `DataContract` and `DataMember` metadata attributes as its `BugReport` server-side counterpart.
- ❑ **IBugReportManagerServiceContract:** This is the same `IBugReportManagerServiceContract` type defined on the server side. Recall that this type defines the service contract of your bug report manager service. Also note that the methods of this client-side `IBugReportManagerServiceContract` service contract are annotated with the same `OperationContract` attributes as its server-side counterpart. Note that the `Action` and `ReplyAction` properties these `OperationContract` metadata attributes are set. The client uses the values of these two properties to keep track of which incoming response message corresponds to which outgoing request message.
- ❑ **BugReportManagerServiceContractClient:** This class acts as the proxy for the server-side `BugReportManagerServiceImpl` class. Recall that the server-side `BugReportManagerServiceImpl` class implements the `IBugReportManagerServiceContract` service contract of the bug report manager service. Because the client-side `BugReportManagerServiceContractClient` class is a proxy for the server-side `BugReportManagerServiceImpl`, it also implements the same contract, that is, the `IBugReportManagerServiceContract`, as can be seen from Listing 12-8.

Besides implementing the service contract that its server-side counterpart implements, a proxy class such as `BugReportManagerServiceContractClient` also inherits from a generic class named `ClientBase<ServiceContract>` where the `ServiceContract` stands for the service contract that the proxy class implements. In this case, the `BugReportManagerServiceContractClient` proxy class inherits the `ClientBase<IBugReportManagerServiceContract>` base class. The `ClientBase<ServiceContract>` generic class encapsulates the logic that allows the proxy class to communicate with the back-end WCF service.

Next, I take a look at the content of the `web.config` file that the `svcutil.exe` tool generates, as shown in Listing 12-9.

Listing 12-9: The web.config File

```
<?xml version="1.0"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
```

Listing 12-9: *(continued)*

```

    <binding name="BasicHttpBinding_IBugReportManagerServiceContract"
      closeTimeout="00:01:00" openTimeout="00:01:00" receiveTimeout="00:10:00"
      sendTimeout="00:01:00" allowCookies="false" bypassProxyOnLocal="false"
      hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
      maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
      messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
      useDefaultWebProxy="true">
      <readerQuotas maxDepth="32" maxStringContentLength="8192"
        maxArrayLength="16384" maxBytesPerRead="4096"
        maxNameTableCharCount="16384"/>
      <security mode="None">
        <transport clientCredentialType="None"
          proxyCredentialType="None" realm=""/>
        <message clientCredentialType="UserName" algorithmSuite="Default"/>
      </security>
    </binding>
  </basicHttpBinding>
</bindings>

<client>
  <endpoint
    address=
      "http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc"
    binding="basicHttpBinding"
    bindingConfiguration="BasicHttpBinding_IBugReportManagerServiceContract"
    contract="IBugReportManagerServiceContract"
    name="BasicHttpBinding_IBugReportManagerServiceContract"/>
</client>

</system.serviceModel>
</configuration>

```

As you can see from Listing 12-9, this `web.config` file contains a `<system.serviceModel>` element that has two child elements named `<bindings>` and `<client>`. The `<bindings>` element contains one or more binding elements that each specifies a particular binding. Recall that each binding is a .NET class that directly or indirectly inherits from the `Binding` base class. In this case, the `<bindings>` element contains a single binding element named `<basicHttpBinding>`, which represents the standard `BasicHttpBinding` binding, which ships with Windows Communication Foundation. Note that this `<basicHttpBinding>` element contains a `<binding>` child element that specifies the values of the properties of the `BasicHttpBinding` binding.

As shown in Listing 12-9, the `<client>` element contains an `<endpoint>` element that represents the service endpoint to which the client communications will be directed. Note that the `address`, `binding`, and `contract` attributes on this `<endpoint>` element are set to the `address`, `binding`, and `contract` of its associated service endpoint. Also note that the `<endpoint>` element exposes an attribute named `bindingConfiguration` whose value is set to the value of the `name` attribute of the `<binding>` child element of the `<basicHttpBinding>` element that specifies the values of the properties of the `BasicHttpBinding` binding. This instructs the client to use the property values specified by this `<binding>` child element. Another point of interest here is the `name` attribute of the `<endpoint>` element. This attribute uniquely identifies its associated `<endpoint>` element among other endpoint

elements. In this case, the `<client>` element contains a single `<endpoint>` element because the bug report manager service exposes a single endpoint. However, in general, a WCF service may expose several endpoints, which means that the `<client>` element may contain several `<endpoint>` elements, one for each service endpoint.

Imperative Approach

The previous section discussed the C# code that the `svcutil.exe` tool generates. Recall that this tool generates the code for three types of classes:

- ❑ **Data classes:** The `BugReport` class is an example of a data class. Instances of these data classes represent the data exchanged between a WCF service and its clients. As discussed earlier, the `DataContractSerializer` serializes these instances into XML on the sender side (be it the WCF service or its client) and deserializes these instances from XML on the receiver side.

As Listing 12-8 shows, the `svcutil.exe` tool parses the metadata document and generates the code for these data classes. Because these client-side data classes are the same as their server-side counterparts, you should be able to manually program these data classes yourself without using the `svcutil.exe` tool.

- ❑ **Service contracts:** The `IBugReportManagerServiceContract` interface shown in Listing 12-8 is an example of a service contract. Again, because these client-side service contracts are the same as their server-side counterparts, you can easily program them without using the `svcutil.exe` tool.
- ❑ **Proxy classes:** The `BugReportManagerServiceContractClient` class is an example of a proxy class. As you can see from Listing 12-8, a proxy class is a class that
 - ❑ Implements a service contract. For example, the `BugReportManagerServiceContractClient` proxy class implements the `IBugReportManagerServiceContract` service contract.
 - ❑ Inherits from `ClientBase<ServiceContract>`. For example, the `BugReportManagerServiceContractClient` proxy class inherits from `ClientBase<IBugReportManagerServiceContract>`.
 - ❑ Its implementation of the operations of the service contract delegates to the corresponding methods of the `Channel` property of the `ClientBase<ServiceContract>`. For example, the `BugReportManagerServiceContractClient` proxy's implementations of the `AddBugReport` and `GetBugReports` operations of the `IBugReportManagerServiceContract` service contract respectively delegate to the `Channel` property of the `AddBugReport` and `GetBugReports` methods of the `ClientBase<IBugReportManagerServiceContract>` class as can be seen from the following excerpt from Listing 12-8:

```
public void AddBugReport(BugReport bugReport)
{
    base.Channel.AddBugReport(bugReport);
}

public BugReport[] GetBugReports()
{
    return base.Channel.GetBugReports();
}
```

- ❑ Exposes five constructors as follows:
 - ❑ Default constructor: Thanks to this constructor you could use the `ObjectDataSource` data source control in `Default.aspx` to invoke the `GetBugReports` method of the `BugReportManagerServiceContractClient` class. Under the hood, the `ObjectDataSource` data source control uses the type information assigned to its `TypeName` property to instantiate an instance of the specified type, which is the `BugReportManagerServiceContractClient` class in this case.

```
public BugReportManagerServiceContractClient() { }
```

- ❑ A constructor that takes a string that contains the endpoint's configuration name. Recall from the previous section that the `<endpoint>` subelement of the `<client>` element exposes an attribute named `name` that uniquely identifies the associated endpoint among other endpoints. This constructor requires you to pass in the value of the `name` attribute of the `<endpoint>` element that represents an endpoint. You'll see an example of this later in this section.

```
public BugReportManagerServiceContractClient(string endpointConfigurationName)
                                           : base(endpointConfigurationName)
{
}
```

- ❑ A constructor that takes two string parameters. The first parameter is the endpoint's configuration name and the second parameter is the network address of the endpoint:

```
public BugReportManagerServiceContractClient(string endpointConfigurationName,
                                           string remoteAddress)
                                           : base(endpointConfigurationName, remoteAddress)
{
}
```

- ❑ A constructor that takes two parameters. The first parameter is a string that contains the endpoint's configuration name and the second parameter is a `EndpointAddress` object that represents the endpoint's address:

```
public BugReportManagerServiceContractClient(string endpointConfigurationName,
                                           EndpointAddress remoteAddress)
                                           : base(endpointConfigurationName, remoteAddress)
{
}
```

- ❑ A constructor that takes two parameters. The first parameter is a `Binding` object that represents the binding of the endpoint and the second parameter is an `EndpointAddress` object that represents the endpoint's address:

```
public BugReportManagerServiceContractClient(Binding binding,
                                           EndpointAddress remoteAddress)
                                           : base(binding, remoteAddress)
{
}
```

Chapter 12: ASP.NET and WCF Integration in IIS 7

You can program all these classes and their methods without using `svcutil.exe` as follows. Add a new Web site named `BugReportManagerServiceClientImperative` to the `ProIIS7AspNetIntegProgCh12` solution. Go back to the Solution Explorer panel of Visual Studio. Drag the `App_Code` directory and `Default.aspx` and `web.config` files from the `BugReportManagerServiceClientSvcUtil` Web site to the `BugReportManagerServiceClientImperative` Web site. This automatically adds a new `App_Code` directory to the `BugReportManagerServiceClientImperative` Web site and copies the `BugReportManagerServiceClient.cs` file into this directory. Basically you're pretending that you've written the content of the `BugReportManagerServiceClient.cs` file yourself instead of using the `svcutil.exe` tool.

Next, go ahead and change the value of the `TypeName` attribute on the `<asp:ObjectDataSource>` in `Default.aspx` file to `ProIIS7AspNetIntegProgCh12.BugReports` as shown in the following code fragment:

```
<asp:ObjectDataSource runat="server" ID="MySource"
    TypeName="ProIIS7AspNetIntegProgCh12.BugReports"
    SelectMethod="GetBugReports" />
```

As you can see, this time around you want the `ObjectDataSource` data source control to invoke the `GetBugReports` method of a new class named `BugReports`, which you've written manually, instead of the proxy class. Delegating to a different class enables you to use any of the five constructors of the proxy class you want. Listing 12-10 presents the implementation of the `BugReports` class.

Listing 12-10: The BugReports Class

```
namespace ProIIS7AspNetIntegProgCh12
{
    using System.ServiceModel;

    public class BugReports
    {
        public static BugReport[] GetBugReports()
        {
            BugReportManagerServiceContractClient proxy =
                new BugReportManagerServiceContractClient();
            return proxy.GetBugReports();
        }

        public static BugReport[] GetBugReports2()
        {
            BugReportManagerServiceContractClient proxy =
                new BugReportManagerServiceContractClient(
                    "BasicHttpBinding_IBugReportManagerServiceContract");
            return proxy.GetBugReports();
        }

        public static BugReport[] GetBugReports3()
        {
            BasicHttpBinding binding =
                new BasicHttpBinding("BasicHttpBinding_IBugReportManagerServiceContract");
            EndpointAddress address =
                new EndpointAddress(
```

Listing 12-10: *(continued)*

```

        "http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc");
        BugReportManagerServiceContractClient proxy =
            new BugReportManagerServiceContractClient(binding, address);
        return proxy.GetBugReports();
    }
}

```

The `BugReports` class presents three versions of the `GetBugReports` method as follows:

- ❑ `GetBugReports`: This version of the method uses the default constructor of the `BugReportManagerServiceContractClient` proxy class. This is exactly what the `ObjectDataSource` data source control did when you asked the control to directly invoke the `GetBugReports` method on the proxy class itself.
- ❑ `GetBugReports2`: This version of the method uses the proxy constructor that takes the value of the `name` attribute of the `<endpoint>` subelement of the `<client>` element. Recall that this value uniquely identifies an `<endpoint>` subelement among other `<endpoint>` subelements of the `<client>` element. Keep in mind that each `<endpoint>` subelement represents a particular service endpoint with a particular address, binding, and contract. In other words, the `GetBugReports2` method allows you to communicate with the WCF service through any desired endpoint.
- ❑ `GetBugReports3`: This version of the method uses the proxy constructor that takes two parameters. The first parameter references the `Binding` object that represents the binding of the endpoint. The second parameter references the `EndpointAddress` that represents the address of the endpoint. Note that `GetBugReports3` instantiates a `BasicHttpBinding` object, passing in the string that contains the value of the `name` attribute on the `<binding>` subelement of the `<basicHttpBinding>` subelement of the `<bindings>` element in the configuration file shown in Listing 12-9. This string instructs the constructor of the `BasicHttpBinding` class to use the values specified by the `<binding>` element with the specified `name` attribute value to initialize the `BasicHttpBinding` object.

Switching from one version of the `GetBugReports` method to another is as simple as setting the value of the `SelectMethod` property of the `ObjectDataSource` data source control to the name of the desired version of the method. For example, the following instructs the `ObjectDataSource` data source control to use the `GetBugReports3` method:

```

<%@ Page Language="C#" %>
...
<html xmlns="http://www.w3.org/1999/xhtml">
...
<body>
    <form id="form1" runat="server">
        ...
        <asp:ObjectDataSource runat="server" ID="MySource"
            TypeName="ProIIS7AspNetIntegProgCh12.BugReports"
            SelectMethod="GetBugReports3" />
    </form>
</body>
</html>

```

Taking Advantage of ASP.NET and WCF Integration in IIS 7

As discussed earlier, one of the great advantages of hosting WCF services in IIS is their deep integration with ASP.NET applications. This allows you to treat WCF services running in your Web application like any other component of your application such as ASP.NET Web pages. This section shows you how to take advantage of this deep integration in your Web applications.

Now, go back to the BugReportManagerServiceHost WCF Service Web site and add a new Web Form named `Default.aspx`. Add the code shown in Listing 12-11 to the `Default.aspx` file. If you access the `Default.aspx` page from your browser, you'll get the result shown in Figure 12-5.

Listing 12-11: The Default.aspx File of the BugReportManagerServiceHost Web Site

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:DetailsView ID="DetailsView1" DataSourceID="MySource" runat="server"
            Height="50px" Width="125px" BackColor="LightGoldenrodYellow" BorderColor="Tan"
            BorderWidth="1px" CellPadding="2" ForeColor="Black" GridLines="None"
            AutoGenerateInsertButton="True" AutoGenerateRows="False" DefaultMode="Insert">
            <CommandRowStyle HorizontalAlign="Center" />
            <editrowstyle backcolor="DarkSlateBlue" forecolor="GhostWhite"
                HorizontalAlign="Center" />
            <AlternatingRowStyle BackColor="PaleGoldenrod" />
            <Fields>
                <asp:BoundField DataField="sender" HeaderText="Sender" />
                <asp:BoundField DataField="subject" HeaderText="Subject" />
                <asp:BoundField DataField="body" HeaderText="Body" />
            </Fields>
        </asp:DetailsView>

        <asp:ObjectDataSource runat="server" ID="MySource"
            TypeName="ProIIS7AspNetIntegProgCh12.BugReportManager"
            InsertMethod="AddBugReport" />
    </form>
</body>
</html>
```

As Figure 12-5 shows, the `Default.aspx` page of the BugReportManagerServiceHost Web site consists of `DetailsView` server control bound to an `ObjectDataSource` data source control. Note that the `DefaultMode` attribute on the `<asp:DetailsView>` tag is set to `Insert` to display the `DetailsView` server control in its `Insert` mode, where you can insert a new bug report. As the following excerpt from Listing 12-11 shows, the `TypeName` property of the `ObjectDataSource` data source control bound to the



Figure 12-5

DetailsView server control is set to the fully qualified name of the BugReportManager static class. Recall that this class represents the bug report manager system.

```
<asp:ObjectDataSource runat="server" ID="MySource"
  TypeName="ProIIS7AspNetIntegProgCh12.BugReportManager"
  InsertMethod="AddBugReport" />
```

In other words, this code instructs the ObjectDataSource data source control to use the AddBugReport method of the BugReportManager class as an insert method to insert a new bug report. As you can see, the Default.aspx page directly interacts with the bug report manager system. This is very different from the Default.aspx pages of the BugReportManagerServiceClientWebRef, BugReportManagerServiceClientSvcUtil, and BugReportManagerServiceClientImperative Web sites, where these pages interact with the bug report manager system via the bug report manager WCF service.

Now go ahead and enter a few bug reports into the system. Next, visit the Default.aspx page of one of the client applications, that is, BugReportManagerServiceClientWebRef, BugReportManagerServiceClientSvcUtil, or BugReportManagerServiceClientImperative. You should get the result shown in Figure 12-6. Note that this page displays all the bug reports that you just entered into the system.

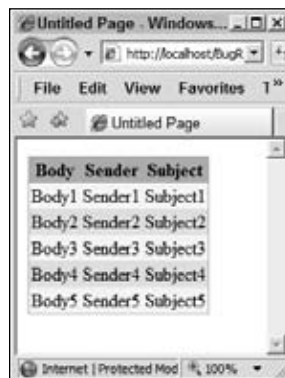


Figure 12-6

As you can see, you have two Web pages up and running, the `Default.aspx` page of the `BugReportManagerServiceHost` Web site shown in Figure 12-5, and the `Default.aspx` page of one of the client Web sites shown in Figure 12-6. The former page allows you to enter new bug reports, whereas the latter page allows you to view all the reported bugs. Now use the `Default.aspx` page to enter a few more bug reports into the system. Refresh the client page. Notice that the client page displays all the reported bugs, including those that you just entered. The `Default.aspx` page directly interacts with the bug report manager system, whereas the client page uses the bug report manager WCF service to interact with the system. Notice that the `Default.aspx` page and your bug report manager WCF service both belong to the same Web site, that is, the `BugReportManagerServiceHost` Web site. This means that the `Default.aspx` page and the service both are loaded into the same application domain. That is exactly why you can enter new bug reports in one page and view them all in another page.

To help you understand this important point, revisit the implementation of the `BugReportManager` class shown in Listing 12-1. As this code listing shows, the `BugReportManager` class stores all bug reports in an `ArrayList` kept in memory. The `Default.aspx` page of the `BugReportManagerServiceHost` Web site uses the `AddBugReport` method of the `BugReportManager` class to add a new bug report to this *in-memory* `ArrayList` storage. The `Default.aspx` page of the client Web sites, on the other hand, uses the `GetBugReports` operation of the service to retrieve all bug reports from this *in-memory* `ArrayList` storage. In other words, both the `Default.aspx` page of the `BugReportManagerServiceHost` Web site and the service can access the same *in-memory* `ArrayList` storage. This is possible because both of them are loaded into the same application domain. As you can see, the deep integration of ASP.NET and WCF services in IIS 7 opens up new programming opportunities that would not be possible otherwise.

Using Different Bindings

The previous sections showed you how to develop a WCF service to enable a piece of software such as the bug report manager system to communicate with the outside world. As discussed, every WCF service exposes one or more endpoints, each with an address, a binding, and a contract. The binding of an endpoint specifies the communication protocols through which the endpoint communicates with the outside world. In this section, you see how to enable your WCF service to communicate with different clients through different sets of communication protocols (bindings).

As discussed in Chapter 2, two main IIS 7 components are involved for each type of transport protocol:

- ❑ **Protocol listener:** A protocol listener is a component that listens for incoming requests over a particular transport protocol. For example, `HTTP.SYS` is the protocol listener that listens for incoming requests over the HTTP transport protocol. The `Net.Tcp Port Sharing Service`, on the other hand, is the protocol listener that listens for incoming requests over the TCP transport protocol.
- ❑ **Protocol listener adapter:** A protocol listener adapter is a component that adapts a particular type of protocol listener to Windows Process Activation Service (WAS). For example, `Net.Tcp Listener Adapter` is the protocol listener adapter that adapts the `Net.Tcp Port Sharing Service` (`Net.Tcp` protocol listener) to Windows Process Activation Service.

When a request over a specific transport protocol arrives, the associated protocol listener picks it up. The protocol listener adapter then informs the WAS that a request for a specified application pool has arrived. The WAS checks whether a worker process has already been assigned to the application pool. If

not, it spawns a new worker process and assigns the task of processing requests for the application pool to this worker process, which in turn picks up the request from its associated queue and processes it.

Therefore, the first order of business in enabling your WCF service to communicate with its clients through a new transport protocol such as TCP is to ensure that the protocol listener and protocol listener adapter components for that transport protocol are up and running. Because you want to enable the bug report manager WCF service to communicate with its clients through the TCP transport protocol, you need to ensure that the Net.Tcp Port Sharing Service, which is the protocol listener for the TCP transport protocol, and Net.Tcp Listener Adapter, which is the protocol listener adapter for the TCP transport protocol, are both up and running. Follow these steps to accomplish this. Select Start → Run and run `services.msc` to launch the Services management console shown in Figure 12-7.



Figure 12-7

Right-click Net.Tcp Listener Adapter and select Properties from the popup menu to launch the Net.Tcp Listener Adapter Properties dialog box shown in Figure 12-8. Select the Automatic option from the Startup type combo box and click OK. Make sure you start the adapter if it hasn't already started. To start the adapter, right-click the adapter and select Start from the popup menu.



Figure 12-8

Chapter 12: ASP.NET and WCF Integration in IIS 7

Repeat these two steps for the Net.Tcp Port Sharing Service protocol listener. So far, you have ensured that the TCP protocol listener and listener adapter are up and running. Next, you need to enable the application that hosts the bug report manager WCF service to accept incoming requests over the TCP transport protocol. Recall that the bug report manager WCF service is hosted in a Web application named `BugReportManagerServiceHost`. Therefore, you need to enable this application to accept incoming requests over the TCP transport protocol. Take these steps to accomplish this task:

1. Open the `applicationHost.config` file in your favorite editor.
2. Search for the `<application>` element that represents the `BugReportManagerServiceHost` Web application in the `applicationHost.config` file, add an attribute named `enabledProtocols`, and set its value to `"http, net.tcp"` to enable this Web application to accept incoming requests on both the HTTP and TCP transport protocols, as shown in the following:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="Default Web Site" id="1" serverAutoStart="true">
        <application path="/BugReportManagerServiceHost"
          enabledProtocols="http, net.tcp">
          <virtualDirectory path="/"
            physicalPath="C:\inetpub\wwwroot\BugReportManagerServiceHost" />
        </application>
      </site>
    </sites>
  </system.applicationHost>
</configuration>
```

Next, you need to configure the Net.Tcp Port Sharing Service protocol listener to listen on the appropriate binding on the site that contains the Web application that hosts the bug report manager WCF service. Take these steps to accomplish this:

1. Open the `applicationHost.config` file in your favorite editor.
2. Search for the `<site>` element that represents the site that contains the `BugReportManagerServiceHost` Web application and add the `<binding>` element shown in boldface in the following excerpt from the `applicationHost.config` file to the `<bindings>` subelement of this `<site>` element:

```
<configuration>
  <system.applicationHost>
    <sites>
      <site name="Default Web Site" id="1" serverAutoStart="true">
        <application path="/BugReportManagerServiceHost"
          enabledProtocols="http, net.tcp">
          <virtualDirectory path="/"
            physicalPath="C:\inetpub\wwwroot\BugReportManagerServiceHost" />
        </application>
        <bindings>
          <binding protocol="http" bindingInformation="*:80:" />
          <binding protocol="net.tcp" bindingInformation="808:+" />
        </bindings>
      </site>
    </sites>
  </system.applicationHost>
</configuration>
```

```

        </bindings>
    </site>
</sites>
</system.applicationHost>
</configuration>

```

Next, you need to enable the bug report manager WCF service itself for accepting incoming requests over the TCP transport protocol. Take these steps to accomplish this:

1. Open the `web.config` file of the `BugReportManagerServiceHost` Web application. Recall that this application hosts the bug report manager WCF service.
2. Add a new `<endpoint>` child element to the `<service>` element that represents the WCF service:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="ProIIS7AspNetIntegProgCh12.BugReportManagerServiceImpl"
        behaviorConfiguration="BugReportManagerService_ServiceMetadata">

        <endpoint address="" binding="basicHttpBinding"
          contract="ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract"/>

        <endpoint address="" binding="netTcpBinding"
          contract="ProIIS7AspNetIntegProgCh12.IBugReportManagerServiceContract"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="BugReportManagerService_ServiceMetadata">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

Now if you access the following URL from your browser:

```
http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc?wsdl
```

you'll get the result shown in Figure 12-9. This figure displays the content of the WSDL document that describes the bug report manager service. Note that the WSDL document now contains a `<wsdl:binding>` element with the name attribute value of `NetTcpBinding_IBugReportManagerServiceContract`. This binding enables the clients of the service to communicate with the service through the TCP transport protocol.

That wraps up the server-side configurations. Next, you develop a new client Web application that communicates with the service through the TCP transport protocol.

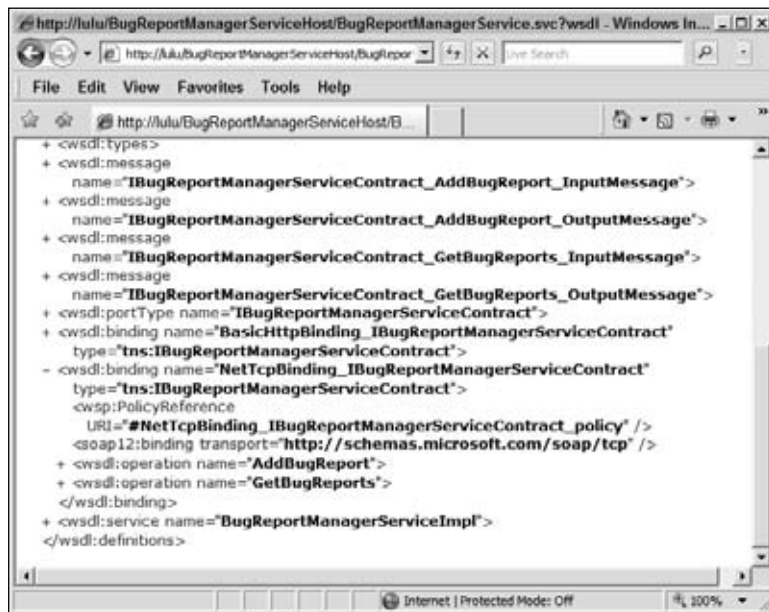


Figure 12-9

Now add a new Web site named `BugReportManagerServiceClientSvcUtil2` to the `ProIIS7AspNetIntegProgCh12` solution. In the Solution Explorer window of Visual Studio, drag the `Default.aspx` file from the `BugReportManagerServiceHost` Web site to the `BugReportManagerServiceClientSvcUtil2` Web site. This will automatically copy this file into the `BugReportManagerServiceClientSvcUtil2` Web site. Recall that this `Default.aspx` file contains the Web page shown in Figure 12-5, which allows the user to add new bug reports to the bug report manager system.

Next, navigate to your favorite directory and use the following command:

```
svcutil.exe /out:BugReportManagerServiceclient.cs /config:Web.config  
http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc
```

As discussed earlier, when you run the preceding command, the `svcutil.exe` tool will automatically do the following:

- ❑ Download the metadata document (WSDL document) that describes the bug report manager service from the specified URL:
`http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc`
- ❑ Generate the code for a proxy class named `BugReportManagerServiceClient` and store the code in the specified `BugReportManagerServiceClient.cs` file in the current directory.
- ❑ Generate the required configuration settings and store them in the specified `web.config` file in the current directory.

Now add the App_Code directory to your BugReportManagerServiceClientSvcUtil2 Web site. Right-click App_Code and select Add Existing Item to launch the Add Existing Item dialog. Navigate to the directory where you used the previous command to run svcutil.exe to add the BugReportManagerServiceClient.cs file to the App_Code directory. Now, right-click BugReportManagerServiceClientSvcUtil2 again and select Add Existing Item to launch the Add Existing Item dialog. Navigate to the directory where you used the previous command to run the svcutil.exe tool to add the web.config file to the BugReportManagerServiceClientSvcUtil2 Web site. Listing 12-12 presents the content of this web.config file.

Listing 12-12: The web.config File

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IBugReportManagerServiceContract"
          closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00"
          sendTimeout="00:01:00" allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
          maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
          useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
              realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
      <netTcpBinding>
        <binding name="NetTcpBinding_IBugReportManagerServiceContract"
          closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00"
          sendTimeout="00:01:00" transactionFlow="false" transferMode="Buffered"
          transactionProtocol="OleTransactions"
          hostNameComparisonMode="StrongWildcard"
          listenBacklog="10" maxBufferPoolSize="524288" maxBufferSize="65536"
          maxConnections="10" maxReceivedMessageSize="65536">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
          <security mode="Transport">
            <transport clientCredentialType="Windows"
              protectionLevel="EncryptAndSign" />
            <message clientCredentialType="Windows" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

(Continued)

Listing 12-12: (continued)

```
</binding>
</netTcpBinding>
</bindings>
<client>

  <endpoint
    address="http://localhost/BugReportManagerServiceHost/
              BugReportManagerService.svc"
    binding="basicHttpBinding"
    bindingConfiguration="BasicHttpBinding_IBugReportManagerServiceContract"
    contract="IBugReportManagerServiceContract"
    name="BasicHttpBinding_IBugReportManagerServiceContract" />

  <endpoint address="net.tcp://localhost/BugReportManagerServiceHost/
              BugReportManagerService.svc"
    binding="netTcpBinding"
    bindingConfiguration="NetTcpBinding_IBugReportManagerServiceContract"
    contract="IBugReportManagerServiceContract"
    name="NetTcpBinding_IBugReportManagerServiceContract">
    <identity>
      <servicePrincipalName value="host/serverName" />
    </identity>
  </endpoint>
</client>
</system.serviceModel>
</configuration>
```

As you can see from Listing 12-12:

- ❑ The `<client>` element contains two `<endpoint>` subelements. The first `<endpoint>` subelement represents the service endpoint with the network address of `http://localhost/BugReportManagerServiceHost/BugReportManagerService.svc` that allows the client to communicate with the endpoint through the `BasicHttpBinding` binding. As discussed earlier, this binding uses HTTP as the transport protocol for sending and receiving SOAP messages. The second `<endpoint>` subelement of the `<client>` element represents the service endpoint with the network address of `net.tcp://localhost/BugReportManagerServiceHost/BugReportManagerService.svc` that allows the client to communicate with the endpoint through the `NetTcpBinding` binding. This binding uses TCP as the transport protocol for sending and receiving SOAP messages.
- ❑ The `<bindings>` element contains two subelements. The first subelement is `<basicHttpBinding>`, which was discussed earlier in this chapter. This binding specifies the values of the properties of the `BasicHttpBinding` binding. The second subelement is `<netTcpBinding>`, which specifies the values of the properties of the `NetTcpBinding` binding.

Now open the `Default.aspx` file of the `BugReportManagerServiceClientSvcUtil2` client Web site and make the changes shown in highlighted portion of Listing 12-13.

Listing 12-13: The Default.aspx Page of the BugReportManagerServiceClientSvcUtil2 Client Application

```

<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:DetailsView ID="DetailsView1" DataSourceID="MySource" runat="server"
            Height="50px" Width="125px" BackColor="LightGoldenrodYellow" BorderColor="Tan"
            BorderWidth="1px" CellPadding="2" ForeColor="Black" GridLines="None"
            AutoGenerateInsertButton="true" AutoGenerateRows="false" DefaultMode="Insert">
            <AlternatingRowStyle BackColor="PaleGoldenrod" />
            <Fields>
                <asp:BoundField DataField="sender" HeaderText="Sender" />
                <asp:BoundField DataField="subject" HeaderText="Subject" />
                <asp:BoundField DataField="body" HeaderText="Body" />
            </Fields>
        </asp:DetailsView>
        <asp:ObjectDataSource runat="server" ID="MySource"
            TypeName="ProIIS7AspNetIntegProgCh12.BugReports"
            InsertMethod="AddBugReport" />
    </form>
</body>
</html>

```

As you can see, these changes instruct the `ObjectDataSource` data source control to use the `AddBugReport` method of the `BugReports` class as the `Insert` method to add a new bug report into your bug report manager system. Now, right-click the `App_Code` directory of the `BugReportManagerServiceClientSvcUtil2` client application in the Solution Explorer and select `Add New Item` to add a new source file named `BugReports.cs`. Next, add the code shown in Listing 12-14 to this source file.

Listing 12-14: The BugReports Class

```

namespace ProIIS7AspNetIntegProgCh12
{
    using System.ServiceModel;

    public class BugReports
    {
        public static void AddBugReport(string sender, string subject, string body)
        {
            BugReportManagerServiceContractClient proxy =
                new BugReportManagerServiceContractClient(
                    "NetTcpBinding_IBugReportManagerServiceContract");
            BugReport bugReport = new BugReport();
            bugReport.Sender = sender;
        }
    }
}

```


Listing 12-14: (continued)

```
        bugReport.Subject = subject;
        bugReport.Body = body;
        proxy.AddBugReport(bugReport);
    }
}
```

As you can see from Listing 12-14, the `BugReports` class exposes a static method named `AddBugReport`. Note that this method does *not* use the default constructor of the `BugReportManagerServiceContractClient` proxy class to create an instance of this proxy class. Instead, it uses the constructor that takes the string that contains the name of a specified endpoint.

Because you want your `BugReportManagerServiceClientSvcUtil2` Web application to direct its communications to the service endpoint that uses the TCP transport protocol, you must ensure that Windows Communication Foundation uses the second `<endpoint>` subelement of the `<client>` element. As the you've seen, this `<endpoint>` subelement represents a service endpoint with the network address of `net.tcp://localhost/BugReportManagerServiceHost/BugReportManagerService.svc` that uses the `NetTcpBinding` binding to communicate with its clients. As Listing 12-14 shows, the `AddBugReport` method of the `BugReports` class passes the value of the `name` attribute of this `<endpoint>` subelement into the constructor of the `BugReportManagerServiceClient` proxy class when it is instantiating an instance of this proxy class. This instructs Windows Communication Foundation to direct the communications of your `BugReportManagerServiceClientSvcUtil2` Web application to the service endpoint specified by the specified `<endpoint>` subelement of the `<client>` element.

Putting It All Together

Add a new Web Form named `Default2.aspx` to the `BugReportManagerServiceHost` Web application and add the code shown in Listing 12-15 to this file.

Listing 12-15: The Default2.aspx File

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:GridView ID="GridView1" DataSourceID="MySource" runat="server"
            BackColor="LightGoldenrodYellow" BorderColor="Tan" BorderWidth="1px"
            CellPadding="2" ForeColor="Black" GridLines="None">
            <HeaderStyle BackColor="Tan" Font-Bold="True" />
            <AlternatingRowStyle BackColor="PaleGoldenrod" />
        </asp:GridView>
```

Listing 12-15: (continued)

```

    <br />
    <asp:ObjectDataSource runat="server" ID="MySource"
        TypeName="ProIIS7AspNetIntegProgCh12.BugReportManager"
        SelectMethod="GetBugReports" />
    </form>
</body>
</html>

```

Note that the `ObjectDataSource` data source control in `Default2.aspx` directly calls the `GetBugReports` method on the `BugReportManager` static class. In other words, this Web page directly interacts with the bug report manager system.

Now you're ready for the final test, where you will have several Web pages up and running simultaneously as shown in Figure 12-10. Take these steps to set up the final test:

1. Access the `Default.aspx` Web page of the `BugReportManagerServiceHost` Web site from your browser. You should see the Web page shown in Figure 12-10. Recall that this Web page is a regular ASP.NET page that does *not* use the bug report manager WCF service. Instead it directly invokes the `AddBugReport` method on the `BugReportManager` static class to add new bug reports.
2. Access the `Default2.aspx` Web page of the `BugReportManagerServiceHost` Web site from your browser. You should see the Web page shown in Figure 12-11. This Web page is also a regular ASP.NET page that does *not* use the bug report manager WCF service. Instead it directly invokes the `GetBugReports` method on the `BugReportManager` static class to display all bug reports.
3. Access the `Default.aspx` Web page of the `BugReportManagerServiceClientSvcUtil2` Web site from your browser. You should see the Web page shown in Figure 12-12. Recall that this Web page uses the `AddBugReport` operation of the bug report manager WCF service through the TCP transport protocol to add new bug reports.
4. Access the `Default.aspx` Web page of the `BugReportManagerServiceClientSvcUtil` Web site from your browser. You should see the Web page shown in Figure 12-13. Recall that this Web page uses the `GetBugReports` operation of the bug report manager WCF service through the HTTP transport protocol to display all bug reports.

Note that I've added a label to the original `Default.aspx` Web pages to help you distinguish these pages. You can easily add the same labels to these Web pages for your own testing if you want.

Now follow these steps to test your applications:

1. Use the Web page shown in Figure 12-10 to add a new bug report to the bug report manager system through a regular ASP.NET request processing mechanism.
2. Refresh the Web page shown in Figure 12-11 to retrieve this bug report from your bug report manager system through a regular ASP.NET request processing mechanism.

- 3. Refresh the Web page shown in Figure 12-13 to retrieve this bug report from your bug report manager system through the bug report manager WCF service using HTTP transport protocol.
- 4. Use the Web page shown in Figure 12-14 to add a new bug report to your bug report manager system through the bug report manager WCF service using TCP transport protocol.
- 5. Repeat Steps 2 and 3 to see the new bug report added to the system.

This example clearly shows the deep integration of ASP.NET and Windows Communication Foundation in the IIS 7 environment.



Figure 12-10



Figure 12-12



Figure 12-11



Figure 12-13

Summary

This chapter provided you with in-depth coverage of the ASP.NET and Windows Communication Foundation integration in the IIS 7 environment and showed you how to take advantage of this integration in your own Web applications. Because WCF services of your applications are loaded into the same application domain as the rest of the components of your applications, and because their compilation goes through the same ASP.NET dynamic compilation process as any other component of your application, you can treat WCF services of your applications as you would any other components of your applications, such as Web pages. These services can seamlessly interact with the rest of the components of your application, opening up new programming opportunities that were not possible before.

Index

A

“ “ value, 29

“. “ value, 29

<access> element, 55

accessing configuration sections in

 <system.applicationHost> section group,
 108–113

 add application pool, 108–109

 add binding, 110–111

 add virtual directory, 112–113

 add Web application, 111–112

 add Web site, 109–110

accessing specified attribute of specified
configuration section, 103, 104–105

 Program.cs file, 105

AcquireRequestState event, 316

action attribute, 42

Actions pane, 63

activity event types, 542

Add Application Pool, 63

Add child elements, 117

“Add collection item” link, 180, 185, 240, 242

ADD command, 81

Add Connection String task form, 487

Add Managed Handler task form, 306, 421

Add Managed Module link, 324, 421

Add Managed Module task form, 325

 checkbox, 325

 GUI elements, 325

Add method

 BindingCollection class, 98

 ConfigurationElementCollectionBase<T> class, 87

 ScheduleCollection class, 93

Add Provider task form, 402

Add Roles Wizard, 14

 confirm installation selections in, 18

 installation results, 19

 launching, 15

 preliminary instruction, 15

 select role services in, 17

 select server roles in, 16

“Add URL rewriter rule” link, 364, 366

Add Web Site, 66–67

Add XML elements, 407, 410, 415, 443

AddAt method, 87

AddBugReport method, 611

AddCollectionItem method, 278–279

 of MyCollectionPage, 243–244

addElement attribute, 121

adding/removing element from specified collection
element of specified configuration section, 103,
106–108

 add new element, 106–107

 remove element, 107–108

AddItem method, 239–240, 363–364

AddProviderForm task form, 402, 421, 490, 491

 parts, 402, 432

AddUrlRewriterRule method, 367–368, 386

administration.config file, 103, 104, 269, 280, 283,
285, 381, 389

 MyConfigSectionModuleProvider module provider

 registration with, 285–286

 RssModuleProvider module provider and, 535–536

ADO.NET, 299

 classes, 398

 SQL Server-specific classes, 398

Advanced Settings dialog, 65

Allow value, 31

allowDefinition attribute, 128

AllowsAdd, 88

AllowsClear, 88

AllowsRemove, 88

<anonymousAuthentication> element, 56

APP object, 79

APPCMD tool. See appcmd.exe tool

appcmd.exe tool (APPCMD tool), 76–81

 IIS Manager v., 77

 location, 77

 object model, 77–79

 syntax, 79–80

 XML elements/attributes and, 76–81

Application class, 98–99

 GetWebConfiguration method, 99

application domains, 593

<application> elements, 98, 99, 100

Application Level, 27

application pools, 63–64

- adding, in <system.applicationHost>, 108–109
- editing, 65

ApplicationCollection class, 99–100

<applicationDefaults> element, 45

ApplicationDomain class, 85, 587, 593–594

- members, 594

ApplicationDomainCollection class, 85, 587, 594–595

- members, 595

applicationHost.config file, 64, 571

- editing, 61
- RssHandler registered by, 304
- structure, 36–60
- UrlRewriterModule HTTP module and, 322–323
- XML elements/attributes in, 77–79

application-level configuration file, 133–134

ApplicationPool class, 88–89, 595

- members, 595
- methods, 89
- properties, 89

ApplicationPoolCollection class, 94–95

- members, 95

ApplicationPoolCollection container, 103

ApplicationPoolCpu class, 93–94

<applicationPoolDefaults> element, 42

ApplicationPoolPeriodicRestart class, 91–92

ApplicationPoolProcessModel class, 89–90

ApplicationPoolRecycling class, 90–91

<applicationPools>, 37–42

- element, 88
- section, 37–38

ApplicationPools collection property, 103

applications, 43

- root, 43

ApplyChanges method, 193, 213–214, 510–511

APPPool object, 79

AppPoolName, 591

ArrayList, 218, 219, 237, 239, 242, 277, 278, 299, 308, 309, 361, 362, 363

Articles table, 292

ArticlesDB database, 488

Articles.xml file, 571–572

ASP.NET, 1. See also integrated architecture, IIS7/ASP.NET

- configuration settings, 68–70
- pages, 51
- provider-based services, 400
- and WCF integration
 - in IIS7 environment, 605–649

ASPNET_schema.xml, 65

assembly, public key token of, 282, 286, 304

AssemblyDownloadService, 169

AssemblyQualifiedName method, 282, 389

AssociatedModule property, 165

<attribute> elements, 120

- in <collection> element, 121, 122

attribute-based programming, WCF Service Model, 609

attributes. See specific attributes

AuthenticateRequest event, 316

<authentication>, 55–58

authentication modules

- managed, 57–58
- native, 55–57

<authorization>, 58–60

AuthorizeRequest event, 316

B

back button, 222

back-end Web server, 184

- communication scenarios

- MyCollectionItemTaskForm, 184–186

- MyCollectionPage, 184–186

- MyConfigSectionPage, 184–186

- UrlRewriterPage module page, 348–349

- UrlRewriterRuleTaskForm task form, 349–351

BaseForm class, 168

- GetService method of, 168

BaseForm constructor, 168

<basicAuthentication>

- configuration section, 119, 120
- element, 57

BeginRequest event, 316

Binding class, 96–97

- properties, 97

<binding> element, 96, 97

BindingCollection class, 97–98

- methods, 98

bindingInformation attribute, 96

bindings. See also communication protocols

- adding, in <system.applicationHost>, 110–111
- WCF endpoint, 608–609, 638–646
- developing, 610

Boolean Property, 182

breadcrumbs bar, 222

bug report manager system, 606–607

- implementation, 606–607
- WCF and, 607–649

bug reports, 606

- in memory, 606, 638

BugReport class, 612–614, 630

- implementation, 612–613

BugReportCollection class, 612

BugReportManager class, 607
BugReportManagerService, 611
 in *ProllS7AspNetIntegProgCh12*, 611
BugReportManagerServiceClient.cs file, 628
 content of, 628–630
BugReportManagerServiceClientImperative, 634
 in *ProllS7AspNetIntegProgCh12*, 634
BugReportManagerServiceClientSvcUtil, 627
 in *ProllS7AspNetIntegProgCh12*, 627
BugReportManagerServiceClientSvcUtil2
 client application, 645
 Default.aspx page of, 645
 in *ProllS7AspNetIntegProgCh12*, 642
BugReportManagerServiceClientWebRef, 625
 in *ProllS7AspNetIntegProgCh12*, 625
BugReportManagerServiceContractClient class, 630
BugReportManagerServiceHost, 617
 in *ProllS7AspNetIntegProgCh12*, 617
 Web application, 646
 Default2.aspx file and, 646–647
 Web site, 636
 Default.aspx file of, 636–638
BugReportManagerServiceImpl class, 615–617
 implementation, 615–616
BugReports class, 634–635, 645–646
build provider, 618
Button1_Click event handler, 425
 implementation, 431–432

C

callback methods for TextChanged event, of textbox controls, 507
CanApplyChanges property, 193, 202, 507
CancelChanges method, 193, 215–216, 511–512
CancelEventArgs class, 204
CanNavigateBack property, 157
CanNavigateForward property, 157
CanRefresh property, 194, 518
 of *ModulePage* class, 224
 of *MyCollectionPage*, 257
 of *MyConfigSectionPage*, 224
Category string property, 158
CausesNavigation property, 162
“Change identifier” link, 181, 182, 185, 240, 242, 243
Channel class, 449
 RssHandler HTTP handler, 295
<channel> element, 292
Channel layer, 605
channelDescription, 296
channelLink, 296

channelTitle, 296
checkbox, Add Managed Module task form, 325
CheckBox control, 184, 199
classes. *See* managed classes, IIS7/ASP.NET integrated imperative management API; *specific classes*
classic ASP pages, 50–51
Clear child elements, 118
Clear method, 87
ClearChannelSettings method, 510
clearElement attribute, 121
ClearSettings method, 211
ClientIPAddr property, 593
client-side managed code, 175–269
 class diagram, 176
 RSS provider-based service and, 493–526
 UrlRewriterModule and, 346–381
clonePropertyBag collection, 215
CLR. *See* Common Language Runtime
Collection element
 <fileExtensions>, 122
 <hiddenSegments>, 122
 steps for defining, 122–123
<collection> element, 121–124
 <attribute> elements in, 121, 122
 attributes, 121
collection XML elements, 117
 child elements, 117–118
 examples, 118, 119
ComboBox control, 184, 199
commandText, 297
 SqlRssProvider, 474
commandType, 297
 SqlRssProvider, 474
CommitChanges method, 103
Common Language Runtime (CLR), 37, 38
communication protocols, 14, 35, 36, 96, 608, 610, 620, 638
<compilation> section, of root web.config file, 25, 26
compiler type-checking support, 135, 208, 336, 340, 362, 390
ConditionalAttribute metadata attribute, 550
<configSections> section, 32–34
configSource attribute, 31, 32
configuration collections, 26
configuration elements, 26
configuration files, 116
configuration properties, 26
configuration sections, 26, 116
 rules, 25

configuration-based programming, WCF Service Model, 609

ConfigurationElement class, 86
members, 86

ConfigurationElementCollectionBase<T> class, 86–88
methods, 87
properties, 88

ConfigurationModuleProvider class, 282

ConfigurationPath property, 156

ConfigurationReadOnlyDelegationState static field, 535

ConfigurationReadWriteDelegationState static field, 535

ConfigurationSection class, 101–102
properties, 102

ConfigurationSectionName property, 282, 389

Configure RSS provider link, 489

ConfigureProvider method, 436, 437, 438
adding/removing/renaming/updating providers of service, 438–440
setting default provider of service, 438, 440–442
steps to set default provider, 442–444

ConfigureRoleProvider method, 439

ConfigureRssProvider method, 516

Connection class, 155–156
service container, 154
service provider, 153

Connection object, 156

connection strings, 297, 473–474

Connection Strings item, 486

ConnectionManager, 152

Connections pane, 62

ConnectionStringAttributeName property, 409

connectionStringName, 297

ConnectionStringRequired property, 409

ConnectionStringsPage module list page, 486, 487, 488

console application. See also <myConfigSection> configuration section
IIS7/ASP.NET integrated imperative extensibility model and, 141–142
<myConfigSection> and, 128–134

constructors. See specific constructors

containing XML elements, 117
examples, 118, 119

Content View, 63

control adapters, 394

Count property, 88, 590, 592, 595

<cpu> element, 42, 89
attributes, 42
imperative representation of, 93, 94

CreateElement, 87

CreateHeader method, 170, 221
implementation, 223

CreateMainArea method, 170, 221

CreateMenuBar method, 170, 221

CreateNavigationItem method, 232, 233

CreateStatusBar method, 170, 221

Critical event type, 541, 542

CurrentItem property, 157

CurrentModule property, 593

custom module service proxy, implementation of, 502–503

custom provider base class, 448–449

custom provider collection, 449–450

custom provider configuration settings class, implementation of, 493–498

custom provider feature, implementation of, 498–500

custom section PropertyBag wrapper class, implementation of, 500–502

D

DataContractAttribute metadata attribute, 614

DataContractSerializer, 613

DataMemberAttribute metadata attribute, 614

Default Document feature, 72, 73, 75

Default Web Site
Home page, RSS item in, 488, 489
node, 69, 73, 76, 227, 404

Default2.aspx file, 646
BugReportManagerServiceHost Web application and, 646–647

Default.aspx
file, 626
of BugReportManagerServiceHost Web site, 636–638
page, 598
of BugReportManagerServiceClientSvcUtil2 client application, 645

<defaultDocument> section, 45–46
configuration section, 118

defaultValue attribute, 120

Define Trace Conditions, 574, 575

delegation, 73–76

DelegationState class, 534–535

“Delete collection item” link, 181, 240, 242

DELETE command, 81

DELETE database operation, 79, 81

“Delete URL rewriter rule” link, 347, 348, 364, 366

DeleteCollectionItem method, 279
of MyCollectionPage, 244–246

DeleteUrlRewriterRule method, 368–369, 387

Deny value, 31

Description

property, 165
string property, 158

deserialization, WCF and, 613

DictionaryEntry objects, 189, 190

<digestAuthentication> element, 57

<directoryBrowse> section, 46

DisableRss method, 515–516, 528–529

disallowOverlappingRotation attribute, 41

disallowRotationOnConfigChange attribute, 41

DisplayName property, 553

Dispose method, 267, 301, 315, 320

DOM

data model, 484
random-access XML API, 302

DownloadandDisplayProviders method, 419

DoWorkEventArgs event data class, 204

DoWorkEventHandler, 204, 235, 265, 359, 378, 441

E

Edit Provider task form, 402, 403

Edit Web Site Failed Request Tracing Settings dialog, 572, 573

<element> element, 121. *See also specific elements*

Enabled property, 159

EnableRss method, 515, 528–529

encoding protocol, 609

endpoints, WCF, 608, 610, 638

adding/updating/removing/configuring, 611
addresses, 609
bindings, 608–609, 610, 638–646
contracts, 609

EndRequest event, 317

Error event, 317

Error event type, 541, 542

event data class, 204

event handler delegates, 204

event handlers, MyConfigSectionPage, 200–201. *See also specific event handlers*

event types. *See also specific event types*

activity, 542
severity, 542

events. *See specific events*

EventTypeFilter filter, 538, 563

instantiation and attachment, 563–570
declarative, 563–565
imperative, 566–570

Execute method, 167, 168

F**Failed Request Tracing**

link button, 572
module, 537, 538, 572
Rules page, 573, 574
Rules Wizard, 574

Feature Delegation page, 75

feature modules

IIS7, 2–7
IIS-ApplicationDevelopment, 4
IIS-CommonHttpFeatures, 3
IIS-FTPPublishingService, 7
IIS-HealthAndDiagnostics, 4
IIS-Performance, 5
IIS-Security, 5
IIS-WebServerManagementTools, 6–7
WAS-WindowsActivationService, 7

FeatureName property, 407

features, 14

Features View, 63

<fileExtensions> Collection element, 122

flags attribute, 55

forward button, 222

functional areas, 2

G

GAC. *See* Global Assembly Cache

gacutil.exe tool, 284

GenerateRss static method, 299–302

of RssHelper class, 300

GetAdministrationConfiguration method, 103

GetApplicationHostConfiguration method, 103

GetAttributeValue, 86

GetBugReports method, 611, 635

versions of, 635

GetChannelValues method, 511

GetCollection, 86

GetCollectionItems method, 235, 277–278

GetDataReader, 299

GetEnumerator, 87, 590, 592, 595

GetGroups method, 255–256

Overridden, 256

GetHandler method, 333

arguments, 334

GetModuleDefinition method, 282, 388

RssModuleProvider module provider and, 533

GetMyConfigSectionSection method, 274

GetPropertyValue method, 164

GetProviders method, 523–524, 529–530

GetSection method

GetSection method, 96, 104, 106, 130, 142, 272, 310, 345, 384, 385, 392

GetService method, 267

- of `BaseForm` class, 168
- of `WebMgrShellApplication` class, 168

GetSettings method, 203–205, 275, 508, 530

- implementation, 203–205

GetTaskItems method, 216, 217, 365–366

- implementation, 216–220

GetUrlRewriterSection method, 384–385

GetUrlRewriterSettings method, 359–360, 385–386

GetValues method, 214–215

GetWebApplication, 96

GetWebConfiguration method, 99

- `Application` class, 99
- `ServerManager` class, 103

GetWorkerProcess, 590

Global Assembly Cache (GAC), 269, 283, 284, 286, 293, 303, 306, 308, 311, 320, 325, 331, 334, 340, 421

<globalModules> section, 46–48

Group method, of `ModuleListPage`, 255

Groupings property, of `ModuleListPage`, 253, 254

GroupTaskItem

- class, 162–163
- task item, 174

H

handler factories. See **managed handler factories**

Handler Mappings module page, 305–306

handlers, 48. See also **managed handlers**

<handlers> section, 48–50

- `<add>` elements, 50
- attributes, 50

HasChanges property, 194, 201–202, 507

help button, 222

<hiddenSegments> Collection element, 122

HierarchyPanel, 171

History collection property, 157

home button, 222

HttpApplication objects, 316

- firing of ordered events, 316–317
- pool, 316

HttpContext object, 291

HTTPS protocol, 184

HybridDictionary class, 190, 191

I

IBugReportManagerServiceContract service contract, 612, 630

- methods, 611

IConnectionManager interface, 152

identityType attribute, 40

IDictionary interface, 189, 190, 208, 276, 413, 414, 434

IDisposable interface, 301

idleTimeout attribute, 40

IEnumerable object, 174

IExtensibilityManager, 411

IHttpHandler interface, 290

- members, 291

IHttpHandlerFactory interface, 333

- members, 333–334

IHttpModule interface, 315

- methods, 315

IIS-

- `ApplicationDevelopment`, 4
 - feature modules, 4
- `ASP`, 4
 - update dependencies, 12
- `ASPNET`, 4
 - update dependencies, 12
- `BasicAuthentication`, 5
- `CGI`, 4
- `ClientCertificateMappingAuthentication`, 5
- `CommonHttpFeatures`, 3
 - feature modules, 3
- `CustomLogging`, 4
- `DefaultDocument`, 3
- `DigestAuthentication`, 5
- `DirectoryBrowsing`, 3
- `FTPManagement`, 7
- `FTPPublishingService`, 7
 - feature modules, 7
- `FTPServer`, 7
- `HealthAndDiagnostics`, 4
 - feature modules, 4
- `HttpCompressionDynamic`, 5
- `HttpCompressionStatic`, 5
- `HttpErrors`, 3
- `HttpLogging`, 4
- `HttpRedirect`, 3
- `HttpTracing`, 4
- `IISCertificateMappingAuthentication`, 5
- `IPSecurity`, 5
- `ISAPIExtensions`, 4
- `ISAPIFilter`, 4
- `LegacyScripts`, 7
 - update dependencies, 12
- `LegacySnapIn`, 7
- `LoggingLibraries`, 4
- `ManagementConsole`, 6
 - update dependencies, 12

- ManagementScriptingTools, 6
 - update dependencies, 12
- ManagementService, 6
 - update dependencies, 12
- Metabase, 7
- NetFxExtensibility, 4
 - update dependencies, 12
- ODBCLogging, 4
- Performance feature modules, 5
- RequestFiltering, 5
- RequestMonitor, 4
- Security feature modules, 5
- ServerSideIncludes, 4
- StaticContent, 3
- URLAuthorization, 5
- WebServer, 3
 - update dependencies, 12
 - updates, 3–5
- WebServerManagementTools, 6
 - feature modules, 6–7
- WebServerRole, 2
 - updates, 2
- WindowsAuthentication, 5
- WMICompatibility, 7
- IIS7. See Internet Information Services 7.0**
- IIS7 Manager. See Internet Information Services Manager**
- IIS7 runtime objects, up-to-date runtime data for, 567–598**
- IIS7/ASP.NET integrated architecture. See integrated architecture, IIS7/ASP.NET**
- IIS_schema.xml, 65**
- IsTraceListener, 538**
 - instantiation and attachment, 539, 557–562
 - SourceFilter filter and, 569
- Image property, 159, 162**
- IManagementHost interface, 152**
- IManagementUIService interface, 152**
- IModuleChildPage interface, 230–231, 232**
- imperative programming, WCF Service Model, 609**
- ImperativeManagement directory, 340, 341**
- INavigationService interface, 152, 156–157**
- IndexOf, 87, 590, 592, 595**
- Information event type, 541, 542**
- Inherit value, 31**
- Init method, 315, 320**
- Initialize method, 267**
 - of Module base class, 269, 381
 - of ProviderConfigurationConsolidatedPage module page, 444
 - of RolesModule module, 411–412
 - RssService class, 464, 465
 - SqlRssProvider, 471–473
 - XmlRssProvider, 480–482
- InitializeComponent method**
 - MyCollectionItemTaskForm and, 262–264
 - MyConfigSectionPage's user interface and, 196–200
 - RssPage module page, 504–507
 - RssSettingsForm task form, 522–523
 - UrlRewriterRuleTaskForm and, 376–378
- InitializeListPage method, 233, 234**
 - UrlRewriterPage module and, 358–359
- InitializeUI method, 210, 212, 510**
- INSERT database operation, 79, 81**
- InstantiateProviders method, 458–459**
- integrated administration, IIS7/ASP.NET, 11**
- integrated architecture, IIS7/ASP.NET, 1–21**
 - extensibility models, 116
 - main components, 115
- integrated configuration system, IIS7/ASP.NET, 10–11, 23–60**
 - APPCMD, 76–81
 - characteristics/benefits, 23–24, 28
 - distributed, 26–28
 - extending RSS provider-based service, 450–453
 - extensibility model, 115–134
 - hierarchical, 24–28, 68, 73–74
 - IIS Manager, 62–76
 - IIS Manager and hierarchical, 68–73
 - management options, 61–62
 - programmatic interaction with, 103–113
 - steps for extending, 124–125, 337–338
- integrated declarative schema extension markup language, IIS7/ASP.NET, 116–117, 337, 338**
 - XML constructs
 - <rss>configuration section and, 451–453
 - XML elements/attributes, 117–124
- integrated graphical management system, IIS7/ASP.NET, 11, 62–76, 145–174. See also Internet Information Services Manager**
 - extending
 - RSS provider-based service and, 485–493
 - extensibility model, 115, 116, 145, 146, 175–287
- integrated imperative management API, IIS7/ASP.NET, 11**
 - extending
 - RSS provider-based service and, 454–462
 - extensibility model, 115, 116, 134–140
 - console application for, 141–142
 - managed classes, 86–103
 - types, 84
 - IIS7 troubleshooting, 85, 587
 - XML constructs, 84–85, 587

integrated providers model, IIS7/ASP.NET, 397–444

- in action, 400–405
- extending, 445–536. *See also* RSS provider-based service
- recipe, 445–447

integrated request processing pipeline, IIS7/ASP.NET, 8–10

- advantage of, 393
- extensibility model, 116, 289–396
 - through configurable managed components, 336–337
 - through managed code, 289–290
- modular architecture, 115
- plugging into
 - custom managed handler factories, 336
 - custom managed handlers, 302–315
 - custom managed modules, 322–332
 - RssHandler HTTP handler, 302–315
 - UrlRewriterModule HTTP module, 322–332

integrated tracing and diagnostics, IIS7/ASP.NET, 537–604. *See also* tracing

- components, 537–540

integration of ASP.NET and WCF services, in IIS7 environment, 605–649

IntelliSense support for strongly-typed properties, Visual Studio, 135, 208, 336, 340, 362, 390

Internet Information Services 7.0 (IIS7), 1

- architecture
 - extensible, 8
 - modular, 1–2, 11, 115
- ASP.NET and WCF integration in, 605–649
- configuration settings
 - application-specific, 73
 - IIS Manager and, 70–73
 - web.config file with, 73
- feature modules, 2–7
- ISAPI mode, 38, 39, 48
- setup options, 11–12
 - command-line, 20
 - Server Manager tool, 14–19
 - unattended, 20–21
 - upgrade, 21
 - Windows Features dialog, 13–14
- troubleshooting integrated imperative management types, 85, 587

Internet Information Services (IIS7) Manager, 62–76, 145–174

- APPCMD tool v., 77
- ASP.NET configuration settings, 68–70
 - machine level, 68–69
 - site level, 69–70
- capabilities, 145
- IIS7 configuration settings, 70–73

- IIS7/ASP.NET hierarchical configuration and, 68–73
- launching

- command-line, 62
- GUI-based, 62
- module pages, 146–148
- object model, 152
 - classes in, 152–166
- page navigation, 149
- RssHandler HTTP handler registration
 - at IIS7 Web-server level, 304–309
 - RssHandlerCh8 application and, 312–313
- tasks, 149–151
- UrlRewriterModule HTTP module registration
 - at IIS7 Web-server level, 323–327
 - UrlRewriterModuleCh8 application and, 323–327
- user interface extension
 - procedures/tasks, 176
- Visual Studio and automatic running of, 285
- XML elements/attributes and, 62–76

Invoke method, 188

InvokeMethod method, 164–165

IPropertyEditingService interface, 152

IProviderConfigurationService interface, 405, 436–444

IRSCA_AppDomain, 587

IRSCA_AppPool, 587

IRSCA_RequestData, 587

IRSCA_RequestReader, 587

IRSCA_VirtualSite, 588

IRSCA_W3SVC, 588

IRSCA_WAS, 588

IRSCA_WorkerProcess, 588

ISAPI mode, IIS7, 38, 39, 48

IsapiFilterModule, 48

IsapiModule, 48

IsAssignableFrom method, 232

IsEnabled property, 165

IServiceContainer interface, 153–154

IServiceProvider interface, 152–153

isExternalTraceSource attribute, 578

- <rss> configuration section and, 578
- RSS_Schema.xml file and, 578–579
- RssSection class and, 578–580
- RssService class and, 578

isExternalTraceSource property, 585

- RssSectionInfo class and, 585–586

IsHeading property, 159, 163

IsNew property, 158

IsReusable property, 291

- RssHandler HTTP handler, 296

Item class

- RssHandler HTTP handler, 295–296
- SqlRssProvider, 469–470

Item indexer, 590

Item property, 88, 592, 595
itemDescription field, 475
itemDescriptionField, 296
itemDescriptionXPath, 481
itemLink field, 475
itemLinkField, 296
itemLinkFormatString, 297
itemLinkFormatString field, 475
itemLinkXPath, 481
Items property, 163
itemTitle field, 475
itemTitleField, 296
itemTitleXPath, 481
itemXPath, 481

K

KeyValuePair objects, 166, 173

L

Label control, 199
LargelImage property, 165
Level property, of **SourceSwitch**, 554
limit attribute, 42
LIST command, 80
ListViewItem class, 239
loading specified configuration file, 103, 104
LoadRss method, 298–299
 RssService class, 465–466
 SqlRssProvider, 475
 XmlRssProvider, 482–483
LoadSettings method, 412, 435
localInfo private field, 207
<location> tags, 28–31
 characteristics, 30
logEventOnRecycle attribute, 41
LogRequest event, 317, 600–604
LongDescription property, 165

M

Machine Level 1, 27
Machine Level 2, 27
machine-level configuration file, 128–131
managed authentication modules, 57–58
managed classes, IIS7/ASP.NET integrated imperative management API, 86–103. *See also specific classes*
 category, 84
 IIS7 troubleshooting, 85
 integrated configuration system, 84
 diagram, 83–85

managed code

client-side, 175–269
 class diagram, 176
 RSS provider-based service and, 493–526
 UrlRewriterModule and, 346–381
 extending integrated pipeline through, 289–290
 object-oriented, 83
 server-side, 175, 269–286
 deployment, 283–286
 RSS provider-based service and, 526–536
 steps to write, 269
 UrlRewriterModule and, 381–390

managed handler factories, 333–336

configurability of, 336
 custom
 developing, 334–335
 plugging into integrated request processing pipeline, 336
 definition, 290

managed handlers, 290–315

configurability of, 336
 custom, 291–302
 plugging into integrated request processing pipeline, 302–315
 definition, 289–290
 server-side logic that retrieves, 308–309

managed modules, 315–333

configurability of, 336
 custom, 318–332
 developing, 318–321
 definition, 290
 important points, 318–319
 server-side logic that retrieves HTTP, 326–327

Managed pipeline mode drop-down list, 63

ManagedEngine module, 47

managedPipelineMode attribute, 38–39

ManagementConfigurationPath class, 154–155

ManagementFrame constructor, 170, 221

methods, 221
 OnCommandBarRefresh method of, 223–224

ManagementGroupBox, 199

controls, 199, 200
 hierarchy, 199

ManagementPanel, 264

control, 377
 hierarchy, 264

ManagementScope enumeration, 155

ManagementUIService, 152, 169

MapRequestHandler event, 316

maxProcesses attribute, 40

MemberName property, 163

MembershipProviderConfigurationFeature subclass, 406, 407

MessageTaskItem

class, 159–161
definition, 161
properties, 161
task item, 174

MessageTaskItemType enumeration, 161

MessageType property, 161

MethodName property, 161, 162

methods. *See specific methods*

MethodTaskItem

class, 161–162
definition, 161–162
properties, 162
task item, 174

MethodTaskItemUsages enumeration, 162

property, 162

Microsoft.Web.Administration namespace, 83, 84

Microsoft.Web.Administration.dll assembly, 104, 105, 107, 108, 141, 177, 340, 344

modeling software, 609

ModifiedKeys property, PropertyBag class, 190, 191, 276

Module class, 153, 267–268

Initialize method of, 269, 381
methods, 267

module dialog pages, 147

module list pages, 147

module pages, 146–148, 177

custom, 148
implementation of, 503–518
for <myConfigSection> configuration section, 179

module properties pages, 147–148

module service, 270

custom, 270

ModuleDefinition object, 533

ModuleDialogPage class, 147, 193

overridable members, 193–194
Refresh method of, 225–226

ModuleListPage, 147

Group method of, 255
Groupings property of, 253, 254
ModulePage class and, 233
MyCollectionPage and, 233

ModuleListPageGrouping class, 253–254

ModulePage class, 177, 194

CanRefresh property of, 224
class hierarchy, 177–178
MyCollectionPage and, 233
Refresh method of, 225
UrlRewriterPage and, 358

ModulePageInfo class, 165–166

ModulePropertiesPage, 147–148

ModuleProvider class, 281, 282

Modules module list page, 324

<modules> section, 52–55

<add> elements, 52
attributes, 52
contents, 53–54

ModuleService class, 270

ModuleServiceMethodAttribute metadata attribute, 273, 381, 383

ModuleServiceProxy, 186–189

MyApplication project, Visual Studio and, 422

MyApplicationPool node, 64

MyClass1 class, 422–423

read/write properties, 423

MyClass1Property1 property, 423, 427

MyClass1Property2 property, 423, 427

MyClass1Property3 property, 423, 427, 428, 430

MyClass1Property4 property, 423, 427, 430

MyClass1Property5 property, 423, 427, 428, 429

MyClass1Property6 property, 423, 424, 427, 429

MyClass2 class, 424

MyCollection class, 136–138, 177

myCollectionIntAttr attribute, 179, 184, 185

MyCollectionIntProperty, 209

MyCollectionItem class, 136, 177

myCollectionItemBoolAttr attribute, 179

myCollectionItemIdentifier attribute, 179

MyCollectionItemInfo class, 238

MyCollectionItemListViewItem class, 239

MyCollectionItemTaskForm

in action, 179–183
communications with back-end Web server, 184–186
constructors, 262
implementation, 258–267
IntializeComponent method and, 262–264

MyCollectionPage, 179

in action, 179–183

AddCollectionItem method of, 243–244

adding support for new task items, 240–247

CanRefresh property of, 257

communications with back-end Web server, 184–186

declaration of members, 229–230

DeleteCollectionItem method of, 244–246

implementation, 229–257

ModuleListPage class and, 233

OnActivated method and, 235

overrides Tasks property, 243

UpdateCollectionItem method of, 246–247

MyConfigSection class, 139–140, 177

Visual Studio and, 176

<myConfigSection> configuration section, 125, 178–179

- console application, 128–134
 - application-level configuration file, 133–134
 - machine-level configuration file, 128–131
 - site-level configuration file, 131–133
- content, 126–127
- module pages for, 179
- MyCollection class, 136–138, 177
- MyCollectionItem class, 136, 177
- MyConfigSection class, 139–140, 177
- MyConfigSectionEnum type, 140, 177
- MyNonCollection class, 138, 177
- MY_schema.xml in, 125
 - content, 126–127
- portions, 125
- registration, 128
- representative implementation, 125
- strongly-typed objects and, 134–135
- in <system.webServer> group, 125, 128

MyConfigSection module page, 179–180

- group boxes, 180

myConfigSectionBoolAttr attribute, 179, 184, 185

MyConfigSectionBoolProperty, 209

MyConfigSectionEnum type, 140, 177

myConfigSectionEnumAttr attribute, 179, 184, 185

MyConfigSectionEnumObject class, 211–212

MyConfigSectionEnumProperty, 209

MyConfigSectionInfo class, 207–210

- benefits, 208
- implementation, 207–208
- discussion, 209–210

MyConfigSectionModule, 268–269

MyConfigSectionModuleProvider custom module provider, 281–282

- register with administration.config file, 285–286

MyConfigSectionModuleService Server-Side class, 188, 189, 270–273

MyConfigSectionModuleServiceProxy class, 186, 187

MyConfigSectionPage, 179

- in action, 179–183
 - adding support for new task items, 216–221
- ApplyChanges method, 213–214
- CanApplyChanges property, 202
- CanRefresh property of, 224
- communications with back-end Web server, 184–186
- constructor, 196
- declarations of members, 194–196
- event handlers, 200–201
- GetSettings method, 203–205
- HasChanges property, 201–202
- implementation, 193–229

- InitializeComponent method, 196–200

- OnActivated method, 202–203

- overrides Tasks property, 220–221

- user interface, 198–199

- group boxes, 199–200

- InitializeComponent method and, 196–200

- “View collections items” link

- adding, 216–221

MyEnum enumeration, 423

MyForm Windows Form, 427

- Button1_Click event handler, 431–432

- observations about, 427–428

MyForm.cs file, 422

- content of, 425–426

MyForm.Designer.cs file, content of, 424–425

myListener, 558

MyModule HTTP module, 600–604

- example of XML file supported by, 600

MyNamespace, 176, 422

MyNamespace.Server.MyConfigSectionModuleProvider, 286

MyNonCollection class, 138, 177

myNonCollectionTimeSpanAttr attribute, 179, 184, 185

MyNonCollectionTimeSpanProperty, 209

myNonCollectionTimeSpanPropertyTextBox control, 200

myobj object, 427

MY_schema.xml, 125

MySqlConnection string, 487, 488

MySqlRssProvider, 492

- ProviderConfigurationConsolidatedPage module page with, 492

myTraceSource, 545, 559

MyXmlRssProvider, 491

- ProviderConfigurationConsolidatedPage module page with, 492

N

name attribute

- <attribute> element, 120

- <handlers> section, 50

- <modules> section, 52

- <section> element, 128

Name textbox, 325

native authentication modules, 55–57

Navigate method, 157, 232

NavigateBack method, 157

NavigateForward method, 157

navigation items, 156, 232

navigation service, 149, 156, 232

NavigationData object, 156

NavigationEventArgs class, 157, 158
NavigationEventHandler, 157
NavigationItem class, 156
NavigationService class, 152
 implementation, 231
.NET Framework, and release cycle of Visual Studio, 28
.NET reflection, 421, 428, 430
.NET Roles page, 405, 410
Net.Tcp Listener Adapter, 639
Net.Tcp Port Sharing Service, 639
 protocol listener, 640
NewItem property, 158
non-ASP.NET URLs, **UrlRewriterModule** and rewriting, 393
 postback problems, 393–395
non-collection XML elements, 118
 examples, 118, 119

O

object-oriented
 APIs, 8, 46
 managed code, 83
 programming, 135, 208, 336, 340, 362, 390
objects. *See specific objects*
ObjectState enumeration type, 596
ObjectStateFormatter class, 209
OldItem property, 158
OnAccept method, 265, 441
 RssSettingsForm task and, 524
 UrlRewriterRuleTaskForm and, 378
OnActivated method, 194, 202–203
 MyCollectionPage and, 235
 of ProviderConfigurationConsolidatedPage, 418
 RssPage module page overrides, 508
 UrlRewriterPage and, 359
OnCommandBarRefresh method, of **ManagementFrame**, 223–224
OnFeatureComboBoxSelectedIndexChanged event handler, 418
 portion of implementation, 418–419
OnGroup method, 256
 Overridden, 256
OnListViewAfterLabelEdit method, 248–251
OnListViewBeforeLabelEdit method, 247–248
OnListViewDoubleClick method, 251
 UrlRewriterPage and, 370–371
OnListViewKeyUp method, 252
 UrlRewriterPage and, 371
OnListViewSelectedIndexChanged method, 252
 UrlRewriterPage and, 370

OnmyCollectionIntPropertyTextBoxTextChanged method, 200, 202
OnmyCollectionItemBoolPropertyCheckBoxChanged method, 264
OnmyCollectionItemIdentifierTextBoxTextChanged method, 264
OnmyConfigSectionBoolAttrCheckBoxCheckedChanged event handler, 200, 201
OnmyConfigSectionEnumPropertyComboBoxSelectedIndexChanged event handler, 200, 201
OnmyNonCollectionTimeSpanPropertyTextBoxTextChanged method, 200, 202
OnNavigationPerformed event handler, 170, 172–174
OnOkButtonClick method, 433
 implementation, 433–435
OnRefresh method, 193, 226, 227, 228, 229, 518
OnValueChanged method
 of SourceSwitch class, 553
 of Switch base class, 553
OnWorkerCompleted method, 266–267, 378, 380, 441
 UrlRewriterRuleTaskForm and, 380
OnWorkerDoWork method, 265–266, 441
 UrlRewriterRuleTaskForm and, 378–379
OnWorkerGetCollectionItems method, 235–236
OnWorkerGetCollectionItemsCompleted method, 236–238, 239
OnWorkerGetSettings method, 205, 508–509
OnWorkerGetSettingsCompleted method, 205–207, 210, 509–510
OnWorkerGetUrlRewriterSettings method, 360
OnWorkerGetUrlRewriterSettingsCompleted method, 360–362, 363
OperationContractAttribute metadata attribute, 611
Overridden GetGroups method, 256
Overridden OnGroup method, 256
overrideMode, 30
 Allow value, 31
 Deny value, 31
 Inherit value, 31
 property, 102
overrideModeDefault, 128

P

Page module page, 156
page navigation, 149–150
PageContainerPanel, 171
PageHandlerFactory, 10, 51, 290, 333, 334, 335
PageHeader class, 221–222
PageTaskList class, 217–218
 implementation, 240–241

- RssPage module page, 512–515
 - UrlRewriterPage module page, 364–366
 - PageType** object, 156, 165
 - ParentPage** property, 230, 231
 - password** attribute, 40
 - path** attribute, 50
 - <periodicRestart>** element, 42, 91, 92, 93
 - PhysicalPath**, 594
 - pingingEnabled** attribute, 40
 - pingInterval** attribute, 41
 - pingResponseTime** attribute, 41
 - PipelineState**
 - enumeration type, 593
 - property, 593
 - pkgmgr.exe** command-line tool, 20
 - options, 20
 - PostAcquireRequestState** event, 317
 - PostAuthenticateRequest** event, 316
 - PostAuthorizeRequest** event, 316
 - PostLogRequest** event, 317
 - PostMapRequestHandler** event, 316
 - PostReleaseRequestState** event, 317
 - PostRequestHandlerExecute** event, 317
 - PostResolveRequestCache** event, 316
 - PostUpdateRequestCache** event, 317
 - preCondition** attribute
 - <handlers> section, 50
 - <modules> section, 52
 - PreRequestHandlerExecute** event, 317
 - PreSendRequestContent** event, 317
 - PreSendRequestHeaders** event, 317
 - ProcessGuid**, 591
 - ProcessID**, 591, 593
 - <processModel>** element, 39–40, 89, 90
 - attributes, 40–41
 - imperative representation of, 89, 90
 - ProcessRequest** method, 291
 - RssHandler's implementation of, 297–298
 - ProfileProviderConfigurationFeature** subclass, 406, 407
 - Program.cs** file, 105
 - ProllIS7AspNetIntegProgCh12**, 606
 - BugReportManagerService in, 611
 - BugReportManagerServiceClientImperative in, 634
 - BugReportManagerServiceClientSvcUtil in, 627
 - BugReportManagerServiceClientSvcUtil2 in, 642
 - BugReportManagerServiceClientWebRef in, 625
 - BugReportManagerServiceHost in, 617
 - properties.** *See specific properties*
 - PropertyBag** class, 189–193
 - constructors, 189–190
 - CreatePropertyBagFromState method, 192–193
 - GetState method, 192
 - indexer property, 190–191
 - ModifiedKeys property, 190, 191, 276
 - PropertyEditingService**, 152, 169
 - PropertyGrid** control, 421, 422, 427
 - edit property in, 431–432
 - protocol** attribute, 96
 - protocol listener adapters**, 35, 638
 - protocol listeners**, 34, 638
 - provider configuration service**, 436. *See also*
 - IProviderConfigurationService interface
 - provider pattern**, 400
 - provider-based services**, 397. *See also* integrated
 - providers model, IIS7/ASP.NET; RSS provider-based service; workflow
 - ASP.NET, 400
 - configurations, 438–442
 - custom, 407–410
 - configuration settings, 410
 - need for, 398
 - ProviderBaseType** property, 407
 - ProviderCollectionPropertyName** property, 407, 408
 - ProviderConfigurationConsolidatedPage** module page, 401, 404, 410, 489, 490
 - Initialize method of, 444
 - MySqlRssProvider in, 492
 - MyXmlRssProvider in, 492
 - OnActivated method, 418
 - ProviderConfigurationModule** class, 436–437
 - ProviderConfigurationService** property, 440, 517
 - ProviderConfigurationSettingNames** array, 408, 434
 - ProviderConfigurationSettings** class, 405, 412–415
 - implementation, 412
 - ProviderFeature** abstract base class, 405
 - internal implementation, 406
 - properties, 407–410
 - subclasses, 406
 - virtual properties, 409
 - ProviderHelper** helper class, 457–460
 - providers**, 400. *See also* workflow
 - ProviderSettings** class, 454–455
 - ProviderSettingsCollection** class, 455–457
 - proxies**, 184–186
 - public key token**, of assembly, 282, 286, 304
 - PublicKeyToken** attribute, 304
- ## Q
- queueLength** attribute, 39
- ## R
- ReadOnly** property, 194, 209, 210, 237, 518
 - ReadOnlyDescription** property, 237

Really Simple Syndication. See RSS

Recipes

- accessing configuration sections in
 - <system.applicationHost> section group, 108–113
- add application pool, 108–109
- add binding, 110–111
- add virtual directory, 112–113
- add Web application, 111–112
- add Web site, 109–110
- accessing specified attribute of specified configuration section, 103, 104–105
 - Program.cs file, 105
- adding/removing element from specified collection
 - element of specified configuration section, 103, 106–108
 - add new element, 106–107
 - remove element, 107–108
- for extension of integrated providers model, 445–447
- loading specified configuration file, 103, 104

Recycle method, 89

<recycling> element, 41–42, 89, 92, 93

- attributes, 41
- imperative representation of, 90, 91

refresh button, 222

Refresh method, 227, 228, 257

- of ModuleDialogPage class, 225–226
- of ModulePage class, 225

RegisterExtension method, 412

related activity identifier, 548

ReleaseHandler method, 334

ReleaseRequestState event, 317

Remove child elements, 117–118

Remove method

- BindingCollection class, 98
- ConfigurationElementCollectionBase<T> class, 87

RemoveAt method

- BindingCollection class, 98
- ConfigurationElementCollectionBase<T> class, 87

removeElement attribute, 121

Repeater control, 292

Request class, 85, 587, 592–593

- properties, 593

RequestCollection class, 85, 587, 591–592

- methods/properties, 592

<requestFiltering> configuration section, 121–122

RequestLoggingModuleService class, 586–587

resetInterval attribute, 42

ResolveRequestCache event, 316

Resume event type, 541, 542

RetrieveData method, 483–484

rewriting non-ASP.NET URLs, UriRewriterModule and, 393

- postback problems, 393–395

role management service, 397

role services, 14

roles, 14. See also Add Roles Wizard

Roles provider-based service, 409

- RolesProviderConfigurationFeature information on, 410

RolesModule module, 410

- Initialize method of, 411–412

RolesProviderConfigurationFeature subclass, 406, 407, 411

- information on Roles provider-based service, 410
- registration, 412

RolesProviderConfigurationSettings class, 412

- implementation, 413–414

root application, 43

root web.config file, 69, 73

- <compilation> section, 25, 26
- IIS7 configuration settings in, 73

RscalerInterop internal class, 588–589

RSS (Really Simple Syndication), 291

- 2.0 format, 291–292
- document, 291–292

<rss> configuration section, 454

- isExternalTraceSource and, 578
- registration, 453
- traceSource and, 578
- XML constructs of integrated declarative schema
 - extension markup language and, 451–453

<rss> document element, 292

RSS item, in Default Web Site Home page, 488, 489

Rss project, 447, 570, 572

RSS provider-based service, 445

- client-side managed code and, 493–526
 - implementation steps, 493
- custom providers, 467–485
 - SqlRssProvider, 467–477
 - XmlRssProvider, 477–485
- default provider for, 492–493
- extending integrated configuration system, 450–453
- extending integrated graphical management system, 485–493
 - workflow, 485
- extending integrated imperative management system, 454–462
- implementation, 448–466
- preliminary setup, 447
- server-side managed code and, 526–536
 - implementation steps, 526

RssEnabled property, 518

RssHandler HTTP handler, 291, 292, 293

applicationHost.config file and registering, 304
 channel class, 295
 compilation options, 303
 add reference, 303
 constructor, 296–297
 private fields initialized by, 296–297
 definition, 398–400
 implementation, 293–294
 IsReusable property, 296
 item class, 295–296
 new version
 RssService and, 466
 plugging into integrated request processing pipeline, 302–315
 ProcessRequest method implemented by, 297–298
 registration, IIS7 Web server-level, 303–304
 declarative approach, 304
 graphical approach/IIS7 Manager, 304–309
 imperative approach, 304, 309–310
 undo, 311
 registration levels, 302–303
 registration, RssHandlerCh8 application, 311–314
 declarative approach, 311–312
 graphical approach/IIS7 Manager, 312–313
 imperative approach, 313–314
 shortcomings, 398
 using, 314–315

RssHandlerCh8 application, RssHandler HTTP handler registration with, 311–314**RssHandlerConsoleApplication project, 309, 310****RssHandlerProj, Visual Studio and, 293, 306, 311****RssHelper class, 299, 300**

GenerateRss method of, 300, 400
 SqlRssProvider, 470–471

RssModule module, 524–526**RssModuleProvider module provider, 531–536**

GetModuleDefinition method overridden by, 533
 registration, 535–536

RssModuleService class, 526–531

implementation, 527–528

RssModuleServiceProxy class, 502–503**RssPage module page, 503–518, 582–585**

declaration of members, 503–504
 InitializeComponent method, 504–507
 OnActivated method overridden by, 508
 PageTaskList class, 512–515
 Tasks property overridden by, 517

RssProvider base class, 448**RssProviderCollection class, 449–450****RssProviderConfigurationFeature class, 498–500**

tasks, 500

RssProviderConfigurationSettings class, 493–498**RSS-related fields, 475****RSS_Schema.xml file, 451–452**

content, 451
 isExternalTraceSource and, 578–579
 traceSource and, 578–579

RssSection class, 460–462

isExternalTraceSource and, 579–580
 traceSource and, 579–580

RssSectionInfo class, 500–502

IsExternalTraceSource property and, 585–586
 programming benefits, 500–501
 TraceSource property and, 585–586

RssService class, 400, 580–582

implementation, 462–466
 Initialize method, 464, 465
 isExternalTraceSource attribute and, 578
 LoadRss method, 465–466
 RssHandler HTTP handler and, 466
 TraceEvent method and, 548–550
 traceSource attribute and, 578
 TraceSource class and, 543–545

RssSettingsForm task form, 518–524

constructor, 522
 implementation, 519–522
 InitializeComponent method, 522–523
 OnAccept method and, 524

RssWebSite node, 573**Runtime Status and Control API (RSCA), 587–589****RunWorkerCompletedEventArgs event data class, 205****RunWorkerCompletedEventHandler, 205, 235, 265, 359, 378, 441****S**

<schedule> element, 42, 93

ScheduleCollection class, 93

Add method, 93

scriptProcessor attribute, 50

<section> element, 128

section groups, 25. *See also specific section groups*
 rules, 25

SectionName property, 408

sections. *See specific sections*

<sectionSchema> element, 119–120, 339

security protocol, 609

<security> section, 55

elements, 55–60

SELECT database operation, 79, 80

Select Trace Providers, 575

SelectedProvider property, 408

SelectedProviderPropertyName property, 408–409

serialization, WCF and, 613

Server Manager tool, 14–19

launching, 14

ServerManager class, 102–103, 589–590

collection properties, 103

methods, 103

server-side managed code, 175, 269–286

deployment, 283–286

RSS provider-based service and, 526–536

steps to write, 269

UrlRewriterModule and, 381–390

service containers, 153–154

Service Model layer, 605, 609

service providers, 152, 153

ServiceContainer class, 154, 167

ServiceContractAttribute metadata attribute, 611

ServiceMetadataBehavior service behavior, 622

WCF service and, 622–625

services, 152. *See also specific services*

interfaces, 152

ServiceType property, 282, 389

Session State

icon, 68

Mode Settings, 69

page, 227

SET command, 81

Set Default Provider link button, 405

“Set Default Provider...” link button, 440

Set default provider task form, 493

Set Feature Delegation, 75

SetAttributeValue, 86

SetDefaultProvider method, 516–517

SetItemGroup method, 257

SetPropertyValue method, 164

Settings property, 409

SetUIReadOnly method, 210

severity event types, 542

<sharedListeners> section, 558, 559

ShellMainForm class, 169–170

ShouldTrace method, of SourceSwitch, 554

shutdownTimeLimit attribute, 40

Site class, 95–96, 596

members, 596

methods, 96

Site Level, 27

SITE object, 79

SiteCollection container, 103

<siteDefaults> element, 45

Siteld property, 593

site-level configuration file, 131–133

“SiteName,” 29

“SiteName/AppName,” 29–30

sites, 43

<sites>, 43–45

element, 95

Sites collection property, 103

Sites node (Web Sites node), 62, 66–68

SmallImage property, 165

smpAffinitized attribute, 42

smpProcessorAffinityMask attribute, 42

SourceFilter filter, 538, 568–570

constructor, 569

lisTraceListener and, 569

implementation, 568

SourceLevels enumeration, 542–543

members, 542–543

<sources> section, 551

SourceSwitch class, 538, 551

under the hood, 553–557

Level property, 554

OnValueChanged method of, 553

properties, 553

ShouldTrace method of, 554

SQL Server-specific classes, ADO.NET, 398

SqlDataReader, 298, 299

SqlRssProvider, 466–477, 491

class, 467–469

commandText, 474

commandType, 474

database, 477

implementation, 467–477

Initialize method, 471–473

Item class, 469–470

LoadRss method, 475

member fields, 480

private fields, 475

registration, 475–477

RssHelper class, 470–471

sslFlags attribute, 55

Start event type, 541, 542

Start method

ApplicationPool class, 89

Site class, 96

StartAsyncTask method, 204, 205, 206, 235, 265, 359, 378, 441

startupTimeLimit attribute, 40

State property, 591

<staticContent> configuration section, 119, 121

schema, 124

stop button, 222

Stop event type, 541, 542

Stop method

ApplicationPool class, 89

Site class, 96

strongly-named assemblies, 269, 283, 293, 303, 311, 320, 322, 334, 340, 421

strongly-typed

- objects, 134–135
 - benefits, 135
 - <myConfigSection> and, 134–135
- properties, IntelliSense support for, 135, 208, 336, 340, 362, 390

SupportsScope property, 282, 389

Suspend event type, 541, 542

svcutil.exe tool, 627

- WCF client and, 625, 627–632
- web.config file and, 630–631

Switch base class, 538, 553

- OnValueChanged method of, 553

switches, 538, 550

- as configurable components, 551
- instantiation and attachment, 539, 550–557
 - declarative, 550–554
 - imperative, 554–557
- local, 551
- shared, 539, 551

<switches> element, 551

SwitchSetting property, 553

<system.applicationHost> section group, 108

- accessing configuration sections in, 108–113
 - adding application pool, 108–109
 - adding binding, 110–111
 - adding virtual directory, 112–113
 - adding Web application, 111–112
 - adding Web site, 109–110
- child elements, 36–45

<system.diagnostics> section, 551, 558

System.ServiceModel.Activation.HttpHandler HTTP handler, 618

System.ServiceModel.Activation.ServiceBuildProvider build provider, 618, 619

<system.web> element, 25

System.Web.Security.RoleProvider, 410

<system.webServer> section group, 45, 340

- <myConfigSection> configuration section in, 125, 128
- sections, 45–60

System.Windows.Forms.GroupBox control, 199

T

task forms, 150. *See also specific task forms*

task items. *See specific task items*

task panels, 147, 149

TaskItem class, 158–159

- properties, 158–159
- subclasses, 159–163

TaskList class, 163–165, 216–217

- definition, 163
- GetTaskItems method, 216, 217
 - implementation, 216–220
- methods, 164–165

TaskListCollection class, 166

tasks, 149–151

Tasks property, 193

- MyCollectionPage overrides, 243
- MyConfigSectionPage overrides, 220–221
- RssPage module page overrides, 517
- UrlRewriterPage overrides, 366–367

TCP transport protocol, 639

- incoming requests over, 640

Text string property, 159

TextBox control, 184, 185

textbox controls, callback methods for TextChanged event of, 507

TextChanged event of textbox controls, callback methods for, 507

TextTaskItem

- class, 159
- task item, 174

TimeElapsed property, 593

TimeInModule property, 593

Title property, 165

“TRACE” conditional compilation symbol, 539, 550

- defining, 550

trace events, 537

- adding, 539, 546–550

trace filters, 538, 562

- instantiation and attachment, 540, 562–570

trace listeners, 538

- instantiation and attachment, 539, 557–562
 - declarative, 558–560
 - imperative, 560–562
- local, 558
- shared, 539, 558

Trace Output file, 576–578

trace sources, 538

- instantiation, 540–545

TraceData method, 546–547

TraceEvent method, 546

- overloads, 546

TraceEventCache object, 558

TraceEventType enumeration, 541–542

- members, 541–542

TraceFilter base class, 562–563

TraceInformation method, 547–548

TraceListener base class, 557

- tracing methods of, 557–558

traceSource attribute, 578

- <rss> configuration section and, 578
- RSS_Schema.xml file and, 578–579
- RssSection class and, 579–580
- RssService class and, 578

TraceSource class, 538

- constructors, 540
- RssService class and, 543–545

TraceSource property, 585

- RssSectionInfo class and, 585–586

TraceTransfer method, 548

tracing, 537–570

- components, 537–540
- configurable, 578–587
- tasks, 538–540
 - from within code, 538–550
 - from configuration file, 539–540, 550–570

Transfer event type, 541, 542

transport protocols, 110, 111, 608

- TCP 639

type attribute

- <attribute> element, 120
- <handlers> section, 50
- <modules> section, 52

Type combo box, 325, 326

type-checking support, compiler, 135, 208, 336, 340, 362, 390

U

unattended setup option, 20–21

unattend.xml file, 20, 21

Unload member, 594

“Update collection item” link, 181, 240, 242

UPDATE database operation, 79, 81

update dependencies, 12

- IIS-ASP, 12
- IIS-ASPNET, 12
- IIS-LegacyScripts, 12
- IIS-ManagementConsole, 12
- IIS-ManagementScriptingTools, 12
- IIS-ManagementService, 12
- IIS-NetFxExtensibility, 12
- IIS-WebServer, 12

“Update URL rewriter rule” link, 347, 348, 364, 366

UpdateChannelSettings method, 530–531

UpdateCollectionItem method, 279–280

- of MyCollectionPage, 246–247

UpdateCollectionItemIdentifier method, 185, 243

UpdateRequestCache event, 317

UpdateSettings method, 275–276

“UpdateSettings” string value, 188

UpdateTasks method, 173, 174

UpdateUI method, 170

UpdateUIState method, 201

UpdateUrlRewriterRule method, 369–370, 387

upgrading, IIS7 setup and, 21

up-to-date runtime data, for IIS7 runtime objects, 597–598

Url property, 593

<urlRewriter> configuration section, 338

- registering, 339–340

UrlRewriter.browser file, content of, 395

UrlRewriterConsoleApplication, 344

UrlRewriterControlAdapter, 394

UrlRewriterHandlerFactory HTTP handler factory, 334–335

- problems with, 337

UrlRewriterHandlerFactoryProj, Visual Studio and, 334

UrlRewriterHtmlTextWriter class, 394–395

UrlRewriterModule HTTP module, 318–321

- applicationHost.config file and registering, 322–323
- compilation options, 322
- implementation, 320–321
- plugging into integrated request processing pipeline, 322–332
 - problems with, 337
- registration, IIS7 Web-server level
 - declarative option, 322, 323
 - graphical option/IIS7 Manager, 322, 323–327
 - imperative option, 322, 327–328
 - undo, 329
- registration levels, 322
- registration, URLRewriterModuleCh8 application, 329–332
 - declarative option, 329
 - graphical/IIS7 Manager option, 329–331
 - imperative option, 331–332
- role of, 318, 319
- using, 332–333

UrlRewriterModule managed module

- configurability, 391–393
 - problems, 390–391
- configuration support for, 338–339, 390
- graphical management support for, 346–390
 - client-side managed code, 346–381
 - server-side managed code, 381–390
- imperative management classes, testing of, 344–345
- imperative management support for, 340–345, 390
- rewriting non-ASP.NET URLs, 393
- postback problems, 393–395

UrlRewriterModuleCh8 application, UrlRewriterModule HTTP module registration with, 329–332

UrlRewriterModuleConsoleApplication project, 327–328
UrlRewriterModuleProvider custom module provider, 387–389

registration, 379–380

UrlRewriterModuleService server-side class, 351, 382–383

UrlRewriterModuleServiceProxy class, 350–351

UrlRewriterPage class, 350–358

ModulePage class and, 358

UrlRewriterPage module page, 346, 347, 351

communications with back-end server, 348–349

InitializeListPage method and, 358–359

OnActivated method and, 359

OnListViewDoubleClick method and, 370–371

OnListViewKeyUp method and, 371

OnListViewSelectedIndexChanged method and, 370

overrides Tasks property, 366–367

PageTaskList class, 364–366

registration, 380–381

UrlRewriterProj, **Visual Studio** and, 320, 322, 325, 329

UrlRewriterProj2, 340

ImperativeManagement directory, 340, 341

setup, 340

Visual Studio and, 340, 344

UrlRewriterRule class, 341–342

UrlRewriterRuleInfo class, 362

UrlRewriterRuleListViewItem class, 363

UrlRewriterRules class, 342–343

tasks of, 343

<urlRewriterRules> Collection element, 339

UrlRewriterRuleTaskForm task form, 347, 348

communications with back-end server, 349–351

constructors, 375–376

implementation, 371–375

InitializeComponent method and, 376–378

OnAccept method and, 378

OnWorkerCompleted method and, 380

OnWorkerDoWork method and, 378–379

URLREWRITER_schema.xml file, content of, 339

UrlRewriterSection class, 343–344

user membership service, 397

user profile service, 397

UserData property, 161, 162

userName attribute, 40

V

Validate method, 433

ValidateUserInputs method, 213

Value property, 553

VDIR object, 79

verb attribute, 50

Verb property, 593

Verbose event type, 541, 542

VerticalLayoutPanel, 171

View Application Pools link, 149

“View collections items” link, 180, 185, 216, 219, 225

adding to MyConfigSectionPage, 216–221

ViewCollectionItems method, 219–220

virtual directory, 44, 45

adding, in <system.applicationHost>, 112–113

Virtual Directory Level, 27

VirtualDirectory class, 100–101

<virtualDirectory> elements, 44, 100, 101

VirtualDirectoryCollection class, 101

<virtualDirectoryDefaults> element, 45

VirtualPath, 594

Visual Studio

configuration, 283

console applications added in, 105, 106, 107, 108,

134, 141

C#, 128, 131

IIS7 Manager and, 285

IntelliSense support for strongly-typed properties, 135,

208, 336, 340, 362, 390

MyApplication project and, 422

MyConfigSection added in, 176

.NET Framework and release cycle of, 28

private key, 283

Rss HandlerProj and, 293, 306, 311

UrlRewriterHandlerFactoryProj and, 334

UrlRewriterProj and, 320, 322, 325, 329

UrlRewriterProj2 and, 340, 344

Web application creation, 134

W

Warning event type, 541, 542

WAS. See **Windows Process Activation Service**

WAS-WindowsActivationService, 7

WCF. See **Windows Communications Foundation**

WCF Visual Studio Extensions, 606

Web applications, 146

adding, in <system.applicationHost>, 111–112

BugReportManagerServiceHost, 646

Default2.aspx file and, 646–647

Web garden, 42

Web sites

adding, in <system.applicationHost>, 109–110

BugReportManagerServiceHost, 636

Default.aspx file of, 636–638

Web sites node. See **Sites node**

web.config file, 570–571, 601

svcutil.exe and, 630–631

WebMgrShellApplication class, 153, 167

GetService method of, 168

WebServiceHandlerFactory handler factory, 334

Win32ManagementHost, 152

Windows Communications Foundation (WCF), 52, 605

behaviors, 611

bug report manager system and, 607–649

client, 625

adding Web reference in, 625–627

developing, 625–635

svcutil.exe tool and, 625, 627–632

endpoints, 608, 610, 638

addresses, 609

bindings, 608–609, 610, 638–646

contracts, 609

integration of ASP.NET and

in IIS7 environment, 605–649

taking advantage of, 636–638

layered framework, 605, 609

required software, 605–606

serialization/deserialization in, 613

service behaviors, 622

service contract, 611

developing, 611–614

implementing, 614–617

Service Model, 605, 609–610

attribute-based programming, 609

configuration-based programming, 609

imperative programming, 609

services, 608

administrative tasks, 619–625

advantages, 636–638

developing, 610–611

hosting, 617–619

imperative tasks, 610–619

ServiceMetadataBehavior and, 622–625

Windows Features dialog, 13–14

Windows Process Activation Service (WAS), 16, 34–35

<windowsAuthentication enabled> element, 57

wizard forms, 150–151

WorkerProcess class, 85, 587, 590–591, 594

properties, 591

WorkerProcessCollection class, 85, 103, 587, 590

members, 590

WorkerProcesses collection property, 103

WorkerProcessState enumeration, 591

workflow

for adding new provider to provider-based service, 402, 416, 420–435

for displaying ProviderConfigurationConsolidatedPage module list page, 401, 415, 416–418

for extending integrated graphical management system, 485

for updating provider of provider-based service, 416, 435–436

for viewing providers of provider-based service, 416, 418–419

workspace pane, 63

World Wide Web. See WWW

WriteAttribute method, 301, 395

WWW (World Wide Web) Publishing Service, 35–36

X

XML elements/attributes. See also specific XML elements and attributes

APPCMD tool and, 76–81

applicationHost.config file with, 77–79

IIS Manager and, 62–76

IIS7/ASP.NET integrated declarative schema extension markup language, 117–124

categories, 117–118

XML nodes, 481

XmlDataDocument, 483, 484

XmlDocument, 483, 484

XmlReader class, 301, 484

XmlRssProvider, 448, 477–485, 491

implementation, 478–480

Initialize method, 480–482

LoadRss method, 482–483

member fields, 480

registration, 484–485

sample XML document, 481

XmlWriter class, 301, 484

XmlWriterSettings object, 300

XPath engine, 483

XPath expressions, 481–482

XPathDocument, 483, 484

XPathNavigator

random-access XML API, 302

XPath engine, 483

XPathNodeIterator, 482



Take your library wherever you go.

Now you can access more than 200 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

• ASP.NET	• .NET
• C#/C++	• Open Source
• Database	• PHP/MySQL
• General	• SQL Server
• Java	• Visual Basic
• Mac	• Web
• Microsoft Office	• XML