

Microsoft

Programming

Microsoft®

ASP.NET 4



Dino Esposito

Programming Microsoft® ASP.NET 4

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2011 by Dino Esposito

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2011920853
ISBN: 978-0-7356-4338-3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Roger LeBlanc
Editorial Production: Waypoint Press
Technical Reviewer: Scott Galloway
Cover: Tom Draper Design

Body Part No. X17-45994

To Silvia, with love

Contents at a Glance

Part I	The ASP.NET Runtime Environment	
1	ASP.NET Web Forms Today	3
2	ASP.NET and IIS	27
3	ASP.NET Configuration	63
4	HTTP Handlers, Modules, and Routing	119
Part II	ASP.NET Pages and Server Controls	
5	Anatomy of an ASP.NET Page	169
6	ASP.NET Core Server Controls	217
7	Working with the Page	269
8	Page Composition and Usability	319
9	ASP.NET Input Forms	365
10	Data Binding	411
11	The <i>ListView</i> Control	471
12	Custom Controls	513
Part III	Design of the Application	
13	Principles of Software Design	565
14	Layers of an Application	593
15	The Model-View-Presenter Pattern	615
Part IV	Infrastructure of the Application	
16	The HTTP Request Context	645
17	ASP.NET State Management	675
18	ASP.NET Caching	721
19	ASP.NET Security	779
Part V	The Client Side	
20	Ajax Programming	839
21	jQuery Programming	899

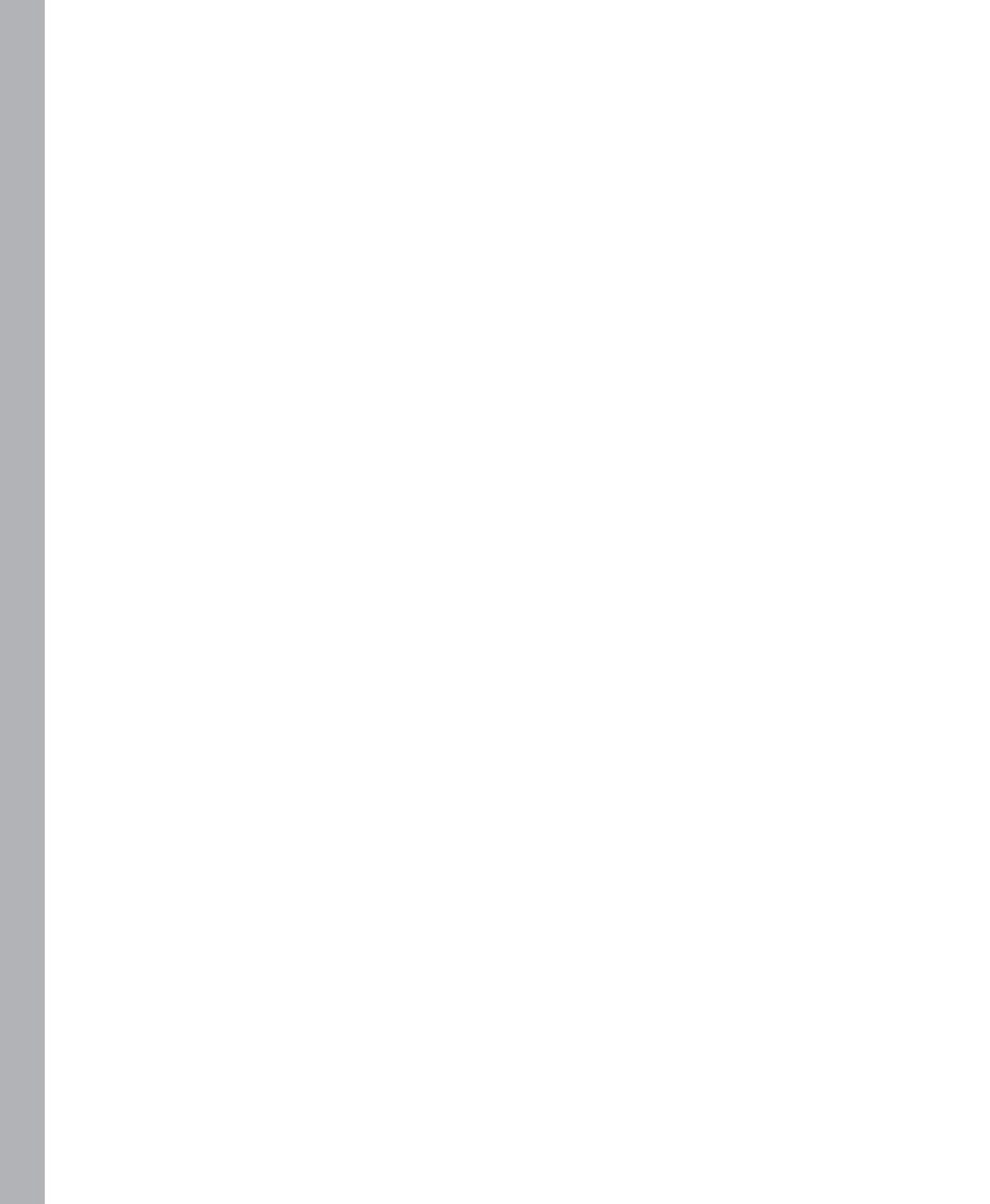


Table of Contents

Acknowledgments	xvii
Introduction	xix

Part I The ASP.NET Runtime Environment

1 ASP.NET Web Forms Today	3
The Age of Reason of ASP.NET Web Forms	4
The Original Strengths	4
Today's Perceived Weaknesses	8
How Much Is the Framework and How Much Is It You?	11
The AJAX Revolution	14
Moving Away from Classic ASP.NET	15
AJAX as a Built-in Feature of the Web	19
ASP.NET of the Future	20
ASP.NET MVC	21
ASP.NET Web Pages	25
Summary	26
2 ASP.NET and IIS	27
The Web Server Environment	28
A Brief History of ASP.NET and IIS	28
The Journey of an HTTP Request in IIS	31
Some New Features in IIS 7.5	37
Deploying ASP.NET Applications	39
XCopy Deployment for Web Sites	40
Packaging Files and Settings	43
Site Precompilation	52
Configuring IIS for ASP.NET Applications	55
Application Warm-up and Preloading	59

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

3	ASP.NET Configuration	63
	The ASP.NET Configuration Hierarchy	63
	Configuration Files	64
	The <i><location></i> Section	68
	The <i><system.web></i> Section	71
	Other Top-Level Sections	105
	Managing Configuration Data	110
	Using the Configuration API	110
	Encrypting a Section	113
	Summary	117
4	HTTP Handlers, Modules, and Routing	119
	Writing HTTP Handlers	121
	The <i>IHttpHandler</i> Interface	121
	The Picture Viewer Handler	128
	Serving Images More Effectively	133
	Advanced HTTP Handler Programming	141
	Writing HTTP Modules	149
	The <i>IHttpModule</i> Interface	149
	A Custom HTTP Module	151
	Examining a Real-World HTTP Module	154
	URL Routing	156
	The URL Routing Engine	157
	Routing in Web Forms	160
	Summary	165
Part II ASP.NET Pages and Server Controls		
5	Anatomy of an ASP.NET Page	169
	Invoking a Page	170
	The Runtime Machinery	170
	Processing the Request	174
	The Processing Directives of a Page	179
	The <i>Page</i> Class	190
	Properties of the <i>Page</i> Class	191
	Methods of the <i>Page</i> Class	194
	Events of the <i>Page</i> Class	198
	The Eventing Model	199
	Asynchronous Pages	201

The Page Life Cycle	209
Page Setup	209
Handling the Postback	212
Page Finalization	214
Summary	215
6 ASP.NET Core Server Controls	217
Generalities of ASP.NET Server Controls	218
Properties of the <i>Control</i> Class	218
Methods of the <i>Control</i> Class	228
Events of the <i>Control</i> Class	229
Other Features	230
HTML Controls	235
Generalities of HTML Controls	236
HTML Container Controls	239
HTML Input Controls	246
The <i>HtmImage</i> Control	252
Web Controls	253
Generalities of Web Controls	253
Core Web Controls	256
Miscellaneous Web Controls	262
Summary	268
7 Working with the Page	269
Dealing with Errors in ASP.NET Pages	269
Basics of Exception Handling	270
Basics of Page Error Handling	272
Mapping Errors to Pages	278
Error Reporting	283
Page Personalization	285
Creating the User Profile	285
Interacting with the Page	292
Profile Providers	300
Page Localization	303
Making Resources Localizable	304
Resources and Cultures	308
Adding Resources to Pages	312
Using Script Files	312
Using Cascading Style Sheets and Images	315
Summary	317

8	Page Composition and Usability	319
	Page Composition Checklist	319
	Working with Master Pages	320
	Writing a Content Page	323
	Processing Master and Content Pages	329
	Programming the Master Page	333
	Styling ASP.NET Pages	336
	Page Usability Checklist	344
	Cross-Browser Rendering	344
	Search Engine Optimization	348
	Site Navigation	351
	Configuring the Site Map	357
	Testing the Page	361
	Summary	364
9	ASP.NET Input Forms	365
	Programming with Forms	365
	The <i>HtmlForm</i> Class	366
	Multiple Forms	368
	Cross-Page Postings	374
	Validation Controls	379
	Generalities of Validation Controls	379
	Gallery of Controls	382
	Special Capabilities	387
	Working with Wizards	397
	An Overview of the <i>Wizard</i> Control	397
	Adding Steps to a Wizard	402
	Navigating Through the Wizard	405
	Summary	409
10	Data Binding	411
	Foundation of the Data Binding Model	411
	Feasible Data Sources	412
	Data-Binding Properties	415
	Data-Bound Controls	421
	List Controls	421
	Iterative Controls	427
	View Controls	432
	Data-Binding Expressions	434

Simple Data Binding	434
The <i>DataBinder</i> Class	436
Managing Tables of Data.....	438
The <i>GridView</i> 's Object Model	439
Binding Data to the Grid	443
Working with the <i>GridView</i>	451
Data Source Components	456
Internals of Data Source Controls	456
The <i>ObjectDataSource</i> Control.....	459
Summary.....	469
11 The <i>ListView</i> Control.....	471
The <i>ListView</i> Control.....	471
The <i>ListView</i> Object Model	472
Defining the Layout of the List.....	479
Building a Tabular Layout	480
Building a Flow Layout.....	485
Building a Tiled Layout	487
Styling the List	493
Working with the <i>ListView</i> Control	496
In-Place Editing	496
Conducting the Update.....	499
Inserting New Data Items	501
Selecting an Item	505
Paging the List of Items.....	507
Summary.....	511
12 Custom Controls	513
Extending Existing Controls	514
Choosing a Base Class.....	514
A Richer <i>HyperLink</i> Control.....	515
Building Controls from Scratch	518
Base Class and Interfaces	518
Choosing a Rendering Style	520
The <i>SimpleGaugeBar</i> Control	522
Rendering the <i>SimpleGaugeBar</i> Control.....	527
Building a Data-Bound Control	533
Key Features.....	533
The <i>GaugeBar</i> Control	535

Building a Composite Templated Control	543
Generalities of Composite Data-Bound Controls	544
The <i>BarChart</i> Control	547
Adding Template Support	556
Summary	561

Part III Design of the Application

13 Principles of Software Design	565
The Big Ball of Mud	566
Reasons for the Mud	566
Alarming Symptoms	567
Universal Software Principles	569
Cohesion and Coupling	569
Separation of Concerns	571
SOLID Principles	573
The Single Responsibility Principle	573
The Open/Closed Principle	575
Liskov's Substitution Principle	576
The Interface Segregation Principle	579
The Dependency Inversion Principle	580
Tools for Dependency Injection	583
Managed Extensibility Framework at a Glance	584
Unity at a Glance	587
Summary	591
14 Layers of an Application	593
A Multitiered Architecture	594
The Overall Design	594
Methodologies	595
The Business Layer	596
Design Patterns for the BLL	596
The Application Logic	602
The Data Access Layer	605
Implementation of a DAL	605
Interfacing the DAL	608
Using an Object/Relational Mapper	610
Beyond Classic Databases	613
Summary	614

15	The Model-View-Presenter Pattern	615
	Patterns for the Presentation Layer	615
	The MVC Pattern	616
	The MVP Pattern	619
	The MVVM Pattern	621
	Implementing Model View Presenter	623
	Abstracting the View	624
	Creating the Presenter	626
	Navigation	632
	Testability in Web Forms with MVP	636
	Writing Testable Code	637
	Testing a Presenter Class	639
	Summary	642

Part IV Infrastructure of the Application

16	The HTTP Request Context	645
	Initialization of the Application	645
	Properties of the <i>HttpApplication</i> Class	645
	Application Modules	646
	Methods of the <i>HttpApplication</i> Class	647
	Events of the <i>HttpApplication</i> Class	648
	The <i>global.asax</i> File	651
	Compiling <i>global.asax</i>	652
	Syntax of <i>global.asax</i>	653
	The <i>HttpContext</i> Class	656
	Properties of the <i>HttpContext</i> Class	656
	Methods of the <i>HttpContext</i> Class	658
	The <i>Server</i> Object	660
	Properties of the <i>HttpServerUtility</i> Class	660
	Methods of the <i>HttpServerUtility</i> Class	660
	The <i>HttpResponse</i> Object	663
	Properties of the <i>HttpResponse</i> Class	664
	Methods of the <i>HttpResponse</i> Class	667
	The <i>HttpRequest</i> Object	670
	Properties of the <i>HttpRequest</i> Class	670
	Methods of the <i>HttpRequest</i> Class	673
	Summary	674

17	ASP.NET State Management	675
	The Application's State	676
	Properties of the <i>HttpApplicationState</i> Class	676
	Methods of the <i>HttpApplicationState</i> Class	677
	State Synchronization	678
	Tradeoffs of Application State	679
	The Session's State	680
	The Session-State HTTP Module	680
	Properties of the <i>HttpSessionState</i> Class	685
	Methods of the <i>HttpSessionState</i> Class	686
	Working with a Session's State	686
	Identifying a Session	687
	Lifetime of a Session	693
	Persist Session Data to Remote Servers	695
	Persist Session Data to SQL Server	699
	Customizing Session State Management	704
	Building a Custom Session State Provider	704
	Generating a Custom Session ID	708
	The View State of a Page	710
	The <i>StateBag</i> Class	711
	Common Issues with View State	712
	Programming the View State	715
	Summary	720
18	ASP.NET Caching	721
	Caching Application Data	721
	The <i>Cache</i> Class	722
	Working with the ASP.NET <i>Cache</i>	725
	Practical Issues	732
	Designing a Custom Dependency	737
	A Cache Dependency for XML Data	739
	SQL Server Cache Dependency	743
	Distributed Cache	744
	Features of a Distributed Cache	745
	AppFabric Caching Services	747
	Other Solutions	753
	Caching ASP.NET Pages	755
	ASP.NET and the Browser Cache	756

Making ASP.NET Pages Cacheable	758
The <i>HttpCachePolicy</i> Class.....	763
Caching Multiple Versions of a Page.....	765
Caching Portions of ASP.NET Pages.....	768
Advanced Caching Features	774
Summary.....	777
19 ASP.NET Security	779
Where the Threats Come From.....	779
The ASP.NET Security Context	781
Who Really Runs My ASP.NET Application?	781
Changing the Identity of the ASP.NET Process	784
The Trust Level of ASP.NET Applications.....	786
ASP.NET Authentication Methods	789
Using Forms Authentication	791
Forms Authentication Control Flow.....	792
The <i>FormsAuthentication</i> Class.....	796
Configuration of <i>Forms</i> Authentication	798
Advanced Forms Authentication Features	801
The Membership and Role Management API	806
The <i>Membership</i> Class	807
The Membership Provider.....	812
Managing Roles.....	817
Quick Tour of Claims-Based Identity	821
Claims-Based Identity.....	822
Using Claims in ASP.NET Applications	824
Security-Related Controls	825
The <i>Login</i> Control	826
The <i>LoginName</i> Control.....	828
The <i>LoginStatus</i> Control.....	829
The <i>LoginView</i> Control.....	830
The <i>PasswordRecovery</i> Control.....	832
The <i>ChangePassword</i> Control.....	833
The <i>CreateUserWizard</i> Control.....	834
Summary.....	835

Part V The Client Side

20	Ajax Programming	839
	The Ajax Infrastructure	840
	The Hidden Engine of Ajax	840
	JavaScript and Ajax	845
	Partial Rendering in ASP.NET	851
	The <i>ScriptManager</i> Control	852
	The <i>UpdatePanel</i> Control	860
	Considerations Regarding Partial Rendering	865
	Configuring for Conditional Refresh	866
	Giving Feedback to the User	870
	The Ins and Outs of Partial Rendering	876
	REST and Ajax	879
	Scriptable Services	880
	JSON Payloads	890
	JavaScript Client Code	893
	Summary	897
21	jQuery Programming	899
	Power to the Client	899
	Programming within the Browser	900
	The Gist of jQuery	903
	Working with jQuery	905
	Detecting DOM Readiness	906
	Wrapped Sets	908
	Operating on a Wrapped Set	915
	Manipulating the DOM	920
	The jQuery Cache	923
	Ajax Capabilities	925
	Cross-Domain Calls	929
	Summary	932
	Index	933

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Acknowledgments

As is usual for a book, the cover of this book shows only the name of the author, but in no way can an author produce a book all alone. In fact, a large ensemble of people made this book happen. First, I want to thank **Devon Musgrave** for developing the idea and scheduling new books for me to author at an amazingly quick pace for the next two years!

Next comes **Roger LeBlanc**, whom I've had the pleasure to have as a copy editor on previous books of mine—including the first edition of this *Programming ASP.NET* book (Microsoft Press, 2003). This time, Roger assisted me almost every day—not just as the copy editor, but also as the development manager. I dare to say that as my English gets a little bit better every year, the amount of copy editing required does not amount to much for a diligent editor like Roger. So he decided to take on extra tasks.

In the middle of this project, I had to take a short break to have back surgery. The surgery increased the number of lengths I could swim and improved my tennis game, especially the penetration of my first serve and my top-spin backhand, but it put a temporary stop to my progress on the book. As a result, Roger and I had to work very hard to get the book completed on a very tight schedule.

Steve Sagman handled the production end of the book—things like layout, art, indexing, proofreading, prepping files for printing, as well as the overall project management. Here, too, the tight schedule required a greater effort than usual. Steve put in long days as well as weekends to keep everything on track and to ensure this edition equals or exceeds the high standards of previous editions.

Scott Galloway took the responsibility of ensuring that this book contains no huge technical mistakes or silly statements. As a technical reviewer, Scott provided me with valuable insights, especially about the rationale of some design decisions in ASP.NET. Likewise, he helped me understand the growing importance JavaScript (and unobtrusive JavaScript) has today for Web developers. Finally, Scott woke me up to the benefits of Twitter, as tweeting was often the quickest way to get advice or reply to him.

To all of you, I owe a monumental “Thank you” for being so kind, patient, and accurate. Working with you is a privilege and a pleasure, and it makes me a better author each time. And I still have a long line of books to author.

My final words are for Silvia, Francesco, and Michela, who wait for me and keep me busy. But I'm happy only when I'm busy.

—Dino

Introduction

In the fall of 2004, at a popular software conference I realized how all major component vendors were advertising their ASP.NET products using a new word—Ajax. Only a few weeks later, a brand new module in my popular ASP.NET master class made its debut—using Ajax to improve the user experience. At its core, Ajax is a little thing and fairly old too—as I presented the engine of it (*XmlHttpRequest*) to a C++ audience at TechEd 2000, only four weeks before the public announcement of the .NET platform.

As emphatic as it may sound, that crazy little thing called Ajax changed the way we approach Web development. Ajax triggered a chain reaction in the world of the Web. Ajax truly represents paradigm shift for Web applications. And, as the history of science proves, a paradigm shift always has a deep impact, especially in scenarios that were previously stable and consolidated. We are now really close to the day we will be able to say “the Web” without feeling the need to specify whether it contains Ajax or not. Just the Web—which has a rich client component, a made-to-measure layer of HTTP endpoints to call, and interchangeable styles.

Like it or not, the more we take the Ajax route, the more we move away from ASP.NET Web Forms. In the end, it’s just like getting older. Until recently, Web Forms was a fantastic platform for Web development. The Web, however, is now going in a direction that Web Forms can’t serve in the same stellar manner.

No, you didn’t pick up the wrong book, and you also did not pick up the wrong technology for your project.

It’s not yet time to cease ASP.NET Web Forms development. However, it’s already time for you to pay a lot more attention to aspects of Web development that Web Forms specifically and deliberately shielded you from for a decade—CSS, JavaScript, and HTML markup.

In my ASP.NET master class, I have a lab in which I first show how to display a data-bound grid of records with cells that trigger an Ajax call if clicked. I do that in exactly the way one would do it—as an ASP.NET developer. Next, I challenge attendees to rewrite it without inline script and style settings. And yes—a bit perversely—I also tell anyone who knows jQuery not to use it. The result is usually a thoughtful and insightful experience, and the code I come up with gets better every time. ASP.NET Web Forms is not dead, no matter what ASP.NET MVC—the twin technology—can become. But it’s showing signs of age. As a developer, you need to recognize that and revive it through robust injections of patterns, JavaScript and jQuery code, and Ajax features.

In this book, I left out some of the classic topics you found in earlier versions, such as ADO.NET and even LINQ-to-SQL. I also reduced the number of pages devoted to controls. I brought in more coverage of ASP.NET underpinnings, ASP.NET configuration, jQuery, and patterns and design principles. Frankly, not a lot has changed in ASP.NET since version 2.0.

Because of space constraints, I didn't cover some rather advanced aspects of ASP.NET customization, such as expression builders, custom providers, and page parsers. For coverage of those items, my older book *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006) is still a valid reference in spite of the name, which targets the 2.0 platform. The new part of this book on principles of software design is a compendium of another pretty successful book of mine (actually coauthored with Andrea Saltarello)—*Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008).

Who Should Read This Book?

This is not a book for novice developers and doesn't provide a step-by-step guide on how to design and code Web pages. So the book is not appropriate if you have only a faint idea about ASP.NET and expect the book to get you started with it quickly and effectively. Once you have grabbed hold of ASP.NET basic tasks and features and need to consolidate them, you enter the realm of this book.

You won't find screen shots here illustrating Microsoft Visual Studio wizards, nor any mention of options to select or unselect to get a certain behavior from your code. Of course, this doesn't mean that I hate Visual Studio or that I'm not recommending Visual Studio for developing ASP.NET applications. Visual Studio is a great tool to use to write ASP.NET applications but, judged from an ASP.NET perspective, it is only a tool. This book, instead, is all about the ASP.NET core technology.

I do recommend this book to developers who have knowledge of the basic steps required to build simple ASP.NET pages and easily manage the fundamentals of Web development. This book is not a collection of recipes for cooking good (or just functional) ASP.NET code. This book begins where recipes end. It explains to you the how-it-works, what-you-can-do, and why-you-should-or-should-not aspects of ASP.NET. Beginners need not apply, even though this book is a useful and persistent reference to keep on the desk.

System Requirements

You'll need the following hardware and software to build and run the code samples for this book:

- Microsoft Windows 7, Microsoft Windows Vista, Microsoft Windows XP with Service Pack 2, Microsoft Windows Server 2003 with Service Pack 1, or Microsoft Windows 2000 with Service Pack 4.
- Any version of Microsoft Visual Studio 2010.

- Internet Information Services (IIS) is not strictly required, but it is helpful for testing sample applications in a realistic runtime environment.
- Microsoft SQL Server 2005 Express (included with Visual Studio 2008) or Microsoft SQL Server 2005, as well as any newer versions.
- The Northwind database of Microsoft SQL Server 2000 is used in most examples in this book to demonstrate data-access techniques throughout the book.
- 766-MHz Pentium or compatible processor (1.5-GHz Pentium recommended).
- 256 MB RAM (512 MB or more recommended).
- Video (800 x 600 or higher resolution) monitor with at least 256 colors (1024 x 768 High Color 16-bit recommended).
- CD-ROM or DVD-ROM drive.
- Microsoft Mouse or compatible pointing device.

Code Samples

All of the code samples discussed in this book can be downloaded from the book's Companion Content page accessible via following address:

<http://www.microsoftpressstore.com/title/9780735643383>.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site:

1. Go to *www.microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata & Updates tab
5. Click View/Submit Errata.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://www.microsoft.com/learning/booksurvey>.

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>.*

Chapter 4

HTTP Handlers, Modules, and Routing

Advice is what we ask for when we already know the answer but wish we didn't.

—Erica Jong

HTTP handlers and modules are truly the building blocks of the ASP.NET platform. Any requests for a resource managed by ASP.NET are always resolved by an HTTP handler and pass through a pipeline of HTTP modules. After the handler has processed the request, the request flows back through the pipeline of HTTP modules and is finally transformed into markup for the caller.

The *Page* class—the base class for all ASP.NET runtime pages—is ultimately an HTTP handler that implements internally the page life cycle that fires the well-known set of page events, including postbacks, *Init*, *Load*, *PreRender*, and the like. An HTTP handler is designed to process one or more URL extensions. Handlers can be given an application or machine scope, which means they can process the assigned extensions within the context of the current application or all applications installed on the machine. Of course, this is accomplished by making changes to either the site's *web.config* file or a local *web.config* file, depending on the scope you desire.

HTTP modules are classes that handle runtime events. There are two types of public events that a module can deal with. They are the events raised by *HttpApplication* (including asynchronous events) and events raised by other HTTP modules. For example, *SessionStateModule* is one of the built-in modules provided by ASP.NET to supply session-state services to an application. It fires the *End* and *Start* events that other modules can handle through the familiar *Session_End* and *Session_Start* signatures.

In Internet Information Services (IIS) 7 integrated mode, modules and handlers are resolved at the IIS level; they operate, instead, inside the ASP.NET worker process in different runtime configurations, such as IIS 7 classic mode or IIS 6.

HTTP modules and handlers are related to the theme of request routing. Originally developed for ASP.NET MVC, the URL routing engine has been incorporated into the overall ASP.NET platform with the .NET Framework 3.5 Service Pack 1. The URL routing engine is a system-provided HTTP module that hooks up any incoming requests and attempts to match the requested URL to one of the user-defined rewriting rules (known as *routes*). If a match exists, the module locates the HTTP handler that is due to serve the route and goes with it. If no match is found, the request is processed as usual in Web Forms, as if no URL routing engine was ever in the middle. What makes the URL routing engine so beneficial to

applications? It actually enables you to use free-hand and easy-to-remember URLs that are not necessarily bound to physical files in the Web server.

In this chapter, we'll explore the syntax and semantics of HTTP handlers, HTTP modules, and the URL routing engine.

The ISAPI Extensibility Model of IIS

A Web server generally provides an application programming interface (API) for enhancing and customizing the server's capabilities. Historically speaking, the first of these extension APIs was the Common Gateway Interface (CGI). A CGI module is a new application that is spawned from the Web server to service a request. Nowadays, CGI applications are almost never used because they require a new process for each HTTP request, and this approach poses severe scalability issues and is rather inadequate for high-volume Web sites.

More recent versions of Web servers supply an alternate and more efficient model to extend the capabilities of the server. In IIS, this alternative model takes the form of the ISAPI interface. When the ISAPI model is used, instead of starting a new process for each request, the Web server loads a made-to-measure component—namely, a Win32 dynamic-link library (DLL)—into its own process. Next, it calls a well-known entry point on the DLL to serve the request. The ISAPI component stays loaded until IIS is shut down and can service requests without any further impact on Web server activity. The downside to such a model is that because components are loaded within the Web server process, a single faulty component can tear down the whole server and all installed applications. Some effective countermeasures have been taken over the years to smooth out this problem. Today, IIS installed applications are assigned to application pools and each application pool is served by a distinct instance of a worker process.

From an extensibility standpoint, however, the ISAPI model is less than optimal because it requires developers to create Win32 unmanaged DLLs to endow the Web server with the capability of serving specific requests, such as those for ASPX resources. Until IIS 7 (and still in IIS 7 when the classic mode is configured), requests are processed by IIS and then mapped to some ISAPI (unmanaged) component. This is exactly what happens with plain ASPX requests, and the ASP.NET ISAPI component is *aspnet_isapi.dll*. In IIS 7.x integrated mode, you can add managed components (HTTP handlers and HTTP modules) directly at the IIS level. More precisely, the IIS 7 integrated mode merges the ASP.NET internal runtime pipeline with the IIS pipeline and enables you to write Web server extensions using managed code. This is the way to go.

Today, if you learn how to write HTTP handlers and HTTP modules, you can use such skills to customize how any requests that hit IIS are served, and not just requests that would be mapped to ASP.NET. You'll see a few examples in the rest of the chapter.

Writing HTTP Handlers

As the name suggests, an HTTP handler is a component that handles and processes a request. ASP.NET comes with a set of built-in handlers to accommodate a number of system tasks. The model, however, is highly extensible. You can write a custom HTTP handler whenever you need ASP.NET to process certain types of requests in a nonstandard way. The list of useful things you can do with HTTP handlers is limited only by your imagination.

Through a well-written handler, you can have your users invoke any sort of functionality via the Web. For example, you could implement click counters and any sort of image manipulation, including dynamic generation of images, server-side caching, or obstructing undesired linking to your images. More in general, an HTTP handler is a way for the user to send a command to the Web application instead of just requesting a particular page.

In software terms, an HTTP handler is a relatively simple class that implements the *IHttpHandler* interface. An HTTP handler can either work synchronously or operate in an asynchronous way. When working synchronously, a handler doesn't return until it's done with the HTTP request. An asynchronous handler, on the other hand, launches a potentially lengthy process and returns immediately after. A typical implementation of asynchronous handlers is asynchronous pages. An asynchronous HTTP handler is a class that implements a different interface—the *IHttpAsyncHandler* interface.

HTTP handlers need be registered with the application. You do that in the application's *web.config* file in the `<httpHandlers>` section of `<system.web>`, in the `<handlers>` section of `<system.webServer>` as explained in Chapter 3, "ASP.NET Configuration," or in both places. If your application runs under IIS 7.x in integrated mode, you can also configure HTTP handlers via the Handler Mappings panel of the IIS Manager.

The *IHttpHandler* Interface

Want to take the splash and dive into HTTP handler programming? Well, your first step is getting the hang of the *IHttpHandler* interface. An HTTP handler is just a managed class that implements that interface. As mentioned, a synchronous HTTP handler implements the *IHttpHandler* interface; an asynchronous HTTP handler, on the other hand, implements the *IHttpAsyncHandler* interface. Let's tackle synchronous handlers first.

The contract of the *IHttpHandler* interface defines the actions that a handler needs to take to process an HTTP request synchronously.

Members of the *IHttpHandler* Interface

The *IHttpHandler* interface defines only two members: *ProcessRequest* and *IsReusable*, as shown in Table 4-1. *ProcessRequest* is a method, whereas *IsReusable* is a Boolean property.

TABLE 4-1 Members of the *IHttpHandler* Interface

Member	Description
<i>IsReusable</i>	This property provides a Boolean value indicating whether the HTTP runtime can reuse the current instance of the HTTP handler while serving another request.
<i>ProcessRequest</i>	This method processes the HTTP request from start to finish and is responsible for processing any input and producing any output.

The *IsReusable* property on the *System.Web.UI.Page* class—the most common HTTP handler in ASP.NET—returns *false*, meaning that a new instance of the HTTP request is needed to serve each new page request. You typically make *IsReusable* return *false* in all situations where some significant processing is required that depends on the request payload. Handlers used as simple barriers to filter special requests can set *IsReusable* to *true* to save some CPU cycles. I'll return to this subject with a concrete example in a moment.

The *ProcessRequest* method has the following signature:

```
void ProcessRequest(HttpContext context);
```

It takes the context of the request as the input and ensures that the request is serviced. In the case of synchronous handlers, when *ProcessRequest* returns, the output is ready for forwarding to the client.

A Very Simple HTTP Handler

The output for the request is built within the *ProcessRequest* method, as shown in the following code:

```
using System.Web;
namespace AspNetGallery.Extensions.Handlers
{
    public class SimpleHandler : IHttpHandler
    {
        public void ProcessRequest(HttpContext context)
        {
            const String htmlTemplate = "<html><head><title>{0}</title></head><body>" +
                "<h1>Hello I'm: " +
                "<span style='color:blue'>{1}</span></h1>" +
                "</body></html>";

            var response = String.Format(htmlTemplate,
                "HTTP Handlers", context.Request.Path);
            context.Response.Write(response);
        }
        public Boolean IsReusable
        {
            get { return false; }
        }
    }
}
```

You need an entry point to be able to call the handler. In this context, an entry point into the handler's code is nothing more than an HTTP endpoint—that is, a public URL. The URL must be a unique name that IIS and the ASP.NET runtime can map to this code. When registered, the mapping between an HTTP handler and a Web server resource is established through the *web.config* file:

```
<configuration>
  <system.web>
    <httpHandlers>
      <add verb="*"
          path="hello.axd"
          type="Samples.Components.SimpleHandler" />
    </httpHandlers>
  </system.web>
  <system.webServer>
    <validation validateIntegratedModeConfiguration="false" />
    <handlers>
      <add name="Hello"
          preCondition="integratedMode"
          verb="*"
          path="hello.axd"
          type="Samples.Components.SimpleHandler" />
    </handlers>
  </system.webServer>
</configuration>
```

The `<httpHandlers>` section lists the handlers available for the current application. These settings indicate that *SimpleHandler* is in charge of handling any incoming requests for an endpoint named *hello.axd*. Note that the URL *hello.axd* doesn't have to be a physical resource on the server; it's simply a public resource identifier. The *type* attribute references the class and assembly that contain the handler. Its canonical format is *type[,assembly]*. You omit the assembly information if the component is defined in the *App_Code* or other reserved folders.



Important As noted in Chapter 3, you usually don't need both forms of an HTTP handler declaration in `<system.web>` and `<system.webServer>`. You need the former only if your application runs under IIS 6 (Windows Server 2003) or if it runs under IIS 7.x but is configured in classic mode. You need the latter only if your application runs under IIS 7.x in integrated mode. If you have both sections, you enable yourself to use a single *web.config* file for two distinct deployment scenarios. In this case, the `<validation>` element is key because it prevents IIS 7.x from strictly parsing the content of the configuration file. Furthermore, as discussed in Chapter 3, the `<httpHandlers>` and `<httpModules>` sections help in testing handlers and modules within Visual Studio if you're using the embedded ASP.NET Development Server (also known as, Cassini).

If you invoke the *hello.axd* URL, you obtain the results shown in Figure 4-1.

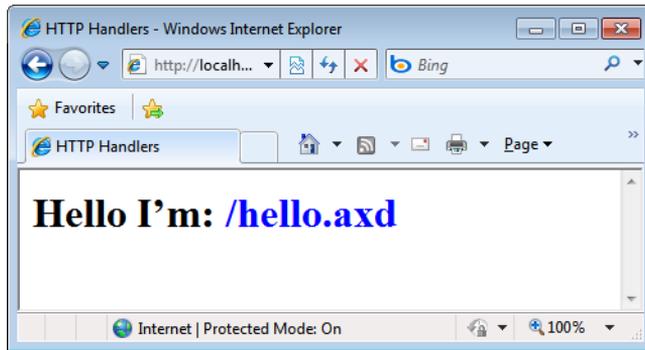


FIGURE 4-1 A sample HTTP handler that answers requests for *hello.axd*.

The technique discussed here is the quickest and simplest way of putting an HTTP handler to work, but there is more to know about the registration of HTTP handlers and there are many more options to take advantage of.



Note It's more common to use the ASHX extension for a handler mapping. The AXD extension is generally reserved for resource handlers that inject embedded content such as images, scripts, and so forth.

Registering the Handler

An HTTP handler is a class and must be compiled to an assembly before you can use it. The assembly must be deployed to the *Bin* directory of the application. If you plan to make this handler available to all applications, you can copy it to the global assembly cache (GAC). The next step is registering the handler with an individual application or with all the applications running on the Web server.

You already saw the script you need to register an HTTP handler. Table 4-2 expands a bit more on the attributes you can set up.

TABLE 4-2 Attributes Required to Register an HTTP Handler in `<system.web>`

Attribute	Description
<i>path</i>	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, <i>*.aspx</i> .
<i>type</i>	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application's private <i>Bin</i> directory and then in the system global assembly cache.
<i>validate</i>	If this attribute is set to <i>false</i> , ASP.NET loads the assembly with the handler on demand. The default value is <i>true</i> .
<i>verb</i>	Indicates the list of the supported HTTP verbs—for example, GET, PUT, and POST. The wildcard character (*) is an acceptable value and denotes all verbs.

All attributes except for *validate* are mandatory. When *validate* is set to *false*, ASP.NET delays as much as possible loading the assembly with the HTTP handler. In other words, the assembly will be loaded only when a request for it arrives. ASP.NET will not try to preload the assembly, thus catching earlier any errors or problems with it.

Additional attributes are available if you register the handler in `<system.webServer>`. They are listed in Table 4-3.

TABLE 4-3 Attributes Required to Register an HTTP Handler in `<system.webServer>`

Attribute	Description
<i>allowPathInfo</i>	If this attribute is set to <i>true</i> , the handler processes full path information in the URL or just the last section. It is set to <i>false</i> by default.
<i>modules</i>	Indicates the list of HTTP modules (comma-separated list of names) that are enabled to intercept requests for the current handler. The standard list contains only the <i>ManagedPipelineHandler</i> module.
<i>name</i>	Unique name of the handler.
<i>path</i>	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, <i>*.aspx</i> .
<i>preCondition</i>	Specifies conditions under which the handler will run. (More information appears later in this section.)
<i>requireAccess</i>	Indicates the type of access that a handler requires to the resource, either read, write, script, execute, or none. The default is script.
<i>resourceType</i>	Indicates the type of resource to which the handler mapping applies: file, directory, or both. The default option, however, is <i>Unspecified</i> , meaning that the handler can handle requests for resources that map to physical entries in the file system as well as to plain commands.
<i>responseBufferLimit</i>	Specifies the maximum size, in bytes, of the response buffer. The default value is 4 MB.
<i>scriptProcessor</i>	Specifies the physical path of the ISAPI extension or CGI executable that processes the request. It is not requested for managed handlers.
<i>type</i>	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application's private <i>Bin</i> directory and then in the system global assembly cache.
<i>verb</i>	Indicates the list of the supported HTTP verbs—for example, GET, PUT, and POST. The wildcard character (*) is an acceptable value and denotes all verbs.

The reason why the configuration of an HTTP handler might span a larger number of attributes in IIS is that the `<handlers>` section serves for both managed and unmanaged handlers. If you configure a managed handler written using the ASP.NET API, you need only *preCondition* and *name* in addition to the attributes you would specify in the `<httpHandlers>` section.

Preconditions for Managed Handlers

The *preCondition* attribute sets prerequisites for the handler to run. Prerequisites touch on three distinct areas: bitness, ASP.NET runtime version, and type of requests to respond. Table 4-4 lists and explains the various options:

TABLE 4-4 Preconditions for an IIS 7.x HTTP Handler

Precondition	Description
<i>bitness32</i>	The handler is 32-bit code and should be loaded only in 64-bit worker processes running in 32-bit emulation.
<i>bitness64</i>	The handler is 64-bit and should be loaded only in native 64-bit worker processes.
<i>integratedMode</i>	The handler should respond only to requests in application pools configured in integrated mode.
<i>ISAPIMode</i>	The handler should respond only to requests in application pools configured in classic mode.
<i>runtimeVersionv1.1</i>	The handler should respond only to requests in application pools configured for version 1.1 of the ASP.NET runtime.
<i>runtimeVersionv2.0</i>	The handler should respond only to requests in application pools configured for version 2.0 of the ASP.NET runtime.

Most of the time you use the *integratedMode* value only to set preconditions on a managed HTTP handler.

Handlers Serving New Types of Resources

In ASP.NET applications, a common scenario when you want to use custom HTTP handlers is that you want to loosen yourself from the ties of ASPX files. Sometimes you want to place a request for a nonstandard ASP.NET resource (for example, a custom XML file) and expect the handler to process the content and return some markup.

More in general, you use HTTP handlers in two main situations: when you want to customize how known resources are processed and when you want to introduce new resources. In the latter case, you probably need to let IIS know about the new resource. Again, how you achieve this depends on the configuration of the application pool that hosts your ASP.NET applications.

Suppose you want your application to respond to requests for *.report* requests. For example, you expect your application to be able to respond to a URL like */monthly.report?year=2010*. Let's say that *monthly.report* is a server file that contains a description of the report your handler will then create using any input parameters you provide.

In integrated mode, you need to do nothing special for this request to go successfully. Moreover, you don't even need to add a *.report* or any other analogous extension. You

can specify any custom URL (much like you do in ASP.NET MVC) and as long as you have a handler properly configured, it will work.

In classic mode, instead, two distinct pipelines exist in IIS and ASP.NET. The extension, in this case, is mandatory to instruct IIS to recognize that request and map it to ASP.NET, where the HTTP handler actually lives. As an example, consider that when you deploy ASP.NET MVC in classic mode you have to tweak URLs so that each controller name has an *.mvc* suffix. To force IIS to recognize a new resource, you must add a new script map via the IIS Manager, as shown in Figure 4-2.

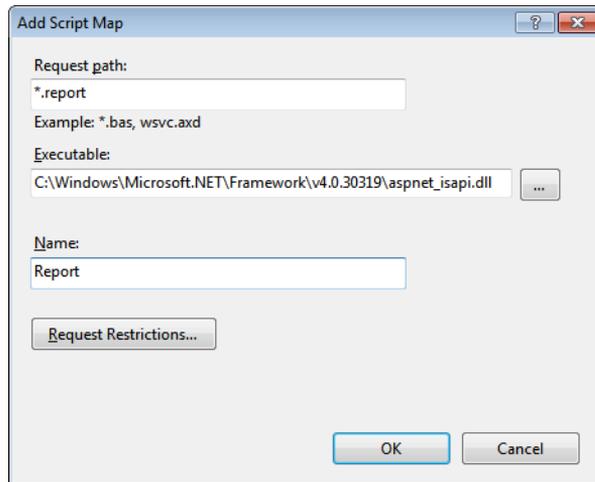


FIGURE 4-2 Adding an IIS script map for *.report* requests.

The executable is the ISAPI extension that will be bridging the request from the IIS world to the ASP.NET space. You choose the *aspnet_isapi* DLL from the folder that points to the version of the .NET Framework you intend to target. In Figure 4-2, you see the path for ASP.NET 4.



Note In Microsoft Visual Studio, if you test a sample *.report* resource using the local embedded Web server, nothing happens that forces you to register the *.report* resource with IIS. This is just the point, though. You're not using IIS! In other words, if you use the local Web server, you have no need to touch IIS; you do need to register any custom resource you plan to use with IIS before you get to production.

Why didn't we have to do anything special for our first example, *hello.axd*? Because AXD is a system extension that ASP.NET registers on its own and that sometimes also can be used for registering custom HTTP handlers. (AXD is not the recommended extension for custom handlers, however.)

Now let's consider a more complex example of an HTTP handler.

The Picture Viewer Handler

To speed up processing, IIS claims the right to personally serve some typical Web resources without going down to any particular ISAPI extensions. The list of resources served directly by IIS includes static files such as images and HTML files.

What if you request a GIF or a JPG file directly from the address bar of the browser? IIS retrieves the specified resource, sets the proper content type on the response buffer, and writes out the bytes of the file. As a result, you'll see the image in the browser's page. So far so good.

What if you point your browser to a virtual folder that contains images? In this case, IIS doesn't distinguish the contents of the folder and returns a list of files, as shown in Figure 4-3.

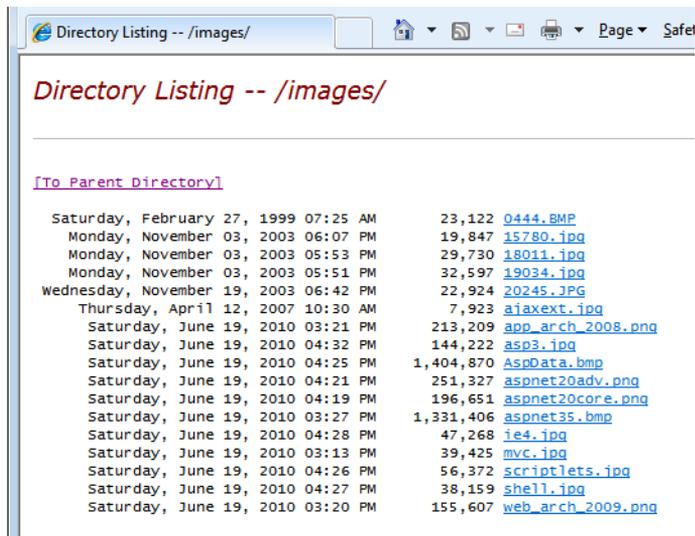


FIGURE 4-3 The standard IIS-provided view of a folder.

Wouldn't it be nice if you could get a preview of the contained pictures instead?

Designing the HTTP Handler

To start out, you need to decide how to let IIS know about your wishes. You can use a particular endpoint that, when appended to a folder's name, convinces IIS to yield to ASP.NET and provide a preview of contained images. Put another way, the idea is to bind your picture viewer handler to a particular endpoint—say, *folder.axd*. As mentioned earlier in the chapter, a fixed endpoint for handlers doesn't have to be an existing, deployed resource. You make the *folder.axd* endpoint follow the folder name, as shown here:

```
http://www.contoso.com/images/folder.axd
```

The handler processes the URL, extracts the folder name, and selects all the contained pictures.



Note In ASP.NET, the *.axd* extension is commonly used for endpoints referencing a special service. *Trace.axd* for tracing and *WebResource.axd* for script and resources injection are examples of two popular uses of the extension. In particular, the *Trace.axd* handler implements the same logic described here. If you append its name to the URL, it will trace all requests for pages in that application.

Implementing the HTTP Handler

The picture viewer handler returns a page composed of a multirow table showing as many images as there are in the folder. Here's the skeleton of the class:

```
class PictureViewerInfo
{
    public PictureViewerInfo() {
        DisplayWidth = 200;
        ColumnCount = 3;
    }
    public int DisplayWidth;
    public int ColumnCount;
    public string FolderName;
}

public class PictureViewerHandler : IHttpHandler
{
    // Override the ProcessRequest method
    public void ProcessRequest(HttpContext context)
    {
        PictureViewerInfo info = GetFolderInfo(context);
        string html = CreateOutput(info);

        // Output the data
        context.Response.Write("<html><head><title>");
        context.Response.Write("Picture Web Viewer");
        context.Response.Write("</title></head><body>");
        context.Response.Write(html);
        context.Response.Write("</body></html>");
    }

    // Override the IsReusable property
    public bool IsReusable
    {
        get { return true; }
    }
    ...
}
```

Retrieving the actual path of the folder is as easy as stripping off the *folder.axd* string from the URL and trimming any trailing slashes or backslashes. Next, the URL of the folder is mapped to a server path and processed using the .NET Framework API for files and folders to retrieve all image files:

```
private static IList<FileInfo> GetAllImages(DirectoryInfo di)
{
    String[] fileTypes = { "*.bmp", "*.gif", "*.jpg", "*.png" };
    var images = new List<FileInfo>();
    foreach (var files in fileTypes.Select(di.GetFiles).Where(files => files.Length > 0))
    {
        images.AddRange(files);
    }
    return images;
}
```

The *DirectoryInfo* class provides some helper functions on the specified directory; for example, the *GetFiles* method selects all the files that match the given pattern. Each file is wrapped by a *FileInfo* object. The method *GetFiles* doesn't support multiple search patterns; to search for various file types, you need to iterate for each type and accumulate results in an array list or equivalent data structure.

After you get all the images in the folder, you move on to building the output for the request. The output is a table with a fixed number of cells and a variable number of rows to accommodate all selected images. For each image file, a new ** tag is created through the *Image* control. The *width* attribute of this file is set to a fixed value (say, 200 pixels), causing browsers to automatically resize the image. Furthermore, the image is wrapped by an anchor that links to the same image URL. As a result, when the user clicks on an image, the page refreshes and shows the same image at its natural size.

```
private static String CreateOutputForFolder(PictureViewerInfo info, DirectoryInfo di)
{
    var images = GetAllImages(di);

    var t = new Table();
    var index = 0;
    var moreImages = true;

    while (moreImages)
    {
        var row = new TableRow();
        t.Rows.Add(row);

        for (var i = 0; i < info.ColumnCount; i++)
        {
            var cell = new TableCell();
            row.Cells.Add(cell);
        }
    }
}
```

```

var img = new Image();
var fi = images[index];
img.ImageUrl = fi.Name;
img.Width = Unit.Pixel(info.DisplayWidth);

var a = new HtmlAnchor {Href = fi.Name};
a.Controls.Add(img);
cell.Controls.Add(a);

index++;
moreImages = (index < images.Count);
if (!moreImages)
    break;
}
}
}

```

You might want to make the handler accept some optional query string parameters, such as the width of images and the column count. These values are packed in an instance of the helper class *PictureViewerInfo* along with the name of the folder to view. Here's the code to process the query string of the URL to extract parameters if any are present:

```

var info = new PictureViewerInfo();
var p1 = context.Request.Params["Width"];
var p2 = context.Request.Params["Cols"];
if (p1 != null)
    info.DisplayWidth = p1.ToInt32();
if (p2 != null)
    info.ColumnCount = p2.ToInt32();

```

ToInt32 is a helper extension method that attempts to convert a numeric string to the corresponding integer. I find this method quite useful and a great enhancer of code readability. Here's the code:

```

public static Int32 ToInt32(this String helper, Int32 defaultValue = Int32.MinValue)
{
    Int32 number;
    var result = Int32.TryParse(helper, out number);
    return result ? number : defaultValue;
}

```

Figure 4-4 shows the handler in action.



FIGURE 4-4 The picture viewer handler in action with a given number of columns and a specified width.

Registering the handler is easy too. You just add the following script to the `<httpHandlers>` section of the `web.config` file:

```
<add verb="*"
    path="folder.axd"
    type="PictureViewerHandler, AspNetGallery.Extensions" />
```

You place the assembly in the GAC and move the configuration script to the global `web.config` to extend the settings to all applications on the machine. If you're targeting IIS 7 integrated mode, you also need the following:

```
<system.webServer>
  <handlers>
    <add name="PictureFolder"
        preCondition="integratedMode"
        verb="*" />
```

```
        path="folder.axd"  
        type="PictureViewerHandler, AspNetGallery.Extensions" />  
    </handlers>  
</system.webServer>
```

Serving Images More Effectively

Any page you get from the Web these days is topped with so many images and is so well conceived and designed that often the overall page looks more like a magazine advertisement than an HTML page. Looking at the current pages displayed by portals, it's rather hard to imagine there ever was a time—and it was only a decade ago—when one could create a Web site by using only a text editor and some assistance from a friend who had a bit of familiarity with Adobe PhotoShop.

In spite of the wide use of images on the Web, there is just one way in which a Web page can reference an image—by using the HTML `` tag. By design, this tag points to a URL. As a result, to be displayable within a Web page, an image must be identifiable through a URL and its bits should be contained in the output stream returned by the Web server for that URL.

In many cases, the URL points to a static resource such as a GIF or JPEG file. In this case, the Web server takes the request upon itself and serves it without invoking external components. However, the fact that many `` tags on the Web are bound to a static file does not mean there's no other way to include images in Web pages.

Where else can you turn to get images aside from picking them up from the server file system? One way to do it is to load images from a database, or you can generate or modify images on the fly just before serving the bits to the browser.

Loading Images from Databases

The use of a database as the storage medium for images is controversial. Some people have good reasons to push it as a solution; others tell you bluntly they would never do it and that you shouldn't either. Some people can tell you wonderful stories of how storing images in a properly equipped database was the best experience of their professional life. With no fear that facts could perhaps prove them wrong, other people will confess that they would never use a database again for such a task.

The facts say that all database management systems (DBMS) of a certain reputation and volume have supported binary large objects (BLOB) for quite some time. Sure, a BLOB field doesn't necessarily contain an image—it can contain a multimedia file or a long text file—but overall there must be a good reason for having this BLOB supported in Microsoft SQL Server, Oracle, and similar popular DBMS systems!

To read an image from a BLOB field with ADO.NET, you execute a SELECT statement on the column and use the *ExecuteScalar* method to catch the result and save it in an array of bytes. Next, you send this array down to the client through a binary write to the response stream. Let's write an HTTP handler to serve a database-stored image:

```
public class DbImageHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext ctx)
    {
        // Ensure the URL contains an ID argument that is a number
        var id = -1;
        var p1 = context.Request.Params["id"];
        if (p1 != null)
            id = p1.ToInt32(-1);
        if (id < 0)
        {
            context.Response.End();
            return;
        }

        var connString = "...";
        const String cmdText = "SELECT photo FROM employees WHERE employeeid=@id";

        // Get an array of bytes from the BLOB field
        byte[] img = null;
        var conn = new SqlConnection(connString);
        using (conn)
        {
            var cmd = new SqlCommand(cmdText, conn);
            cmd.Parameters.AddWithValue("@id", id);
            conn.Open();
            img = (byte[])cmd.ExecuteScalar();
        }

        // Prepare the response for the browser
        if (img != null)
        {
            ctx.Response.ContentType = "image/jpeg";
            ctx.Response.BinaryWrite(img);
        }
    }

    public bool IsReusable
    {
        get { return true; }
    }
}
```

There are quite a few assumptions made in this code. First, we assume that the field named *photo* contains image bits and that the format of the image is JPEG. Second, we assume that images are to be retrieved from a fixed table of a given database through a predefined connection string. Finally, we assume that the URL to invoke this handler includes a query string parameter named *id*.

Notice the attempt to convert the value of the *id* query parameter to an integer before proceeding. This simple check significantly reduces the surface attack area for malicious users by verifying that what is going to be used as a numeric ID is really a numeric ID. Especially when you're inoculating user input into SQL query commands, filtering out extra characters and wrong data types is a fundamental measure for preventing attacks.

The *BinaryWrite* method of the *HttpResponse* object writes an array of bytes to the output stream.



Note If the database you're using is Northwind, an extra step might be required to ensure that the images are correctly managed. For some reason, the SQL Server version of the Northwind database stores the images in the *photo* column of the Employees table as OLE objects. This is probably because of the conversion that occurred when the database was upgraded from the Microsoft Access version. As a matter of fact, the array of bytes you receive contains a 78-byte prefix that has nothing to do with the image. Those bytes are just the header created when the image was added as an OLE object to the first version of Access.

Although the preceding code works like a champ with regular BLOB fields, it must undergo the following modification to work with the *photo* field of the Northwind.Employees database:

```
Response.OutputStream.Write(img, 78, img.Length-78);
```

Instead of using the *BinaryWrite* call, which doesn't let you specify the starting position, use the code shown here.

A sample page to test BLOB field access is shown in Figure 4-5. The page lets users select an employee ID and post back. When the page renders, the ID is used to complete the URL for the ASP.NET *Image* control.

```
var url = String.Format("photo.axd?id={0}", DropDownList1.SelectedValue);  
Image1.ImageUrl = url;
```

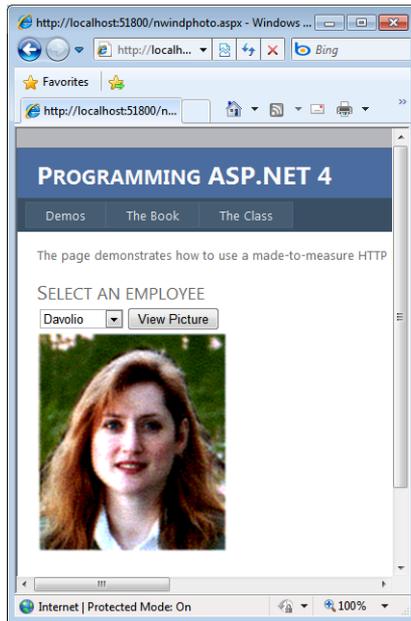


FIGURE 4-5 Downloading images stored within the BLOB field of a database.

An HTTP handler must be registered in the *web.config* file and bound to a public endpoint. In this case, the endpoint is *photo.axd* and the script to enter in the configuration file is shown next (in addition to a similar script in *<system.webServer>*):

```
<httpHandlers>
  <add verb="*"
        path="photo.axd"
        type=" NorthwindPhotoImageHandler, AspNetGallery.Extensions" />
</httpHandlers>
```



Note The preceding handler clearly has a weak point: it hard-codes a SQL command and the related connection string. This means that you might need a different handler for each different command or database to access. A more realistic handler would probably use an external and configurable database-specific provider. Such a provider can be as simple as a class that implements an agreed-upon interface. At a minimum, the interface will supply a method to retrieve and return an array of bytes.

Alternatively, if you want to keep the ADO.NET code in the handler itself, the interface will just supply members that specify the command text and connection string. The handler will figure out its default provider from a given entry in the *web.config* file.

Serving Dynamically Generated Images

Isn't it true that an image is worth thousands of words? Many financial Web sites offer charts and, more often than not, these charts are dynamically generated on the server. Next, they are served to the browser as a stream of bytes and travel over the classic response output stream. But can you create and manipulate server-side images? For these tasks, Web applications normally rely on ad hoc libraries or the graphic engine of other applications (for example, Microsoft Office applications). ASP.NET applications are different and, to some extent, luckier. ASP.NET applications, in fact, can rely on a powerful and integrated graphic engine integrated in the .NET Framework.

In ASP.NET, writing images to disk might require some security adjustments. Normally, the ASP.NET runtime runs under the aegis of the NETWORK SERVICE user account. In the case of anonymous access with impersonation disabled—which are the default settings in ASP.NET—the worker process lends its own identity and security token to the thread that executes the user request of creating the file. With regard to the default scenario, an access-denied exception might be thrown if NETWORK SERVICE (or the selected application pool identity) lacks writing permissions on virtual directories—a pretty common situation.

ASP.NET provides an interesting alternative to writing files on disk without changing security settings: in-memory generation of images. In other words, the dynamically generated image is saved directly to the output stream in the needed image format or in a memory stream.

Writing Copyright Notes on Images

The .NET Framework graphic engine supports quite a few image formats, including JPEG, GIF, BMP, and PNG. The whole collection of image formats is in the *ImageFormat* structure of the *System.Drawing* namespace. You can save a memory-resident *Bitmap* object to any of the supported formats by using one of the overloads of the *Save* method:

```
Bitmap bmp = new Bitmap(file);  
...  
bmp.Save(outputStream, ImageFormat.Gif);
```

When you attempt to save an image to a stream or disk file, the system attempts to locate an encoder for the requested format. The encoder is a module that converts from the native format to the specified format. Note that the encoder is a piece of unmanaged code that lives in the underlying Win32 platform. For each save format, the *Save* method looks up the right encoder and proceeds.

The next example wraps up all the points we've touched on. This example shows how to load an existing image, add some copyright notes, and serve the modified version to the user. In doing so, we'll load an image into a *Bitmap* object, obtain a *Graphics* for that bitmap, and use graphics primitives to write. When finished, we'll save the result to the page's output stream and indicate a particular MIME type.

The sample page that triggers the example is easily created, as shown in the following listing:

```
<html>
<body>
  
</body>
</html>
```

The page contains no ASP.NET code and displays an image through a static HTML ** tag. The source of the image, though, is an HTTP handler that loads the image passed through the query string and then manipulates and displays it. Here's the source code for the *ProcessRequest* method of the HTTP handler:

```
public void ProcessRequest (HttpContext context)
{
    var o = context.Request["url"];
    if (o == null)
    {
        context.Response.Write("No image found.");
        context.Response.End();
        return;
    }

    var file = context.Server.MapPath(o);
    var msg = ConfigurationManager.AppSettings["CopyrightNote"];
    if (File.Exists(file))
    {
        Bitmap bmp = AddCopyright(file, msg);
        context.Response.ContentType = "image/jpeg";
        bmp.Save(context.Response.OutputStream, ImageFormat.Jpeg);
        bmp.Dispose();
    }
    else
    {
        context.Response.Write("No image found.");
        context.Response.End();
    }
}
```

Note that the server-side page performs two different tasks indeed. First, it writes copyright text on the image canvas; next, it converts whatever the original format was to JPEG:

```
Bitmap AddCopyright(String file, String msg)
{
    // Load the file and create the graphics
    var bmp = new Bitmap(file);
    var g = Graphics.FromImage(bmp);

    // Define text alignment
    var strFmt = new StringFormat();
    strFmt.Alignment = StringAlignment.Center;

    // Create brushes for the bottom writing
    // (green text on black background)
    var btmForeColor = new SolidBrush(Color.PaleGreen);
    var btmBackColor = new SolidBrush(Color.Black);
```

```
// To calculate writing coordinates, obtain the size of the
// text given the font typeface and size
var btmFont = new Font("Verdana", 7);
var textSize = g.MeasureString(msg, btmFont);

// Calculate the output rectangle and fill
float x = (bmp.Width-textSize.Width-3);
float y = (bmp.Height-textSize.Height-3);
float w = (x + textSize.Width);
float h = (y + textSize.Height);
var textArea = new RectangleF(x, y, w, h);
g.FillRectangle(btmBackColor, textArea);

// Draw the text and free resources
g.DrawString(msg, btmFont, btmForeColor, textArea);
btmForeColor.Dispose();
btmBackColor.Dispose();
btmFont.Dispose();
g.Dispose();

return bmp;
}
```

Figure 4-6 shows the results.

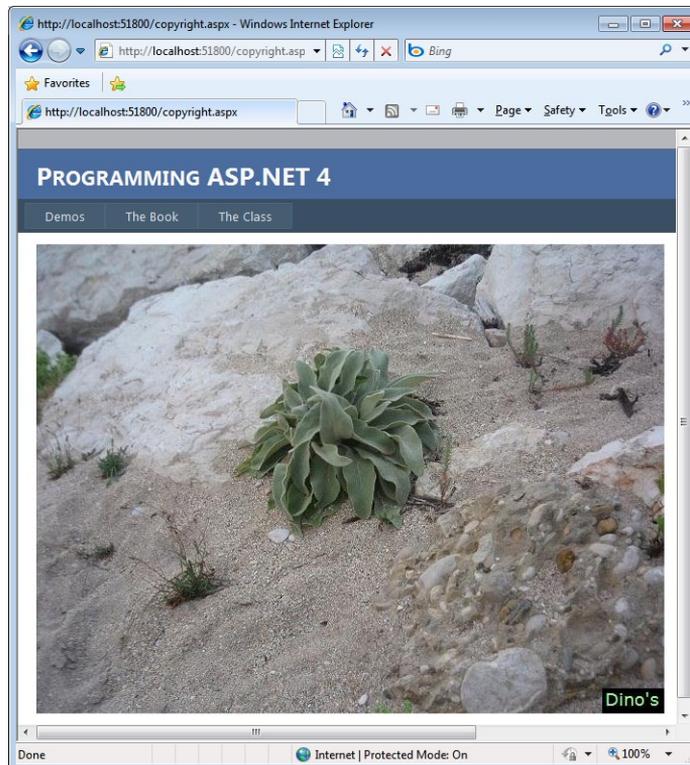


FIGURE 4-6 A server-resident image has been modified before being displayed.

Note that the additional text is part of the image the user downloads on her client browser. If the user saves the picture by using the Save Picture As menu from the browser, the text (in this case, the copyright note) is saved along with the image.



Important All examples demonstrating programmatic manipulation of images take advantage of the classes in the *System.Drawing* assembly. The use of this assembly is not recommended in ASP.NET and is explicitly not supported in ASP.NET Web services. (See <http://msdn.microsoft.com/en-us/library/system.drawing.aspx>.) This fact simply means that you are advised not to use classes in *System.Drawing* because Microsoft can't guarantee it is always safe to use them in all possible scenarios. If your code is currently using *System.Drawing*—the GDI+ subsystem—and it works just fine, you're probably OK. In any case, if you use GDI+ classes and encounter a malfunction, Microsoft will not assist you. Forewarned is forearmed.

You might be better off using an alternative to GDI+, especially for new applications. Which one? For both speed and reliability, you can consider the WPF Imaging API. Here's an interesting post that shows how to use Windows Presentation Foundation (WPF) for resizing images: <http://weblogs.asp.net/bleroy/archive/2010/01/21/server-side-resizing-with-wpf-now-with-jpg.aspx>.

Controlling Images via an HTTP Handler

What if the user requests the JPG file directly from the address bar? And what if the image is linked by another Web site or referenced in a blog post? By default, the original image is served without any further modification. Why is this so?

For performance reasons, IIS serves static files, such as JPG images, directly without involving any external module, including the ASP.NET runtime. In this way, the HTTP handler that does the trick of adding a copyright note is therefore blissfully ignored when the request is made via the address bar or a hyperlink. What can you do about it?

In IIS 6, you must register the JPG extension as an ASP.NET extension for a particular application using IIS Manager. In this case, each request for JPG resources is forwarded to your application and resolved through the HTTP handler.

In IIS 7, things are even simpler for developers. All you have to do is add the following lines to the application's *web.config* file:

```
<system.webServer>
  <handlers>
    <add name="Jpeg"
        preCondition="integratedMode"
        verb="*"
        path="*.jpg"
        type="DynImageHandler, AspNetGallery.Extensions" />
  </handlers>
</system.webServer>
```

You might want to add the same setting also under `<httpHandlers>`, which will be read in cases where IIS 7.x is configured in classic mode:

```
<httpHandlers>
  <add verb="*" path="*.jpg" type="DynImageHandler, AspNetGallery.Extensions"/>
</httpHandlers>
```

This is yet another benefit of the unified runtime pipeline we experience when the ASP.NET application runs under IIS 7 integrated mode.



Note An HTTP handler that needs to access session-state values must implement the *IRequiresSessionState* interface. Like *INamingContainer*, it's a marker interface and requires no method implementation. Note that the *IRequiresSessionState* interface indicates that the HTTP handler requires read and write access to the session state. If read-only access is needed, use the *IReadOnlySessionState* interface instead.

Advanced HTTP Handler Programming

HTTP handlers are not a tool for everybody. They serve a very neat purpose: changing the way a particular resource, or set of resources, is served to the user. You can use handlers to filter out resources based on runtime conditions or to apply any form of additional logic to the retrieval of traditional resources such as pages and images. Finally, you can use HTTP handlers to serve certain pages or resources in an asynchronous manner.

For HTTP handlers, the registration step is key. Registration enables ASP.NET to know about your handler and its purpose. Registration is required for two practical reasons. First, it serves to ensure that IIS forwards the call to the correct ASP.NET application. Second, it serves to instruct your ASP.NET application on the class to load to handle the request. As mentioned, you can use handlers to override the processing of existing resources (for example, *hello.aspx*) or to introduce new functionalities (for example, *folder.axd*). In both cases, you're invoking a resource whose extension is already known to IIS—the *.axd* extension is registered in the IIS metabase when you install ASP.NET. In both cases, though, you need to modify the *web.config* file of the application to let the application know about the handler.

By using the ASHX extension and programming model for handlers, you can also save yourself the *web.config* update and deploy a new HTTP handler by simply copying a new file in a new or existing application's folder.

Deploying Handlers as ASHX Resources

An alternative way to define an HTTP handler is through an *.ashx* file. The file contains a special directive, named *@WebHandler*, that expresses the association between the HTTP

handler endpoint and the class used to implement the functionality. All *.ashx* files must begin with a directive like the following one:

```
<%@ WebHandler Language="C#" Class="AspNetGallery.Handlers.MyHandler" %>
```

When an *.ashx* endpoint is invoked, ASP.NET parses the source code of the file and figures out the HTTP handler class to use from the *@WebHandler* directive. This automation removes the need of updating the *web.config* file. Here's a sample *.ashx* file. As you can see, it is the plain class file plus the special *@WebHandler* directive:

```
<%@ WebHandler Language="C#" Class="MyHandler" %>

using System.Web;

public class MyHandler : IHttpHandler {

    public void ProcessRequest (HttpContext context) {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }

    public bool IsReusable {
        get {
            return false;
        }
    }
}
```

Note that the source code of the class can either be specified inline or loaded from any of the assemblies referenced by the application. When *.ashx* resources are used to implement an HTTP handler, you just deploy the source file and you're done. Just as for XML Web services, the source file is loaded and compiled only on demand. Because ASP.NET adds a special entry to the IIS metabase for *.ashx* resources, you don't even need to enter changes to the Web server configuration.

Resources with an *.ashx* extension are handled by an HTTP handler class named *SimpleHandleFactory*. Note that *SimpleHandleFactory* is actually an HTTP handler factory class, not a simple HTTP handler class. We'll discuss handler factories in a moment.

The *SimpleHandleFactory* class looks for the *@WebHandler* directive at the beginning of the file. The *@WebHandler* directive tells the handler factory the name of the HTTP handler class to instantiate when the source code has been compiled.



Important You can build HTTP handlers both as regular class files compiled to an assembly and via *.ashx* resources. There's no significant difference between the two approaches except that *.ashx* resources, like ordinary ASP.NET pages, will be compiled on the fly upon the first request.

Prevent Access to Forbidden Resources

If your Web application manages resources of a type that you don't want to make publicly available over the Web, you must instruct IIS not to display those files. A possible way to accomplish this consists of forwarding the request to *aspnet_isapi* and then binding the extension to one of the built-in handlers—the *HttpForbiddenHandler* class:

```
<add verb="*" path="*.xyz" type="System.Web.HttpForbiddenHandler" />
```

Any attempt to access an *.xyz* resource results in an error message being displayed. The same trick can also be applied for individual resources served by your application. If you need to deploy, say, a text file but do not want to take the risk that somebody can get to it, add the following:

```
<add verb="*" path="yourFile.txt" type="System.Web.HttpForbiddenHandler" />
```

Should It Be Reusable or Not?

In a conventional HTTP handler, the *ProcessRequest* method takes the lion's share of the overall set of functionality. The second member of the *IHttpHandler* interface—the *IsReusable* property—is used only in particular circumstances. If you set the *IsReusable* property to return *true*, the handler is not unloaded from memory after use and is repeatedly used. Put another way, the Boolean value returned by *IsReusable* indicates whether the handler object can be pooled.

Frankly, most of the time it doesn't really matter what you return—be it *true* or *false*. If you set the property to return *false*, you require that a new object be allocated for each request. The simple allocation of an object is not a particularly expensive operation. However, the initialization of the handler might be costly. In this case, by making the handler reusable, you save much of the overhead. If the handler doesn't hold any state, there's no reason for not making it reusable.

In summary, I'd say that *IsReusable* should be always set to *true*, except when you have instance properties to deal with or properties that might cause trouble if used in a concurrent environment. If you have no initialization tasks, it doesn't really matter whether it returns *true* or *false*. As a margin note, the *System.Web.UI.Page* class—the most popular HTTP handler ever—sets its *IsReusable* property to *false*.

The key point to determine is the following: Who's really using *IsReusable* and, subsequently, who really cares about its value?

Once the HTTP runtime knows the HTTP handler class to serve a given request, it simply instantiates it—no matter what. So when is the *IsReusable* property of a given handler taken into account? Only if you use an HTTP handler factory—that is, a piece of code that dynamically decides which handler should be used for a given request. An HTTP handler

factory can query a handler to determine whether the same instance can be used to service multiple requests and thus optionally create and maintain a pool of handlers.

ASP.NET pages and ASHX resources are served through factories. However, none of these factories ever checks *IsReusable*. Of all the built-in handler factories in the whole ASP.NET platform, very few check the *IsReusable* property of related handlers. So what's the bottom line?

As long as you're creating HTTP handlers for AXD, ASHX, or perhaps ASPX resources, be aware that the *IsReusable* property is blissfully ignored. Do not waste your time trying to figure out the optimal configuration. Instead, if you're creating an HTTP handler factory to serve a set of resources, whether or not to implement a pool of handlers is up to you and *IsReusable* is the perfect tool for the job.

But when should you employ an HTTP handler factory? You should do it in all situations in which the HTTP handler class for a request is not uniquely identified. For example, for ASPX pages, you don't know in advance which HTTP handler type you have to use. The type might not even exist (in which case, you compile it on the fly). The HTTP handler factory is used whenever you need to apply some logic to decide which handler is the right one to use. In other words, you need an HTTP handler factory when declarative binding between endpoints and classes is not enough.

HTTP Handler Factories

An HTTP request can be directly associated with an HTTP handler or with an HTTP handler factory object. An HTTP handler factory is a class that implements the *IHttpHandlerFactory* interface and is in charge of returning the actual HTTP handler to use to serve the request. The *SimpleHandlerFactory* class provides a good example of how a factory works. The factory is mapped to requests directed at *.ashx* resources. When such a request comes in, the factory determines the actual handler to use by looking at the *@WebHandler* directive in the source file.

In the .NET Framework, HTTP handler factories are used to perform some preliminary tasks on the requested resource prior to passing it on to the handler. Another good example of a handler factory object is an internal class named *PageHandlerFactory*, which is in charge of serving *.aspx* pages. In this case, the factory handler figures out the name of the handler to use and, if possible, loads it up from an existing assembly.

HTTP handler factories are classes that implement a couple of methods on the *IHttpHandlerFactory* interface—*GetHandler* and *ReleaseHandler*, as shown in Table 4-5.

TABLE 4-5 Members of the *IHttpHandlerFactory* Interface

Method	Description
<i>GetHandler</i>	Returns an instance of an HTTP handler to serve the request.
<i>ReleaseHandler</i>	Takes an existing HTTP handler instance and frees it up or pools it.

The *GetHandler* method has the following signature:

```
public virtual IHttpHandler GetHandler(  
    HttpContext context,  
    String requestType,  
    String url,  
    String pathTranslated);
```

The *requestType* argument is a string that evaluates to GET or POST—the HTTP verb of the request. The last two arguments represent the raw URL of the request and the physical path behind it. The *ReleaseHandler* method is a mandatory override for any class that implements *IHttpHandlerFactory*; in most cases, it will just have an empty body.

The following listing shows a sample HTTP handler factory that returns different handlers based on the HTTP verb (GET or POST) used for the request:

```
class MyHandlerFactory : IHttpHandlerFactory  
{  
    public IHttpHandler GetHandler(HttpContext context,  
        String requestType, String url, String pathTranslated)  
    {  
        // Feel free to create a pool of HTTP handlers here  
        if(context.Request.RequestType.ToLower() == "get")  
            return (IHttpHandler) new MyGetHandler();  
        else if(context.Request.RequestType.ToLower() == "post")  
            return (IHttpHandler) new MyPostHandler();  
        return null;  
    }  
  
    public void ReleaseHandler(IHttpHandler handler)  
    {  
        // Nothing to do  
    }  
}
```

When you use an HTTP handler factory, it's the factory (not the handler) that you want to register in the ASP.NET configuration file. If you register the handler, it will always be used to serve requests. If you opt for a factory, you have a chance to decide dynamically and based on runtime conditions which handler is more appropriate for a certain request. In doing so, you can use the *IsReusable* property of handlers to implement a pool.

Asynchronous Handlers

An asynchronous HTTP handler is a class that implements the *IHttpAsyncHandler* interface. The system initiates the call by invoking the *BeginProcessRequest* method. Next, when the method ends, a callback function is automatically invoked to terminate the call. In the .NET Framework, the sole *HttpApplication* class implements the asynchronous interface. The members of the *IHttpAsyncHandler* interface are shown in Table 4-6.

TABLE 4-6 Members of the *IHttpAsyncHandler* Interface

Method	Description
<i>BeginProcessRequest</i>	Initiates an asynchronous call to the specified HTTP handler
<i>EndProcessRequest</i>	Terminates the asynchronous call

The signature of the *BeginProcessRequest* method is as follows:

```
IAsyncResult BeginProcessRequest(
    HttpContext context,
    AsyncCallback cb,
    Object extraData);
```

The *context* argument provides references to intrinsic server objects used to service HTTP requests. The second parameter is the *AsyncCallback* object to invoke when the asynchronous method call is complete. The third parameter is a generic cargo variable that contains any data you might want to pass to the handler.



Note An *AsyncCallback* object is a delegate that defines the logic needed to finish processing the asynchronous operation. A delegate is a class that holds a reference to a method. A delegate class has a fixed signature, and it can hold references only to methods that match that signature. A delegate is equivalent to a type-safe function pointer or a callback. As a result, an *AsyncCallback* object is just the code that executes when the asynchronous handler has completed its job.

The *AsyncCallback* delegate has the following signature:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

It uses the *IAsyncResult* interface to obtain the status of the asynchronous operation. To illustrate the plumbing of asynchronous handlers, I'll show you what the HTTP runtime does when it deals with asynchronous handlers. The HTTP runtime invokes the *BeginProcessRequest* method as illustrated here:

```
// Sets an internal member of the HttpContext class with
// the current instance of the asynchronous handler
context.AsyncAppHandler = asyncHandler;

// Invokes the BeginProcessRequest method on the asynchronous HTTP handler
asyncHandler.BeginProcessRequest(context, OnCompletionCallback, context);
```

The *context* argument is the current instance of the *HttpContext* class and represents the context of the request. A reference to the HTTP context is also passed as the custom data sent to the handler to process the request. The *extraData* parameter in the *BeginProcessRequest* signature is used to represent the status of the asynchronous operation. The *BeginProcessRequest* method returns an object of type *HttpAsyncResult*—a class that implements the *IAAsyncResult* interface. The *IAAsyncResult* interface contains a property named *AsyncState* that is set with the *extraData* value—in this case, the HTTP context.

The *OnCompletionCallback* method is an internal method. It gets automatically triggered when the asynchronous processing of the request terminates. The following listing illustrates the pseudocode of the *HttpRuntime* private method:

```
// The method must have the signature of an AsyncCallback delegate
private void OnHandlerCompletion(IAAsyncResult ar)
{
    // The ar parameter is an instance of HttpAsyncResult
    HttpContext context = (HttpContext) ar.AsyncState;

    // Retrieves the instance of the asynchronous HTTP handler
    // and completes the request
    IHttpAsyncHandler asyncHandler = context.AsyncAppHandler;
    asyncHandler.EndProcessRequest(ar);

    // Finalizes the request as usual
    ...
}
```

The completion handler retrieves the HTTP context of the request through the *AsyncState* property of the *IAAsyncResult* object it gets from the system. As mentioned, the actual object passed is an instance of the *HttpAsyncResult* class—in any case, it is the return value of the *BeginProcessRequest* method. The completion routine extracts the reference to the asynchronous handler from the context and issues a call to the *EndProcessRequest* method:

```
void EndProcessRequest(IAAsyncResult result);
```

The *EndProcessRequest* method takes the *IAAsyncResult* object returned by the call to *BeginProcessRequest*. As implemented in the *HttpApplication* class, the *EndProcessRequest* method does nothing special and is limited to throwing an exception if an error occurred.

Implementing Asynchronous Handlers

Asynchronous handlers essentially serve one particular scenario—a scenario in which the generation of the markup is subject to lengthy operations, such as time-consuming database stored procedures or calls to Web services. In these situations, the ASP.NET thread in charge of the request is stuck waiting for the operation to complete. Because threads are valuable resources, lengthy tasks that keep threads occupied for too long are potentially the perfect scalability killer. However, asynchronous handlers are here to help.

The idea is that the request begins on a thread-pool thread, but that thread is released as soon as the operation begins. In *BeginProcessRequest*, you typically create your own thread and start the lengthy operation. *BeginProcessRequest* doesn't wait for the operation to complete; therefore, the thread is returned to the pool immediately.

There are a lot of tricky details that this bird's-eye description just omitted. In the first place, you should strive to avoid a proliferation of threads. Ideally, you should use a custom thread pool. Furthermore, you must figure out a way to signal when the lengthy operation has terminated. This typically entails creating a custom class that implements *IAsyncResult* and returning it from *BeginProcessRequest*. This class embeds a synchronization object—typically a *ManualResetEvent* object—that the custom thread carrying the work will signal upon completion.

In the end, building asynchronous handlers is definitely tricky and not for novice developers. Very likely, you are more interested in having asynchronous pages than in generic asynchronous HTTP handlers. With asynchronous pages, the “lengthy task” is merely the *ProcessRequest* method of the *Page* class. (Obviously, you configure the page to execute asynchronously only if the page contains code that starts I/O-bound and potentially lengthy operations.)

ASP.NET offers ad hoc support for building asynchronous pages more easily and more comfortably than through HTTP handlers.



Caution I've seen several ASP.NET developers use an *.aspx* page to serve markup other than HTML markup. This is not a good idea. An *.aspx* resource is served by quite a rich and sophisticated HTTP handler—the *System.Web.UI.Page* class. The *ProcessRequest* method of this class entirely provides for the page life cycle as we know it—*Init*, *Load*, and *PreRender* events, as well as rendering stage, view state, and postback management. Nothing of the kind is really required if you only need to retrieve and return, say, the bytes of an image. HTTP handlers are an excellent way to speed up particular requests. HTTP handlers are also a quick way to serve AJAX requests without writing (and spinning up) the whole machinery of Windows Communication Foundation (WCF) services. At the very end of the day, an HTTP handler is an endpoint and can be used to serve data to AJAX requests. In this regard, the difference between an HTTP handler and a WCF service is that the HTTP handler doesn't have a free serialization engine for input and output values.

Writing HTTP Modules

So you've learned that any incoming requests for ASP.NET resources are handed over to the worker process for the actual processing. The worker process is distinct from the Web server executable so that even if one ASP.NET application crashes, it doesn't bring down the whole server.

On the way to the final HTTP handler, the request passes through a pipeline of special runtime modules—HTTP modules. An HTTP module is a .NET Framework class that implements the *IHttpModule* interface. The HTTP modules that filter the raw data within the request are configured on a per-application basis within the *web.config* file. All ASP.NET applications, though, inherit a bunch of system HTTP modules configured in the global *web.config* file. Applications hosted under IIS 7.x integrated mode can configure HTTP modules that run at the IIS level for any requests that comes in, not just for ASP.NET-related resources.

An HTTP module can pre-process and post-process a request, and it intercepts and handles system events as well as events raised by other modules.

The *IHttpModule* Interface

The *IHttpModule* interface defines only two methods: *Init* and *Dispose*. The *Init* method initializes a module and prepares it to handle requests. At this time, you subscribe to receive notifications for the events of interest. The *Dispose* method disposes of the resources (all but memory!) used by the module. Typical tasks you perform within the *Dispose* method are closing database connections or file handles.

The *IHttpModule* methods have the following signatures:

```
void Init(HttpApplication app);  
void Dispose();
```

The *Init* method receives a reference to the *HttpApplication* object that is serving the request. You can use this reference to wire up to system events. The *HttpApplication* object also features a property named *Context* that provides access to the intrinsic properties of the ASP.NET application. In this way, you gain access to *Response*, *Request*, *Session*, and the like.

Table 4-7 lists the events that HTTP modules can listen to and handle.

TABLE 4-7 *HttpApplication* Events in Order of Appearance

Event	Description
<i>BeginRequest</i>	Occurs as soon as the HTTP pipeline begins to process the request.
<i>AuthenticateRequest</i> , <i>PostAuthenticateRequest</i>	Occurs when a security module has established the identity of the user.
<i>AuthorizeRequest</i> , <i>PostAuthorizeRequest</i>	Occurs when a security module has verified user authorization.
<i>ResolveRequestCache</i> , <i>PostResolveRequestCache</i>	Occurs when the ASP.NET runtime resolves the request through the output cache.
<i>MapRequestHandler</i> , <i>PostMapRequestHandler</i>	Occurs when the HTTP handler to serve the request has been found. <i>It is fired only to applications running in classic mode or under IIS 6.</i>
<i>AcquireRequestState</i> , <i>PostAcquireRequestState</i>	Occurs when the handler that will actually serve the request acquires the state information associated with the request.
<i>PreRequestHandlerExecute</i>	Occurs just before the HTTP handler of choice begins to work.
<i>PostRequestHandlerExecute</i>	Occurs when the HTTP handler of choice finishes execution. The response text has been generated at this point.
<i>ReleaseRequestState</i> , <i>PostReleaseRequestState</i>	Occurs when the handler releases the state information associated with the current request.
<i>UpdateRequestCache</i> , <i>PostUpdateRequestCache</i>	Occurs when the ASP.NET runtime stores the response of the current request in the output cache to be used to serve subsequent requests.
<i>LogRequest</i> , <i>PostLogRequest</i>	Occurs when the ASP.NET runtime is ready to log the results of the request. Logging is guaranteed to execute even if errors occur. <i>It is fired only to applications running under IIS 7 integrated mode.</i>
<i>EndRequest</i>	Occurs as the last event in the HTTP pipeline chain of execution.

Another pair of events can occur during the request, but in a nondeterministic order. They are *PreSendRequestHeaders* and *PreSendRequestContent*.

The *PreSendRequestHeaders* event informs the *HttpApplication* object in charge of the request that HTTP headers are about to be sent. The *PreSendRequestContent* event tells the *HttpApplication* object in charge of the request that the response body is about to be sent. Both these events normally fire after *EndRequest*, but not always. For example, if buffering is turned off, the event gets fired as soon as some content is going to be sent to the client. Speaking of nondeterministic application events, it must be said that a third nondeterministic event is, of course, *Error*.

All these events are exposed by the *HttpApplication* object that an HTTP module receives as an argument to the *Init* method. You can write handlers for such events in the *global.asax* file of the application. You can also catch these events from within a custom HTTP module.

A Custom HTTP Module

Let's come to grips with HTTP modules by writing a relatively simple custom module named *Marker* that adds a signature at the beginning and end of each page served by the application. The following code outlines the class we need to write:

```
using System;
using System.Web;

namespace AspNetGallery.Extensions.Modules
{
    public class MarkerModule : IHttpModule
    {
        public void Init(HttpApplication app)
        {
            // Register for pipeline events
        }

        public void Dispose()
        {
            // Nothing to do here
        }
    }
}
```

The *Init* method is invoked by the *HttpApplication* class to load the module. In the *Init* method, you normally don't need to do more than simply register your own event handlers. The *Dispose* method is, more often than not, empty. The heart of the HTTP module is really in the event handlers you define.

Wiring Up Events

The sample *Marker* module registers a couple of pipeline events. They are *BeginRequest* and *EndRequest*. *BeginRequest* is the first event that hits the HTTP application object when the request begins processing. *EndRequest* is the event that signals the request is going to be terminated, and it's your last chance to intervene. By handling these two events, you can write custom text to the output stream before and after the regular HTTP handler—the *Page*-derived class.

The following listing shows the implementation of the *Init* and *Dispose* methods for the sample module:

```
public void Init(HttpApplication app)
{
    // Register for pipeline events
    app.BeginRequest += OnBeginRequest;
    app.EndRequest += EndRequest;
}

public void Dispose()
{
}
```

The *BeginRequest* and *EndRequest* event handlers have a similar structure. They obtain a reference to the current *HttpApplication* object from the sender and get the HTTP context from there. Next, they work with the *Response* object to append text or a custom header:

```
public void OnBeginRequest(Object sender, EventArgs e)
{
    var app = (HttpApplication) sender;
    var ctx = app.Context;

    // More code here
    ...

    // Add custom header to the HTTP response
    ctx.Response.AppendHeader("Author", "DinoE");

    // PageHeaderText is a constant string defined elsewhere
    ctx.Response.Write(PageHeaderText);
}

public void OnEndRequest(Object sender, EventArgs e)
{
    // Get access to the HTTP context
    var app = (HttpApplication) sender;
    var ctx = app.Context;

    // More code here
    ...

    // Append some custom text
    // PageFooterText is a constant string defined elsewhere
    ctx.Response.Write(PageFooterText);
}
```

OnBeginRequest writes standard page header text and also adds a custom HTTP header. *OnEndRequest* simply appends the page footer. The effect of this HTTP module is visible in Figure 4-7.

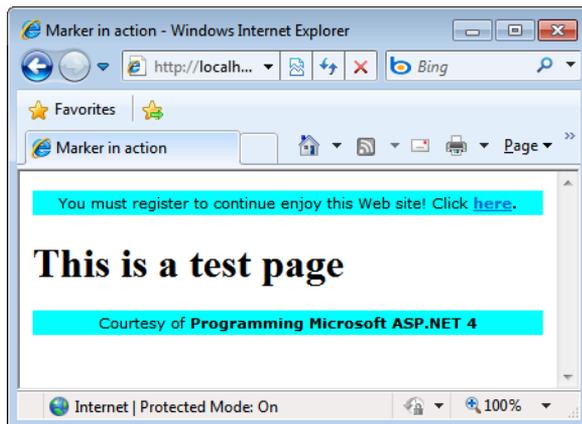


FIGURE 4-7 The *Marker* HTTP module adds a header and footer to each page within the application.

Registering with the Configuration File

You register a new HTTP module by adding an entry to the `<httpModules>` section of the configuration file. The overall syntax of the `<httpModules>` section closely resembles that of HTTP handlers. To add a new module, you use the `<add>` node and specify the `name` and `type` attributes. The `name` attribute contains the public name of the module. This name is used to select the module within the `HttpApplication's Modules` collection. If the module fires custom events, this name is also used as the prefix for building automatic event handlers in the `global.asax` file:

```
<system.web>
  <httpModules>
    <add name="Marker"
        type="MarkerModule, ASPNetGallery.Extensions" />
  </httpModules>
</system.web>
```

The order in which modules are applied depends on the physical order of the modules in the configuration list. You can remove a system module and replace it with your own that provides a similar functionality. In this case, in the application's `web.config` file you use the `<remove>` node to drop the default module and then use `<add>` to insert your own. If you want to completely redefine the order of HTTP modules for your application, you can clear all the default modules by using the `<clear>` node and then re-register them all in the order you prefer.



Note HTTP modules are loaded and initialized only once, at the startup of the application. Unlike HTTP handlers, they apply to any requests. So when you plan to create a new HTTP module, you should first wonder whether its functionality should span all possible requests in the application. Is it possible to choose which requests an HTTP module should process? The `Init` method is called only once in the application's lifetime, but the handlers you register are called once for each request. So to operate only on certain pages, you can do as follows:

```
public void OnBeginRequest(object sender, EventArgs e)
{
    HttpApplication app = (HttpApplication) sender;
    HttpContext ctx = app.Context;
    if (!ShouldHook(ctx))
        return;
    ...
}
```

`OnBeginRequest` is your handler for the `BeginRequest` event. The `ShouldHook` helper function returns a Boolean value. It is passed the context of the request—that is, any information that is available on the request. You can code it to check the URL as well as any HTTP content type and headers.

Accessing Other HTTP Modules

The sample just discussed demonstrates how to wire up pipeline events—that is, events fired by the *HttpApplication* object. But what about events fired by other modules? The *HttpApplication* object provides a property named *Modules* that gets the collection of modules for the current application.

The *Modules* property is of type *HttpModuleCollection* and contains the names of the modules for the application. The collection class inherits from the abstract class *NameObjectCollectionBase*, which is a collection of pairs made of a string and an object. The string indicates the public name of the module; the object is the actual instance of the module. To access the module that handles the session state, you need code like this:

```
var sessionModule = app.Modules["Session"];
sessionModule.Start += OnSessionStart;
```

As mentioned, you can also handle events raised by HTTP modules within the *global.asax* file and use the *ModuleName_EventName* convention to name the event handlers. The name of the module is just one of the settings you need to define when registering an HTTP module.

Examining a Real-World HTTP Module

The previous example gave us the gist of an HTTP module component. It was a simple (and kind of pointless) example, but it was useful to demonstrate what you can do with HTTP modules in a real application. First and foremost, not all applications need custom HTTP modules. ASP.NET comes with a bunch of built-in modules, which are listed in Table 4-8.

TABLE 4-8 Native HTTP Modules

Event	Description
<i>AnonymousIdentificationModule</i>	Manages anonymous identifiers for the ASP.NET application
<i>DefaultAuthenticationModule</i>	Ensures that the <i>User</i> object is always bound to some identity
<i>FileAuthorizationModule</i>	Verifies that the user has permission to access the given file.
<i>FormsAuthenticationModule</i>	Manages Forms authentication
<i>OutputCacheModule</i>	Implements output page caching
<i>ProfileModule</i>	Implements the data retrieval for profile data
<i>RoleManagerModule</i>	Manages the retrieval of role information
<i>ScriptModule</i>	Manages script requests placed through ASP.NET AJAX
<i>SessionStateModule</i>	Manages session state
<i>UrlAuthorizationModule</i>	Verifies that the user has permission to access the given URL
<i>UrlRoutingModule</i>	Implements URL routing
<i>WindowsAuthenticationModule</i>	Manages Windows authentication

All these HTTP modules perform a particular system-level operation and can be customized by application-specific code. Because an HTTP module works on any incoming request, it usually doesn't perform application-specific tasks. From an application perspective, an HTTP module is helpful when you need to apply filters on all requests for profiling, debugging, or functional reasons.

Let's dissect one of the system-provided HTTP modules, which will also slowly move us toward the next topic of this chapter. Enter the URL-routing HTTP module.

The *UrlRoutingModule* Class

In ASP.NET 3.5 Service Pack 1, Microsoft introduced a new and more effective API for URL rewriting. Because of its capabilities, the new API got a better name—*URL routing*. URL routing is built on top of the URL rewriting API, but it offers a richer and higher level programming model. (I'll get to URL rewriting and URL routing in a moment.)

The URL routing engine is a system-provided HTTP module that wires up the *PostResolveRequestCache* event. In a nutshell, the HTTP module matches the requested URL to one of the user-defined rewriting rules (known as *routes*) and finds the HTTP handler that is due to serve that route. If any HTTP handler is found, it becomes the actual handler for the current request. Here's the signature of the module class:

```
public class UrlRoutingModule : IHttpModule
{
    public virtual void PostResolveRequestCache(HttpContextBase context)
    {
        ...
    }

    void IHttpModule.Dispose()
    {
        ...
    }

    void IHttpModule.Init(HttpApplication application)
    {
        ...
    }
}
```

The class implements the *IHttpModule* interface implicitly, and in its initialization phase it registers a handler for the system's *PostResolveRequestCache* event.

The *PostResolveRequestCache* Event

The *PostResolveRequestCache* event fires right after the runtime environment (IIS or ASP.NET, depending on the IIS working mode) has determined whether the response for the current request can be served from the output cache or not. If the response is already cached,

there's no need to process the request and, subsequently, no need to analyze the content of the URL. Any system events that follow *PostResolveRequestCache* are part of the request processing cycle; therefore, hooking up *PostResolveRequestCache* is the optimal moment for taking control of requests that require some work on the server.

The first task accomplished by the HTTP module consists of grabbing any route data contained in the URL of the current request. The module matches the URL to one of the registered routes and figures out the handler for the route.

The route handler is not the HTTP handler yet. It is simply the object responsible for handling the route. The primary task of a route handler, however, is returning the HTTP handler to serve the request.

In the end, HTTP modules are extremely powerful tools that give you control over every little step taken by the system to process a request. For the same reason, however, HTTP modules are delicate tools—every time you write one, it will be invoked for each and every request. An HTTP module is hardly a tool for a specific application (with due exceptions), but it is often a formidable tool for implementing cross-cutting, system-level features.

URL Routing

The whole ASP.NET platform originally developed around the idea of serving requests for physical pages. Look at the following URL:

```
http://northwind.com/news.aspx?id=1234
```

It turns out that most URLs used within an ASP.NET application are made of two parts: the path to the physical Web page that contains the logic to apply, and some data stuffed in the query string to provide parameters. In the URL just shown, the *news.aspx* page incorporates the logic required to retrieve and display the data; the ID for the specific news to retrieve is provided, instead, via a parameter on the query string.

This is the essence of the Page Controller pattern for Web applications. The request targets a page whose logic and graphical layout are saved to disk. This approach has worked for a few years and still works today. The content of the news is displayed correctly, and everybody is generally happy. In addition, you have just one page to maintain, and you still have a way to identify a particular piece of news via the URL.

A possible drawback of this approach is that the real intent of the page might not be clear to users. And, more importantly, search engines usually assign higher ranks to terms contained in the URL. Therefore, an expressive URL provides search engines with an effective set of keywords that describe the page. To fix this, you need to make the entire URL friendlier and more readable. But you don't want to add new Web pages to the application or a bunch

of made-to-measure HTTP handlers. Ideally, you should try to transform the request in a command sent to the server rather than having it be simply the virtual file path name of the page to display.



Note The advent of Content Management Systems (CMS) raised the need to have friendlier URLs. A CMS is an application not necessarily written for a single user and that likely manages several pages created using semi-automatic algorithms. For these tools, resorting to pages with an algorithmically editable URL was a great help. But, alas, it was not a great help for users and search engines. This is where the need arises to expose user-friendly URLs while managing cryptic URLs internally. A URL rewriter API attempts to bridge precisely this gap.

The URL Routing Engine

To provide the ability to always expose friendly URLs to users, ASP.NET has supported a feature called *URL rewriting* since its inception. At its core, URL rewriting consists of an HTTP module (or a *global.asax* event handler) that hooks up a given request, parses its original URL, and instructs the HTTP runtime environment to serve a “possibly related but different” URL.

URL rewriting is a powerful feature; however, it’s not free of issues. For this reason, Microsoft more recently introduced a new API in ASP.NET. Although it’s based on the same underlying URL rewriting, the API offers a higher level of programmability and more features overall—and the URL routing engine in particular.

Originally devised for ASP.NET MVC, URL routing gives you total freedom to organize the layout of the URL recognized by your application. In a way, the URL becomes a command for the Web application; the application is the only entity put in charge of parsing and validating the syntax of the command. The URL engine is the system-provided component that validates the URL. The URL routing engine is general enough to be usable in both ASP.NET MVC and ASP.NET Web Forms; in fact, it was taken out of the ASP.NET MVC framework and incorporated in the general ASP.NET *system.web* assembly a while ago.

URL routing differs in ASP.NET MVC and ASP.NET Web Forms only with regard to how you express the final destination of the request. You use a controller-action pair in ASP.NET MVC; you use an ASPX path in ASP.NET Web Forms.

Original URL Rewriting API

URL rewriting helps you in two ways. It makes it possible for you to use a generic front-end page such as *news.aspx* and then redirect to a specific page whose actual URL is read from a database or any other container. In addition, it also enables you to request user-friendly URLs to be programmatically mapped to less intuitive, but easier to manage, URLs.

Here's a quick example of how you can rewrite the requested URL as another one:

```
protected void Application_BeginRequest(object sender, EventArgs e)
{
    // Get the current request context
    var context = HttpContext.Current;

    // Get the URL to the handler that will physically handle the request
    var newURL = ParseOriginalUrl(context);

    // Overwrite the target URL of the current request
    context.RewritePath(newURL);
}
```

The *RewritePath* method of *HttpContext* lets you change the URL of the current request on the fly, thus performing a sort of internal redirect. As a result, the user is provided the content generated for the URL you set through *RewritePath*. At the same time, the URL shown in the address bar remains as the originally requested one.

In a nutshell, URL rewriting exists to let you decouple the URL from the physical Web form that serves the requests.



Note The change of the final URL takes place on the server and, more importantly, within the context of the same call. *RewritePath* should be used carefully and mainly from within the *global.asax* file. In Web Forms, for example, if you use *RewritePath* in the context of a postback event, you can experience some view-state problems.

One drawback of the URL rewriting API is that as the API changes the target URL of the request, any postbacks are directed to the rewritten URL. For example, if you rewrite *news.aspx?id=1234* to *1234.aspx*, any postbacks from *1234.aspx* are targeted to the same *1234.aspx* instead of to the original URL.

This might or might not be a problem for you and, for sure, it doesn't break any page behavior. However, the original URL has just been fully replaced while you likely want to use the same, original URL as the front end. If this is the case (and most of the time, this is exactly the case), URL rewriting just created a new problem.

In addition, the URL rewriting logic is intrinsically monodirectional because it doesn't offer any built-in mechanism to go from the original URL to the rewritten URL and then back.

URL Patterns and Routes

The URL routing module is a system component that intercepts any request and attempts to match the URL to a predefined pattern. All requested URLs that match a given pattern are processed in a distinct way; typically, they are rewritten to other URLs.

The URL patterns that you define are known as *routes*.

A route contains placeholders that can be filled up with values extracted from the URL. Often referred to as a *route parameter*, a placeholder is a name enclosed in curly brackets { }. You can have multiple placeholders in a route as long as they are separated by a constant or delimiter. The forward slash (/) character acts as a delimiter between the various parts of the route. Here's a sample route:

```
Category/{action}/{categoryName}
```

URLs that match the preceding route begin with the word "Category" followed by two segments. The first segment will be mapped to the *action* route parameter; the second segment will be mapped to the *categoryName* route parameter. As you might have guessed, *action* and *categoryName* are just arbitrary names for parameters. A URL that matches the preceding route is the following:

```
/Category/Edit/Beverages
```

The route is nothing more than a pattern and is not associated with any logic of its own. Invoked by the routing module, the component that ultimately decides *how* to rewrite the matching URL is another one entirely. Precisely, it is the *route handler*.

Technically speaking, a route handler is a class that implements the *IRouteHandler* interface. The interface is defined as shown here:

```
public interface IRouteHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

In its *GetHttpHandler* method, a route handler typically looks at route parameters to figure out if any of the information available needs to be passed down to the HTTP handler (for example, an ASP.NET page) that will handle the request. If this is the case, the route handler adds this information to the *Items* collection of the HTTP context. Finally, the route handler obtains an instance of a class that implements the *IHttpHandler* interface and returns that.

For Web Forms requests, the route handler—an instance of the *PageRouteHandler* class—resorts to the ASP.NET build manager to identify the dynamic class for the requested page resource and creates the handler on the fly.



Important The big difference between plain URL rewriting and ASP.NET routing is that with ASP.NET routing, the URL is not changed when the system begins processing the request. Instead, it's changed later in the life cycle. In this way, the runtime environment can perform most of its usual tasks on the original URL, which is an approach that maintains a consistent and robust solution. In addition, a late intervention on the URL also gives developers a big chance to extract values from the URL and the request context. In this way, the routing mechanism can be driven by a set of rewriting rules or patterns. If the original URL matches a particular pattern, you rewrite it to the associated URL. URL patterns are an external resource and are kept in one place, which makes the solution more maintainable overall.

Routing in Web Forms

To introduce URL routing in your Web Forms application, you start by defining routes. Routes go in the *global.asax* file to be processed at the very beginning of the application. To define a route, you create an instance of the *Route* class by specifying the URL pattern, the handler, and optionally a name for the route. However, you typically use helper methods that save you a lot of details and never expose you directly to the API of the *Route* class. The next section shows some code that registers routes.



Note The vast majority of examples that illustrate routing in both ASP.NET MVC and Web Forms explicitly register routes from within *global.asax*. Loading route information from an external file is not be a bad idea, though, and will make your application a bit more resilient to changes.

Defining Routes for Specific Pages

In *Application_Start*, you invoke a helper method inside of which new routes are created and added to a static route collection object. Here's a sample *global.asax* class:

```
public class Global : System.Web.HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {
        RegisterRoutes(RouteTable.Routes);
    }

    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.MapPageRoute("Category",
            "Category/{action}/{categoryName}",
            "~/categories.aspx",
            true,
            new RouteValueDictionary
            {
                { "categoryName", "beverages" },
                { "action", "edit" }
            });
    }
}
```

All routes for the application are stored in a global container: the static *Routes* property of the *RouteTable* class. A reference to this property is passed to the helper *RegisterRoutes* method invoked upon application start.

The structure of the code you just saw is optimized for testability; nothing prevents you from stuffing all the code in the body of *Application_Start*.

The *MapPageRoute* method is a helper method that creates a *Route* object and adds it to the *Routes* collection. Here's a glimpse of its internal implementation:

```
public Route MapPageRoute(String routeName,
                          String routeUrl,
                          String physicalFile,
                          Boolean checkPhysicalUrlAccess,
                          RouteValueDictionary defaults,
                          RouteValueDictionary constraints,
                          RouteValueDictionary dataTokens)
{
    if (routeUrl == null)
    {
        throw new ArgumentNullException("routeUrl");
    }

    // Create the new route
    var route = new Route(routeUrl,
                          defaults, constraints, dataTokens,
                          new PageRouteHandler(physicalFile, checkPhysicalUrlAccess));

    // Add the new route to the global collection
    this.Add(routeName, route);
    return route;
}
```

The *MapPageRoute* method offers a simplified interface for creating a *Route* object. In particular, it requires you to specify the name of the route, the URL pattern for the route, and the physical ASP.NET Web Forms page the URL will map to. In addition, you can specify a Boolean flag to enforce the application of current authorization rules for the actual page. For example, imagine that the user requests a URL such as *customers/edit/alfki*. Imagine also that such a URL is mapped to *customers.aspx* and that this page is restricted to the admin role only. If the aforementioned Boolean argument is *false*, all users are allowed to view the page behind the URL. If the Boolean value is *true*, only admins will be allowed.

Finally, the *MapPageRoute* method can accept three dictionaries: the default values for URL parameters, additional constraints on the URL parameters, plus custom data values to pass on to the route handler.

In the previous example, we aren't using constraints and data tokens. Instead, we are specifying default values for the *categoryName* and *action* parameters. As a result, an incoming URL such as */category* will be automatically resolved as if it were */category/edit/beverages*.

Programmatic Access to Route Values

The *MapPageRoute* method just configures routes recognized by the application. Its job ends with the startup of the application. The URL routing HTTP module then kicks in for each request and attempts to match the request URL to any of the defined routes.

Routes are processed in the order in which they have been added to the *Routes* collection, and the search stops at the first match. For this reason, it is extremely important that you list your routes in decreasing order of importance—stricter rules must go first.

Beyond the order of appearance, other factors affect the process of matching URLs to routes. One is the set of default values that you might have provided for a route. Default values are simply values that are automatically assigned to defined placeholders in case the URL doesn't provide specific values. Consider the following two routes:

```
{Orders}/{Year}/{Month}
{Orders}/{Year}
```

If you assign the first route's default values for both *{Year}* and *{Month}*, the second route will never be evaluated because, thanks to the default values, the first route is always a match regardless of whether the URL specifies a year and a month.

The URL-routing HTTP module also uses constraints (which I'll say more about in a moment) to determine whether a URL matches a given route. If a match is finally found, the routing module gets the HTTP handler from the route handler and maps it to the HTTP context of the request.

Given the previously defined route, any matching requests are mapped to the *categories.aspx* page. How can this page know about the route parameters? How can this page know about the action requested or the category name? There's no need for the page to parse (again) the URL. Route parameters are available through a new property on the *Page* class—the *RouteData* property.

RouteData is a property of type *RouteData* and features the members listed in Table 4-9.

TABLE 4-9 Members of the *RouteData* Class

Member	Description
<i>DataTokens</i>	List of additional custom values that are passed to the route handler
<i>GetRequiredString</i>	Method that takes the name of a route parameter and returns its value
<i>Route</i>	Returns the current <i>Route</i> object
<i>RouteHandler</i>	Returns the handler for the current route
<i>Values</i>	Returns the dictionary of route parameter values

The following code snippet shows how you retrieve parameters in *Page_Load*:

```
protected void Page_Load(object sender, EventArgs e)
{
    var action = RouteData.GetRequiredString("action");
    ...
}
```

The only difference between using *GetRequiredString* and accessing the *Values* dictionary is that *GetRequiredString* throws if the requested value is not found. In addition, *GetRequiredString* uses protected access to the collection via *TryGetValue* instead of a direct reading.

Structure of Routes

A route is characterized by the five properties listed in Table 4-10.

TABLE 4-10 Properties of the *Route* Class

Property	Description
<i>Constraints</i>	List of additional constraints the URL should fulfill to match the route.
<i>DataTokens</i>	List of additional custom values that are passed to the route handler. These values, however, are not used to determine whether the route matches a URL pattern.
<i>Defaults</i>	List of default values to be used for route parameters.
<i>RouteHandler</i>	The object responsible for retrieving the HTTP handler to serve the request.
<i>Url</i>	The URL pattern for the route.

Constraints, *DataTokens*, and *Defaults* are all properties of type *RouteValueDictionary*. In spite of the fancy name, the *RouteValueDictionary* type is a plain `<String, Object>` dictionary.

Most of the time, the pattern defined by the route is sufficient to decide whether a given URL matches or not. However, this is not always the case. Consider, for example, the situation in which you are defining a route for recognizing requests for product details. You want to make sure of the following two aspects.

First, make sure the incoming URL is of the type `http://server/{category}/{productId}`, where `{category}` identifies the category of the product and `{productId}` indicates the ID of the product to retrieve.

Second, you also want to be sure that no invalid product ID is processed. You probably don't want to trigger a database call right from the URL routing module, but at the very least, you want to rule out as early as possible any requests that propose a product ID in an incompatible format. For example, if product IDs are numeric, you want to rule out anything passed in as a product ID that is alphanumeric.

Regular expressions are a simple way to filter requests to see if any segment of the URL is acceptable. Here's a sample route that keeps URLs with a string product ID off the application:

```
routes.MapPageRoute(
    "ProductInfo",
    "Category/{category}/{productId}/{locale}",
    "~/categories.aspx",
    true,
    new { category = "Beverages", locale="en-us" },
    new { productId = @"\d{8}",
        locale = "[a-z]{2}-[a-z]{2}" }
);
```

The sixth parameter to the *MapPageRoute* method is a dictionary object that sets regular expressions for *productId* and *locale*. In particular, the product ID must be a numeric sequence of exactly eight digits, whereas the locale must be a pair of two-letter strings separated by a dash. The filter doesn't ensure that all invalid product IDs and locale codes are stopped at the gate, but at least it cuts off a good deal of work. An invalid URL is presented as an HTTP 404 failure and is subject to application-specific handling of HTTP errors.

More in general, a route constraint is a condition that a given URL parameter must fulfill to make the URL match the route. A constraint is defined via either regular expressions or objects that implement the *IRouteConstraint* interface.

Preventing Routing for Defined URLs

The ASP.NET URL routing module gives you maximum freedom to keep certain URLs off the routing mechanism. You can prevent the routing system from handling certain URLs in two steps. First, you define a pattern for those URLs and save it to a route. Second, you link that route to a special route handler—the *StopRoutingHandler* class.

Any request that belongs to a route managed by a *StopRoutingHandler* object is processed as a plain ASP.NET Web Forms endpoint. The following code instructs the routing system to ignore any *.axd* requests:

```
// In global.asax.cs
protected void Application_Start(Object sender, EventArgs e)
{
    RegisterRoutes(RouteTable.Routes);
}

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    ...
}
```

All that *IgnoreRoute* does is associate a *StopRoutingHandler* route handler to the route built around the specified URL pattern, thus preventing all matching URLs from being processed.

A little explanation is required for the *{*pathInfo}* placeholder in the URL. The token *pathInfo* simply represents a placeholder for any content following the *.axd* URL. The asterisk (*), though, indicates that the last parameter should match the rest of the URL. In other words, anything that follows the *.axd* extension goes into the *pathInfo* parameter. Such parameters are referred to as *catch-all parameters*.



Note Earlier in this chapter, I presented HTTP handlers as a way to define your own commands for the application through customized URLs. So what's the difference between HTTP handlers and URL routing? In ASP.NET, HTTP handlers remain the only way to process requests; URL routing is an intermediate layer that pre-processes requests and determines the HTTP handler for them. In doing so, the routing module decides whether the URL meets the expectations of the application or not. In a nutshell, URL routing offers a more flexible and extensible API; if you just need one specially formatted URL, though, a direct HTTP handler is probably a simpler choice.

Summary

HTTP handlers and HTTP modules are the building blocks of the ASP.NET platform. ASP.NET includes several predefined handlers and HTTP modules, but developers can write handlers and modules of their own to perform a variety of tasks. HTTP handlers, in particular, are faster than ordinary Web pages and can be used in all circumstances in which you don't need state maintenance and postback events. To generate images dynamically on the server, for example, an HTTP handler is more efficient than a page.

Everything that occurs under the hood of the ASP.NET runtime environment occurs because of HTTP handlers. When you invoke a Web page or an ASP.NET Web service method, an appropriate HTTP handler gets into the game and serves your request.

HTTP modules are good at performing a number of low-level tasks for which tight interaction and integration with the request/response mechanism is a critical factor. Modules are sort of interceptors that you can place along an HTTP packet's path, from the Web server to the ASP.NET runtime and back. Modules have read and write capabilities, and they can filter and modify the contents of both inbound and outbound requests.

In ASP.NET 4, a special HTTP module has been introduced to simplify the management of application URLs and make the whole process more powerful. The URL routing HTTP module offers a programmer-friendly API to define URL patterns, and it automatically blocks calls

for nonmatching URLs and redirects matching URLs to specific pages. It's not much different from old-fashioned URL rewriting, but it offers a greater level of control to the programmer.

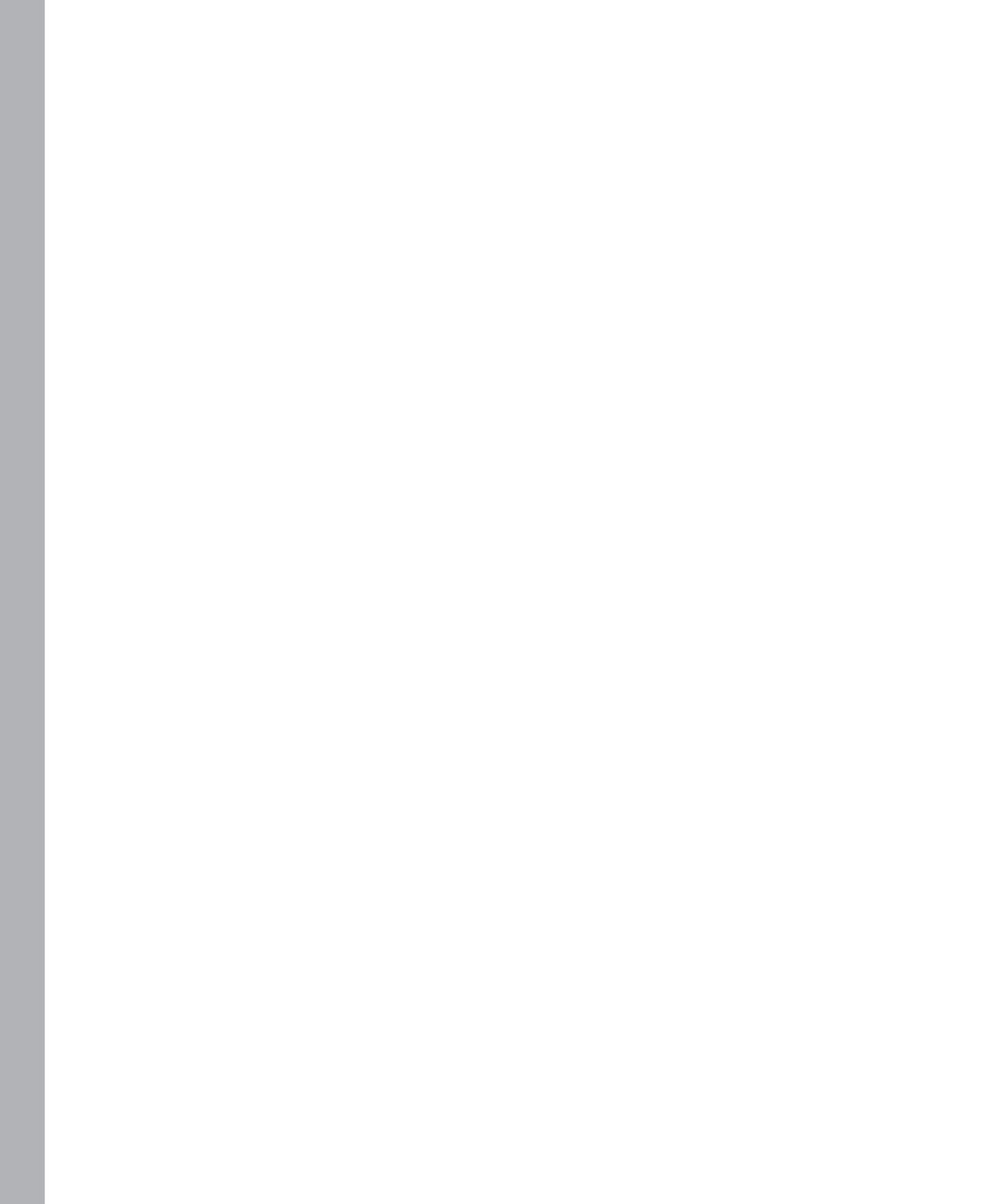
With this chapter, our exploration of the ASP.NET and IIS runtime environment terminates. With the next chapter, we'll begin a tour of the ASP.NET page-related features.

Part II

ASP.NET Pages and Server Controls

In this part:

Chapter 5: Anatomy of an ASP.NET Page	169
Chapter 6: ASP.NET Core Server Controls	217
Chapter 7: Working with the Page	269
Chapter 8: Page Composition and Usability	319
Chapter 9: ASP.NET Input Forms	365
Chapter 10: Data Binding	411
Chapter 11: The <i>ListView</i> Control	471
Chapter 12: Custom Controls	513



Chapter 5

Anatomy of an ASP.NET Page

The wise are instructed by reason; ordinary minds by experience; the stupid, by necessity; and brutes by instinct.

—Cicero

ASP.NET pages are dynamically compiled on demand when first requested in the context of a Web application. Dynamic compilation is not specific to ASP.NET pages alone (.aspx files); it also occurs with services (.svc and asmx files), Web user controls (.ascx files), HTTP handlers (.ashx files), and a few more ASP.NET application files such as the *global.asax* file. A pipeline of run-time modules takes care of the incoming HTTP packet and makes it evolve from a simple protocol-specific payload up to the rank of a server-side ASP.NET object—whether it’s an instance of a class derived from the system’s *Page* class or something else.

The ASP.NET HTTP runtime processes the page object and causes it to generate the markup to insert in the response. The generation of the response is marked by several events handled by user code and collectively known as the *page life cycle*.

In this chapter, we’ll review how an HTTP request for an *.aspx* resource is mapped to a page object, the programming interface of the *Page* class, and how to control the generation of the markup by handling events of the page life cycle.



Note By default in release mode, application pages are compiled in batch mode, meaning that ASP.NET attempts to stuff as many uncompiled pages as possible into a single assembly. The attributes *maxBatchSize* and *maxBatchGeneratedFileSize* in the `<compilation>` section let you limit the number of pages packaged in a single assembly and the overall size of the assembly. By default, you will have no more than 1000 pages per batched compilation and no assembly larger than 1 MB. In general, you don’t want users to wait too long when a large number of pages are compiled the first time. At the same time, you don’t want to load a huge assembly in memory to serve only a small page, or to start compilation for each and every page. The *maxBatchSize* and *maxBatchGeneratedFileSize* attributes help you find a good balance between first-hit delay and memory usage.

Invoking a Page

Let's start by examining in detail how the *.aspx* page is converted into a class and then compiled into an assembly. Generating an assembly for a particular *.aspx* resource is a two-step process. First, the source code of the resource file is parsed and a corresponding class is created that inherits either from *Page* or another class that, in turn, inherits from *Page*. Second, the dynamically generated class is compiled into an assembly and cached in an ASP.NET-specific temporary directory.

The compiled page remains in use as long as no changes occur to the linked *.aspx* source file or the whole application is restarted. Any changes to the linked *.aspx* file invalidate the current page-specific assembly and force the HTTP runtime to create a new assembly on the next request for the page.



Note Editing files such as *web.config* and *global.asax* causes the whole application to restart. In this case, all the pages will be recompiled as soon as each page is requested. The same happens if a new assembly is copied or replaced in the application's *Bin* folder.

The Runtime Machinery

Most of the requests that hit Internet Information Services (IIS) are forwarded to a particular run-time module for actual processing. The only exception to this model is made for static resources (for example, images) that IIS can quickly serve on its own. A module that can handle Web resources within IIS is known as an ISAPI extension and can be made of managed or unmanaged code. The worker process that serves the Web application in charge of the request loads the pinpointed module and commands it through a contracted programming interface.

For example, old-fashioned ASP pages are processed by an ISAPI extension named *asp.dll* whereas files with an *.aspx* extension—classic Web Forms pages—are assigned to an ISAPI extension named *aspnet_isapi.dll*, as shown in Figure 5-1. Extension-less requests like those managed by an ASP.NET MVC application are intercepted at the gate and redirected to completely distinct runtime machinery. (At least this is what happens under IIS 7 in integrated mode. In older configurations, you still need to register a specific extension for the requests to be correctly handled by IIS.)

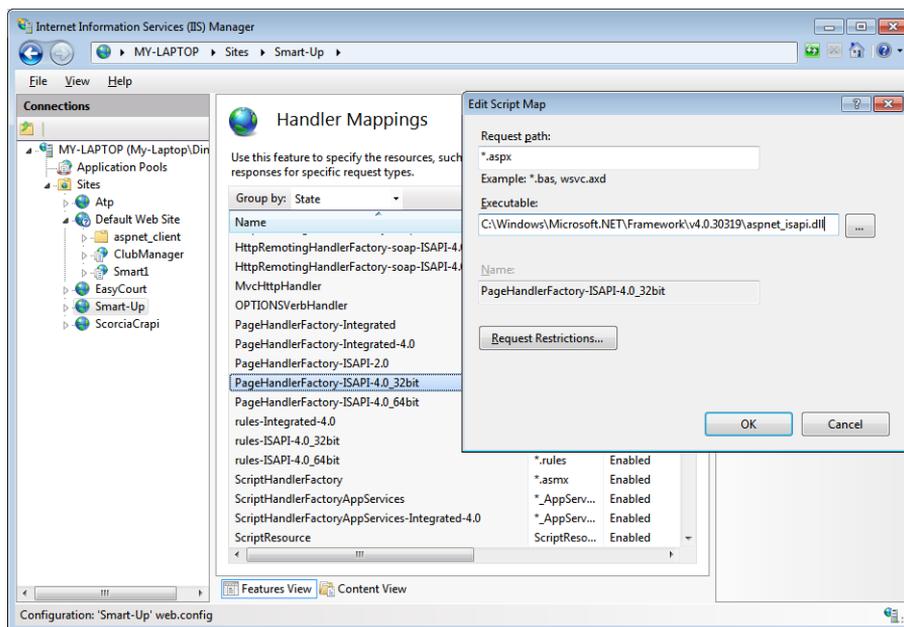


FIGURE 5-1 Setting the handler for resources with an *.aspx* extension.

Resource Mappings

IIS stores the list of recognized resources in the IIS metabase. Depending on the version of IIS you are using, the metabase might be a hidden component or a plain configuration file that an administrator can freely edit by hand. Regardless of the internal implementation, the IIS manager tool provides a user interface to edit the content of the metabase.

Upon installation, ASP.NET modifies the IIS metabase to make sure that *aspnet_isapi.dll* can handle some typical ASP.NET resources. Table 5-1 lists some of these resources.

TABLE 5-1 IIS Application Mappings for *aspnet_isapi.dll*

Extension	Resource Type
<i>.asax</i>	ASP.NET application files. Note, though, that any <i>.asax</i> file other than <i>global.asax</i> is ignored. The mapping is there only to ensure that <i>global.asax</i> can't be requested directly.
<i>.ascx</i>	ASP.NET user control files.
<i>.ashx</i>	HTTP handlers—namely, managed modules that interact with the low-level request and response services of IIS.
<i>.asmx</i>	Files that represent the endpoint of old-fashioned .NET Web services.
<i>.aspx</i>	Files that represent ASP.NET pages.
<i>.axd</i>	Extension that identifies internal HTTP handlers used to implement system features such as application-level tracing (<i>trace.axd</i>) or script injection (<i>webresource.axd</i>).
<i>.svc</i>	Files that represent the endpoint of a Windows Communication Foundation (WCF) service.

In addition, the *aspnet_isapi.dll* extension handles other typical Microsoft Visual Studio extensions such as *.cs*, *.csproj*, *.vb*, *.vbproj*, *.config*, and *.resx*.

As mentioned in Chapter 2, “ASP.NET and IIS,” the exact behavior of the ASP.NET ISAPI extension depends on the process model selected for the application—integrated pipeline (the default in IIS 7 and superior) or classic pipeline. Regardless of the model, at the end of the processing pipeline the originally requested URL that refers to an *.aspx* resource is mapped to, and served through, an instance of a class that represents an ASP.NET Web Forms page. The base class is the *System.Web.UI.Page* class.

Representing the Requested Page

The aforementioned *Page* class is only the base class. The actual class being used by the IIS worker process is a dynamically created derived class. So the ASP.NET HTTP runtime environment first determines the name of the class that will be used to serve the request. A particular naming convention links the URL of the page to the name of the class. If the requested page is, say, *default.aspx*, the associated class turns out to be *ASP.default.aspx*. The transformation rule applies a fixed ASP namespace and replaces any dot (.) with an underscore (_). If the URL contains a directory name, any slashes are also replaced with an underscore.

If no class exists with the specified name in any of the assemblies currently loaded in the AppDomain, the HTTP runtime orders that the class be created and compiled on the fly. This step is often referred to as the *dynamic compilation* of ASP.NET pages.

The source code for the new class is created by parsing the source code of the *.aspx* resource, and it’s temporarily saved in the ASP.NET temporary folder. The parser attempts to create a class with an initializer method able to create instances of any referenced server controls found in the ASPX markup. A referenced server control results from tags explicitly decorated with the *runat=server* attribute and from contiguous literals, including blanks and carriage returns. For example, consider the following short piece of markup:

```
<html>
<body>
<asp:button runat="server" ID="Button1" text="Click" />
</body>
</html>
```

When parsed, it sparks three distinct server control instances: two literal controls and a *Button* control. The first literal comprehends the text “<html><body>” plus any blanks and carriage returns the editor has put in. The second literal includes “</body></html>”.

Next, the *Page*-derived class is compiled and loaded in memory to serve the request. When a new request for the same page arrives, the class is ready and no compile step will ever take place. (The class will be re-created and recompiled only if the source code of the *.aspx* source changes at some point.)

The *ASP.default_aspx* class inherits from *Page* or, more likely, from a class that in turn inherits from *Page*. More precisely, the base class for *ASP.default_aspx* will be a combination of the code-behind, partial class you created through Visual Studio and a second partial class dynamically arranged by the ASP.NET HTTP runtime. The second, implicit partial class contains the declaration of protected properties for any explicitly referenced server controls. This second partial class is the key that allows you to write the following code successfully:

```
// No member named Button1 has ever been explicitly declared in any code-behind
// class. It is silently added at compile time through a partial class.
Button1.Text = ...;
```

Partial classes are a hot feature of .NET compilers. When partially declared, a class has its source code split over multiple source files, each of which appears to contain an ordinary class definition from beginning to end. The keyword *partial*, though, informs the compiler that the class declaration being processed is incomplete. To get full and complete source code, the compiler must look into other files specified on the command line.

Partial Classes in ASP.NET Projects

Partial classes are a compiler feature originally designed to overcome the brittleness of tool-generated code back in Visual Studio 2003 projects. Ideal for team development, partial classes simplify coding and avoid manual file synchronization in all situations in which many authors work on distinct segments of the class logical class.

Generally, partial classes are a source-level, assembly-limited, non-object-oriented way to extend the behavior of a class. A number of advantages are derived from intensive use of partial classes. As mentioned, you can have multiple teams at work on the same component at the same time. In addition, you have a neat and elegant way to add functionality to a class incrementally. In the end, this is just what the ASP.NET runtime does.

The ASPX markup defines server controls that will be handled by the code in the code-behind class. For this model to work, the code-behind class needs to incorporate references to these server controls as internal members—typically, protected members. In Visual Studio, the code-behind class is a partial class that just lacks members' declaration. Missing declarations are incrementally added at run time via a second partial class created by the ASP.NET HTTP runtime. The compiler of choice (C#, Microsoft Visual Basic .NET, or whatever) will then merge the two partial classes to create the real parent of the dynamically created page class.

Processing the Request

So to serve a request for a page named *default.aspx*, the ASP.NET runtime gets or creates a reference to a class named *ASP.default_aspx*. Next, the HTTP runtime environment invokes the class through the methods of a well-known interface—*IHttpHandler*. The root *Page* class implements this interface, which includes a couple of members: the *ProcessRequest* method and the Boolean *IsReusable* property. After the HTTP runtime has obtained an instance of the class that represents the requested resource, invoking the *ProcessRequest* method—a public method—gives birth to the process that culminates in the generation of the final response for the browser. As mentioned, the steps and events that execute and trigger out of the call to *ProcessRequest* are collectively known as the page life cycle.

Although serving pages is the ultimate goal of the ASP.NET runtime, the way in which the resultant markup code is generated is much more sophisticated than in other platforms and involves many objects. The IIS worker process passes any incoming HTTP requests to the so-called HTTP pipeline. The HTTP pipeline is a fully extensible chain of managed objects that works according to the classic concept of a pipeline. All these objects form what is often referred to as the *ASP.NET HTTP runtime environment*.

This ASP.NET-specific pipeline is integrated with the IIS pipeline in place for any requests when the Web application is configured to work in IIS 7 Integrated mode. Otherwise, IIS and ASP.NET use distinct pipelines—an unmanaged pipeline for IIS and a managed pipeline for ASP.NET.

A page request passes through a pipeline of objects that process the original HTTP payload and, at the end of the chain, produce some markup code for the browser. The entry point in this pipeline is the *HttpRuntime* class.

The *HttpRuntime* Class

The ASP.NET worker process activates the HTTP pipeline in the beginning by creating a new instance of the *HttpRuntime* class and then calling its *ProcessRequest* method for each incoming request. For the sake of clarity, note that despite the name, *HttpRuntime.ProcessRequest* has nothing to do with the *IHttpHandler* interface.

The *HttpRuntime* class contains a lot of private and internal methods and only three public static methods: *Close*, *ProcessRequest*, and *UnloadAppDomain*, as detailed in Table 5-2.

TABLE 5-2 Public Methods in the *HttpRuntime* Class

Method	Description
<i>Close</i>	Removes all items from the ASP.NET cache, and terminates the Web application. This method should be used only when your code implements its own hosting environment. There is no need to call this method in the course of normal ASP.NET request processing.
<i>ProcessRequest</i>	Drives all ASP.NET Web processing execution.
<i>UnloadAppDomain</i>	Terminates the current ASP.NET application. The application restarts the next time a request is received for it.

Note that all the methods shown in Table 5-2 have limited applicability in user applications. In particular, you're not supposed to use *ProcessRequest* in your own code, whereas *Close* is useful only if you're hosting ASP.NET in a custom application. Of the three methods in Table 5-2, only *UnloadAppDomain* can be considered for use if, under certain run-time conditions, you realize you need to restart the application. (See the sidebar "What Causes Application Restarts?" later in this chapter.)

Upon creation, the *HttpRuntime* object initializes a number of internal objects that will help carry out the page request. Helper objects include the cache manager and the file system monitor used to detect changes in the files that form the application. When the *ProcessRequest* method is called, the *HttpRuntime* object starts working to serve a page to the browser. It creates a new empty context for the request and initializes a specialized text writer object in which the markup code will be accumulated. A context is given by an instance of the *HttpContext* class, which encapsulates all HTTP-specific information about the request.

After that, the *HttpRuntime* object uses the context information to either locate or create a Web application object capable of handling the request. A Web application is searched using the virtual directory information contained in the URL. The object used to find or create a new Web application is *HttpApplicationFactory*—an internal-use object responsible for returning a valid object capable of handling the request.

Before we get to discover more about the various components of the HTTP pipeline, a look at Figure 5-2 is in order.

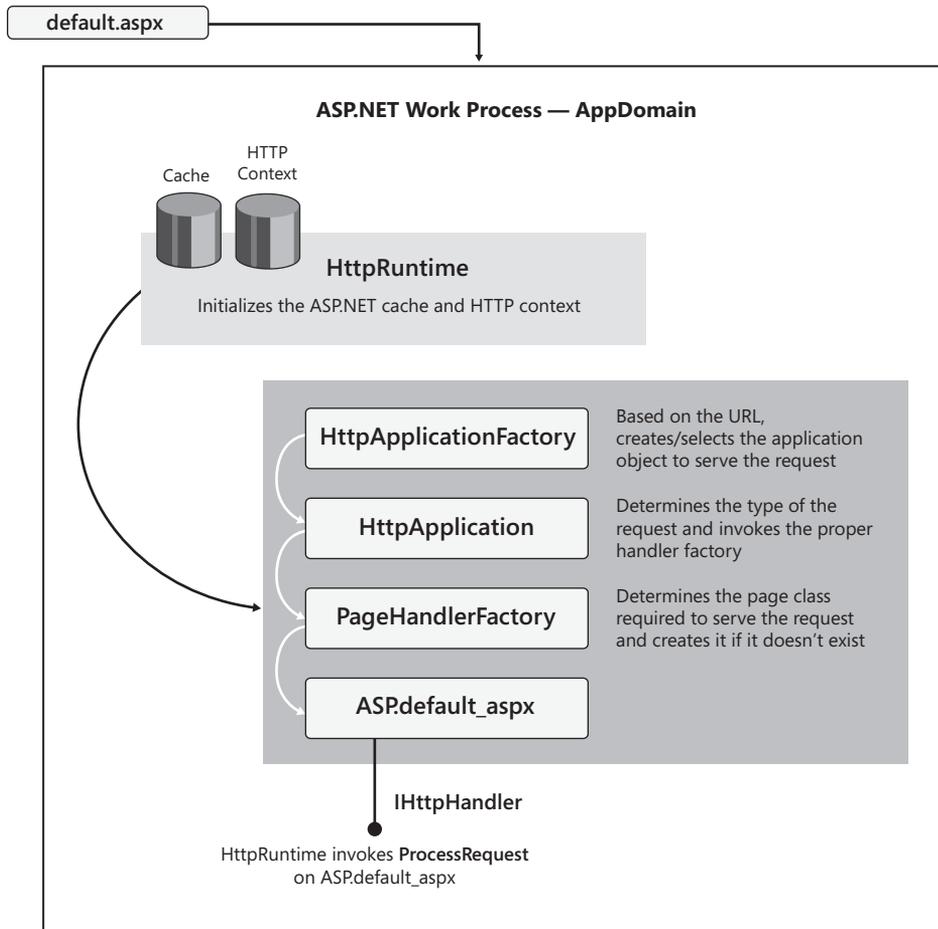


FIGURE 5-2 The HTTP pipeline processing for a page.

The Application Factory

During the lifetime of the application, the *HttpApplicationFactory* object maintains a pool of *HttpApplication* objects to serve incoming HTTP requests. When invoked, the application factory object verifies that an AppDomain exists for the virtual folder the request targets. If the application is already running, the factory picks an *HttpApplication* out of the pool of available objects and passes it the request. A new *HttpApplication* object is created if an existing object is not available.

If the virtual folder has not yet been called for the first time, a new *HttpApplication* object for the virtual folder is created in a new AppDomain. In this case, the creation of an *HttpApplication* object entails the compilation of the *global.asax* application file, if one is

present, and the creation of the assembly that represents the actual page requested. This event is actually equivalent to the start of the application. An *HttpApplication* object is used to process a single page request at a time; multiple objects are used to serve simultaneous requests.

The *HttpApplication* Object

HttpApplication is the base class that represents a running ASP.NET application. A derived HTTP application class is dynamically generated by parsing the contents of the *global.asax* file, if any is present. If *global.asax* is available, the application class is built and named after it: *ASP.global_asax*. Otherwise, the base *HttpApplication* class is used.

An instance of an *HttpApplication*-derived class is responsible for managing the entire lifetime of the request it is assigned to. The same instance can be reused only after the request has been completed. The *HttpApplication* maintains a list of HTTP module objects that can filter and even modify the content of the request. Registered modules are called during various moments of the elaboration as the request passes through the pipeline.

The *HttpApplication* object determines the type of object that represents the resource being requested—typically, an ASP.NET page, a Web service, or perhaps a user control. *HttpApplication* then uses the proper handler factory to get an object that represents the requested resource. The factory either instantiates the class for the requested resource from an existing assembly or dynamically creates the assembly and then an instance of the class. A handler factory object is a class that implements the *IHttpHandlerFactory* interface and is responsible for returning an instance of a managed class that can handle the HTTP request—an HTTP handler. An ASP.NET page is simply a handler object—that is, an instance of a class that implements the *IHttpHandler* interface.

Let's see what happens when the resource requested is a page.

The Page Factory

When the *HttpApplication* object in charge of the request has figured out the proper handler, it creates an instance of the handler factory object. For a request that targets a page, the factory is a class named *PageHandlerFactory*. To find the appropriate handler, *HttpApplication* uses the information in the `<httpHandlers>` section of the configuration file as a complement to the information stored in the IIS handler mappings list, as shown in Figure 5-3.

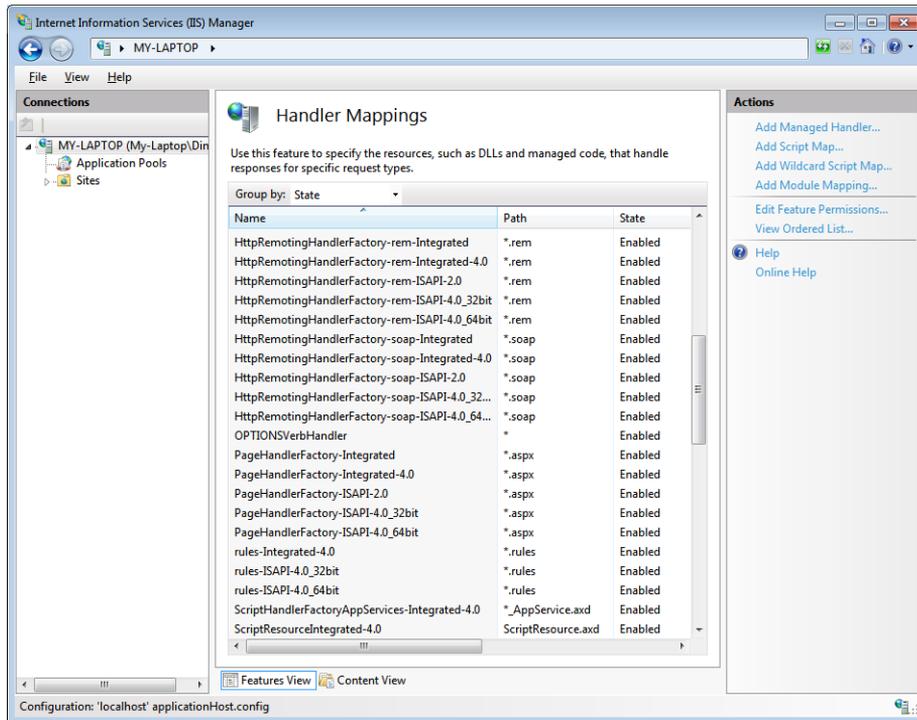


FIGURE 5-3 The HTTP pipeline processing for a page.

Bear in mind that handler factory objects do not compile the requested resource each time it is invoked. The compiled code is stored in an ASP.NET temporary directory on the Web server and used until the corresponding resource file is modified.

So the page handler factory creates an instance of an object that represents the particular page requested. As mentioned, the actual object inherits from the *System.Web.UI.Page* class, which in turn implements the *IHttpHandler* interface. The page object is returned to the application factory, which passes that back to the *HttpRuntime* object. The final step accomplished by the ASP.NET runtime is calling the *IHttpHandler's ProcessRequest* method on the page object. This call causes the page to execute the user-defined code and generate the markup for the browser.

In Chapter 17, “ASP.NET State Management,” we’ll return to the initialization of an ASP.NET application, the contents of *global.asax*, and the information stuffed into the HTTP context—a container object, created by the *HttpRuntime* class, that is populated, passed along the pipeline, and finally bound to the page handler.

What Causes Application Restarts?

There are a few reasons why an ASP.NET application can be restarted. For the most part, an application is restarted to ensure that latent bugs or memory leaks don't affect the overall behavior of the application in the long run. Another reason is that too many changes dynamically made to deployed ASPX pages might have caused too large a number of assemblies (typically, one per page) to be loaded in memory.

Note that any applications that consume more than a certain share of virtual memory are automatically killed and restarted by IIS. In IIS 7, you can even configure a periodic recycle to ensure that your application is always lean, mean, and in good shape.

Furthermore, the hosting environment (IIS or ASP.NET, depending on the configuration) implements a good deal of checks and automatically restarts an application if any the following scenarios occur:

- The maximum limit of dynamic page compilations is reached. This limit is configurable through the *web.config* file.
- The physical path of the Web application has changed, or any directory under the Web application folder is renamed.
- Changes occurred in *global.asax*, *machine.config*, or *web.config* in the application root, or in the *Bin* directory or any of its subdirectories.
- Changes occurred in the code-access security policy file, if one exists.
- Too many files are changed in one of the content directories. (Typically, this happens if files are generated on the fly when requested.)
- You modified some of the properties for the application pool hosting the Web application.

In addition to all this, in ASP.NET an application can be restarted programmatically by calling *HttpRuntime.UnloadAppDomain*.

The Processing Directives of a Page

Processing directives configure the run-time environment that will execute the page. In ASP.NET, directives can be located anywhere in the page, although it's a good and common practice to place them at the beginning of the file. In addition, the name of a directive is case insensitive and the values of directive attributes don't need to be quoted. The most important and most frequently used directive in ASP.NET is *@Page*. The complete list of ASP.NET directives is shown in Table 5-3.

TABLE 5-3 Directives Supported by ASP.NET Pages

Directive	Description
<i>@ Assembly</i>	Links an assembly to the current page or user control.
<i>@ Control</i>	Defines control-specific attributes that guide the behavior of the control compiler.
<i>@ Implements</i>	Indicates that the page, or the user control, implements a specified .NET Framework interface.
<i>@ Import</i>	Indicates a namespace to import into a page or user control.
<i>@ Master</i>	Identifies an ASP.NET master page. (See Chapter 8, "Page Composition and Usability.")
<i>@ MasterType</i>	Provides a way to create a strongly typed reference to the ASP.NET master page when the master page is accessed from the <i>Master</i> property. (See Chapter 8.)
<i>@ OutputCache</i>	Controls the output caching policies of a page or user control. (See Chapter 18, "ASP.NET Caching.")
<i>@ Page</i>	Defines page-specific attributes that guide the behavior of the page compiler and the language parser that will preprocess the page.
<i>@ PreviousPageType</i>	Provides a way to get strong typing against the previous page, as accessed through the <i>PreviousPage</i> property.
<i>@ Reference</i>	Links a page or user control to the current page or user control.
<i>@ Register</i>	Creates a custom tag in the page or the control. The new tag (prefix and name) is associated with the namespace and the code of a user-defined control.

With the exception of *@Page*, *@PreviousPageType*, *@Master*, *@MasterType*, and *@Control*, all directives can be used both within a page and a control declaration. *@Page* and *@Control* are mutually exclusive. *@Page* can be used only in *.aspx* files, while the *@Control* directive can be used only in user control *.ascx* files. *@Master*, in turn, is used to define a very special type of page—the master page.

The syntax of a processing directive is unique and common to all supported types of directives. Multiple attributes must be separated with blanks, and no blank can be placed around the equal sign (=) that assigns a value to an attribute, as the following line of code demonstrates:

```
<%@ Directive_Name attribute="value" [attribute="value"...] %>
```

Each directive has its own closed set of typed attributes. Assigning a value of the wrong type to an attribute, or using a wrong attribute with a directive, results in a compilation error.



Important The content of directive attributes is always rendered as plain text. However, attributes are expected to contain values that can be rendered to a particular .NET Framework type, specific to the attribute. When the ASP.NET page is parsed, all the directive attributes are extracted and stored in a dictionary. The names and number of attributes must match the expected schema for the directive. The string that expresses the value of an attribute is valid as long as it can be converted into the expected type. For example, if the attribute is designed to take a Boolean value, *true* and *false* are its only feasible values.

The @Page Directive

The @Page directive can be used only in .aspx pages and generates a compile error if used with other types of ASP.NET files such as controls and Web services. Each .aspx file is allowed to include at most one @Page directive. Although not strictly necessary from the syntax point of view, the directive is realistically required by all pages of some complexity.

@Page features over 40 attributes that can be logically grouped in three categories: compilation (defined in Table 5-4), overall page behavior (defined in Table 5-5), and page output (defined in Table 5-6). Each ASP.NET page is compiled upon first request, and the HTML actually served to the browser is generated by the methods of the dynamically generated class. The attributes listed in Table 5-4 let you fine-tune parameters for the compiler and choose the language to use.

TABLE 5-4 @Page Attributes for Page Compilation

Attribute	Description
<i>ClassName</i>	Specifies the name of the class that will be dynamically compiled when the page is requested. It must be a class name without namespace information.
<i>CodeFile</i>	Indicates the path to the code-behind class for the current page. The source class file must be deployed to the Web server.
<i>CodeBehind</i>	Attribute consumed by Visual Studio, indicates the path to the code-behind class for the current page. The source class file will be compiled to a deployable assembly.
<i>CodeFileBaseClass</i>	Specifies the type name of a base class for a page and its associated code-behind class. The attribute is optional, but when it is used the <i>CodeFile</i> attribute must also be present.
<i>CompilationMode</i>	Indicates whether the page should be compiled at run time.
<i>CompilerOptions</i>	A sequence of compiler command-line switches used to compile the page.
<i>Debug</i>	A Boolean value that indicates whether the page should be compiled with debug symbols.
<i>Explicit</i>	A Boolean value that determines whether the page is compiled with the Visual Basic <i>Option Explicit</i> mode set to <i>On</i> . <i>Option Explicit</i> forces the programmer to explicitly declare all variables. The attribute is ignored if the page language is not Visual Basic .NET.

Attribute	Description
<i>Inherits</i>	Defines the base class for the page to inherit. It can be any class derived from the <i>Page</i> class.
<i>Language</i>	Indicates the language to use when compiling inline code blocks (<% ... %>) and all the code that appears in the page <script> section. Supported languages include Visual Basic .NET, C#, JScript .NET, and J#. If not otherwise specified, the language defaults to Visual Basic .NET.
<i>LinePragmas</i>	Indicates whether the run time should generate line pragmas in the source code to mark specific locations in the file for the sake of debugging tools.
<i>MasterPageFile</i>	Indicates the master page for the current page.
<i>Src</i>	Indicates the source file that contains the implementation of the base class specified with <i>Inherits</i> . The attribute is not used by Visual Studio and other Rapid Application Development (RAD) designers.
<i>Strict</i>	A Boolean value that determines whether the page is compiled with the Visual Basic <i>Option Strict</i> mode set to <i>On</i> . When this attribute is enabled, <i>Option Strict</i> permits only type-safe conversions and prohibits implicit conversions in which loss of data is possible. (In this case, the behavior is identical to that of C#.) The attribute is ignored if the page language is not Visual Basic .NET.
<i>Trace</i>	A Boolean value that indicates whether tracing is enabled. If tracing is enabled, extra information is appended to the page's output. The default is <i>false</i> .
<i>TraceMode</i>	Indicates how trace messages are to be displayed for the page when tracing is enabled. Feasible values are <i>SortByTime</i> and <i>SortByCategory</i> . The default, when tracing is enabled, is <i>SortByTime</i> .
<i>WarningLevel</i>	Indicates the compiler warning level at which you want the compiler to abort compilation for the page. Possible values are 0 through 4.

Attributes listed in Table 5-5 allow you to control to some extent the overall behavior of the page and the supported range of features. For example, you can set a custom error page, disable session state, and control the transactional behavior of the page.



Note The schema of attributes supported by *@Page* is not as strict as for other directives. In particular, any public properties defined on the page class can be listed as an attribute, and initialized, in a *@Page* directive.

TABLE 5-5 @Page Attributes for Page Behavior

Attribute	Description
<i>AspCompat</i>	A Boolean attribute that, when set to <i>true</i> , allows the page to be executed on a single-threaded apartment (STA) thread. The setting allows the page to call COM+ 1.0 components and components developed with Microsoft Visual Basic 6.0 that require access to the unmanaged ASP built-in objects. (I'll return to this topic in Chapter 16, "The HTTP Request Context.")
<i>Async</i>	If this attribute is set to <i>true</i> , the generated page class derives from <i>IHttpAsyncHandler</i> rather than having <i>IHttpHandler</i> add some built-in asynchronous capabilities to the page.

Attribute	Description
<i>AsyncTimeout</i>	Defines the timeout in seconds used when processing asynchronous tasks. The default is 45 seconds.
<i>AutoEventWireup</i>	A Boolean attribute that indicates whether page events are automatically enabled. It's set to <i>true</i> by default. Pages developed with Visual Studio .NET have this attribute set to <i>false</i> , and page events for these pages are individually tied to handlers.
<i>Buffer</i>	A Boolean attribute that determines whether HTTP response buffering is enabled. It's set to <i>true</i> by default.
<i>Description</i>	Provides a text description of the page. The ASP.NET page parser ignores the attribute, which subsequently has only a documentation purpose.
<i>EnableEventValidation</i>	A Boolean value that indicates whether the page will emit a hidden field to cache available values for input fields that support event data validation. It's set to <i>true</i> by default.
<i>EnableSessionState</i>	Defines how the page should treat session data. If this attribute is set to <i>true</i> , the session state can be read and written to. If it's set to <i>false</i> , session data is not available to the application. Finally, if this attribute is set to <i>ReadOnly</i> , the session state can be read but not changed.
<i>EnableViewState</i>	A Boolean value that indicates whether the page <i>view state</i> is maintained across page requests. The view state is the page call context—a collection of values that retain the state of the page and are carried back and forth. View state is enabled by default. (I'll cover this topic in Chapter 17. "ASP.NET State Management.")
<i>EnableTheming</i>	A Boolean value that indicates whether the page will support themes for embedded controls. It's set to <i>true</i> by default.
<i>EnableViewStateMac</i>	A Boolean value that indicates ASP.NET should calculate a machine-specific authentication code and append it to the view state of the page (in addition to Base64 encoding). The <i>Mac</i> in the attribute name stands for <i>machine authentication check</i> . When the attribute is <i>true</i> , upon postbacks ASP.NET will check the authentication code of the view state to make sure that it hasn't been tampered with on the client.
<i>ErrorPage</i>	Defines the target URL to which users will be automatically redirected in case of unhandled page exceptions.
<i>MaintainScrollPositionOnPostBack</i>	A Boolean value that indicates whether to return the user to the same position in the client browser after postback.
<i>SmartNavigation</i>	A Boolean value that indicates whether the page supports the Microsoft Internet Explorer 5 or later smart navigation feature. Smart navigation allows a page to be refreshed without losing scroll position and element focus.
<i>Theme, StylesheetTheme</i>	Indicates the name of the theme (or style-sheet theme) selected for the page.

Attribute	Description
<i>Transaction</i>	Indicates whether the page supports or requires transactions. Feasible values are <i>Disabled</i> , <i>NotSupported</i> , <i>Supported</i> , <i>Required</i> , and <i>RequiresNew</i> . Transaction support is disabled by default.
<i>ValidateRequest</i>	A Boolean value that indicates whether request validation should occur. If this attribute is set to <i>true</i> , ASP.NET checks all input data against a hard-coded list of potentially dangerous values. This functionality helps reduce the risk of cross-site scripting attacks for pages. The value is <i>true</i> by default.

Attributes listed in Table 5-6 allow you to control the format of the output being generated for the page. For example, you can set the content type of the page or localize the output to the extent possible.

TABLE 5-6 @Page Directives for Page Output

Attribute	Description
<i>ClientTarget</i>	Indicates the target browser for which ASP.NET server controls should render content.
<i>ClientIDMode</i>	Specifies the algorithm to use to generate client ID values for server controls. <i>This attribute requires ASP.NET 4.</i>
<i>CodePage</i>	Indicates the code page value for the response. Set this attribute only if you created the page using a code page other than the default code page of the Web server on which the page will run. In this case, set the attribute to the code page of your development machine. A code page is a character set that includes numbers, punctuation marks, and other glyphs. Code pages differ on a per-language basis.
<i>ContentType</i>	Defines the content type of the response as a standard MIME type. Supports any valid HTTP content type string.
<i>Culture</i>	Indicates the culture setting for the page. Culture information includes the writing and sorting system, calendar, and date and currency formats. The attribute must be set to a non-neutral culture name, which means it must contain both language and country/region information. For example, <i>en-US</i> is a valid value, unlike <i>en</i> alone, which is considered country/region neutral.
<i>LCID</i>	A 32-bit value that defines the locale identifier for the page. By default, ASP.NET uses the locale of the Web server.
<i>MetaDescription</i>	Sets the “description” meta element for the page. The value set through the <i>@Page</i> directive overrides any similar values you might have specified as literal text in the markup. <i>This attribute requires ASP.NET 4.</i>
<i>MetaKeywords</i>	Sets the “keywords” meta element for the page. The value set through the <i>@Page</i> directive overrides any similar values you might have specified as literal text in the markup. <i>This attribute requires ASP.NET 4.</i>
<i>ResponseEncoding</i>	Indicates the character encoding of the page. The value is used to set the <i>CharSet</i> attribute on the content type HTTP header. Internally, ASP.NET handles all strings as Unicode.

Attribute	Description
<i>UICulture</i>	Specifies the default culture name used by Resource Manager to look up culture-specific resources at run time.
<i>ViewStateEncryptionMode</i>	Determines how and if the <i>view state</i> is encrypted. Feasible values are <i>Auto</i> , <i>Always</i> , or <i>Never</i> . The default is <i>Auto</i> , meaning that view state will be encrypted only if an individual control requests that.
<i>ViewStateMode</i>	Determines the value for the page's <i>ViewStateMode</i> property that influences the way in which the page treats the view state of child controls. (More details are available in Chapter 17.) <i>This attribute requires ASP.NET 4.</i>

As you can see, many attributes discussed in Table 5-6 are concern with page localization. Building multilanguage and international applications is a task that ASP.NET, and the .NET Framework in general, greatly simplify.

The @Assembly Directive

The *@Assembly* directive adds an assembly to a collection of assembly names that are used during the compilation of the ASP.NET page so that classes and interfaces in the assembly are available for early binding to the code. You use the *@Assembly* directive when you want to reference a given assembly only from a specific page.

Some assemblies are linked by default for any ASP.NET application. The complete list can be found in the root *web.config* file of the Web server machine. The list is pretty long in ASP.NET 4, but it no longer includes the *System.Web.Mobile* assembly that was there for older versions of ASP.NET. The mobile assembly is now deprecated, but if you're trying to upgrade an existing application to ASP.NET 4 that uses the assembly, you are required to add the assembly explicitly via an *@Assembly* directive or via a custom *<compilation>* section in the application.

Table 5-7 lists some of the assemblies that are automatically provided to the compiler for an ASP.NET 4 application.

TABLE 5-7 Assemblies Linked by Default in ASP.NET 4

Assembly File Name	Description
<i>mscorlib</i>	Provides the core functionality of the .NET Framework, including types, AppDomains, and run-time services
<i>System.dll</i>	Provides another bunch of system services, including regular expressions, compilation, native methods, file I/O, and networking
<i>System.Configuration.dll</i>	Defines classes to read and write configuration data.
<i>System.Core.dll</i>	Provides some other core functionality of the .NET Framework, including LINQ-to-Objects, the time-zone API, and some security and diagnostic classes

<i>System.Data.dll</i>	Defines data container and data access classes, including the whole ADO.NET framework
<i>System.Data.DataSetExtensions.dll</i>	Defines additional functions built over the ADO.NET <i>DataSet</i> object
<i>System.Drawing.dll</i>	Implements the GDI+ features
<i>System.EnterpriseServices.dll</i>	Provides the classes that allow for serviced components and COM+ interaction
<i>System.Web.dll</i>	Indicates the assembly implements the core ASP.NET services, controls, and classes
<i>System.Web.ApplicationServices.dll</i>	Provides classes that enable you to access ASP.NET authentication, roles, and profile functions via a bunch of built-in WCF services
<i>System.Web.DynamicData.dll</i>	Provides classes behind the ASP.NET Dynamic Data framework
<i>System.Web.Entity.dll</i>	Contains the code for the <i>EntityDataSource</i> component that supports Entity Framework
<i>System.Web.Extensions.dll</i>	Contains the code for AJAX extensions to ASP.NET
<i>System.Web.Services.dll</i>	Contains the core code that makes Web services run
<i>System.Xml.dll</i>	Implements the .NET Framework XML features
<i>System.Xml.Linq.dll</i>	Contains the code for the LINQ-to-XML parser

Note that you can modify, extend, or restrict the list of default assemblies by editing the global settings in the root *web.config* file under

```
%Windows%\Microsoft.NET\Framework\v4.0.30319\Config
```

If you do so, changes will apply to all ASP.NET applications run on that Web server. Alternatively, you can modify the assembly list on a per-application basis by editing the `<assemblies>` section under `<compilation>` in the application's specific *web.config* file. Note also that the `<compilation>` section should be used only for global assembly cache (GAC) resident assemblies, not for the private assemblies that you deploy to the *Bin* folder.

By default, the `<compilation>` section in the root *web.config* file contains the following entry:

```
<add assembly="*" />
```

It means that any assembly found in the binary path of the application should be treated as if it were registered through the `@Assembly` directive. To prevent all assemblies found in the *Bin* directory from being linked to the page, remove the entry from the root configuration file. To link a needed assembly to the page, use the following syntax:

```
<%@ Assembly Name="AssemblyName" %>
<%@ Assembly Src="assembly_code.cs" %>
```

The `@Assembly` directive supports two mutually exclusive attributes: `Name` and `Src`. `Name` indicates the name of the assembly to link to the page. The name cannot include the path or the extension. `Src` indicates the path to a source file to dynamically compile and link against the page. The `@Assembly` directive can appear multiple times in the body of the page. In fact, you need a new directive for each assembly to link. `Name` and `Src` cannot be used in the same `@Assembly` directive, but multiple directives defined in the same page can use either.



Note In terms of performance, the difference between `Name` and `Src` is minimal, although `Name` points to an existing and ready-to-load assembly. The source file referenced by `Src` is compiled only the first time it is requested. The ASP.NET runtime maps a source file with a dynamically compiled assembly and keeps using the compiled code until the original file undergoes changes. This means that after the first application-level call, the impact on the page performance is identical whether you use `Name` or `Src`.

Any assemblies you register through the `@Assembly` directive are used by the compiler at compile time, which allows for early binding. After the compilation of the requested ASP.NET file is complete, the assembly is loaded into the application domain, thus allowing late binding. In the end, any assemblies listed through the directive (implicitly through the root configuration or explicitly through the application configuration) is loaded into the AppDomain and referenced on demand.



Important Removing an assembly from the Visual Studio project doesn't help much to keep the AppDomain lean and mean. To ensure you load all the assemblies you want and only the ones you want, you should insert the following code in your configuration file:

```
<assemblies>
  <clear />
  <add assembly="..." />
  ...
  <add assembly="*" />
</assemblies>
```

The `<clear />` tag removes all default configurations; the subsequent tags add just the assemblies your application needs. As you can verify for yourself, the default list will likely load assemblies you don't need.

In debug mode, you can track the list of assemblies actually loaded in the AppDomain for the site using the following code:

```
var assemblies1 = Assembly.GetExecutingAssembly().GetReferencedAssemblies();
var assemblies2 = AppDomain.CurrentDomain.GetAssemblies();
```

The size of the two arrays can vary quite a bit. The former counts just the dynamically referenced assemblies at the current stage of execution. The latter counts the number of assemblies physically loaded in the AppDomain (which can't be unloaded unless you recycle the application).

The *@Import* Directive

The *@Import* directive links the specified namespace to the page so that all the types defined can be accessed from the page without specifying the fully qualified name. For example, to create a new instance of the ADO.NET *DataSet* class, you either import the *System.Data* namespace or specify the fully qualified class name whenever you need it, as in the following code:

```
System.Data.DataSet ds = new System.Data.DataSet();
```

After you've imported the *System.Data* namespace into the page, you can use more natural coding, as shown here:

```
DataSet ds = new DataSet();
```

The syntax of the *@Import* directive is rather self-explanatory:

```
<%@ Import namespace="value" %>
```

@Import can be used as many times as needed in the body of the page. The *@Import* directive is the ASP.NET counterpart of the C# *using* statement and the Visual Basic .NET *Imports* statement. Looking back at unmanaged C/C++, we could say the directive plays a role nearly identical to the *#include* directive. For example, to be able to connect to a Microsoft SQL Server database and grab some disconnected data, you need to import the following two namespaces:

```
<%@ Import namespace="System.Data" %>  
<%@ Import namespace="System.Data.SqlClient" %>
```

You need the *System.Data* namespace to work with the *DataSet* and *DataTable* classes, and you need the *System.Data.SqlClient* namespace to prepare and issue the command. In this case, you don't need to link against additional assemblies because the *System.Data.dll* assembly is linked by default.



Note *@Import* helps the compiler only to resolve class names; it doesn't automatically link required assemblies. Using the *@Import* directive allows you to use shorter class names, but as long as the assembly that contains the class code is not properly referenced, the compiler will generate a type error. In this case, using the fully qualified class name is of no help because the compiler lacks the type definition. You might have noticed that, more often than not, assembly and namespace names coincide. The latest version of Visual Studio (as well as some commercial products such as JetBrains ReSharper) is able to detect when you lack a reference and offers to import the namespace and reference the assembly with a single click. This is pure tooling activity—namespaces and assemblies are totally different beasts.

The *@Implements* Directive

The directive indicates that the current page implements the specified .NET Framework interface. An interface is a set of signatures for a logically related group of functions. An interface is a sort of contract that shows the component's commitment to expose that group of functions. Unlike abstract classes, an interface doesn't provide code or executable functionality. When you implement an interface in an ASP.NET page, you declare any required methods and properties within the `<script>` section. The syntax of the *@Implements* directive is as follows:

```
<%@ Implements interface="InterfaceName" %>
```

The *@Implements* directive can appear multiple times in the page if the page has to implement multiple interfaces. Note that if you decide to put all the page logic in a separate class file, you can't use the directive to implement interfaces. Instead, you implement the interface in the code-behind class.

The *@Reference* Directive

The *@Reference* directive is used to establish a dynamic link between the current page and the specified page or user control. This feature has significant implications for the way you set up cross-page communication. It also lets you create strongly typed instances of user controls. Let's review the syntax.

The directive can appear multiple times in the page. The directive features two mutually exclusive attributes: *Page* and *Control*. Both attributes are expected to contain a path to a source file:

```
<%@ Reference page="source_page" %>  
<%@ Reference control="source_user_control" %>
```

The *Page* attribute points to an *.aspx* source file, whereas the *Control* attribute contains the path of an *.ascx* user control. In both cases, the referenced source file will be dynamically compiled into an assembly, thus making the classes defined in the source programmatically available to the referencing page. When running, an ASP.NET page is an instance of a .NET Framework class with a specific interface made of methods and properties. When the referencing page executes, a referenced page becomes a class that represents the *.aspx* source file and can be instantiated and programmed at will. For the directive to work, the referenced page must belong to the same domain as the calling page. Cross-site calls are not allowed, and both the *Page* and *Control* attributes expect to receive a relative virtual path.



Note Cross-page posting can be considered as an alternate approach to using the *@Reference* directive. Cross-page posting is an ASP.NET feature through which you force an ASP.NET button control to post the content of its parent form to a given target page. I'll demonstrate cross-page posting in Chapter 9, "Input Forms."

The *Page* Class

In the .NET Framework, the *Page* class provides the basic behavior for all objects that an ASP.NET application builds by starting from *.aspx* files. Defined in the *System.Web.UI* namespace, the class derives from *TemplateControl* and implements the *IHttpHandler* interface:

```
public class Page : TemplateControl, IHttpHandler
{
    ...
}
```

In particular, *TemplateControl* is the abstract class that provides both ASP.NET pages and user controls with a base set of functionality. At the upper level of the hierarchy, you find the *Control* class. It defines the properties, methods, and events shared by all ASP.NET server-side elements—pages, controls, and user controls.

Derived from a class—*TemplateControl*—that implements *INamingContainer*, the *Page* class also serves as the naming container for all its constituent controls. In the .NET Framework, the naming container for a control is the first parent control that implements the *INamingContainer* interface. For any class that implements the naming container interface, ASP.NET creates a new virtual namespace in which all child controls are guaranteed to have unique names in the overall tree of controls. (This is a very important feature for iterative data-bound controls, such as *DataGrid*, and for user controls.)

The *Page* class also implements the methods of the *IHttpHandler* interface, thus qualifying it as the handler of a particular type of HTTP requests—those for *.aspx* files. The key element of the *IHttpHandler* interface is the *ProcessRequest* method, which is the method the ASP.NET runtime calls to start the page processing that will actually serve the request.



Note *INamingContainer* is a marker interface that has no methods. Its presence alone, though, forces the ASP.NET runtime to create an additional namespace for naming the child controls of the page (or the control) that implements it. The *Page* class is the naming container of all the page's controls, with the clear exception of those controls that implement the *INamingContainer* interface themselves or are children of controls that implement the interface.

Properties of the *Page* Class

The properties of the *Page* class can be classified in three distinct groups: intrinsic objects, worker properties, and page-specific properties. The tables in the following sections enumerate and describe them.

Intrinsic Objects

Table 5-8 lists all properties that return a helper object that is intrinsic to the page. In other words, objects listed here are all essential parts of the infrastructure that allows for the page execution.

TABLE 5-8 ASP.NET Intrinsic Objects in the *Page* Class

Property	Description
<i>Application</i>	Instance of the <i>HttpApplicationState</i> class; represents the state of the application. It is functionally equivalent to the ASP intrinsic <i>Application</i> object.
<i>Cache</i>	Instance of the <i>Cache</i> class; implements the cache for an ASP.NET application. More efficient and powerful than <i>Application</i> , it supports item priority and expiration.
<i>Profile</i>	Instance of the <i>ProfileCommon</i> class; represents the user-specific set of data associated with the request.
<i>Request</i>	Instance of the <i>HttpRequest</i> class; represents the current HTTP request.
<i>Response</i>	Instance of the <i>HttpResponse</i> class; sends HTTP response data to the client.
<i>RouteData</i>	Instance of the <i>RouteData</i> class; groups information about the selected route (if any) and its values and tokens. (Routing in Web Forms is covered in Chapter 4, "xxx.") <i>The object is supported only in ASP.NET 4.</i>
<i>Server</i>	Instance of the <i>HttpServerUtility</i> class; provides helper methods for processing Web requests.
<i>Session</i>	Instance of the <i>HttpSessionState</i> class; manages user-specific data.
<i>Trace</i>	Instance of the <i>TraceContext</i> class; performs tracing on the page.
<i>User</i>	An <i>IPrincipal</i> object that represents the user making the request.

I'll cover *Request*, *Response*, and *Server* in Chapter 16; *Application* and *Session* are covered in Chapter 17; *Cache* will be the subject of Chapter 19. Finally, *User* and security will be the subject of Chapter 19, "ASP.NET Security."

Worker Properties

Table 5-9 details page properties that are both informative and provide the foundation for functional capabilities. You can hardly write code in the page without most of these properties.

TABLE 5-9 Worker Properties of the *Page* Class

Property	Description
<i>AutoPostBackControl</i>	Gets a reference to the control within the page that caused the postback event.
<i>ClientScript</i>	Gets a <i>ClientScriptManager</i> object that contains the client script used on the page.
<i>Controls</i>	Returns the collection of all the child controls contained in the current page.
<i>ErrorPage</i>	Gets or sets the error page to which the requesting browser is redirected in case of an unhandled page exception.
<i>Form</i>	Returns the current <i>HtmlForm</i> object for the page.
<i>Header</i>	Returns a reference to the object that represents the page's header. The object implements <i>IPageHeader</i> .
<i>IsAsync</i>	Indicates whether the page is being invoked through an asynchronous handler.
<i>IsCallback</i>	Indicates whether the page is being loaded in response to a client script callback.
<i>IsCrossPagePostBack</i>	Indicates whether the page is being loaded in response to a postback made from within another page.
<i>IsPostBack</i>	Indicates whether the page is being loaded in response to a client postback or whether it is being loaded for the first time.
<i>IsValid</i>	Indicates whether page validation succeeded.
<i>Master</i>	Instance of the <i>MasterPage</i> class; represents the master page that determines the appearance of the current page.
<i>MasterPageFile</i>	Gets and sets the master file for the current page.
<i>NamingContainer</i>	Returns <i>null</i> .
<i>Page</i>	Returns the current <i>Page</i> object.
<i>PageAdapter</i>	Returns the adapter object for the current <i>Page</i> object.
<i>Parent</i>	Returns <i>null</i> .
<i>PreviousPage</i>	Returns the reference to the caller page in case of a cross-page postback.
<i>TemplateSourceDirectory</i>	Gets the virtual directory of the page.
<i>Validators</i>	Returns the collection of all validation controls contained in the page.
<i>ViewStateUserKey</i>	String property that represents a user-specific identifier used to hash the view-state contents. This trick is a line of defense against one-click attacks.

In the context of an ASP.NET application, the *Page* object is the root of the hierarchy. For this reason, inherited properties such as *NamingContainer* and *Parent* always return *null*. The *Page* property, on the other hand, returns an instance of the same object (*this* in C# and *Me* in Visual Basic .NET).

The *ViewStateUserKey* property deserves a special mention. A common use for the user key is to stuff user-specific information that is then used to hash the contents of the view state

along with other information. A typical value for the *ViewStateUserKey* property is the name of the authenticated user or the user's session ID. This contrivance reinforces the security level for the view state information and further lowers the likelihood of attacks. If you employ a user-specific key, an attacker can't construct a valid view state for your user account unless the attacker can also authenticate as you. With this configuration, you have another barrier against one-click attacks. This technique, though, might not be effective for Web sites that allow anonymous access, unless you have some other unique tracking device running.

Note that if you plan to set the *ViewStateUserKey* property, you must do that during the *Page_Init* event. If you attempt to do it later (for example, when *Page_Load* fires), an exception will be thrown.

Context Properties

Table 5-10 lists properties that represent visual and nonvisual attributes of the page, such as the URL's query string, the client target, the title, and the applied style sheet.

TABLE 5-10 Page-Specific Properties of the *Page* Class

Property	Description
<i>ClientID</i>	Always returns the empty string.
<i>ClientIDMode</i>	Determines the algorithm to use to generate the ID of HTML elements being output as part of a control's markup. <i>This property requires ASP.NET 4.</i>
<i>ClientQueryString</i>	Gets the query string portion of the requested URL.
<i>ClientTarget</i>	Set to the empty string by default; allows you to specify the type of browser the HTML should comply with. Setting this property disables automatic detection of browser capabilities.
<i>EnableViewState</i>	Indicates whether the page has to manage view-state data. You can also enable or disable the view-state feature through the <i>EnableViewState</i> attribute of the <i>@Page</i> directive.
<i>EnableViewStateMac</i>	Indicates whether ASP.NET should calculate a machine-specific authentication code and append it to the page view state.
<i>EnableTheming</i>	Indicates whether the page supports themes.
<i>ID</i>	Always returns the empty string.
<i>MetaDescription</i>	Gets and sets the content of the <i>description</i> meta tag. <i>This property requires ASP.NET 4.</i>
<i>MetaKeywords</i>	Gets and sets the content of the <i>keywords</i> meta tag. <i>This property requires ASP.NET 4.</i>
<i>MaintainScrollPositionOnPostBack</i>	Indicates whether to return the user to the same position in the client browser after postback.
<i>SmartNavigation</i>	Indicates whether smart navigation is enabled. Smart navigation exploits a bunch of browser-specific capabilities to enhance the user's experience with the page.

Property	Description
<i>StyleSheetTheme</i>	Gets or sets the name of the style sheet applied to this page.
<i>Theme</i>	Gets and sets the theme for the page. Note that themes can be programmatically set only in the <i>PreInit</i> event.
<i>Title</i>	Gets or sets the title for the page.
<i>TraceEnabled</i>	Toggles page tracing on and off.
<i>TraceModeValue</i>	Gets or sets the trace mode.
<i>UniqueID</i>	Always returns the empty string.
<i>ViewStateEncryptionMode</i>	Indicates if and how the view state should be encrypted.
<i>ViewStateMode</i>	Enables the view state for an individual control even if the view state is disabled for the page. <i>This property requires ASP.NET 4.</i>
<i>Visible</i>	Indicates whether ASP.NET has to render the page. If you set <i>Visible</i> to <i>false</i> , ASP.NET doesn't generate any HTML code for the page. When <i>Visible</i> is <i>false</i> , only the text explicitly written using <i>Response.Write</i> hits the client.

The three ID properties (*ID*, *ClientID*, and *UniqueID*) always return the empty string from a *Page* object. They make sense only for server controls.

Methods of the *Page* Class

The whole range of *Page* methods can be classified in a few categories based on the tasks each method accomplishes. A few methods are involved with the generation of the markup for the page; others are helper methods to build the page and manage the constituent controls. Finally, a third group collects all the methods related to client-side scripting.

Rendering Methods

Table 5-11 details the methods that are directly or indirectly involved with the generation of the markup code.

TABLE 5-11 Methods for Markup Generation

Method	Description
<i>DataBind</i>	Binds all the data-bound controls contained in the page to their data sources. The <i>DataBind</i> method doesn't generate code itself but prepares the ground for the forthcoming rendering.
<i>RenderControl</i>	Outputs the HTML text for the page, including tracing information if tracing is enabled.
<i>VerifyRenderingInServerForm</i>	Controls call this method when they render to ensure that they are included in the body of a server form. The method does not return a value, but it throws an exception in case of error.

In an ASP.NET page, no control can be placed outside a `<form>` tag with the `runat` attribute set to `server`. The `VerifyRenderingInServerForm` method is used by Web and HTML controls to ensure that they are rendered correctly. In theory, custom controls should call this method during the rendering phase. In many situations, the custom control embeds or derives an existing Web or HTML control that will make the check itself.

Not directly exposed by the `Page` class, but strictly related to it, is the `GetWebResourceUrl` method on the `ClientScriptManager` class. (You get a reference to the current client script manager through the `ClientScript` property on `Page`.) When you develop a custom control, you often need to embed static resources such as images or client script files. You can make these files be separate downloads; however, even though it's effective, the solution looks poor and inelegant. Visual Studio allows you to embed resources in the control assembly, but how would you retrieve these resources programmatically and bind them to the control? For example, to bind an assembly-stored image to an `` tag, you need a URL for the image. The `GetWebResourceUrl` method returns a URL for the specified resource. The URL refers to a new Web Resource service (`webresource.axd`) that retrieves and returns the requested resource from an assembly.

```
// Bind the <IMG> tag to the given GIF image in the control's assembly
img.ImageUrl = Page.GetWebResourceUrl(typeof(TheControl), GifName));
```

`GetWebResourceUrl` requires a `Type` object, which will be used to locate the assembly that contains the resource. The assembly is identified with the assembly that contains the definition of the specified type in the current AppDomain. If you're writing a custom control, the type will likely be the control's type. As its second argument, the `GetWebResourceUrl` method requires the name of the embedded resource. The returned URL takes the following form:

```
WebResource.axd?a=assembly&r=resourceName&t=timestamp
```

The timestamp value is the current timestamp of the assembly, and it is added to make the browser download resources again if the assembly is modified.

Controls-Related Methods

Table 5-12 details a bunch of helper methods on the `Page` class architected to let you manage and validate child controls and resolve URLs.

TABLE 5-12 Helper Methods of the `Page` Object

Method	Description
<code>DesignerInitialize</code>	Initializes the instance of the <code>Page</code> class at design time, when the page is being hosted by RAD designers such as Visual Studio.
<code>FindControl</code>	Takes a control's ID and searches for it in the page's naming container. The search doesn't dig out child controls that are naming containers themselves.

Method	Description
<i>GetTypeHashCode</i>	Retrieves the hash code generated by <i>ASP.xxx_aspx</i> page objects at run time. In the base <i>Page</i> class, the method implementation simply returns 0; significant numbers are returned by classes used for actual pages.
<i>GetValidators</i>	Returns a collection of control validators for a specified validation group.
<i>HasControls</i>	Determines whether the page contains any child controls.
<i>LoadControl</i>	Compiles and loads a user control from an <i>.ascx</i> file, and returns a <i>Control</i> object. If the user control supports caching, the object returned is <i>PartialCachingControl</i> .
<i>LoadTemplate</i>	Compiles and loads a user control from an <i>.ascx</i> file, and returns it wrapped in an instance of an internal class that implements the <i>ITemplate</i> interface. The internal class is named <i>SimpleTemplate</i> .
<i>MapPath</i>	Retrieves the physical, fully qualified path that an absolute or relative virtual path maps to.
<i>ParseControl</i>	Parses a well-formed input string, and returns an instance of the control that corresponds to the specified markup text. If the string contains more controls, only the first is taken into account. The <i>runat</i> attribute can be omitted. The method returns an object of type <i>Control</i> and must be cast to a more specific type.
<i>RegisterRequiresControlState</i>	Registers a control as one that requires control state.
<i>RegisterRequiresPostBack</i>	Registers the specified control to receive a postback handling notice, even if its ID doesn't match any ID in the collection of posted data. The control must implement the <i>IPostBackDataHandler</i> interface.
<i>RegisterRequiresRaiseEvent</i>	Registers the specified control to handle an incoming postback event. The control must implement the <i>IPostBackEventHandler</i> interface.
<i>RegisterViewStateHandler</i>	Mostly for internal use, the method sets an internal flag that causes the page view state to be persisted. If this method is not called in the prerendering phase, no view state will ever be written. Typically, only the <i>HtmlForm</i> server control for the page calls this method. There's no need to call it from within user applications.
<i>ResolveUrl</i>	Resolves a relative URL into an absolute URL based on the value of the <i>TemplateSourceDirectory</i> property.
<i>Validate</i>	Instructs any validation controls included in the page to validate their assigned information. If defined in the page, the method honors ASP.NET validation groups.

The methods *LoadControl* and *LoadTemplate* share a common code infrastructure but return different objects, as the following pseudocode shows:

```
public Control LoadControl(string virtualPath)
{
    Control ascx = GetCompiledUserControlType(virtualPath);
    ascx.InitializeAsUserControl();
    return ascx;
}
public ITemplate LoadTemplate(string virtualPath)
{
    Control ascx = GetCompiledUserControlType(virtualPath);
    return new SimpleTemplate(ascx);
}
```

Both methods differ from the *ParseControl* method in that the latter never causes compilation but simply parses the string and infers control information. The information is then used to create and initialize a new instance of the control class. As mentioned, the *runat* attribute is unnecessary in this context. In ASP.NET, the *runat* attribute is key, but in practice, it has no other role than marking the surrounding markup text for parsing and instantiation. It does not contain information useful to instantiate a control, and for this reason it can be omitted from the strings you pass directly to *ParseControl*.

Script-Related Methods

Table 5-13 enumerates all the methods in the *Page* class related to HTML and script code to be inserted in the client page.

TABLE 5-13 Script-Related Methods

Method	Description
<i>GetCallbackEventReference</i>	Obtains a reference to a client-side function that, when invoked, initiates a client callback to server-side events.
<i>GetPostBackClientEvent</i>	Calls into <i>GetCallbackEventReference</i> .
<i>GetPostBackClientHyperlink</i>	Appends <i>javascript:</i> to the beginning of the return string received from <i>GetPostBackEventReference</i> . For example: <i>javascript:__doPostBack('CtlID','')</i>
<i>GetPostBackEventReference</i>	Returns the prototype of the client-side script function that causes, when invoked, a postback. It takes a <i>Control</i> and an argument, and it returns a string like this: <i>__doPostBack('CtlID','')</i>
<i>IsClientScriptBlockRegistered</i>	Determines whether the specified client script is registered with the page. <i>It's marked as obsolete.</i>
<i>IsStartupScriptRegistered</i>	Determines whether the specified client startup script is registered with the page. <i>It's marked as obsolete.</i>

Method	Description
<i>RegisterArrayDeclaration</i>	Use this method to add an <i>ECMAScript</i> array to the client page. This method accepts the name of the array and a string that will be used verbatim as the body of the array. For example, if you call the method with arguments such as <i>theArray</i> and <code>"'a', 'b'"</code> , you get the following JavaScript code: <code>var theArray = new Array('a', 'b');</code> <i>It's marked as obsolete.</i>
<i>RegisterClientScriptBlock</i>	An ASP.NET page uses this method to emit client-side script blocks in the client page just after the opening tag of the HTML <code><form></code> element. <i>It's marked as obsolete.</i>
<i>RegisterHiddenField</i>	Use this method to automatically register a hidden field on the page. <i>It's marked as obsolete.</i>
<i>RegisterOnSubmitStatement</i>	Use this method to emit client script code that handles the client <i>OnSubmit</i> event. The script should be a JavaScript function call to client code registered elsewhere. <i>It's marked as obsolete.</i>
<i>RegisterStartupScript</i>	An ASP.NET page uses this method to emit client-side script blocks in the client page just before closing the HTML <code><form></code> element. <i>It's marked as obsolete.</i>
<i>SetFocus</i>	Sets the browser focus to the specified control.

As you can see, some methods in Table 5-13, which are defined and usable in ASP.NET 1.x, are marked as obsolete. In ASP.NET 4 applications, you should avoid calling them and resort to methods with the same name exposed out of the *ClientScript* property.

```
// Avoid this in ASP.NET 4
Page.RegisterArrayDeclaration(...);
```

```
// Use this in ASP.NET 4
Page.ClientScript.RegisterArrayDeclaration(...);
```

The *ClientScript* property returns an instance of the *ClientScriptManager* class and represents the central console for registering script code to be programmatically emitted within the page.

Methods listed in Table 5-13 let you emit JavaScript code in the client page. When you use any of these methods, you actually tell the page to insert that script code when the page is rendered. So when any of these methods execute, the script-related information is simply cached in internal structures and used later when the page object generates its HTML text.

Events of the *Page* Class

The *Page* class fires a few events that are notified during the page life cycle. As Table 5-14 shows, some events are orthogonal to the typical life cycle of a page (initialization, postback,

and rendering phases) and are fired as extra-page situations evolve. Let's briefly review the events and then attack the topic with an in-depth discussion of the page life cycle.

TABLE 5-14 Events a Page Can Fire

Event	Description
<i>AbortTransaction</i>	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction aborts
<i>CommitTransaction</i>	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction commits
<i>DataBinding</i>	Occurs when the <i>DataBind</i> method is called on the page to bind all the child controls to their respective data sources
<i>Disposed</i>	Occurs when the page is released from memory, which is the last stage of the page life cycle
<i>Error</i>	Occurs when an unhandled exception is thrown.
<i>Init</i>	Occurs when the page is initialized, which is the first step in the page life cycle
<i>InitComplete</i>	Occurs when all child controls and the page have been initialized
<i>Load</i>	Occurs when the page loads up, after being initialized
<i>LoadComplete</i>	Occurs when the loading of the page is completed and server events have been raised
<i>PreInit</i>	Occurs just before the initialization phase of the page begins
<i>PreLoad</i>	Occurs just before the loading phase of the page begins
<i>PreRender</i>	Occurs when the page is about to render
<i>PreRenderComplete</i>	Occurs just before the pre-rendering phase begins
<i>SaveStateComplete</i>	Occurs when the view state of the page has been saved to the persistence medium
<i>Unload</i>	Occurs when the page is unloaded from memory but not yet disposed of

The Eventing Model

When a page is requested, its class and the server controls it contains are responsible for executing the request and rendering HTML back to the client. The communication between the client and the server is stateless and disconnected because it's based on the HTTP protocol. Real-world applications, though, need some state to be maintained between successive calls made to the same page. With ASP, and with other server-side development platforms such as Java Server Pages and PHP, the programmer is entirely responsible for persisting the state. In contrast, ASP.NET provides a built-in infrastructure that saves and restores the state of a page in a transparent manner. In this way, and in spite of the underlying stateless protocol, the client experience appears to be that of a continuously executing process. It's just an illusion, though.

Introducing the View State

The illusion of continuity is created by the view state feature of ASP.NET pages and is based on some assumptions about how the page is designed and works. Also, server-side Web controls play a remarkable role. In brief, before rendering its contents to HTML, the page encodes and stuffs into a persistence medium (typically, a hidden field) all the state information that the page itself and its constituent controls want to save. When the page posts back, the state information is deserialized from the hidden field and used to initialize instances of the server controls declared in the page layout.

The view state is specific to each instance of the page because it is embedded in the HTML. The net effect of this is that controls are initialized with the same values they had the last time the view state was created—that is, the last time the page was rendered to the client. Furthermore, an additional step in the page life cycle merges the persisted state with any updates introduced by client-side actions. When the page executes after a postback, it finds a stateful and up-to-date context just as it is working over a continuous point-to-point connection.

Two basic assumptions are made. The first assumption is that the page always posts to itself and carries its state back and forth. The second assumption is that the server-side controls have to be declared with the *runat=server* attribute to spring to life when the page posts back.

The Single Form Model

ASP.NET pages are built to support exactly one server-side *<form>* tag. The form must include all the controls you want to interact with on the server. Both the form and the controls must be marked with the *runat* attribute; otherwise, they will be considered plain text to be output verbatim.

A server-side form is an instance of the *HtmlForm* class. The *HtmlForm* class does not expose any property equivalent to the *Action* property of the HTML *<form>* tag. The reason is that an ASP.NET page always posts to itself. Unlike the *Action* property, other common form properties such as *Method* and *Target* are fully supported.

Valid ASP.NET pages are also those that have no server-side forms and those that run HTML forms—a *<form>* tag without the *runat* attribute. In an ASP.NET page, you can also have both HTML and server forms. In no case, though, can you have more than one *<form>* tag with the *runat* attribute set to *server*. HTML forms work as usual and let you post to any page in the application. The drawback is that in this case no state will be automatically restored. In other words, the ASP.NET Web Forms model works only if you use exactly one server *<form>* element. We'll return to this topic in Chapter 9.

Asynchronous Pages

ASP.NET pages are served by an HTTP handler like an instance of the *Page* class. Each request takes up a thread in the ASP.NET thread pool and releases it only when the request completes. What if a frequently requested page starts an external and particularly lengthy task? The risk is that the ASP.NET process is idle but has no free threads in the pool to serve incoming requests for other pages. This happens mostly because HTTP handlers, including page classes, work synchronously. To alleviate this issue, ASP.NET has supported asynchronous handlers since version 1.0 through the *IHttpAsyncHandler* interface. Starting with ASP.NET 2.0, creating asynchronous pages was made easier thanks to specific support from the framework.

Two aspects characterize an asynchronous ASP.NET page: a tailor-made attribute on the *@Page* directive, and one or more tasks registered for asynchronous execution. The asynchronous task can be registered in either of two ways. You can define a *Begin/End* pair of asynchronous handlers for the *PreRenderComplete* event or create a *PageAsyncTask* object to represent an asynchronous task. This is generally done in the *Page_Load* event, but any time is fine provided that it happens before the *PreRender* event fires.

In both cases, the asynchronous task is started automatically when the page has progressed to a well-known point. Let's dig out more details.



Note An ASP.NET asynchronous page is still a class that derives from *Page*. There are no special base classes to inherit for building asynchronous pages.

The *Async* Attribute

The new *Async* attribute on the *@Page* directive accepts a Boolean value to enable or disable asynchronous processing. The default value is *false*.

```
<%@ Page Async="true" ... %>
```

The *Async* attribute is merely a message for the page parser. When used, the page parser implements the *IHttpAsyncHandler* interface in the dynamically generated class for the *.aspx* resource. The *Async* attribute enables the page to register asynchronous handlers for the *PreRenderComplete* event. No additional code is executed at run time as a result of the attribute.

Let's consider a request for a *TestAsync.aspx* page marked with the *Async* directive attribute. The dynamically created class, named *ASP.TestAsync.aspx*, is declared as follows:

```
public class TestAsync_aspx : TestAsync, IHttpHandler, IHttpAsyncHandler
{
    ...
}
```

TestAsync is the code file class and inherits from *Page* or a class that in turn inherits from *Page*. *IHttpAsyncHandler* is the canonical interface that has been used for serving resources asynchronously since ASP.NET 1.0.

The *AddOnPreRenderCompleteAsync* Method

The *AddOnPreRenderCompleteAsync* method adds an asynchronous event handler for the page's *PreRenderComplete* event. An asynchronous event handler consists of a *Begin/End* pair of event handler methods, as shown here:

```
AddOnPreRenderCompleteAsync (
    new BeginEventHandler(BeginTask),
    new EndEventHandler(EndTask)
);
```

The call can be simplified as follows:

```
AddOnPreRenderCompleteAsync(BeginTask, EndTask);
```

BeginEventHandler and *EndEventHandler* are delegates defined as follows:

```
IAAsyncResult BeginEventHandler(
    object sender,
    EventArgs e,
    AsyncCallback cb,
    object state)
void EndEventHandler(
    IAAsyncResult ar)
```

In the code file, you place a call to *AddOnPreRenderCompleteAsync* as soon as you can, and always earlier than the *PreRender* event can occur. A good place is usually the *Page_Load* event. Next, you define the two asynchronous event handlers.

The *Begin* handler is responsible for starting any operation you fear can block the underlying thread for too long. The handler is expected to return an *IAAsyncResult* object to describe the state of the asynchronous task. When the lengthy task has completed, the *End* handler finalizes the original request and updates the page's user interface and controls. Note that you don't necessarily have to create your own object that implements the *IAAsyncResult* interface. In most cases, in fact, to start lengthy operations you just use built-in classes that already implement the asynchronous pattern and provide *IAAsyncResult* ready-made objects.

The page progresses up to entering the *PreRenderComplete* stage. You have a pair of asynchronous event handlers defined here. The page executes the *Begin* event, starts the lengthy operation, and is then suspended until the operation terminates. When the work has been completed, the HTTP runtime processes the request again. This time, though, the request processing begins at a later stage than usual. In particular, it begins exactly where it left off—that is, from the *PreRenderComplete* stage. The *End* event executes, and the page finally

completes the rest of its life cycle, including view-state storage, markup generation, and unloading.



Important The *Begin* and *End* event handlers are called at different times and generally on different pooled threads. In between the two methods calls, the lengthy operation takes place. From the ASP.NET runtime perspective, the *Begin* and *End* events are similar to serving distinct requests for the same page. It's as if an asynchronous request is split in two distinct steps: a *Begin* step and *End* step. Each request is always served by a pooled thread. Typically, the *Begin* and *End* steps are served by threads picked up from the ASP.NET thread pool. The lengthy operation, instead, is not managed by ASP.NET directly and doesn't involve any of the pooled threads. The lengthy operation is typically served by a thread selected from the operating system completion thread pool.

The Significance of *PreRenderComplete*

So an asynchronous page executes up until the *PreRenderComplete* stage is reached and then blocks while waiting for the requested operation to complete asynchronously. When the operation is finally accomplished, the page execution resumes from the *PreRenderComplete* stage. A good question to ask would be the following: "Why *PreRenderComplete*?" What makes *PreRenderComplete* such a special event?

By design, in ASP.NET there's a single unwind point for asynchronous operations (also familiarly known as the *async point*). This point is located between the *PreRender* and *PreRenderComplete* events. When the page receives the *PreRender* event, the *async point* hasn't been reached yet. When the page receives *PreRenderComplete*, the *async point* has passed.

Building a Sample Asynchronous Page

Let's roll a first asynchronous test page to download and process some RSS feeds. The page markup is quite simple indeed:

```
<%@ Page Async="true" Language="C#" AutoEventWireup="true"
    CodeFile="TestAsync.aspx.cs" Inherits="TestAsync" %>
<html>
<body>
    <form id="form1" runat="server">
        <% = RssData %>
    </form>
</body>
</html>
```

The code file is shown next, and it attempts to download the RSS feed from my personal blog:

```
public partial class TestAsync : System.Web.UI.Page
{
    const String RSSFEED = "http://weblogs.asp.net/despos/rss.aspx";
    private WebRequest req;

    public String RssData { get; set; }

    void Page_Load (Object sender, EventArgs e)
    {
        AddOnPreRenderCompleteAsync(BeginTask, EndTask);
    }

    IAsyncResult BeginTask(Object sender,
                            EventArgs e, AsyncCallback cb, Object state)
    {
        // Trace
        Trace.Warn("Begin async: Thread=" +
                  Thread.CurrentThread.ManagedThreadId.ToString());

        // Prepare to make a Web request for the RSS feed
        req = WebRequest.Create(RSSFEED);

        // Begin the operation and return an IAsyncResult object
        return req.BeginGetResponse(cb, state);
    }

    void EndTask(IAsyncResult ar)
    {
        // This code will be called on a(nother) pooled thread

        using (var response = req.EndGetResponse(ar))
        {
            String text;
            using (var reader = new StreamReader(response.GetResponseStream()))
            {
                text = reader.ReadToEnd();
            }

            // Process the RSS data
            rssData = ProcessFeed(text);
        }

        // Trace
        Trace.Warn("End async: Thread=" +
                  Thread.CurrentThread.ManagedThreadId.ToString());

        // The page is updated using an ASP-style code block in the ASPX
        // source that displays the contents of the rssData variable
    }
}
```

```
String ProcessFeed(String feed)
{
    // Build the page output from the XML input
    ...
}
}
```

As you can see, such an asynchronous page differs from a standard one only for the aforementioned elements—the *Async* directive attribute and the pair of asynchronous event handlers. Figure 5-4 shows the sample page in action.

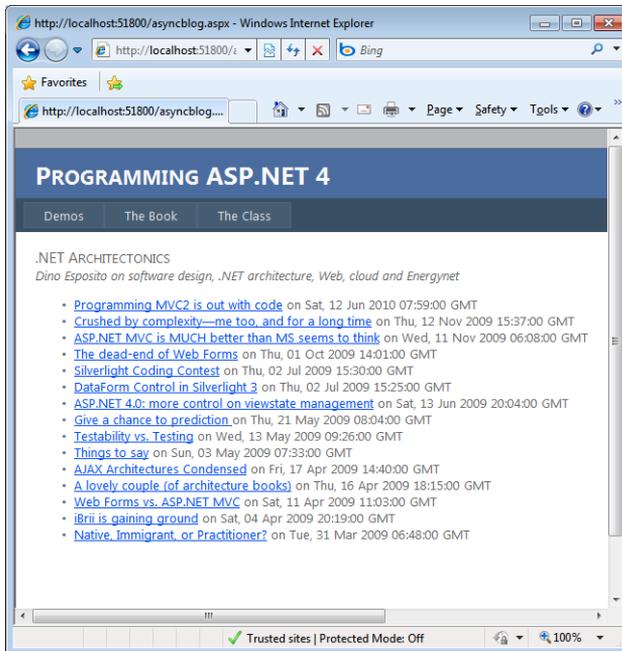


FIGURE 5-4 A sample asynchronous page downloading links from a blog.

It would also be interesting to take a look at the messages traced by the page. Figure 5-5 provides visual clues of it. The *Begin* and *End* stages are served by different threads and take place at different times.

Note the time elapsed between the *Exit BeginTask* and *Enter EndTask* stages. It is much longer than intervals between any other two consecutive operations. It's in that interval that the lengthy operation—in this case, downloading and processing the RSS feed—took place. The interval also includes the time spent to pick up another thread from the pool to serve the second part of the original request.

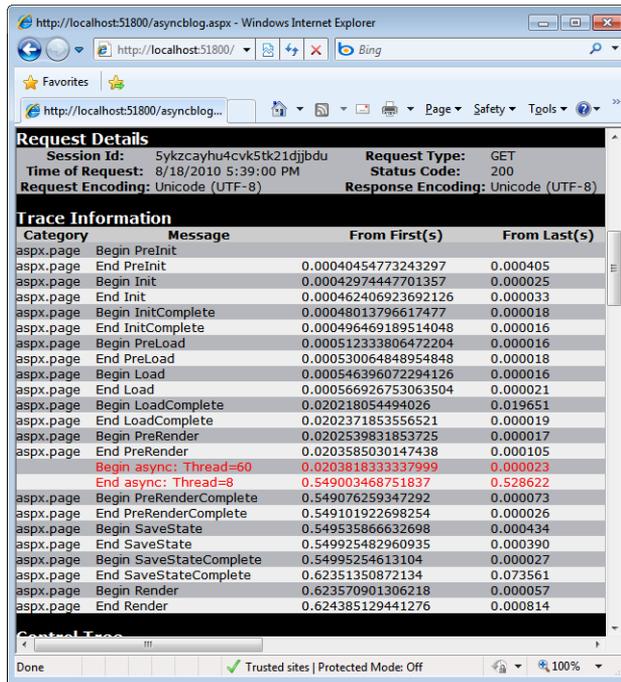


FIGURE 5-5 The traced request details clearly show the two steps needed to process a request asynchronously.

The *RegisterAsyncTask* Method

The *AddOnPreRenderCompleteAsync* method is not the only tool you have to register an asynchronous task. The *RegisterAsyncTask* method is, in most cases, an even better solution. *RegisterAsyncTask* is a *void* method and accepts a *PageAsyncTask* object. As the name suggests, the *PageAsyncTask* class represents a task to execute asynchronously.

The following code shows how to rework the sample page that reads some RSS feed and make it use the *RegisterAsyncTask* method:

```
void Page_Load (object sender, EventArgs e)
{
    PageAsyncTask task = new PageAsyncTask(
        new BeginEventHandler(BeginTask),
        new EndEventHandler(EndTask),
        null,
        null);
    RegisterAsyncTask(task);
}
```

The constructor accepts up to five parameters, as shown in the following code:

```
public PageAsyncTask(
    BeginEventHandler beginHandler,
    EndEventHandler endHandler,
    EndEventHandler timeoutHandler,
    object state,
    bool executeInParallel)
```

The *beginHandler* and *endHandler* parameters have the same prototype as the corresponding handlers you use for the *AddOnPreRenderCompleteAsync* method. Compared to the *AddOnPreRenderCompleteAsync* method, *PageAsyncTask* lets you specify a timeout function and an optional flag to enable multiple registered tasks to execute in parallel.

The timeout delegate indicates the method that will get called if the task is not completed within the asynchronous timeout interval. By default, an asynchronous task times out if it's not completed within 45 seconds. You can indicate a different timeout in either the configuration file or the *@Page* directive. Here's what you need if you opt for the *web.config* file:

```
<system.web>
  <pages asyncTimeout="30" />
</system.web>
```

The *@Page* directive contains an integer *AsyncTimeout* attribute that you set to the desired number of seconds.

Just as with the *AddOnPreRenderCompleteAsync* method, you can pass some state to the delegates performing the task. The *state* parameter can be any object.

The execution of all tasks registered is automatically started by the *Page* class code just before the async point is reached. However, by placing a call to the *ExecuteRegisteredAsyncTasks* method on the *Page* class, you can take control of this aspect.

Choosing the Right Approach

When should you use *AddOnPreRenderCompleteAsync*, and when is *RegisterAsyncTask* a better option? Functionally speaking, the two approaches are nearly identical. In both cases, the execution of the request is split in two parts: before and after the async point. So where's the difference?

The first difference is logical. *RegisterAsyncTask* is an API designed to run tasks asynchronously from within a page—and not just asynchronous pages with *Async=true*. *AddOnPreRenderCompleteAsync* is an API specifically designed for asynchronous pages. That said, a couple of further differences exist.

One is that *RegisterAsyncTask* executes the *End* handler on a thread with a richer context than *AddOnPreRenderCompleteAsync*. The thread context includes impersonation and

HTTP context information that is missing in the thread serving the *End* handler of a classic asynchronous page. In addition, *RegisterAsyncTask* allows you to set a timeout to ensure that any task doesn't run for more than a given number of seconds.

The other difference is that *RegisterAsyncTask* makes the implementation of multiple calls to remote sources significantly easier. You can have parallel execution by simply setting a Boolean flag, and you don't need to create and manage your own *AsyncResult* object.

The bottom line is that you can use either approach for a single task, but you should opt for *RegisterAsyncTask* when you have multiple tasks to execute simultaneously.

Async-Compliant Operations

Which required operations force, or at least strongly suggest, the adoption of an asynchronous page? Any operation can be roughly labeled in either of two ways: CPU bound or I/O bound. *CPU bound* indicates an operation whose completion time is mostly determined by the speed of the processor and amount of available memory. *I/O bound* indicates the opposite situation, where the CPU mostly waits for other devices to terminate.

The need for asynchronous processing arises when an excessive amount of time is spent getting data in and out of the computer in relation to the time spent processing it. In such situations, the CPU is idle or underused and spends most of its time waiting for something to happen. In particular, I/O-bound operations in the context of ASP.NET applications are even more harmful because serving threads are blocked too, and the pool of serving threads is a finite and critical resource. You get real performance advantages if you use the asynchronous model on I/O-bound operations.

Typical examples of I/O-bound operations are all operations that require access to some sort of remote resource or interaction with external hardware devices. Operations on non-local databases and non-local Web service calls are the most common I/O-bound operations for which you should seriously consider building asynchronous pages.



Important Asynchronous operations exist to speed up lengthy operations, but the benefits they provide are entirely enjoyed on the server side. There's no benefit for the end user in adopting asynchronous solutions. The "time to first byte" doesn't change for the user in a synchronous or asynchronous scenario. Using AJAX solutions would give you at least the means to (easily) display temporary messages to provide information about the progress. However, if it's not coded asynchronously on the server, any lengthy operation that goes via AJAX is more harmful for the system than a *slow-but-asynchronous* classic Web Forms page.

The Page Life Cycle

A page instance is created on every request from the client, and its execution causes itself and its contained controls to iterate through their life-cycle stages. Page execution begins when the HTTP runtime invokes *ProcessRequest*, which kicks off the page and control life cycles. The life cycle consists of a sequence of stages and steps. Some of these stages can be controlled through user-code events; some require a method override. Some other stages—or more exactly, substages—are just not public, are out of the developer's control, and are mentioned here mostly for completeness.

The page life cycle is articulated in three main stages: setup, postback, and finalization. Each stage might have one or more substages and is composed of one or more steps and points where events are raised. The life cycle as described here includes all possible paths. Note that there are modifications to the process depending upon cross-page posts, script callbacks, and postbacks.

Page Setup

When the HTTP runtime instantiates the page class to serve the current request, the page constructor builds a tree of controls. The tree of controls ties into the actual class that the page parser created after looking at the ASPX source. Note that when the request processing begins, all child controls and page intrinsics—such as HTTP context, request objects, and response objects—are set.

The very first step in the page lifetime is determining why the run time is processing the page request. There are various possible reasons: a normal request, postback, cross-page postback, or callback. The page object configures its internal state based on the actual reason, and it prepares the collection of posted values (if any) based on the method of the request—either *GET* or *POST*. After this first step, the page is ready to fire events to the user code.

The *PreInit* Event

This event is the entry point in the page life cycle. When the event fires, no master page or theme has been associated with the page as yet. Furthermore, the page scroll position has been restored, posted data is available, and all page controls have been instantiated and default to the properties values defined in the ASPX source. (Note that at this time controls have no ID, unless it is explicitly set in the *.aspx* source.) Changing the master page or the theme programmatically is possible only at this time. This event is available only on the page. *IsCallback*, *IsCrossPagePostback*, and *IsPostBack* are set at this time.

The *Init* Event

The master page, if one exists, and the theme have been set and can't be changed anymore. The page processor—that is, the *ProcessRequest* method on the *Page* class—proceeds and iterates over all child controls to give them a chance to initialize their state in a context-sensitive way. All child controls have their *OnInit* method invoked recursively. For each control in the control collection, the naming container and a specific ID are set, if not assigned in the source.

The *Init* event reaches child controls first and the page later. At this stage, the page and controls typically begin loading some parts of their state. At this time, the view state is not restored yet.

The *InitComplete* Event

Introduced with ASP.NET 2.0, this page-only event signals the end of the initialization substage. For a page, only one operation takes place in between the *Init* and *InitComplete* events: tracking of view-state changes is turned on. Tracking view state is the operation that ultimately enables controls to *really* persist in the storage medium any values that are programmatically added to the *ViewState* collection. Simply put, for controls not tracking their view state, any values added to their *ViewState* are lost across postbacks.

All controls turn on view-state tracking immediately after raising their *Init* event, and the page is no exception. (After all, isn't the page just a control?)



Important In light of the previous statement, note that any value written to the *ViewState* collection before *InitComplete* won't be available on the next postback.

View-State Restoration

If the page is being processed because of a postback—that is, if the *IsPostBack* property is *true*—the contents of the `__VIEWSTATE` hidden field is restored. The `__VIEWSTATE` hidden field is where the view state of all controls is persisted at the end of a request. The overall view state of the page is a sort of call context and contains the state of each constituent control the last time the page was served to the browser.

At this stage, each control is given a chance to update its current state to make it identical to what it was on last request. There's no event to wire up to handle the view-state restoration. If something needs be customized here, you have to resort to overriding the *LoadViewState* method, defined as protected and virtual on the *Control* class.

Processing Posted Data

All the client data packed in the HTTP request—that is, the contents of all input fields defined with the `<form>` tag—are processed at this time. Posted data usually takes the following form:

```
TextBox1=text&DropDownList1=selectedItem&Button1=Submit
```

It's an &-separated string of name/value pairs. These values are loaded into an internal-use collection. The page processor attempts to find a match between names in the posted collection and ID of controls in the page. Whenever a match is found, the processor checks whether the server control implements the *IPostBackDataHandler* interface. If it does, the methods of the interface are invoked to give the control a chance to refresh its state in light of the posted data. In particular, the page processor invokes the *LoadPostData* method on the interface. If the method returns *true*—that is, the state has been updated—the control is added to a separate collection to receive further attention later.

If a posted name doesn't match any server controls, it is left over and temporarily parked in a separate collection, ready for a second try later.



Note As mentioned, during the processing of posted data, posted names are matched against the ID of controls in the page. Which ID? Is it the *ClientID* property, or rather, is it the *UniqueID* property? Posted names are matched against the unique ID of page controls. Client IDs are irrelevant in this instance because they are not posted back to the server.

The *PreLoad* Event

The *PreLoad* event merely indicates that the page has terminated the system-level initialization phase and is going to enter the phase that gives user code in the page a chance to further configure the page for execution and rendering. This event is raised only for pages.

The *Load* Event

The *Load* event is raised for the page first and then recursively for all child controls. At this time, controls in the page tree are created and their state fully reflects both the previous state and any data posted from the client. The page is ready to execute any initialization code related to the logic and behavior of the page. At this time, access to control properties and view state is absolutely safe.

Handling Dynamically Created Controls

When all controls in the page have been given a chance to complete their initialization before display, the page processor makes a second try on posted values that haven't been matched to existing controls. The behavior described earlier in the "Processing Posted Data"

section is repeated on the name/value pairs that were left over previously. This apparently weird approach addresses a specific scenario—the use of dynamically created controls.

Imagine adding a control to the page tree dynamically—for example, in response to a certain user action. As mentioned, the page is rebuilt from scratch after each postback, so any information about the dynamically created control is lost. On the other hand, when the page's form is submitted, the dynamic control there is filled with legal and valid information that is regularly posted. By design, there can't be any server control to match the ID of the dynamic control the first time posted data is processed. However, the ASP.NET framework recognizes that some controls could be created in the *Load* event. For this reason, it makes sense to give it a second try to see whether a match is possible after the user code has run for a while.

If the dynamic control has been re-created in the *Load* event, a match is now possible and the control can refresh its state with posted data.

Handling the Postback

The postback mechanism is the heart of ASP.NET programming. It consists of posting form data to the same page using the view state to restore the call context—that is, the same state of controls existing when the posting page was last generated on the server.

After the page has been initialized and posted values have been taken into account, it's about time that some server-side events occur. There are two main types of events. The first type of event signals that certain controls had the state changed over the postback. The second type of event executes server code in response to the client action that caused the post.

Detecting Control State Changes

The whole ASP.NET machinery works around an implicit assumption: there must be a one-to-one correspondence between some HTML input tags that operate in the browser and some other ASP.NET controls that live and thrive in the Web server. The canonical example of this correspondence is between `<input type="text">` and *TextBox* controls. To be more technically precise, the link is given by a common ID name. When the user types some new text into an input element and then posts it, the corresponding *TextBox* control—that is, a server control with the same ID as the input tag—is called to handle the posted value. I described this step in the "Processing Posted Data" section earlier in the chapter.

For all controls that had the *LoadPostData* method return *true*, it's now time to execute the second method of the *IPostBackDataHandler* interface: the *RaisePostDataChangedEvent* method. The method signals the control to notify the ASP.NET application that the state of the control has changed. The implementation of the method is up to each control. However, most controls do the same thing: raise a server event and give page authors a way to kick

in and execute code to handle the situation. For example, if the *Text* property of a *TextBox* changes over a postback, the *TextBox* raises the *TextChanged* event to the host page.

Executing the Server-Side Postback Event

Any page postback starts with some client action that intends to trigger a server-side action. For example, clicking a client button posts the current contents of the displayed form to the server, thus requiring some action and a new, refreshed page output. The client button control—typically, a hyperlink or a submit button—is associated with a server control that implements the *IPostBackEventHandler* interface.

The page processor looks at the posted data and determines the control that caused the postback. If this control implements the *IPostBackEventHandler* interface, the processor invokes the *RaisePostBackEvent* method. The implementation of this method is left to the control and can vary quite a bit, at least in theory. In practice, though, any posting control raises a server event letting page authors write code in response to the postback. For example, the *Button* control raises the *onclick* event.

There are two ways a page can post back to the server—by using a submit button (that is, `<input type="submit">`) or through script. A submit HTML button is generated through the *Button* server control. The *LinkButton* control, along with a few other postback controls, inserts some script code in the client page to bind an HTML event (for example, *onclick*) to the form's *submit* method in the browser's HTML object model. We'll return to this topic in the next chapter.



Note The *UseSubmitBehavior* property exists on the *Button* class to let page developers control the client behavior of the corresponding HTML element as far as form submission is concerned. By default, a *Button* control behaves like a submit button. By setting *UseSubmitBehavior* to *false*, you change the output to `<input type="button">`, but at the same time the *onclick* property of the client element is bound to predefined script code that just posts back. In the end, the output of a *Button* control remains a piece of markup that ultimately posts back; through *UseSubmitBehavior*, you can gain some more control over that.

The *LoadComplete* Event

The page-only *LoadComplete* event signals the end of the page-preparation phase. Note that no child controls will ever receive this event. After firing *LoadComplete*, the page enters its rendering stage.

Page Finalization

After handling the postback event, the page is ready for generating the output for the browser. The rendering stage is divided in two parts: pre-rendering and markup generation. The pre-rendering substage is in turn characterized by two events for pre-processing and post-processing.

The *PreRender* Event

By handling this event, pages and controls can perform any updates before the output is rendered. The *PreRender* event fires for the page first and then recursively for all controls. Note that at this time the page ensures that all child controls are created. This step is important, especially for composite controls.

The *PreRenderComplete* Event

Because the *PreRender* event is recursively fired for all child controls, there's no way for the page author to know when the pre-rendering phase has been completed. For this reason, ASP.NET supports an extra event raised only for the page. This event is *PreRenderComplete*.

The *SaveStateComplete* Event

The next step before each control is rendered out to generate the markup for the page is saving the current state of the page to the view-state storage medium. Note that every action taken after this point that modifies the state could affect the rendering, but it is not persisted and won't be retrieved on the next postback. Saving the page state is a recursive process in which the page processor walks its way through the whole page tree, calling the *SaveViewState* method on constituent controls and the page itself. *SaveViewState* is a protected and virtual (that is, overridable) method that is responsible for persisting the content of the *ViewState* dictionary for the current control. (We'll come back to the *ViewState* dictionary in Chapter 19.)

ASP.NET server controls can provide a second type of state, known as a "control state." A control state is a sort of private view state that is not subject to the application's control. In other words, the *control state* of a control can't be programmatically disabled, as is the case with the view state. The control state is persisted at this time, too. Control state is another state storage mechanism whose contents are maintained across page postbacks much like the view state, but the purpose of the control state is to maintain necessary information for a control to function properly. That is, state behavior property data for a control should be kept in the control state, while user interface property data (such as the control's contents) should be kept in the view state.

The *SaveStateComplete* event occurs when the state of controls on the page have been completely saved to the persistence medium.



Note The view state of the page and all individual controls is accumulated in a unique memory structure and then persisted to storage medium. By default, the persistence medium is a hidden field named `__VIEWSTATE`. Serialization to, and deserialization from, the persistence medium is handled through a couple of overridable methods on the *Page* class: *SavePageStateToPersistenceMedium* and *LoadPageStateFromPersistenceMedium*. For example, by overriding these two methods you can persist the page state in a server-side database or in the session state, dramatically reducing the size of the page served to the user. Hold on, though. This option is not free of issues, and we'll talk more about it in Chapter 19.

Generating the Markup

The generation of the markup for the browser is obtained by calling each constituent control to render its own markup, which will be accumulated in a buffer. Several overridable methods allow control developers to intervene in various steps during the markup generation—begin tag, body, and end tag. No user event is associated with the rendering phase.

The *Unload* Event

The rendering phase is followed by a recursive call that raises the *Unload* event for each control, and finally for the page itself. The *Unload* event exists to perform any final clean-up before the page object is released. Typical operations are closing files and database connections.

Note that the unload notification arrives when the page or the control is being unloaded but has not been disposed of yet. Overriding the *Dispose* method of the *Page* class—or more simply, handling the page's *Disposed* event—provides the last possibility for the actual page to perform final clean up before it is released from memory. The page processor frees the page object by calling the method *Dispose*. This occurs immediately after the recursive call to the handlers of the *Unload* event has completed.

Summary

ASP.NET is a complex technology built on top of a substantially thick—and, fortunately, solid and stable—Web infrastructure. To provide highly improved performance and a richer programming toolset, ASP.NET builds a desktop-like abstraction model, but it still has to rely on HTTP and HTML to hit the target and meet end-user expectations.

It is exactly this thick abstraction layer that has been responsible for the success of Web Forms for years, but it's being questioned these days as ASP.NET MVC gains acceptance and prime-time use. A thick abstraction layer makes programming quicker and easier, but it necessarily takes some control away from developers. This is not necessarily a problem, but its impact depends on the particular scenario you are considering.

There are two relevant aspects in the ASP.NET Web Forms model: the process model and the page object model. Each request of a URL that ends with *.aspx* is assigned to an application object working within the CLR hosted by the worker process. The request results in a dynamically compiled class that is then instantiated and put to work. The *Page* class is the base class for all ASP.NET pages. An instance of this class runs behind any URL that ends with *.aspx*. In most cases, you won't just build your ASP.NET pages from the *Page* class directly, but you'll rely on derived classes that contain event handlers and helper methods, at the very minimum. These classes are known as code-behind classes.

The class that represents the page in action implements the ASP.NET eventing model based on two pillars: the single form model (page reentrancy) and server controls. The page life cycle, fully described in this chapter, details the various stages (and related substages) a page passes through on the way to generate the markup for the browser. A deep understanding of the page life cycle and eventing model is key to diagnosing possible problems and implementing advanced features quickly and efficiently.

In this chapter, I mentioned controls several times. Server controls are components that get input from the user, process the input, and output a response as HTML. In the next chapter, we'll explore the internal architecture of server controls and other working aspects of Web Forms pages.

Index

Symbols

`$.ajax` function, 926
`$.getScript` function, 930
`$.parseJSON` function, 928
`@xxx` syntax, 25–26

A

absolute expiration, 731
abstraction, 575–576
 ASP.NET MVC and, 24
 importance of, 19
 of views, 624–626
Accept-Charset attribute, 82
access
 rules for, 818–819
 securing with roles, 358
access control lists (ACLs), 790–791
AcquireRequestState event, 32, 650
.acsx files, *@Control* directive for, 180
actions in ASP.NET MVC
 applications, 22
Active Record pattern, 599–600
 DAL and, 606
Active Server Pages (ASP), 3
Adapter property, 231
adapters, 605
 control, 230–231
 CSS-friendly, 232
 writing, 232
adaptive rendering, 230–232
AdCreated event, 263
Add Managed Modules dialog box, 37
Add method, 726
`<add>` tag, 287
AddOnPreRenderCompleteAsync method, 202–203, 207–208
AddValidationCallback method, 764
ADO.NET
 classes, binding data to, 413–414
 images, reading, 134
AdRotator controls, 262–263, 268
advertisement banners, 262–263
AggregateCacheDependency
 class, 738–739
aggregates, 600–601

AJAX, 14–20, 313, 337, 839–840
 advent of, 8
 ASP.NET support for, 3
 benefits of, 840
 Browser-Side Templating pattern, 840
 as built-in part of Web, 19
 cross-domain calls, 850–851
 Data-for-Data model, 17
 events, jQuery handlers for, 927
 HTML Message pattern, 839–840
 HTTP façade, 881. *See also* HTTP façade
 infrastructure, 840–851
 interaction model, 17
 JavaScript and, 845–851
 jQuery support, 925–928
 JSON for, 892–893
 Module Pattern, 849
 out-of-band HTTP requests, 841–842
 page methods, 895–897
 partial rendering, 851–879
 remote validation via, 385
 REST and, 879–897
 scriptable services, 880–889
 ScriptManager control, 852–860
 SEO and, 351
 SOP and, 929
 server controls and, 267–268
 UpdatePanel control, 860–865
 WCF services, hosting, 881
 XMLHttpRequest object and, 840, 845
AJAX calls, replacing with postbacks, 10
AJAX-enabled services, 883
 configuration settings, 107–108
ajax function, 926
AJAX HTML helpers, 20
AJAX postbacks, 868
AlachiSoft NCache, 755
allowAnonymous attribute, 294
allowDefinition attribute, 67
AllowDirectoryBrowsing property, 43
allowLocation attribute, 67–68, 71
allowOverride attribute, 70–71
AllowPartiallyTrustedCallers attribute, 789
allowPolicy attribute, 109

AlternatingItemTemplate
 property, 483
AltSerialization class, 696
Amazon RDS, 613
Amazon SimpleDB, 613
anchor controls, 243–244
animations, 916–917
anonymous access, 781–782
anonymous accounts,
 impersonating through, 785
anonymous functions, 846
anonymous ID, 294, 671
anonymous identification feature, 73–74
anonymous users. *See also* user profiles
 user profiles for, 294–295
`<anonymousIdentification>`
 section, 73–74, 76
AOP, 571
Apache Web servers, 27
AppDomain, *ASP.page_aspx* class,
 obtaining, 35
AppendDataBoundItems property,
 419–420
appendTo function, 921
AppFabric, 747–748
AppFabric Caching Services (ACS), 748–753
 architecture of, 748–751
 client-side configuration, 751–752
 programming, 752–753
 storing output caching in, 777
 unnamed caches, 751
AppFabric Hosting Services, 748
App_GlobalResources folder, 304
Application Controller pattern, 632
application data, caching, 721–744
application deployment, 39–62
 application warm-up and preloading, 59–62
 files and settings, packaging, 43–51
 IIS configuration, 55–59
 mode settings, 81–82
 site precompilation, 52–55
 with XCopy, 40–43
@Application directive, 653–654
application directives for global.asax, 653–654

- application events,
 - nondeterministic, 33
- application factory, 176–177
- application logic, 596, 602–605
 - remote deployment, 603–604
- Application* object, 721
 - writing to, 679
- application pages. *See* pages
- application pools
 - defined, 29
 - identity of, custom, 38–39
 - identity of, modifying, 39
 - initialization of, 59
 - process recycling, 55–56
 - warmup of, 59–62
 - working mode for, 30
- `<applicationPool>` section, 95
- Application Request Routing, 37
- application restarts, 38, 56–58, 170
 - causes of, 179
- application root folder, 786
- application state, 675–679. *See also* *HttpApplicationState* class
 - global state, storing, 679
 - synchronization of operations, 678–679
- application warm-up, 59–62
 - application pools, configuring, 60–61
 - autostart provider, 61
 - behavior of, 59
 - specifying actions, 61–62
- Application_End* event handler, 648
- Application_Error* event handler, 283–284
- Application_Error* stub, 275
- applicationHost.config* file
 - editing, 60, 93
 - mappings in, 37
- application-hosting environment
 - configuration settings, 84
- application-level configuration settings, 111
 - accessing, 111–112
 - changing, 65
 - processing of, 65
 - updating, 112–113
- application-level tracing, 100–101
- ApplicationManager* class, 34
- application scope, 119
- application services,
 - centralization of, 30
- applications. *See also* ASP.NET applications; Web applications
 - binding pages to master, 326
 - claims-based identity,
 - configuring for, 825
 - composable parts, 585
 - cookies, sharing, 801–802
 - data storage, 923
 - debugging, 284–285
 - domain logic, 596
 - error handling, 275–277
 - global.asax* file, 651–655
 - inactive, unloading, 84
 - initialization code, 905–906
 - initialization of, 645–651
 - isolation between, 29
 - maintainability, 565
 - object-oriented design, 599
 - permissions for, 788–789
 - plugin-based, 584
 - resources embedded in, 659–660
 - security features of, 780. *See also* security
 - symptoms of deterioration, 567–569
 - theme settings, 340
 - trust level of, 786–789
 - virtual folder for, 645
- Application_Start* event, 32
 - route definitions in, 160
- Application_Start* event handler, 648
- Application_Xxx* notation, 36
- ApplyAppPathModifier* method, 690
- AppSettings* collection, accessing, 111–112
 - `<appSettings>` section, 67, 105–106
- App_Themes* folder, 339
- ArtOfTest, 363
- .aspx* extension, 768
- ASHX extension and resources
 - for handler mappings, 124
 - for HTTP handlers, 141–142
- .asmx* ASP.NET Web services, 881
- ASP pages. *See also* pages
 - processing of, 170
- `<asp:Content>` tag, 325, 329–330
- aspect-oriented programming (AOP), 571
- ASP.global_asax* class, 652
- ASP.NET
 - adoption of, 3
 - authentication API, 108
 - authentication methods, 789–791. *See also* authentication
 - browser information storage, 230
 - configuration hierarchy, 63–110
 - configuration in, 63. *See also* configuration files
 - HTTP modules built-in, 154. *See also* HTTP modules
 - and IIS, history of, 28–31
 - improvements to, 3
 - introduction of, 3
 - membership API, 88
 - perfect framework,
 - characteristics of, 18–19
 - programming model, 19
 - runtime environment, 27. *See also* runtime environment
 - site navigation API, 352–358
 - stateful behavior, 6–7
 - vulnerability patch, 64
 - worker process, standalone, 28–29
 - writing files on disk, 137
- ASP.NET 4, 20–21
- ASP.NET applications. *See also* Web applications
 - custom configuration data, 105–106
 - deploying, 39–62
 - error-handling settings, 80–81
 - health monitoring, 83
 - HTTP modules, registering, 37
 - identity of, 87
 - IIS, configuring for, 55–59
 - installing, 40
 - partial-trust applications, 103
 - preloading, 59–62
 - restarts of, 56–58
 - session-state information, 98–100
 - site-level settings, 108–110
 - termination tracking, 57
 - trust levels, 101–104
 - warmup of, 59–62
- ASP.NET cache. *See* Cache object; caching
- ASP.NET compiler tool, 53–54
 - parameters of, 54
 - target directory support, 53
- ASP.NET Development Server
 - `<webServer>` section and, 109
- ASP.NET HTTP runtime, 174
 - page processing, 169, 174. *See also* page life cycle; pages

- ASP.NET MVC, 4, 21–25
 - abstraction, building with, 18–19
 - control over markup, 24
 - features of, 21–22
 - language, changing on the fly, 311
 - localizing applications in, 306
 - new paradigm of, 14
 - opting out of built-in features, 25
 - request processing, 24–25, 170
 - requests, 22
 - runtime environment, 22–24, 27
 - runtime stack, 23
 - Selective Update model, 20
 - separation of concerns, 23
 - simplicity, 24–25
 - state, maintaining, 23
 - testing code-behind, 363
 - URL patterns, 23
 - URL routing in, 157
 - visual components, 24
- ASP.NET pages. *See also* pages
- ASP.Net permission set, 103
- ASP.NET requests. *See also* HTTP requests
 - processing, 34–35
 - responses, building, 35–36
- ASP.NET security context, 781–791
- ASP.NET site administration tool, 302
- ASP.NET temporary directory, 172
 - compiled code in, 178
- ASP.NET Web Forms, 3–4. *See also* Web Forms
 - culture, setting, 309
 - HTML rendering, 9
 - localized text, 306–307
 - processing and rendering, separating, 9–10
 - request processing, 9
 - SEO and, 350–351
 - testing, 363
 - URL routing in, 157, 160–166
- ASP.NET Web Pages, 25–26
 - audience of, 25
 - @xxx syntax, 25–26
- ASP.NET Web services, 885–887
- aspnet_client* directory, 104
- AspNetCompatibilityRequirements* attribute, 889
- aspnet_compiler -v* command, 53
- aspnet.config* file, 95
- aspnetdb.mdf* file, 293–294
 - structure of, 302
- AspNetInternalProvider* provider, 79
- aspnet_isapi.dll*, 28–30, 92, 170
 - mapping to resources, 30, 171–172
- aspnet_regiis.exe*, connection strings, encrypting with, 114
- aspnet_regsql.exe*, 80
- AspNetSqlProfileProvider*, 301
- aspnet_state.exe*, 697–699
- AspNetXmlSiteMapProvider* class, 100
- ASP.page_aspx* class, 35
- ASPState* database, 701–702
- ASPStateTempApplications* table, 702
- ASPStateTempSessions* table, 702
- <*asp:substitution*> control, 775
- .aspx* files
 - in ASP.NET MVC projects, 22
 - compiling, 284–285
 - handler for, 190
 - @Page directive for, 180, 181
 - serving markup with, 148
- .aspx* pages. *See also* pages
 - URLs, mapping to, 36
- .aspx* source files, changes in, 170, 173
- ASPX templates, 217. *See also* markup
 - ASPXANONYMOUS cookie, 74
- ASPXROLES, 97
- assemblies
 - business logic, 596
 - debug mode, 284
 - default linked assemblies, 185–186
 - early and late binding and, 187
 - generating, 170
 - loading, 187
 - modifying list of, 186
 - number of pages in, 169
 - referencing, 188
 - referencing from pages, 185
 - unloading vs. recompiling, 56
- @Assembly directive, 185–187, 653–654
 - attributes of, 187
- AssociatedControlID* property, 260–261
- Async attribute, 201–202
- AsyncCallback* objects, 146
- asynchronous handlers, 121, 146–147, 201. *See also* HTTP handlers
 - adding to page, 202
 - implementing, 147–148
- Asynchronous JavaScript and XML. *See* AJAX
- asynchronous pages, 121, 201–209
 - AddOnPreRenderCompleteAsync* method, 202–203
 - Async attribute, 201–202
 - building, 203–206
 - operations for, 208
 - PreRenderComplete* stage, 202–203
 - RegisterAsyncTask* method, 206–207
- asynchronous postbacks, 868, 869
 - concurrent, 877–878
 - events of, 872–874
 - triggers for, 872
- asynchronous requests, 95
- asynchronous tasks
 - registering, 201–203, 206–207
 - within pages, 207
- AsyncPostBackError* event, 857
- AsyncPostBackTrigger* class, 869
- attr* function, 922–923
- attribute filters, 912–913
- AttributeCollection* class, 238
- attributes, directive, 181
- Attributes* collection, 237
- AuthenticateRequest* event, 32, 650, 820
- AuthenticateUser* function, 794
- AuthenticateUser* method, 794
- authentication, 789–791
 - Basic authentication, 782
 - claims-based identity, 821–825
 - configuration settings, 74–76
 - Digest authentication, 782
 - Forms authentication, 783, 791–806
 - of HTTP requests, 32
 - integrated Windows authentication, 782
 - login pages, 792
 - LoginStatus* control, 829–830
 - LoginView* control, 830–832
 - None authentication, 789
 - password changes and, 833–834
 - principal objects, custom, 804–806
 - over secured sockets, 803–804
 - sign-outs, 795–796
 - state of, 829
 - user authentication, 784, 794–795
 - of view state, 713
 - Windows authentication, 790–791
 - Windows CardSpace, 791
- authentication API, 108
- authentication modules, 650

<authentication> section, 74–76, 790, 792
 authentication tickets, 792–793
 encoding of, 800
 getting and setting, 798
 securing, 803–804
 storage in cookies, 799–800
 authorization, 76–77
 file authorization, 790–791
 of HTTP requests, 32
 reauthorization, forcing, 663
 URL authorization, 791
 <authorization> section, 76–77
AuthorizationStoreRoleProvider, 821
AuthorizeRequest event, 32, 650
 AutoDetect, 801
 AutoMapper, 605
 automated test frameworks, 638
 autonomous views, 616
AutoPostBack property, 258
 autostart providers, 61
 Preload method, 62
 autostarting Web applications, 38–39
 .axd extension, 127, 129
 for handler mappings, 124

B

Balsamiq Mockups, 624
BarChart control, 547–556
 BarChartItem class, 550
 control hierarchy, 549
 events of, 553–554
 item object, 548–551
 Items property, 550
 properties of, 547
 style properties, 548
 using, 555–556
 base classes, 513
 choosing, 514–515
 extending, 515
 inheriting from, 514–515
 unit testing and, 656
BaseCompareValidator class, 381
BaseDataBoundControl class, 514
BaseDataList class, 514
BaseValidator class, 379–380, 380–381
 Basic authentication, 782
basicHttpBinding model, 885
 batch mode compilation, 169
Begin/End asynchronous handlers, 201–203
BeginProcessRequest method, 147–148

signature of, 146
beginRequest event, 873, 874
BeginRequest event, 32, 151, 649
BeginRequest event handler, 151–152
BeginXxx methods, 94
behaviorConfiguration attribute, 884
 big ball of mud (BBM), 566
 BigTable (Google), 614
 binary large objects (BLOBs),
 database support for, 133
BinaryFormatter class, 539, 696–697
BinaryWrite method, 135, 669
bind function, 918–919
bind method, 907
 binding containers, 226–227
BindingContainer property, 226
 BLL, 593, 596–605
 application logic, 602–605
 design patterns for, 596–602
 bound data
 adding to controls, 551–553
 getting, 540–544
 tracking and caching, 536–538
bound fields, 445
BoundField class, 445
 browser cache, 316, 755
 behavior of, 756
 browser capabilities, 671–672
 <*browserCaps*> section, 77–78, 347
 browser definition files, 345–346
 .browser extension, 77, 230–232, 328
 editing files, 346
 browser IDs, detecting, 344
 browser information
 reading, 345
 repository for, 344
 storage of, 230–231, 328
Browser property, 77, 344
 browser providers, 78
 browser-led processing model, 840–841
 browsers
 browser-capabilities providers, 346–348
 bypassing, 20
 characteristics and capabilities
 of, enumerating, 77, 671–672
 cross-browser rendering, 344–348
 data storage and, 923
 definition files, 345–346

device-specific master pages
 and, 327–329
 DOM and DHTML support, 842
 geo-location capabilities, 312
 Google Chrome, 902
 IDs of, 328
 JavaScript background
 compiler, 901
 JavaScript engines, 902
 programming in, 900–903
 Same-Origin Policy, 929
 script downloads, 313
 scripting engines, 901–902
 up-level browsers, 393
 uploading files from, 249–251
XMLHttpRequest object
 support, 843
 browser-sensitive rendering,
 234–235
 Browser-Side Templating (BST),
 840
BulletedList control, 426–427
 business logic, 596
 modeling, 597
 business logic layer, 593, 596–605
Button class *UseSubmitBehavior*
 property, 213
 button clicks, processing of, 6
 button controls, 257–258
 command buttons, 247, 259
 command name, 498
 for JavaScript event handlers,
 917–918
 rendering as images, 447
 button fields, 445–447
Button1_Click function, 5

C

cache, jQuery, 923–925
Cache class, 722–725
 Cache object, working with,
 725–732
 methods of, 723–724
 properties of, 722–723
 cache items
 attributes of, 725–726
 dependencies of, 725, 728. *See also* dependencies
 expiration policy, 731–732
 priority of, 730–731
 cache manager, 175
Cache object, 676, 721, 722. *See also* cache items; caching
 cache synchronization, 736
 callback function, 732
 clearing, 735

- Cache object (*continued*)
 - data expiration, 731–732
 - dependencies, 728
 - dependencies, broken, 740
 - for globally shared information, 679
 - inserting new items, 725–727
 - limitations of, 744
 - memory pressure statistics, 732
 - priority of items, 730–731
 - removal callbacks, 729–730
 - removing items from, 727
 - scavenging, 731
 - session state expiration policy, 694
- cache settings, 78–80
- Cache-Control header, 757, 761
- CacheControl property, 665
- cached pages, returning, 32
- CacheDependency class, 728, 737
 - constructors, 728
 - deriving from, 737
 - members of, 737
- CacheDependency object, 725, 728
 - aggregate dependencies, 738–739
 - change notifications, 738
 - custom, 737–739
 - for SQL Server, 743–745
 - testing, 742
 - for XML data, 739–742
- caching, 463–464, 721–778
 - of application data, 721–744
 - cacheability of pages, 758–762
 - Cache class, 722–725
 - custom dependency, designing, 737–739
 - DAL, relation to, 734–735
 - database dependency, creating, 743–745
 - dependencies, 722, 738–739. *See also* dependencies
 - distributed cache, 744–755
 - vs. fetching, 733
 - hashtable, 724–725
 - internal structure of, 724
 - isolating caching layer, 734–735
 - of multiple versions of pages, 765–768
 - of page output, 721, 755–777
 - pages, 665–666
 - and performance, 733
 - per request, 737
 - of portions of pages, 768–774
 - removal callbacks, 726
 - sliding expiration, 723, 726, 731–732
 - update callbacks, 726
 - Web cache, 755
 - Windows Server AppFabric, 747–753
 - XML data, cache dependency for, 739–742
- caching profiles, 774–775
- < caching > section, 73, 78–80
- caching services
 - AppFabric, 747–753
 - architecture of, 748–751
 - client-side configuration, 751–752
 - programming, 752–753
- CacheItemPriority enumeration, 730
- CacheItemRemovedReason enumeration, 727
- CacheMultiple class, 724
- CacheProfile attribute, 774
- CacheSingle class, 724
- Calendar control, 263–264, 267
- Cancel buttons, 876
- CAS, 101
- CAS policies, 103
- cascading style sheets. *See* CSS
- Cassandra, 614
- Cassini, 48
 - < webServer > section and, 109
- Castle Active Record, 600
- catalogs, 585–586
- catch blocks, 270–271
- CausesValidation property, 248, 394
- CDNs, 313–314
- CGI, 120
- ChangePassword control, 833–834
- ChangePassword method, 812, 834
- Chatty anti-pattern, 604
- check box fields, 448
- check boxes, 259–260
- CheckBoxList control, 422–424
- child controls
 - hierarchy of, 544–545
 - initialization, 210
 - managing, methods for, 195–197
 - postbacks, detecting from, 866–868
 - for rendering, 528–532
 - state of, persisting, 6–7
 - storage of, 229
 - unique names for, 190
- child filters, 912
- child requests, free threads for, 86
- ChildrenAsTriggers, 867–869
- Chirpy, 314
- claims, 822, 823–824
- claims-based identity, 821–825
 - using, 824–825
 - workflow, 822–823
- claims-based Windows Identity Foundation (WIF), 76
- class names, resolving, 188
- classes
 - adapters, 605
 - closed for modification, 575–576
 - code-behind classes, 12. *See also* code-behind classes
 - coupling, 570–571
 - dynamically generated, 170
 - hierarchy of, 13
 - imports and exports, 585
 - page classes, 12–13
 - partial classes, 173
 - preconditions, 578
 - responsibilities of, 573–574
 - splitting, 574
 - system classes, 12
- classic ASP, 4
- classic ASP.NET, 4. *See also* Web Forms
 - moving away from, 15–19
- cleanup code in exception handling, 272
- click-throughs, 515
- client behavior, controlling, 213
- client cache, 748–753
- client certificates, 782
- client data, processing, 211
- client IDs of controls, 220
- client script files, storage of, 104–105
- client script manager, reference to, 195
- client side, 839. *See also* AJAX
 - events for user feedback, 872–874
 - JSON representations on, 890
 - powering, 899–905
- client Web programming, 3
- clientIDMode attribute, 91
- ClientIDMode property, 223
 - Predictable option, 224–225
 - Static option, 224
- ClientID property, 211, 220
- ClientIDRowSuffix property, 226
- ClientScript object, 856

- ClientScript property, methods exposed by, 198
- ClientScriptManager class
 - GetWebResourceUrl method, 195
- clientScriptsLocation attribute, 104
- client-side behavior, testing, 361–363
- client-side message boxes, 501
- client-side validation, 393–394
- closures in JavaScript, 847–848
- cloud databases, 613
- CLR exceptions, 270. *See also* error handling
- CLR security zones, 786
- CMS, 157
- code
 - sandboxing, 789
 - testability, 636–642
 - for Web pages, 3
- code access security (CAS), 101, 103
- code blocks in server-side
 - <head> sections, 243
- Code Contracts API, 578
- code declaration blocks, 654
- code-behind classes, 217
 - defined, 12
 - hierarchy of, 13
 - removing, 627–628
 - server control references, 173
 - testing, 361
 - WebMethod attribute, 895–896
- cohesion, 569–571
- collections
 - binding data to, 412–413
 - in profiles, 289
- COM, 843
- command buttons, 247, 259, 498–499
 - custom, 499
- command names, 498
- Common Gateway Interface (CGI), 120
- common language runtime (CLR), 270, 786
- CompareValidator control, 380, 382–383, 386
- compiled pages, 170
- master pages, 329
- Component Object Model (COM), 843
- composable parts, 585
- composite controls, 521. *See also* controls; server controls
 - child controls, hierarchy of, 544–545
 - collections of items, 547
 - defined, 519
 - composite data-bound controls. *See also* controls; server controls
 - adding bound data, 551–553
 - building, 543–561
 - data item object and collection support, 547
 - hierarchy and data separation, 546–547
 - receiving data, 545
 - template support, 556–561
 - CompositeControl class, 514
 - as base class, 519
 - CompositeDataBoundControl class, 514
 - deriving from, 544
 - compressed responses, 79
 - concerns, separation of, 571–572
 - concurrent calls, 877–878
 - conditional refreshes, 866–870
 - CONFIG directory, 64
 - <configProtectedData> section, 107
 - <configSections> element, 67
 - configuration, declarative, 589–590
 - <configuration> element, 50, 66
 - main children of, 66
 - configuration errors, 269
 - configuration files, 64–68. *See also* individual section names
 - accessing, 63
 - add, remove, and clear elements, 68
 - <anonymousIdentification> section, 73–74
 - application-level settings, 111
 - <appSettings> section, 67, 105–106
 - <authentication> section, 74–76
 - <authorization> section, 76–77
 - <browserCaps> section, 77–78, 347
 - <caching> section, 78–80
 - changes to, 63
 - <configProtectedData> section, 107
 - <configSections> element, 67
 - <configuration> element, 66
 - <connectionStrings> section, 106
 - creation of, 63
 - <customErrors> section, 80–81, 278–279
 - custom sections, creating, 116–117
 - custom sections, registering, 117
 - <deployment> section, 81–82
 - <EncryptedData> section, 114
 - encrypting, 107, 113–116
 - <globalization> section, 82, 309
 - handler factories, registering, 145
 - <handlers> section, 125
 - <healthMonitoring> section, 83
 - <hostingEnvironment> section, 84
 - <httpCookies> section, 84–85
 - <httpHandlers> section, 82, 123
 - HTTP modules, registering with, 153
 - <httpModules> section, 82, 153
 - <httpRuntime> section, 85–87
 - <identity> section, 87
 - <location> section, 68–71
 - <machineKey> section, 87–88
 - machine-level settings, 111
 - machinewide settings, 70
 - managing, 110–117
 - <membership> section, 88–89
 - opening, 111
 - <pages> section, 89–92
 - <processModel> section, 92–95
 - <profile> section, 96–97, 286–287
 - <properties> section, 286
 - protection of, 64
 - <providers> section, 301
 - <roleManager> section, 97
 - <section> element, 67
 - <sectionGroup> element, 67–68
 - sections, declaring, 67
 - <securityPolicy> section, 97–98
 - <sessionState> section, 98–100
 - <siteMap> section, 100
 - <system.serviceModel> section, 67
 - <system.web.extensions> section, 107–108
 - <system.web> section, 71–73
 - <system.webServer> section, 108–110
 - <trace> section, 100–101
 - tree of, 64–65
 - <trust> section, 101–104
 - unmodifiable settings, 70–71
 - <urlMappings> section, 104

- configuration files (*continued*)
 - user names and passwords in, 87
 - <*webControls*> section, 104–105
 - <*xhtmlConformance*> section, 105
 - configuration management API, 110–113
 - configuration section handlers, 116
 - ConfigurationManager* class, 111
 - OpenMachineConfiguration* method, 112
 - ConfigurationProperty* attribute, 117
 - connectionString* attribute, 106
 - connection strings
 - configuration settings, 106
 - encrypting, 114
 - for out-of-process session state, 700
 - from profile providers to database engine, 301
 - connectionStringName* attribute, 80
 - ConnectionStrings* collection, accessing, 111–112
 - <*connectionStrings*> section, 106
 - constructors, overloaded, 583
 - container controls, 239–240
 - Container* keyword, 560
 - content. *See also* data
 - default content, 323–324
 - downloading cross-domain, 929–930
 - inline content, 316–317
 - Content* controls, 324–326
 - content delivery networks (CDNs), 313–314
 - content filters, 912
 - Content Management Systems (CMS), fiendly URLs and, 157
 - content pages
 - binding definitions, 327
 - binding to master pages, 326
 - Content controls, 324–326
 - content placeholders, 320–321, 323
 - defined, 320
 - @*MasterType* directive, 335–336
 - processing, 329–334
 - serving to user, 329–330
 - source code of, 325
 - title of, 326
 - writing, 323–328
- ContentPlaceHolder* controls, 320–323
 - ContentTemplate* property, 863
 - contract attribute, 884
 - contracts, MEF, 585
 - control adapters, 230–231
 - writing, 232
 - Control* class, 190, 218, 514. *See also* controls; server controls
 - ClientIDMode* property, 223
 - deriving controls from, 513
 - events of, 229–230
 - extending, 515
 - IComponent* interface, 218
 - IDisposable* interface, 218
 - interfaces of, 218
 - methods of, 228–229
 - properties of, 218–228
 - RenderingCompatibility* property, 233
 - vs. *WebControl* class, 519
 - @*Control* directive, 180
 - control IDs
 - matching to posted names, 211
 - retrieving, 915
 - control properties
 - persistence modes, 558
 - varying output caching by, 770–772
 - control skins, 235
 - control state, 214, 718
 - programming, 718–719
 - ControlAdapter* class, 230
 - controllers
 - in ASP.NET MVC, 21–22
 - defined, 616–617
 - role of, 618
 - ControlParameter* class, 462
 - controlRenderingCompatibility-Version* attribute, 232
 - controls
 - data-bound controls, 421–434. *See also* data-bound controls dynamically created, handling, 211–212
 - and input tags, correspondence between, 212
 - naming container for, 190
 - Page* class methods related to, 195–197
 - prerendering stage, 214
 - state changes, detecting, 212–213
 - unloading, 215
 - validating groups of, 394–395
 - validation, support of, 382
 - view-state tracking, 210
 - Controls* collection, 229
 - adding controls to, 521
 - dynamically added controls, 266
 - <*controls*> section, 91
 - ControlStyle* property, 254, 255
 - ControlToValidate* property, 381
 - cookieless attribute, 74, 800–801
 - cookieless sessions, 688–691
 - CreateUninitialized* method, 706
 - issues with, 689–690
 - Search-Engine Optimization and, 691
 - security and, 690–691
 - cookies, 675, 687–688
 - configuration settings, 84–85
 - cookieless sessions, 688–691
 - customizing, 804–806
 - Forms authentication through, 799–800
 - for HTTP façade, 887–888
 - for role information, 97
 - sharing between applications, 801–802
 - usage of, 74
 - Copy Web Site function (Visual Studio), 40–42
 - CopyFrom* method, 255
 - CouchDB, 614
 - coupling, 569–571
 - between modules, 575
 - between presentation and business layers, 604
 - C++ server programming, 3
 - CPU bound operations, 208
 - CreateChildControls* method
 - binding and nonbinding modes, 545
 - overloaded version, 545–547
 - overriding, 521, 544
 - pseudocode, 544
 - CreateUninitialized* method, 706
 - CreateUser* method, 811
 - CreateUserWizard* control, 834–835
 - credentials
 - collecting, 794
 - getting, 822
 - cross-browser rendering, 344–348
 - cross-domain calls
 - in AJAX, 850–851
 - jQuery and, 929–932
 - Cross-Origin Resource Sharing (CORS), 929

cross-page communication, 189–190

cross-page posting, 365, 374–379

- detecting, 377–378
- @*PreviousPageType* directive, 376–377
- redirecting to another page, 378
- validation and, 395–396
- view state information, 374–375

cross-site scripting (XSS), 780

- GET and, 886–887

CSS, 319

- applying to elements, 917
- ASP.NET support for, 3
- embedded vs. inline, 316
- for *ListView* styling, 474, 480, 482, 494–497
- minimizing impact of, 315–317
- style sheets, 339
- vs. themes, 220, 343, 357
- use of, 255

CSS Control Adapter Toolkit (CSSCAT), 232

css function, 917

CSS-based selectors, 909–910

CssClass property, 493–494

CSS-friendly markup code, 232–234

CssStyleCollection class, 254

culture

- changing, 310–312
- names of, 82, 309
- resource assemblies, creating, 308
- setting, in ASP.NET Web Forms, 309
- setting, in .NET, 308–309

Culture attribute, 860

Culture property, 860

Cunningham wiki, 570

Current property, 657, 897

CurrentCulture property, 308–309

CurrentUICulture property, 308–317

custom controls, 513–561. *See also* server controls

- building from scratch, 518–533
- control state, 718–719
- control tree, building, 521–522
- Control* vs. *WebControl*, 519
- correct rendering of, 195
- data-bound composite controls, building, 543–561
- data-bound controls, building, 533–543

- embedded resources, 195
- extending existing controls, 514–518
- interfaces for, 519
- markup, writing to HTML text writer object, 527–528
- object model definition, 523
- object model implementation, 523–526
- rendering style, 520–522
- template support, 556–561
- usage scenario, defining, 515–516

custom object caching, 463

custom resources, HTTP handlers for, 126–127

custom types in profiles, 289–290

Customer Relationship Management (CRM) systems, 613

<*customErrors*> section, 80–81, 278–279, 857

customization themes, 338

CustomValidator control, 380, 383–385, 395–396, 504

D

DAL, 593, 596, 598, 605–614

- Active Record pattern and, 606
- alternatives to, 613–614
- caching, relation to, 734–735, 747
- conceptual view of, 608
- database independence, 608–609
- domain model organization, 602
- Domain Model pattern and, 607–608
- implementation of, 605–608
- interfacing, 608–610
- O/RM implementation, 610–613
- Repository pattern for, 609–610
- responsibilities of, 607
- Table Module pattern and, 606

data

- deleting, 465–468
- editing, 454–455
- paging, 451–453
- retrieving, 460–461
- sorting, 453–454
- storing on server, 65
- updating, 465–468

data access layer. *See* DAL

data binding

- application-level, 326
- class diagram, 415
- customizing controls for, 533–543
- DataBinder* class, 436–438
- data source controls, 456–468
- data sources, 412–415
- defined, 411
- Eval* method, 436–438
- to *GridView* control, 443–451
- HtmlSelect* control support of, 245
- with *ListView* control, 477–479
- ObjectDataSource* class, 459–469
- page-level, 326
- parent controls, identifying, 226
- process of, 411
- separating from control-building code, 551–553
- for server control RAD designers, 218
- simple, 434–436, 533
- syntax, 434–435
- Web pages vs. desktop applications, 414

data contracts

- for AJAX-enabled WCF services, 885
- preservation of, 892

data controls, binding to data source controls, 474

data eviction, 747

data expiration in cache, 731–732

data function, 923

data item classes, defining, 536–538

data item containers, 227

data keys containers, 227

data models

- for WAP projects, defining, 290–292
- for Web site projects, defining, 286–287

data paging with *ListView* control, 507–511

data representation, JSON for, 890–893

data source controls, 456–468

- data controls, binding to, 474
- defined, 456
- hierarchical controls, 457–458
- ID of, 417
- named views, 456–458
- parameters, using, 462–463
- tabular controls, 456

- data source objects
 - interface implementation, 534
 - mapping to control properties, 535–536
- data sources, 412–415, 721
 - adding items to, 501–505
 - ADO.NET classes, 413–414
 - collection classes, 412–413
 - key fields, setting, 420
 - queryable objects, 414–415
 - specifying, 416
 - updates to, 454–455
 - viewing records, 432–433
- data tables, managing, 438–455
- data transfer objects (DTOs), 605
- data validation. *See also* validation
 - configuration settings, 87
 - for cookies, 799
- database dependencies, 80
 - for caching, 743–745
- database management systems (DBMS), BLOB support, 133
- databases
 - cloud databases, 613
 - images, loading from, 133–136
 - session state, storing in, 699–704
 - sharding, 612
- DataBind* method, 411
 - calling, 414, 561
- DataBinder* class, 436–438, 541
- DataBinding* event, 556
- data-binding expressions, 434–438
 - evaluation of, 435
 - generating and parsing, 437
 - implementing, 436
- data-binding properties, 411, 415–421
 - AppendDataBoundItems* property, 419–420
 - DataKeyField* property, 420
 - DataMember* property, 417–418
 - DataSourceID* property, 417
 - DataSource* property, 416–417
 - DataTextField* property, 418
 - DataValueField* property, 419
- data-bound controls, 421–434
 - bound data, getting, 540–544
 - bound fields, 445
 - building, 533–543
 - composite, 543–544
 - data-bound properties, adding, 534
 - data item classes, 534–535
 - DataPager* control for, 507–508
 - data source fields to control
 - properties, mapping, 535–536
 - defined, 411
 - events of, 556
 - iterative controls, 427–432
 - key features of, 533–534
 - list controls, 421–427
 - receiving data, 545
 - types of, 411
 - view controls, 432–434
- DataBound* event, 556
- DataBoundControl* class, 514
- DataBoundLiteralControl* class, 434
- DataContract* attribute, 891–892
- DataContractJsonSerializer* class, 891
- Data-for-Data model, 17
- DataGrid* control, 431–432
 - autoreverse sorting, 718
- DataItemContainer* property, 227
- DataKeyField* property, 420
- DataKeyNames* property, 421, 467–468, 474, 500
- DataKeysContainer* property, 227
- DataList* control, 430–431
- DataMember* property, 417–418
- DataPager* control, 507–511
 - embedding, 508–509
 - properties of, 508
 - types of, 509
- DataPagerField* class, 509
- DataSet* class, 414, 598
- DataSource* property, 416–417, 474
- DataSourceID* property, 417, 442, 474, 477
- DataSourceView* class, 457
 - methods of, 458
- DataTextField* property, 418
- DataTextFormatString* property, 418
- DataValueField* property, 419
- Db4O, 614
- debug mode, 284–285
- debug script files, 859–860
- debugging pages, 284–285
- declarative authorization, 77
- decryption* attribute, 87
- decryption keys, specifying, 87–88
- decryptionKey* attribute, 87
- default ASP.NET account
 - changing, 784–786
 - privileges of, 785–786
- default.aspx* skeleton, 625
- defaultProvider* attribute, 79, 88, 97, 100
- defaultRedirect* attribute, 81
- defaultUrl* attribute, 799
- delegate classes, 146
- delete operations, in *ListView* control, 500–501
- DeleteUser* method, 811
- denial of service (DoS), 780
- dependencies, 568
 - aggregate, 738
 - broken, 740
 - of cached items, 722, 725, 728
 - database dependencies, 743–745
 - decreasing number of, 570
 - isolating, 572
 - polling, 739, 742
 - resolving, 588–589
- Dependency Injection, 582–591
- Dependency Inversion principle, 572, 580–583
- dependency-changed* event, 730
- deploy packages
 - building, 45–47
 - contents of, 45
 - running, 44
- deployment. *See* application deployment
- deployment precompilation, 53–55
 - update support, 54–55
- <*deployment*> section, 81–82
- derived classes
 - generating, 177
 - naming convention for, 172
 - substitution principle and, 576–578
 - URLs, linking to, 172
- description* attribute, 349
- description* meta tag, 349
- deserialization of session state, 695–697, 710
- design patterns
 - Active Record pattern, 599–600
 - for BLL, 596–602
 - Browser-Side Templating, 840
 - Domain Model pattern, 600–602
 - HTML Message, 839–840
 - Module Pattern, 849
 - MVC pattern, 616–618
 - MVP pattern, 619–621, 623–636
 - MVVM pattern, 621–623
 - for presentation layer, 615–623
 - Repository pattern, 609–610
 - Service Layer pattern, 602

- design patterns (*continued*)
 - Table Module pattern, 598
 - Transaction Script pattern, 597–598
 - Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, and Vlissides), 575
 - design principles, 569–572
 - detach* function, 922
 - detach* method, 921
 - DetailsView* control, 432
 - DataKeyNames* property, 467
 - vs. *ListView* control, 476
 - development environment, *web*.
 - config* file for, 51
 - development frameworks, characteristics of, 18–19
 - device-specific content, ignoring, 92
 - die* function, 919
 - Digest authentication, 782
 - Dijkstra, Edsger W., 571
 - direct scripting, 19–20
 - directives
 - location of, 179
 - processing directives, 179–190
 - DirectoryInfo* class, 130
 - disabled* attribute, 875
 - disableExpiration* attribute, 79
 - DisplayIndex* property, 493
 - Dispose* method, 149, 648
 - for custom HTTP modules, 151
 - overriding, 215
 - Disposed* event, handling, 215
 - distributed cache, 744–755
 - of ACS, 748–751
 - AlachiSoft NCache, 755
 - data eviction, 747
 - design of, 745
 - features of, 745–747
 - freshness of data, 747
 - high availability, 746
 - Memcached, 753–754
 - NorthScale Memcached Server, 755
 - read-through and write-through capabilities, 747
 - ScaleOut StateServer, 755
 - SharedCache*, 754
 - storing output caching in, 777
 - topology of, 746
 - distributed caching system, 745
 - DOM (Document Object Model)
 - adding elements to, 920–922
 - assumptions about, 8
 - elements, accessing, 91
 - evolution of, 842
 - functional approach combined with, 903
 - manipulating in jQuery, 920–923
 - modifying elements in, 922–923
 - queries, running over, 904, 908
 - readiness for scripting, detecting, 906–907
 - readyState* property, 906
 - removing elements from, 922
 - updating content with, 842–843
 - DOM trees
 - adding to DOM, 920–922
 - creating, 920, 922
 - toggling, 921–922
 - domain* attribute, 84, 802
 - domain logic, 596
 - domain model, defined, 600
 - Domain Model pattern, 600–602
 - DAL and, 607–608
 - Domain-Driven Design (DDD), 601
 - DOMContentLoaded* event, 906
 - DoPostBackWithOptions* function, 374
 - download experience, optimizing, 312–317
 - DPAPI protection provider, 107, 115
 - DropDownList* control, 421–422
 - binding data to, 534
 - HTML server control for, 245
 - DTOs, 605
 - dummy objects, 640
 - Duration* attribute, 759, 761
 - dynamic compilation, 52, 169
 - dynamic controls, handling, 211–212
 - Dynamic HTML (DHTML), 839, 842
 - dynamic* keyword, 334
 - Dynamic Language Runtime (DLR) component, 335
 - dynamic resources. *See* pages
 - dynamic* type, 375–376
 - dynamic user interfaces, 18–19
- E**
- each* function, 908–909
 - eavesdropping, 780
 - ECMAScript, 900
 - edit template
 - defining, 497–498
 - predefined buttons, 498–499
 - editing data, 454–455
 - Eich, Brendan, 900
 - ELMAH, 284
 - empty fields, 387
 - empty* function, 922
 - EnableClientScript* property, 393
 - enableKernelCacheForVaryByStar* attribute, 79
 - EnablePageMethods* property, 896–897
 - EnablePaging* property, 464
 - EnablePartialRendering* property, 863
 - EnableScriptGlobalization* property, 860
 - EnableScriptLocalization* property, 315
 - EnableTheming* property, 235, 342
 - EnableViewState* attribute, 715
 - EnableViewStateMac* attribute, 713
 - EnableViewState* property, 227
 - enableWebScript* attribute, 883–884, 893
 - encapsulation, 572
 - encoders, 137
 - encoding, 82, 661
 - <*EncryptedData*> section, 114
 - encryption
 - of configuration files, 113–116
 - for cookies, 799
 - key containers, 115
 - of view state, 712–713
 - XML encryption, 107
 - encryption keys, specifying, 87–88
 - encryption providers, 107
 - choosing, 115–116
 - EncType* property, 249
 - endpoints, JSONP-enabled, 930
 - EndProcessRequest* method, 147
 - EndRequest* event, 33, 651, 874, 875
 - EndRequest* event handler, 32, 151–152
 - EndRequestEventArgs* control, 875
 - EnsureChildControls* method, 522
 - Entity Framework, 458, 611–612
 - error codes, 64, 81
 - Error* event handler, defining, 274
 - Error* events, 33, 274, 651
 - error handling, 269–285
 - configuration settings, 80–81
 - Error* event handler, 651
 - error pages, 273–274
 - error reporting, 283–285
 - errors, mapping to pages, 278–282

error handling (*continued*)
 exception handling, 270–272
 exception logging, 277
 for fatal errors, 283–285
 global, 275–277
 handler precedence, 277
 HTTP errors, 280–281
 page error handling, 272–278
 page-level, 274–275
 partial rendering, 857
 reporting in e-mail messages, 276
 robustness of, 278

Error Logging Modules And
 Handlers, 284

error messages
 custom, 278–279
 displaying, 275, 388–389
 summary and display of, 391–392

error pages, 273–274
 code-behind, 281
 context-sensitive, 282
 custom, 279–280
 custom, specifying, 81
 for local and remote users, 279
 sensitive information on, 275, 278

error reporting, 283–285
 <error> tag, 280

ErrorMessage property, 388, 392–393

errors
 HTTP 500 errors, 277
 mapping to pages, 278–282
 session state and, 695
 types of, 269

Esposito, Dino, 26

Eval method, 436–438
 syntax, 437

Event object, 919

events
 of *BarChart* control, 553–554
 canceling, 407–408
 of *Control* class, 229–230
 of data-bound controls, 556
 of *GridView*, 442–443
 handling, 151–152. *See also* HTTP handlers
 for health monitoring, 83
 of *HttpApplication* class, 150, 648–651
 in IIS messaging pipeline, 32–33
 of *ListView* control, 474–476
 order of firing, 649–650
 of *Page* class, 198–199

personalization events,
 298–299
 of *ScriptManager* control, 856

Exception class, 272

exception handling, 270–272
 ASP.NET-specific facilities for,
 270
 cleanup code, 272
 finally blocks, 271–272
 guidelines for, 271–272
 in .NET, 270–272
 try/catch blocks, 270

exceptions
 built-in types, 271
 function of, 270–271
 getting information about, 282
 retrieving, 276
 self-logging, 284
 unhandled, 272

Execute method, 661–663
 overloads of, 662

ExecuteRequestHandler event,
 33, 650
 handler for, registering, 34

ExecuteScalar method, 134

expiration callbacks, 706

expiration policies
 for cached items, 726, 731–732,
 747
 for session-state items, 694

expired cache items, automatic
 scavenging, 79

Expires HTTP header, 756

Expires property, 665

ExpiresAbsolute property, 665

exports, 585–587

extensions, *aspnet_isapi.dll*
 handling of, 171–172

external style sheet files, linking
 to, 242

F

Factory attribute, 884

fakes, 640

fatal exceptions, 283–285

feedback
 client-side events for, 872–874
 for partial page updates, 875
 progress screen, 871–872
 for users, 870–876

fetching vs. caching, 733

ffSite.master file, 327

Fiddler, 363

fields, defined, 413

file authorization, 790–791

file system monitor, 175

file types, searching for, 130

FileAuthorizationModule HTTP
 module, 790

files
 copying to target site, 42
 packaging, 43–51
FileSystemWatcher object, 738,
 742

FileUpload control, 261–262

filter function, 914

filters, 911–914

finally blocks, 271

FindControl method, 375

find function, 914

Firebug Web development tool,
 317

Firesheep, 803

fixed identities, impersonating,
 785

fixednames parameter, 55

flow layouts, 485–487
 item layout, 486–487

Foote, Brian, 566

forbidden resources, blocking
 access to, 652–653

ForeColor property, 381

form filters, 913–914

form submissions, 6
 client behavior, controlling, 213

<form> tags, 365
 multiple, 368–373
runat, visibility of, 371–373

format strings, defined, 556

Forms authentication, 783,
 791–806
 advanced features, 801–806
 attributes of, 798–799
 configuration of, 75, 798–801
 control flow, 792–796
 cookie-based, 799–800
 cookieless, 800–801
 cookies, sharing among
 applications, 801–802
 custom types of, 76
 encryption and decryption
 keys, 87
 external applications for, 803

FormsAuthentication class,
 796–798

<forms> section, 798–799
 with HTTPS, 889

Login control, 826–828
 resources protected by, 792
 security concerns, 793, 804
 setting up, 792
 user credentials, collecting, 794

<forms> element, 74–75

<forms> section, 798–799
FormsAuthentication class, 796–798
 methods of, 797–798
 properties of, 796–797
SignOut method, 795–796
FormView control, 432, 433
 vs. *ListView* control, 476
 FTP, copying files to target with, 42
FullTrust permission set, 103
 functional programming, 845–846
 JQuery and, 905
 functions
 closures, 847–848
 defined, 846

G

GAC
 assemblies in, 786
 HTTP handlers in, 124
 Gamma, Erich, 575
 gauge control, 523–533
GaugeBar control, 535–544
 data item object, 536–538
 mapping fields to properties, 535–536
PerformDataBinding method, 540–544
 GDI+ subsystem, 140
 geo-location capabilities, 312
get accessor, 538
get function, 909
GET verb, 365
 enabling, 886–887
 kernel caching and, 762
 posting forms with, 367
GetAuthCookie method, 798
GetConfig method, 658
GetDataItem function, 438
GetEnumerator method, 678
GetFiles method, 130
GetGlobalResourceObject method, 307, 659–660
GetHandler method, 145
getJSON function, 927–928
GetLastError, 276
GetLocalResourceObject method, 660
GetPropertyValue method, 541
GetPropertyValue property, 290–291
GetRequiredString method, 163
getScript function, 927
GetUser method, 811

GetVaryByCustomString method, 767
GetWebApplicationSection method, 658
GetWebResourceUrl method, 195, 315
 global assembly cache (GAC)
 assemblies in, 786
 HTTP handlers in, 124
 global error handling, 275–277
 global resources, 304–305, 307
 retrieving, 659–660
 global themes, 339
global.asax file, 651–655
 aliasing file name, 655
 application directives, 653–654
Application_Error stub, 275
 blocking access to, 652–653
 C# code skeleton, 652
 changes to, 653
 code declaration blocks, 654
 compiling, 652–653
 contents of, 645
 editing, 170
 extending, 36–37
 in precompiled assembly, 652
 routes in, 160
 server-side <object> tags, 654–655
 static properties in, 655
 syntax of, 653–655
 globalization, 860
 configuration settings, 82
 <globalization> section, 82, 860
 culture settings, 309
 Google Chrome browser, 902
 Google Gears, 312
GridView control, 433
 accessibility properties, 440
 appearance properties, 440
 behavior properties, 439
 binding data to, 443–451
 bound fields, 445
 button fields, 445–447
 check box fields, 448
 columns, configuring, 444–445
DataKeyNames property, 467
 default user interface, 452
 editing data, 454–455
 events of, 442–443
 hyperlink fields, 447–448
 image fields, 448–449
 interaction with host page, 451
 vs. *ListView* control, 477, 482–483
 object model, 439–443

paging data, 451–453
 predefined types, 440
 Predictable algorithm implementation, 225–226
 sorting data, 453–454
 state properties, 441
 style properties, 440
 templated fields, 450–451
 templating properties, 441–442
GroupItemCount property, 488, 489
GroupPlaceholderID property, 473
GroupSeparatorTemplate property, 489
GroupTemplate property, 487
 Guthrie, Scott, 57
 Gzip compression, 314

H

handler factories, 142, 144–145
IsReusable property and, 144
 handler factory objects, 177
 handlers, 5–6
 for page requests, 177–178
 <handlers> section, 125
hashAlgorithmType attribute, 88
 hashing algorithms, specifying, 87
 <head> section, code blocks in, 243
HeadContent placeholder, 326
 health monitoring system, 83
 <healthMonitoring> section, 83
heartbeat event, interval for, 83
heartbeatInterval attribute, 83
 Helm, Richard, 575
 hidden fields
 creating, control for, 261–262
 tracking values in, 7
 view state information, saving in, 200
HiddenField control, 261–262
 hidden-field tampering, 780
hide function, 915–917
HierarchicalDataBoundControl class, 514
HierarchicalDataSourceView class, 458
 high availability of distributed cache, 746
 <hostingEnvironment> section, 84
HostSecurityManager object, 103
hostSecurityPolicyResolverType attribute, 104

- HTML
 - ASP.NET MVC control over, 24
 - control over, 8
 - downloading, 928
 - literal strings, 7
 - syntax, 3
- HTML attributes, setting, 237–238
- HTML controls, correct rendering of, 195
- HTML elements
 - ID of, 16
 - predictable IDs, 91
- HTML encoding, 661
- HTML forms, 200
- HTML input controls, 246–252
 - command buttons, 247
 - HtmlImage* controls, 252
 - state changes, detecting, 248–249
 - for uploading files, 249–251
 - validation of input fields, 248
- HTML markup. *See also* markup
 - adaptive rendering of, 230–232
 - client ID, 220–223
 - control over, 234
- HTML Message (HM) pattern, 19, 839–840
- HTML output, layout of, 8
- HTML pages, JavaScript code in, 16
- HTML responses, arranging, 5
- HTML server controls, 217, 235–252. *See also* server controls
 - base class, 237
 - container controls, 239–240
 - external style sheet files, linking, 242
 - generic controls, 237
 - header information, 241–242
 - hierarchy of, 239
 - HTML attributes, mappings to, 237–238
 - HTML syntax support, 236
 - meta information, managing, 243
 - namespace definition, 237
 - page postbacks with, 244
 - predefined controls, 236
 - properties of, 237
 - runat=server* attribute, 235
 - state changes, detecting, 248–249
- HTML tables, 484–485
- HTML text writer object, 520
 - writing markup to, 527–528
- HtmlAnchor* class, 243–244
- HtmlButton* class *CausesValidation* property, 248
- HtmlButton* controls, 240
- HtmlContainerControl* class, 239, 366
- HtmlControl* class, 236–237
- HtmlForm* class, 200, 365–367
 - methods of, 367
 - properties of, 366–367
 - Visible* property, 371
- HtmlForm* controls, 240
- HtmlGenericControl* class, 237
- HtmlHead* controls, 241
- HtmlImage* controls, 252
- HtmlInputButton* class, 247
 - CausesValidation* property, 248
- HtmlInputCheckBox* controls, 260
- HtmlInputControl* class, 246
- HtmlInputFile* controls, 249–251
- HtmlInputImage* controls, 247
- HtmlInputRadioButton* control, 260
- HtmlInputReset* controls, 247
- HtmlInputSubmit* controls, 247
- HtmlLink* controls, 242–243
- HtmlMeta* controls, 243
- HtmlSelect* controls, 244–245
 - data binding support, 245
- HtmlTextArea* control, 245
- HTTP access error 403
 - (forbidden), 64
- HTTP endpoints, 123
 - binding handlers to, 128–129
- HTTP errors
 - handling, 280–281
 - HTTP 302, 280, 349
 - HTTP 403, 64
 - HTTP 404, 280–281
 - HTTP 500, 277
- HTTP façade, 880–881
 - ASP.NET Web services, 885–887
 - JSON content, 890
 - protecting services in, 887–888
 - proxy for, 893–895
 - trusting, 888–889
 - WCF services, 881–885
- HTTP GET, enabling, 886–887
- HTTP handlers, 11
 - for AJAX presentation layer, 881
 - for AJAX requests, 148
 - allowPolicy* attribute, 109
 - alternate, specifying, 36
 - ASHX resources, defining as, 141–142
 - asynchronous handlers, 146–148
 - binding to endpoints, 136
 - calling, 123
 - declaring, 123
 - determining, 32, 35
 - forbidden resources, preventing access to, 143
 - functionality of, 33, 119
 - handler factories, 142, 144–145
 - IHttpAsyncHandler* interface, 121
 - IHttpHandler* interface, 121, 121–127
 - images, controlling with, 140–141
 - images, database-stored, serving, 134–135
 - images, dynamically generated, serving, 137
 - images, serving, 128–133
 - loading, 125
 - mapping, 124
 - for new types of resources, 126–127
 - picture viewer handler, 128–133
 - precondition attribute, 109, 126
 - preconditions on, 126
 - ProcessRequest* method, 138
 - query string parameters, accepting, 131
 - registering, 82, 121, 124–125, 132, 141
 - reusing, 143–144
 - ScriptResource.axd*, 859
 - session state, access to, 141
 - synchronous or asynchronous mode, 121, 201
 - <*system.webServer*> section and, 108–109
 - vs. URL routing, 165
 - uses of, 141
 - writing, 36, 121–148
- HTTP headers
 - fresh and stale resources, determining, 756
 - programmatically setting, 763–764
 - sending, 33
 - for static resources, 758
 - varying output caching by, 767
- HTTP modules, 646–647
 - built-in, 154
 - for current application, 154
 - custom modules, 151–154
 - Dispose* method, 151
 - events handled by, 149–154
 - events raised by, 119
 - functionality of, 119
 - Init* method, 151

HTTP modules (*continued*)
 loading and initialization of, 153
 order of application, 153
 pre- and post-processing of requests, 149
 registering, 37, 82, 153
 role management, 97
 system modules, inheritance of, 149
 <*system.webServer*> section and, 108–109
 URL routing engine, 119–120, 155, 157–159
 writing, 149–156

HTTP pipeline, 176. *See also* page life cycle
 activation of, 174
 HTTP requests, parsing to, 174
HttpRequest class, 174–176

HTTP protocol stack, 29

HTTP requests
 anonymous ID, 671
 asynchronous requests, 95
 authentication of, 650
 authorization of, 650
 client data, 671
 delays caused by, 59
 filtering, 155, 164
 free threads and, 86
 handling of, 645
 HTTP module processing of, 149
 IIS processing of, 30–37
 information about, encapsulation of, 656
 information about connection, 672
 information about request, 670–672
 input validation, 674
 logging, 33, 651
 out-of-band, 841–842
 output for, 122, 130–131
 parsing to HTTP pipeline, 174
 processing, 5, 9–10, 11, 27
 processing with IIS worker process, 29
 processing with standalone worker process, 28–29
 queuing, 95
 reducing number of, 316–317
 routing, 24
 saving to disk, 673
 script-led requests, 890
 sending, 844
 serving from cache, 650
 URL rewriting, 658–659

HTTP responses
 cache policy, 665–666
 compressed responses, 79
 encapsulation of, 663
HttpResponse object, 663–670
 large file transmission, 669
 response filters, 666–667

HTTP runtime
 page request processing, 174
 reason for processing pages, 209

HTTP verbs, 879

HttpApplication class
 events of, 648–651
 IIS pipeline events and, 34
 methods, 647–648
 properties of, 645–646

HttpApplication object, 177, 645
 Error event handler, 275–276
 events raised by, 119
 handling events from, 151–153
 pooling of, 645

HttpApplicationFactory, 175–177

HttpRequest class, 676
 methods of, 677–678
 properties of, 676–677
 synchronization of operations, 678–679

HttpBrowserCapabilities class, 77, 345

HttpBrowserCapabilitiesBase class, 345

HttpCachePolicy class, 666
 methods of, 763–764
 properties of, 763

HttpCapabilitiesDefaultProvider class, 347

HttpCapabilitiesProvider class, 78

HttpContext class, 656–660, 676
 methods of, 658–660
 properties of, 656–657

HttpContext object, role of, 645

HttpCookie object, 688

HttpCookieMode enumerated type, 688–689

<*httpCookies*> section, 84–85

HttpForbiddenHandler class, 143

<*httpHandlers*> section, 82
 handler information in, 177–178
 handlers list, 123

<*httpModules*> section, 82
 registering modules in, 153

HttpOnly cookie attribute, 84–85

httpOnlyCookies attribute, 84

HttpPostedFile class, 250

HttpRequest class
 methods of, 673–674

properties of, 670–673

HttpRequest object, 670–674

HttpResponse class
 methods of, 667–670
 properties of, 664–667

HttpResponse object, 663–670
BinaryWrite method, 135
 large file transmission, 669
 output caching, 669
 response filters, 666–667
WriteSubstitution method, 776

HttpRuntime class, 174–176
 public static methods, 174–175
UnloadAppDomain, 179

HttpRuntime object, 30

<*httpRuntime*> section, 85–87
maxRequestLength attribute, 251

HTTPS, Forms authentication with, 889

HttpServerUtility class
 methods of, 660–663
 properties of, 660

HttpRequest class, 676, 680
 methods of, 686
 properties of, 685–686
 synchronization mechanism, 685–686

HttpRequest object, creating, 682–683

http.sys, 29

HttpValidationStatus enumeration, 764

HttpWatch, 363

HyperLink control, 258
 customizing, 515–518
 hyperlink fields, 447–448

I

IAAsyncResult interface, 146

IButtonControl interface, 257, 374

ICollection interface, 412, 685

IComponent interface, 218

IControlBuilderAccessor interface, 218

IControlDesignerAccessor interface, 218

ICustomTypeDescriptor interface, 413

ID autogeneration mechanism, 220–222

ID selectors, 909

IDataBindingsAccessor interface, 218

identity providers, 823

- <identity> section, 87
 - worker process identity information, 784
- Identity Selector, 791
- IDictionary interface, 413
- IDisposable interface, 218
- idleTimeout attribute, 84
- IEditableTextControl interface, 260
- IEnumerable interface, 412, 417, 724
- ieSite.master file, 327
- IExpressionsAccessor interface, 218
- <iframe> tags, 929–930
- <ignoreDeviceFilters> section, 92
- IHttpAsyncHandler interface, 121, 146, 201
 - implementing, 201
- IHttpHandler interface, 11, 36, 121–127, 174
 - members of, 121–122
 - ProcessRequest method, 174
- IHttpHandlerFactory interface, 144, 177
- IHttpModule interface, 37, 149–150
 - module implementation of, 646
- IIS
 - administrator-level extensions, 37
 - ASP.NET applications, configuring for, 55–59
 - and ASP.NET, history of, 28–31
 - authentication tasks, 790
 - Classic mode, 31
 - handler mappings list, 177–178
 - HTTP request processing, 31–37
 - Integrated Pipeline mode, 30–31
 - ISAPI interface, 120
 - messaging pipeline, 32–35
 - new features, 37–39
 - process model and, 95
 - resources served by, 128, 140
 - runtime environment, 28
 - settings, propagating to destination, 48–49
 - unified architecture of, 30
 - warm-up feature, 59–62
 - worker process, 29
- IIS 7 integrated mode
 - HTTP handlers, registering, 132
 - HTTP handlers and modules and, 108–110
 - module and handler resolution, 119
- IIS Application Warm-up module, 59–62
- IIS Express, 25
- IIS kernel caching, 761–762, 766
- IIS Manager
 - Add Managed Modules dialog box, 37
 - applicationHost.config file, editing in, 93
 - Application Pool Identity dialog box, 39
 - Handler Mappings panel, 121
 - IIS Application Warm-up feature, 60
 - mapping HTTP modules, 37
- IIS metabase, 171
- IIS pipeline, 174
- IIS script maps, 127
- IIS SEO Toolkit, 351–352
- IIS Web projects, defined, 48
- IList interface, 412
- Image controls, 259
- image fields, 448–449
- image inlining, 316
- image URLs, 259
- ImageButton controls, 259
- ImageFormat structure, 137
- ImageMap controls, 259
- images
 - advertisements, 262–263
 - buttons, rendering as, 447
 - controlling with HTTP handler, 140–141
 - copyright notes, writing on, 137–140
 - databases, loading from, 133–136
 - display of, 259
 - display of, configuring, 252
 - dynamically generated, serving, 137
 - grouping into sprites, 315–316
 - hyperlinks displayed as, 258
 - minimizing impact of, 315–317
 - referencing on Web pages, 133
 - saving, 137
 - serving, 128–133
 - skin-compliant, 340
 - writing to disk, 137
- tag, 133
- immobility, 568
- impersonate attribute, 87
- impersonation
 - through anonymous account, 785
 - of fixed identities, 785
 - per-request, 785
 - thread identity and, 783–785
- @Implements directive, 189
- @Import directive, 188, 653–654
- imports, 585–587
- INamingContainer interface, 190, 221, 519, 558
- information hiding, 571–572
- inheritance
 - from base class, 515
 - prototype feature and, 849
- Inherits attribute, 96
- Init events, 210
- Init method, 149, 648
 - for custom HTTP modules, 151
- InitComplete events, 210
- initialization
 - of applications, 38, 645–651
 - of child controls, 210
 - completion of, 210
 - modules, 646–647
 - of page, 210
- initialization code, placement of, 905–906
- Initialize method, 808
- InitializeCulture method, 310
- initializeRequest event, 873
- inline content, 316–317
- in-memory generation of images, 137
- InnerException property, 276
- innerHTML property, 842–843, 920
- in-place precompilation, 53
- InProc session state, 682, 694–695, 705
- in-process calls, 603
- input controls
 - accessing, 375
 - multiple validators for, 390–391
 - and validation controls, linking, 381–382
 - Wizard control, 374
- <input> element, 246
- input field filters, 914
- input fields, validation of, 248
- input forms
 - cross-page posting, 374–379
 - HtmlForm class, 366–367
 - logical forms, 368
 - login capabilities, 368–369
 - multiple forms, 368–374
 - MultiView control, 373–374
 - mutually exclusive, 371–373
 - nesting, 370
 - parent objects, 367
 - single-form model, 365–366, 368

- input forms (*continued*)
 - unique name, 367
 - validation controls, 379–396
 - wizards, 397–409
- input tags and controls,
 - correspondence between, 212
- input validation, 674
 - validation controls, 379–396
 - in wizards, 404
- Insert* method, 726
- insert templates, 501–505
 - position of, 502–503
 - validation for, 504–505
- insertAfter* function, 921
- insertBefore* function, 921
- InsertItemPosition* property, 502
- InsertItemTemplate* property, 501
- installer files, deployment with, 42–43
- Integrated Pipeline mode (IIS), 30–31
- integrated Windows
 - authentication, 782
- integration testing, 566
 - defined, 49
- Interface Segregation principle, 579–580
- interfaces, 319
 - for custom controls, 519–520
 - defined, 189
 - implementing, 189
 - in MVP pattern, 625
- intermediate page classes, 13
- Internet Explorer
 - IE9 Developer toolbar, 317
 - JavaScript engine, 902
 - XMLHttpRequest* object
 - instantiation, 843
- Internet Information Services.
 - See IIS
- intrinsic objects, 191
- invalid input, 386
- inversion of control (IoC), 582
 - frameworks, 584
 - vs. MEF, 584–585
 - Unity, 587–592
- I/O bound operations, 208
- I/O threads, 94–95
- IPageableItemContainer* interface
 - definition of, 510
- IParserAccessor* interface, 218
- IPostBackDataHandler* interface, 211, 245, 519–520
 - implementing, 249
- IPostBackEventHandler* interface, 213, 520

- IRepeatInfoUser* interface, 430
- IRequiresSessionState* interface, 141
- IRouteHandler* interface,
 - definition of, 159
- ISAPI extensions, 170
- ISAPI filters for cookieless authentication, 800
- ISAPI interface, 120
- IsAuthenticated* property, 829
- IsCrossPagePostBack* property, 377–378
- ISessionIDManager* interface, 708
 - methods of, 709
- IsInPartialRendering* property, 862
- ISO/IEC 9126 paper, 565
- isolation, testing in, 641–642
- IsReusable* property, 121–122, 174
 - handler pooling, 143
- IStateManager* interface, 534, 711
- IsValid* property, 379, 387–388
- IsViewStateEnabled* property, 228
- ItemCanceling* event, 503
- ItemCommand* event, 476
- ItemDataBound* event, 491–492
- ItemEditing* event, 498
- ItemInserted* event, 504
- ItemPlaceholderID* property, 473
- Item* property, 723, 726
- item templates
 - selected item template, 505–507
 - setting up, 501–503
- ITemplate* type, 557
- Items* property, 423, 657
- ItemTemplate* property, 472, 483
- iterative controls, 411, 427–432
 - DataGrid* control, 431–432
 - DataList* control, 430–431
 - vs. list controls, 427–428
 - Repeater* control, 428–430
- ITextControl* interface, 260
- IUrlResolutionService* interface, 218

J

- Java Server Pages (JSP), 3
- JavaScript
 - AJAX and, 845–851
 - in browsers, 899–900
 - in client pages, 198
 - client-side code, 903
 - closures in, 847–848
 - drawbacks of, 902–903

- functional programming in, 845–846
- goals of, 900
- in HTML pages, 16
- initialization code, 905–906
- introduction of, 900
- JSON and, 890
- libraries, 899
- message boxes, 501
- for navigation systems, 352
- object model for, 903
- Object* type, 846
- OOP in, 846–847
- prototypes in, 848–849
- proxy class, generating, 893–894
- ready event, 906
- scripting engine, 901–902
- skills with, 3
- unobtrusive, 918
- writing and importing, 899
- JavaScript background compiler, 901
- JavaScript Object Notation.
 - See JSON
- JavaScriptSerializer* class, 890–891
- JetBrains ReSharper, 188, 270
- Johnson, Ralph, 575
- jQuery function, 905
- jQuery library, 18–19, 313, 422, 846, 877, 899, 903–905, 905–931
 - AJAX support, 925–928
 - benefits of, 904
 - binding and unbinding functions, 918–919
 - cache, 923–925
 - chainability, 909, 915
 - cross-domain calls and, 929–932
 - Data Link plug-in, 931
 - DOM manipulation, 920–923
 - DOM readiness, detecting, 906–907
 - DOM tree, adding to DOM, 920–922
 - DOM tree, creating, 920
 - filters, 911–914
 - functional programming and, 905
 - Globalization plug-in, 931
 - HTML, downloading, 928
 - IntelliSense support, 904
 - JSON, getting, 927–928
 - linking, 903
 - live binding, 919
 - load function, 907

jQuery library (*continued*)
 Microsoft support, 931
 naming convention, 904
 query capabilities, 908
 ready function, 906–907
 root object, 904–905
 selectors, 909–911
 Templates plug-in, 931
 UI, 18, 877
 wrapped sets, 905, 908–919
.js file, 903
/js suffix, 893
 JScript, 900
 JSON, 890–893
 DataContract attribute,
 891–892
 format, 890
 getting, 927–928
 vs. XML, 892–893
 JSON format, 20
 JSON object, 890
 JSON serialization, 108
 JSON with Padding (JSONP),
 929–931
json2.js file, 890

K

kernel caching, 58–59, 761–762,
 766
 enabling, 79
 key, cached item, 725
 key containers, 115
keywords attribute, 349
keywords meta tag, 349
 Kuhn, Thomas, 15

L

Label control, accessibility
 feature, 260
 LABjs, 313
 language, changing on the fly,
 310–312
language attribute, 654
 large file transmission, 669
 Law of Demeter, 377
 layers, 593
 business logic layer, 596–605
 data access layer, 605–614
 page and database coupling
 and, 762
 service layer, 602–603
 SoC and, 593
 in three-tiered architecture, 593

layouts, 320. *See also* master
 pages
 flow layouts, 485–487
 multicolumn layouts, 485
 tabular layouts, 480–485
 tiled layouts, 487–493
LayoutTemplate property, 472
legacyCasModel attribute, 103
length property, 909
 lengthy tasks, asynchronous
 operation for, 201–209
 lifetime managers, 590–592
LinkButton control, 213
 LINQ, 414–415
 LINQ-to-SQL, 458, 610–611
 linking to design patterns with,
 599
 Liskov's substitution principle,
 576–578
 list controls, 411, 421–427
 BulletedList control, 426–427
 CheckBoxList control, 422–424
 DataTextField property, 418
 DataValueField property, 419
 DropDownList control, 421–422
 Items property, 423
 vs. iterative controls, 427–428
 ListBox control, 425
 RadioButtonList control, 424–
 425, 427
 receiving data, 545
 SelectedIndexChanged event,
 425
 ListBox control, 425
 HTML server control for, 245
 ListControlClass, 514
 listeners, 101
ListItemCollection class, 423
ListView control, 433–434,
 471–512
 alternate item rendering,
 483–484
 buttons in layout, 496–511
 CSS styling, 480, 482, 494–497
 data-based operations, 496
 data binding, 471, 477–479
 data-binding properties, 474
 data-related properties,
 492–493
 editing, in-place, 496–499
 events of, 474–476, 498–499
 flow layout, 485–487
 vs. *GridView* control, 482–483
 ItemTemplate property, 472
 layout template, 479
 LayoutTemplate property, 472
 list layout, 479–480

new data items, adding,
 501–505
 object model, 472–479
 vs. other view controls, 476–477
 paging capabilities, 507–512
 populating dynamically,
 491–493
 Predictable algorithm
 implementation, 225
 properties of, 472–474
 rendering capabilities, 471
 selecting items, 505–507
 sorting items, 511
 style properties, 494
 styling the list, 493–496
 tabular layout, 480–485
 template properties, 473, 480
 tiled layout, 487–493
 updates, conducting, 499–501
 user-interface properties, 474
 literal controls, 252
 literals, 260
 live binding, 919
 live function, 919
Load events, 211
load function, 907, 928
LoadComplete event, 213
LoadControl method, 197
LoadControlState method, 719
loadFromCache function, 925
loadFromSource function, 925
LoadPageStateFromPersistence-
 Medium method, 215
LoadPostData method, 211
LoadTemplate method, 197
LoadViewState method, 540
 overriding, 539
 Local Security Authority (LSA), 88
 local storage, 923
 local themes, 339
 local users, error pages for,
 273–274
 localization, 303–312
 culture, changing, 310–312
 culture, setting in .NET,
 308–309
 culture, setting in Web Forms,
 309
 global resources, 307
 localized text, 306–307
 local resources, 304–305, 308
 of navigation system, 360
 resources, localizable, 304–308
 of script files, 314–315
 of site map information,
 359–361

<location> section, 68–71
allowLocation attribute, 71
allowOverride attribute, 70–71
Path attribute, 69–70
Locator attribute, 50
locking mechanisms on *Cache* object, 736
logging
 of HTTP requests, 651
 of request results, 33
logging exceptions, 277
logic of pages, testing, 361
logical forms, 368. *See also* input forms
login capabilities of input forms, 368–369
Login control, 826–828
 appearance of, 827
 customizing, 827
 events of, 828
 membership API and, 826
login pages, 792–793
 for AJAX-enabled pages, 887–888
 credentials, collecting through, 794
 in external applications, 803
 layout of, 826
 login templates, 831
LoginName control, 828–829
LoginStatus control, 829–830
 properties of, 830
LoginView control, 830–832
 properties of, 830
LogRequest event, 33, 651
low coupling, interface-based programming, 572

M

machine scope, 119
machine.config file
 for configuration, 63
 configuring, 65
 default modules list, 647
 <location> element, 68, 69
 location of, 64
 <processModel> section, 92
machine.config.comments file, 64
machine.config.default file, 64
 <*machineKey*> section, 87–88
machine-level configuration settings, 70, 111
MailDefinition element, 832–833
maintainability, 565
 importance of, 569
maintenance, 565

Managed Extensibility Framework (MEF), 584–587
managed HTTP modules, defined, 37
MapPageRoute method, 161
mappedUrl attribute, 104
mappings
 between fake URLs and real endpoints, 104
 between HTTP handlers and Web server resources, 123
 between properties and section attributes, 117
 between security levels and policy files, 97
MapRequestHandler event, 32, 650
markup
 ASPX templates, 217
 CSS-friendly, 232–234
 generating, 194–195, 215
 graphical aspects, configuring, 422
 server-side programming, 839
 style-ignorant, 255
 writing, 520–521
 writing to HTML text writer object, 527–528
markup files (ASPX), compilation of, 52
markup mix, 3
Martin, Robert, 573
Master attribute, 327
 @*Master* directive, 180, 320–323
 attributes of, 322
master page properties, 333
 exposing, 333
 invoking, 334–335
master pages, 180, 319–324
 ASPX markup for, 222, 225
 binding content pages to, 326
 changing, 209
 compiling, 329
 ContentPlaceHolder controls, 320–322
 default content, defining, 323–324
 device-specific, 327–329
 dynamic changes to, 336–337
 @*Master* directive, 320–323
 nested forms and, 370–371
 nesting, 330–333
 processing, 329–334
 programming, 333–336
 sample page, 321
 UpdatePanel controls in, 864–865

Master property, 333, 336
MasterPage class, 321, 329
MasterPageFile attribute, 326, 327, 336–337
@*MasterType* directive, 335–336
max-age HTTP header, 756
maxBatchGeneratedFileSize attribute, 169
maxBatchSize attribute, 169
maxPageStateFieldLength attribute, 90
maxRequestLength attribute, 251
MEF, 584–587
membership API, 88, 806–821
 data stores, integrating with, 812
 login controls and, 826
 Membership class, 807–812
 membership provider, 812–817
Membership class, 806–812
 advanced functionality of, 808
 methods of, 808–809
 properties of, 807–808
 membership providers, 300, 812–817
 choosing, 809
 configuration settings, 88–89
 configuring, 816–817
 custom, 815–816
 extending, 815
 key issues of, 816
 list of, 808
 MembershipProvider base class, 813–814
 ProviderBase class, 813
 <*membership*> section, 88–89
MembershipProvider class, 813–814
MembershipUser object, 811–812
Memcached, 614, 753–754
memory usage, polling for, 79
MergeWith method, 255
meta tags, 349
MethodName property, 775
methods
 of *Control* class, 228–229
 page methods, transforming into, 895–896
 testing, 641
 of Web controls, 255–256
Meyer, Bertrand, 575
Microsoft Dynamics CRM, 613–614
Microsoft Internet Explorer. *See* Internet Explorer
Microsoft Internet Information Services. *See* IIS

Microsoft .NET Framework. *See* .NET Framework

Microsoft Silverlight. *See* Silverlight

Microsoft SQL Express, 284

Microsoft SQL Server. *See* SQL Server

Microsoft Visual Basic, 3

Microsoft Visual Studio. *See* Visual Studio

Microsoft.Practices.Unity assembly, 587

Microsoft.Practices.Unity.Configuration assembly, 590

Microsoft.Web.Administration assembly, 112

Microsoft.XmlHttp object, 843

MigrateAnonymous event, 299–300

minifiers, 314

MobileCapabilities class, 77

mocks, 640

mode attribute, 74–75

Mode attribute, 81

model

- defined, 616
- role in MVC, 617
- role in MVP, 620

Model-View-Controller, 616–618, 620

Model-View-ViewModel, 615, 621–623

mod_mono module, 27

modular code, 571–572

Module Pattern, 849

modules, 646–647

Modules property, 154, 646

MongoDB, 614

Moq, 640

.*msi* files, 43

MSUnit, 638

multicolumn layouts, 485

multipart/form-data submissions, 249–251

multiserver environments

- <*machineKey*> settings, 88

multitiered architecture, 594–595. *See also* three-tiered architecture

MultiView control, 266–268, 373–374

MVC pattern, 616–618

- vs. MVP pattern, 620

MVP pattern, 14, 619–621

- implementing, 623–636
- interface, 625
- vs. MVC pattern, 620

- navigation, 632–636
- presenter, creating, 626–632
- presenter isolation, 641
- testability of code, 636–642
- view, abstracting, 624–626

MVVM pattern, 615, 621–623

myHandler function, 930

N

name conflicts, avoiding, 221

NameObjectCollectionBase class, 154, 676

namespaces, linking to pages, 188

<*namespaces*> section, 91

naming containers, 221, 226–227

NamingContainer property, 226–227

native requests, defined, 37

NavigateUrl property, 517

navigation

- implementing, 632–636
- linear and nonlinear, 397
- through wizards, 405–409

navigation system, 351–357

- localizing, 360
- SiteMap* class, 355–356
- site map configuration, 357–360
- site map information, 352–353
- SiteMapPath* controls, 356–358
- site map providers, 354–355

navigation workflow, defining, 633–634

nesting

- forms, 370–371
- master pages, 330–333

.NET Framework

- ASP.NET modules, 646–647
- Code Contracts API, 578
- configuration scheme, 63
- culture, setting, 308–309
- Dynamic Language Runtime component, 335
- exception handling, 270–272
- graphic engine, 137
- Managed Extensibility Framework, 584–587
- predefined server controls, 236

.NET Framework root folder, 786

NetFrameworkConfigurationKey container, 115

Netscape, 900

NETWORK SERVICE account, 38, 783

- privileges of, 785–786

NHibernate, 612

- None* authentication, 789

NorthScale Memcached Server, 755

NoSQL solutions, 614, 745

Nothing permission set, 103

NotifyDependencyChanged method, 738, 741

null identities, 32

null reference exceptions, 270

numRecompilesBeforeAppRestart attribute, 56

O

object caching, 463

object model. *See also* DOM

- defined, 600
- updatable, 842

<*object*> tags, server-side, 654–655

Object type, 846

ObjectCreating event, 463

ObjectDataSource control, 458, 459–469

- caching support, 463
- data retrieval, 460–461
- deleting data, 465–468
- existing business and data layers and, 463
- paging, setting up, 464–465
- properties of, 459–460
- updating data, 465–468

ObjectDisposing event, 464

object-oriented design, 599

object-oriented programming. *See* OOP

object/relational mapper, 610–613

OCP, 575–576

omitVaryStar attribute, 79

OnClickClick property, 258

OnCompletionCallback method, 147

one-click attacks, 780

- security against, 193

onload event, 906

- order of running, 907

OOP, 4, 569

- in JavaScript, 846–847
- substitution principle and, 576

Open Authorization (oAuth), 20

Open Data (oData), 20

Open/Closed Principle (OCP), 575–576

OpenID, 76

OpenMachineConfiguration method, 112

OpenWebConfiguration method, 111

OperationContract attribute, 884

optimized internal serializer, 696

originUrl attribute, 102, 787

O/RM, 610–613

- Code-Only mode, 611
- Entity Framework, 611–612
- Linq-to-SQL, 610–611
- NHibernate, 612
- SQL code of, 612

out-of-band HTTP requests, 841–842

output cache

- execute now capabilities, 32
- saving pages to, 33

output cache profiles, 79–80

output cache providers, 79, 776–777

output caching, 28, 33, 58–59, 669, 721, 755–777. *See also* caching

- caching profiles, 774–775
- capabilities of, 758–759
- configuration settings, 79
- configuring, 58
- dependency object, adding, 762–763
- IIS kernel caching, 761–762
- of multiple versions of page, 765–768
- @*OutputCache* directive, 759–760
- page output duration, 761
- page output location, 760–761
- of portions of page, 768–774
- postbacks, dealing with, 766
- post-cache substitution, 775–776
- server-side caching, 761
- sharing output of user controls, 772–773
- of static vs. interactive pages, 766
- of user controls, 770–775
- validation of page, 764–765
- varying by control, 770–772
- varying by custom string, 767–768
- varying by headers, 767
- varying by parameters, 765

@*OutputCache* directive, 759–760

- CacheProfile* attribute, 774
- Shared* attribute, 773
- VaryByCustom* attribute, 767–768

VaryByParam attribute, 765–766

<*outputCacheProfiles*> section, 774

OutputCacheProvider class, 79

overloaded constructors, 583

P

packaging

- files, 43–51
- settings, 43–51

page caching, 665–666. *See also* caching

- output caching, 669

Page class, 36, 119, 190–208

- AddOnPreRenderCompleteAsync* method, 202–203
- Async* attribute, 201–202
- context properties, 193–194
- controls-related methods, 195–197
- Dispose* method, 215
- eventing model, 199
- events of, 198–199
- intrinsic objects, 191
- LoadPageStateFromPersistence-Medium* method, 215
- as naming container for controls, 190
- ProcessRequest* method, 210
- rendering methods, 194–195
- SavePageStateToPersistence-Medium* method, 215
- script-related methods, 197–198
- ViewStateUserKey* property, 192–193
- worker properties, 191–193

page classes

- derived, 12
- intermediate, 13

page composition, 319–345

- content pages, processing, 329–334
- content pages, writing, 323–328
- master pages, 320–324
- master pages, processing, 329–334
- master pages, programming, 333–336
- styling pages, 336–344

page controller entities

- code-behind classes, 12–13. *See also* code-behind classes
- implementation of, 5

Page Controller pattern, 11–14, 156, 618

- effectiveness of, 14
- HTTP handler components, 11
- revisiting, 14
- server-side definitions, 5

page development

- error handling, 269–285
- page localization, 303–312
- page personalization, 285–303
- resources, adding to pages, 312–317

@*Page* directive, 180–185

- Async* attribute, 201–202
- configuration settings, 89–92
- EnableViewStateMac* attribute, 713, 715
- page behavior attributes, 182–184
- page compilation attributes, 181–182
- page output attributes, 184–185

page execution

- external page, embedding, 661–662
- helper methods, 660–663
- server-side redirection, 663
- page handler factory, 177–179
- page life cycle, 11–12, 119, 169, 174, 209–215
- in ASP.NET MVC, 22
- client data, processing, 211
- finalization, 214–215
- InitComplete* events, 210
- Init* events, 210
- LoadComplete* event, 213
- Load* events, 211
- managing, 177
- markup, generating, 215
- Page* class events, 198–200
- postback, 212–213
- PreInit* event, 209
- PreLoad* event, 211
- PreRenderComplete* event, 214
- PreRender* event, 214
- restructuring, 14
- setup, 209–212
- Unload* event, 215

page localization, 185, 303–312

page methods, 895–897

- objects accessible from, 897

page objects, creation of, 178

page output, dependency object, 762–763

page personalization, 285–303

- page processing. *See also* partial rendering
 - browser-led model, 840–841
- page requests
 - context information, 175
 - processing, 174–179
 - reason for processing, 209
- page usability, 344–364
- PageAsyncTask* object, creating, 201
- pageBaseType* attribute, 90
- PageHandlerFactory* class, 144, 177–178
- Page_Load* event
 - profile properties, initializing, 295
 - presenter instance in, 626–627
- pageLoaded* event, 873
- pageLoading* event, 873
- PageMethods* class, 896
- page-output caching. *See* output caching
- PageRequestManager* client object, 872
- PagerSettings* class, 441
- pages, 6
 - advertisement banners on, 262–263
 - asynchronous pages, 121, 201–209
 - batch mode compilation, 169
 - behavior of, 757
 - cacheability of, 757, 758–762
 - content pages. *See* content pages
 - culture, setting, 309
 - debugging, 284–285
 - download experience, optimizing, 312–317
 - dynamic compilation, 52
 - embedding external pages in current page, 661–663
 - error handling for, 272–278
 - errors, mapping to, 278–282
 - header information, 241
 - heaviness of, 10, 17
 - HTML forms on, 200
 - initialization code, 905–906
 - interaction model, 17
 - invoking, 170–173
 - layout of, 320. *See also* master pages
 - life cycle of. *See* page life cycle
 - master pages. *See* master pages
 - namespaces, linking to, 188
 - page behavior, 182–184
 - partial rendering, 865. *See also* partial rendering
 - passing values between, 379
 - performance and quality analysis, 317
 - placeholders on, 265–266
 - postbacks. *See* postbacks
 - processing directives, 179–190
 - protecting access to, 784
 - recompilation of, 170
 - rendering, 214–215
 - requested, representation of, 172–173
 - resources, adding, 312–317
 - run-time modules, 170–171
 - script files, linking to, 312–314
 - scripts, adding to, 858–859
 - serving to user, 329–330
 - sharing of output, 773
 - single server-side form support, 200
 - size thresholds, 714
 - styling, 336–344
 - testing for usability, 361–364
 - themes, applying to, 340–341
 - titles of, 348
 - unloading, 215
 - updating asynchronously, 16
 - usability, 319, 344–364
 - user controls, converting to, 769
 - user controls, linking to, 189–190
 - view state, enabling and disabling, 227
 - view state, persisting, 200
 - XML documents, injecting into, 264–265
- Pages-for-Forms model, 16–17
- `<pages>` section, 89–92
 - child sections, 91–92
 - controlRenderingCompatibilityVersion* attribute, 232
- paging data, 451–453
 - setting up for, 464–465
- paradigm shifts, 15
- parameters, varying output caching by, 765
- ParseControl* method, 197
- parser errors, 269
- partial caching, 768–774
- partial classes, 173
 - code-behind classes, 173
- partial rendering, 19–20, 851–879
 - asynchronous postbacks, concurrent, 877–878
 - benefits and limitations of, 876–877
 - error handling, 857
 - example of, 861
 - migrating pages to, 865
 - polling and, 872, 878–879
 - vs. postbacks, 860, 878–879
 - postbacks, detecting from child controls, 866–868
 - postbacks, triggering, 868–869
 - refresh conditions, 866
 - script loading, 858–859
 - ScriptManager* control, 852–860
 - ScriptManagerProxy* control, 857–858
 - UpdatePanel* control, 860–866
- partial trust permission set
 - changing name of, 103
- PartialCaching* attribute, 770
- partial-trust applications, 103
- partition resolvers, 704
- partitioned cache topologies, 746
- partitioned cache with H/A, 746
- Passport authentication, 76
- PasswordRecovery* control, 832–833
- passwords
 - changing, 833–834
 - managing, 812
 - retrieving or resetting, 832–833
- Path* attribute, 69
- path* property, 895
- pathInfo* placeholder, 165
- patterns. *See* design patterns
- Patterns & Practices group, 587
- pending operations, canceling, 876
- percentagePhysicalMemoryUsedLimit* attribute, 78
- performance
 - analyzing pages for, 317
 - caching and, 733, 759, 761
 - closures and prototypes and, 849
 - DHTML and, 843
 - download experience, optimizing, 312–317
 - exception handling and, 270
 - nesting pages and, 330
 - site precompilation and, 52
 - Substitution* control calls and, 776
 - view state size and, 713–715
- PerformDataBinding* method, overriding, 535, 540
- permission sets, configuring, 103

- permissions, for applications, 788–789
- PermissionSetName* attribute, 103
- per-request impersonation, 785
- PersistenceMode* attribute, 558
- personalization, 285–303
 - anonymous user information, migrating, 299–300
 - anonymous user profiles, 294–295
 - interaction with page, 292–300
 - personalization events, 298–299
 - profile database, creating, 292–294
 - profile providers, 300–303
 - user profiles, creating, 285–292
 - user-specific information, 299
- Personalize* event, 298
- picture viewer handler, 128–133
- PictureViewerInfo* class, 129
- pipeline events, wiring up, 151–153
- PipelineRuntime* objects, invoking, 34
- Placeholder* controls, 265–266, 557
- placeholders, 320–321, 323, 333
 - contents of, 326
 - default content, 323–324
- POCO code generator, 601
- policy files and security levels, mappings between, 97–98
- polling, 739, 742
 - partial rendering and, 872, 878–879
 - timer-based, 878–879
- pollTime* attribute, 80
- port 80, 27
- port numbers, changing, 100
- positional filters, 911
- PostAcquireRequestState* event, 33, 650
- PostAuthenticateRequest* event, 32, 650
- PostAuthorizeRequest* event, 32, 650
- postback controls, 387
- postBackElement* property, 873
- postbacks, 4–6, 766
 - destination of, 365
 - detecting from child controls, 866–868
 - full, 869–870
 - handling, 5, 212–213
 - HTML server controls for, 244
 - partial rendering and, 860, 865
 - replacing with AJAX calls, 10
 - via scripts, 374
 - SEO and, 350
 - through submit buttons, 374
 - to target page, 374–376. *See also* cross-page posting
- Timer* control for, 878–879
- triggering, 258, 868–869
- user interface, disabling, 874–875
- PostBackTrigger* object, 870
- PostBackUrl* property, 374
- post-cache substitution, 775–776
- posted data
 - processing, 211
 - retrieving, 369
 - saving, 250–251
 - testing, 363
- posted names, matching to
 - control IDs, 211
- posting acceptor, 250
- PostLogRequest* event, 33, 651
- PostMapRequestHandler* event, 32, 650
- POST* method, 365
 - posting forms with, 367
- PostReleaseRequestState* event, 33, 651
- PostRequestHandlerExecute* event, 33, 650
- PostResolveRequestCache* event, 32, 155–156, 650
- PostUpdateRequestCache* event, 33, 651
- precedence of themes, 341
- precondition attribute, 109, 126
- preconditions, 578
- predictable IDs, 91
- PreInIt* events, 209
- PreLoad* event, 211
- Preload* method, 62
- prependTo* function, 921
- PreRender* event, 214, 230
- PreRenderComplete* event, 214
 - asynchronous handlers for, 201
 - Begin/End* asynchronous handlers for, 201
 - page processing, 202, 203
- PreRequestHandlerExecute* event, 33, 650
- PreSendRequestContent* event, 33, 150
- PreSendRequestHeaders* event, 33, 150
- presentation layer, 269, 593, 596. *See also* pages in AJAX, 880
- DAL, invoking from, 608
- design patterns for, 615–623
- navigation workflow, binding to, 636
- presentation logic, 217
- Presentation Model (PM) pattern, 621
- presenter
 - creating, 626–632
 - data retrieval, 628–629
 - defined, 619
 - instance of, getting, 626–627
 - navigation and, 632–636
 - Refresh* method, 628
 - role in MVVM pattern, 622
 - role of, 621
 - service layer, connecting to, 629–630
 - sharing with Windows applications, 631–632
 - testing, 639–642
 - testing in isolation, 641–642
 - __PREVIOUSPAGE hidden field, 374
- PreviousPage* property, 375
- @*PreviousPageType* directive, 376–377
- principal objects, custom, 804–806
- priority, cached item, 726, 730–731
- privateBytesLimit* attribute, 78
- process model
 - configuration settings, 92–95
 - IIS integrated mode and, 95
 - optimizing, 94
- process recycling, 28, 55–56
 - configuration file changes and, 65
 - event log entries for, 57
 - unexpected restarts and, 56–58
- processing, 14
 - separating from rendering, 9–10
- processing directives, 179–190
 - @*Assembly* directive, 185–187
 - @*Implements* directive, 189
 - @*Import* directive, 188
 - @*Page* directive, 181–185
 - @*Reference* directive, 189–190
 - syntax, 180
 - typed attributes, 180–181
- <*processModel*> section, 92–95
- processRequestInApplicationTrust* attribute, 102

ProcessRequest method, 36, 121–122, 138, 210
HttpRuntime, 174–175
IHttpHandler, 174, 178, 190
 profile API, 108
 access layer, 300
 storage layer, 300
 for Web site projects, 286–287
 profile class, 291–292
 defining, 287–289
Profile property, 288, 295
 attributes of, 96
 profile providers, 300–303
 configuring, 300–302
 connection strings, 301
 custom, 302–303
 functionality of, 96
 SQL Express, 286
 <profile> section, 96–97, 286–287
ProfileEventArgs class, 299
ProfileModule, 298–299
Programming Microsoft ASP.NET MVC (Esposito), 26, 268
ProgressTemplate property, 871
 properties
 of *Control* class, 218–228
 defined, 413
 of Web controls, 253–254
 <properties> section, 286
 property values, varying output
 caching by, 770
 protected members, 173
 protection providers, choosing, 115–116
ProtectSection method, 114
 protocol stack, 29
 prototype object, 848–849
 prototype property, 847
 prototypes, 848–849
Provider property, 808
ProviderBase class, 813
providerName attribute, 106
 providers
 browser-capabilities providers, 346–348
 browser providers, 77, 78
 defined, 300
 encryption providers, 107, 115–116
 membership providers, 88
 output cache providers, 79
 profile providers, 96, 300–303
 registering, 347
 role providers, 97
 site map providers, 100, 354–355
 store providers, 99

Providers property, 815
 <providers> section, 99, 301
 proxy cache, 755–756
 proxy classes, *PageMethods* class, 896
 proxy methods, JavaScript, 893–895
 public methods, invoking, 886

Q

queries
 filter function, 914
 filters, 911–914
 find function, 914
 selectors, 909–911
 visibility operators, 915–917
 query results. *See* wrapped sets
 queryable objects, 414–415

R

RAD, 4, 437
 designer data bindings, 218
 paradigm, 569, 615
 RadioButtonList control, 424–425, 427
 radio buttons, 259–260
RaisePostBackEvent method, 213, 387
RaisePostDataChangedEvent method, 212, 245
 Random Number Generator (RNG) cryptographic provider, 687
RangeValidator control, 380, 386
 Rapid Application Development. *See* RAD
 Raven, 614
 raw data, passing, 17
 reader/writer locking mechanisms, 683–684
 reading methods, synchronization mechanism, 678
ready function, 906–907
 multiple calls to, 907
 order of running, 907
readyState property, 906
readyStateChange event, 906
 reauthorization, 663
 Red Gate SmartAssembly, 284
redirect attribute, 81
RedirectFromLoginPage method, 794, 796
 redirects, 634–635
RedirectToLoginPage method, 796
 @*Reference* directive, 189–190
 refreshing. *See* updating
 conditional, 866–870
RegisterAsyncTask method, 206–208
RegisterHiddenField method, 261
RegisterInstance method, 588
RegisterRoutes method, 160
RegisterType method, 587
RegisterXXX methods, 855–856
 regular expressions, validating, 385
RegularExpressionValidator control, 380, 385
 release script files, 859–860
ReleaseHandler method, 145
ReleaseRequestState event, 33, 650
 Remote Scripting (RS), 841
 removal callbacks, 726
 defining, 729–730
Remove method, 727
Render method, 231, 520–521
 overriding, 528
RenderControl method, 230
Renderer class, 575
 rendering
 browser-sensitive, 234–235
 child controls for, 528–532
 cross-browser rendering, 344–348
 custom controls, 520–522, 527–533
 legacy and CSS-friendly modes, 232–234
 separating from processing, 9–10
 SimpleGaugeBar control, 527–533
 templates, 560–561
 rendering engine, entry point into, 231
 rendering methods, 194–195
RenderingCompatibility property, 233
Repeater control, 221, 416, 428–430
RepeaterItem class, 429
RepeaterItemCollection class, 429
 replay attacks, 803, 804
 replicated cache topologies, 746
Repository pattern, 609–610
 Representational State Transfer (REST), 879–897
 request handlers, determining, 32

- request life cycle, 22
 - events in, 32–34
 - handlers, writing, 36–37
 - Request* object *Browser* property, 77
 - request* property, 873
 - request routing, 119–120
 - Request.Headers* collection, 275
 - RequiredFieldValidator* control, 380, 382, 386–387
 - RequireJS*, 313
 - requirements churn, 567
 - RequirementsMode* property, 889
 - requireSSL* attribute, 84, 803–804
 - reset buttons, 247
 - ResetPassword* method, 812
 - Resolve* method, 588–589
 - ResolveRequestCache* event, 32, 650
 - resolver types, 103
 - resource assemblies, defined, 304
 - resources
 - adding to pages, 312–317
 - custom resources, 126–127
 - declarative vs. programmatic, 306
 - defined, 304
 - embedding, 195
 - forbidden, preventing access to, 143
 - global resources, 304–305, 307
 - lifetime of, 648
 - localizable, 304–308
 - local resources, 304–305, 308
 - mapping to handlers, 171–172
 - retrieving, 659–660
 - served by IIS, 128
 - \$Resources* expression builder, 307
 - \$Resources* expressions, 359
 - ResourceUICultures* property, 315
 - response bodies, sending, 33
 - response filters, 651, 666–667
 - Response.Redirect* method, 275, 663
 - REST, 879–897
 - consuming, 893
 - HTTP verbs and, 879
 - JavaScript proxy for, 893–895
 - webHttpBinding* model, 883
 - RESX files, 304
 - culture information in, 308
 - editing, 306
 - site map information in, 359–360
 - ReturnUrl* parameter, 799
 - RewritePath* method, 104, 158, 658–659
 - Rich Internet Application (RIA) services, 20
 - rigid software, 567–568
 - Rijndael encryption, 799
 - role management, 97, 817–821
 - LoginView* control, 830–832
 - role management API, 108, 817–819
 - role manager HTTP module, 820
 - role manager providers, 300
 - <roleManager>* section, 97
 - RoleProvider* class, 820–821
 - role providers, 820–821
 - built-in, 821
 - custom, 821
 - role management, 97
 - roles
 - defined, 817
 - login templates based on, 831–832
 - securing access with, 358
 - Roles* class, 806, 807, 819–820
 - methods of, 819
 - properties of, 820
 - Route* class, 160
 - route handlers, 159
 - tasks of, 156
 - route parameters, 159
 - accessing programmatically, 162
 - RouteData* property, 162
 - routes, 158–159
 - default values, 162
 - defining, 160
 - HTTP handler for, 155
 - processing order, 162
 - storing, 160
 - structure of, 163–164
 - RouteTable* class, 160
 - RouteValueDictionary* type, 163
 - routing API, 104
 - RowUpdating* event, 443, 455
 - RSA-based protection provider, 107, 115
 - runAllManagedModulesForAllRequests* attribute, 109
 - runat* attribute, 7, 197, 217
 - runat="server"* attribute, 200
 - for HTML controls, 235
 - for Web controls, 253
 - runManagedModulesForWebDavRequests* attribute, 109
 - runtime compilation, 52
 - runtime environment, 27
 - ASP.NET MVC, 22–24
 - asynchronous handlers, dealing with, 146–147
 - configuration settings for, 71–73, 85–87, 89–92
 - of early ASP.NET, 28–29
 - of early IIS, 29
 - of IIS, 30
 - IIS 5.0, 28
 - Windows 2000, 28
 - runtime errors, 269. *See also* error handling
 - runtime event handling, 119. *See also* HTTP handlers
 - runtime page processing modules, 170–173
- ## S
- Same Origin Policy (SOP), 850, 881, 929
 - sandboxing, 789
 - SaveAs* method, 251
 - SaveControlState* method, 719
 - SavePageStateToPersistence-Medium* method, 215
 - SaveViewState* method, 214
 - overriding, 539
 - saving posted files, 250
 - scalability, 744. *See also* distributed cache
 - ScaleOut StateServer, 755
 - scavenging, 731
 - schema-less storage, 614
 - scope
 - application scope, 119
 - machine scope, 119
 - script code
 - emitting in client pages, 198
 - for page postbacks, 213
 - script files
 - aggregating, 315
 - embedded vs. inline, 316–317
 - linking to pages, 312–314
 - localized, 314–315
 - minifying, 314
 - moving to bottom of page, 312–313
 - script interceptors, 20
 - script maps, 127
 - script resource management, 852
 - <script>* tags, 312, 858, 929–930
 - defer attribute, 313
 - scriptable services, 880–889
 - scripting engines, 901–902
 - script-led requests, JSON for, 890–893

- ScriptManager* control, 315, 851, 852–860
 - events of, 856
 - on master pages, 865
 - methods of, 854–855
 - properties of, 852–854
- ScriptManagerProxy* control, 852, 857–858, 865
- ScriptModule* HTTP module, 897
- ScriptReference* class, 859
- ScriptResource.axd* HTTP handler, 859
- scriptResourceHandler* element, 107
- scripts
 - composite, 859
 - debug files, 859–860
 - globalization, 860
 - loading, 858–859
 - Page* class methods related to, 197–198
 - postbacks via, 374
 - release files, 859–860
- Scripts* collection, 858
- ScriptService* attribute, 886
- search
 - for files, 130
 - on input forms, 368–369
- search engine optimization. *See* SEO
- search engines, expressive URLs and, 156
- `<section>` element, 67
- section handlers, specifying, 116
- `<sectionGroup>` element, 67–68
- SectionInformation* object
 - ProtectSection* method, 114
 - UnprotectSection* method, 114
- Secure Sockets Layer (SSL), 782
 - authentication tickets, securing with, 803–804
- secured sockets, authentication over, 803–804
- security
 - application trust levels and, 786–789
 - ASP.NET pipeline level, 781, 784
 - ASP.NET security context, 781–791
 - authentication methods, 789–791
 - claims-based identity, 821–825
 - cookieless sessions and, 690–691
 - default ASP.NET account, changing, 784–786
 - error handling, 81
 - filtering user input, 135
 - Forms authentication, 791–806
 - HTTP error handling and, 281
 - HttpOnly* attribute and, 85
 - IIS level, 781–783
 - input validation, 674
 - JavaScript callers and, 880
 - membership API, 806–821
 - planning for, 779
 - role management, 817–821
 - server controls for, 825–835
 - of session state data, 699
 - threats, types of, 779–780
 - trust level and policy file mappings, 97–98
 - of view state, 192–193, 712–713
 - worker process level, 781, 783–784
- Security Token Service (STS), 824
- security trimming, 358
 - implementing, 354
- security zones, 786
- `<securityPolicy>` section, 97–98
- Select* buttons, 505–506
- selected item templates, 505–507
- SelectedIndexChanged* event, 425
- SelectedItem* property, 424
- SelectedItemTemplate* property, 505
- selection in *ListView* control, 505–507
- selective updates, 19–20
- selectors, 909–911
 - compound, 910–911
- SelectParameters* collection, 462
- Selenium, 363
- self-logging exceptions, 284
- semi-dynamic content, caching, 58
- sendCacheControlHeader* attribute, 79
- sensitive data, securing, 780
- SEO, 348–351
 - ASP.NET Web Forms and, 350–351
 - cookieless sessions and, 691
 - measuring, 351–352
 - meta tags, 349
 - page titles, 348
 - query strings, 349
 - redirects, 349
 - Server.Transfer* and, 275
 - subdomains, 349
- separation of concerns (SoC). *See* SoC
- serialization
 - of session state, 695–697, 710
 - XML format, 890
- server attacks, 779
- server caching, 761
 - post-cache substitution and, 776
- server controls, 4–5, 7–8. *See also* Control class; controls
 - adaptive rendering, 230–232
 - in AJAX, 267–268, 851
 - browser-sensitive rendering, 234–235
 - control containers, 226–227
 - control state, 214, 718–719
 - CSS-friendly markup, 232–234
 - ctIXX* string IDs, 223
 - custom controls, 513–562
 - data-bound, 411. *See also* data binding
 - data source controls, 456–468
 - HTML and CSS friendly, 337
 - HTML server controls, 217, 235–252
 - identifying, 220–226
 - instances of, 172
 - literal controls, 252
 - name conflicts, avoiding, 221
 - naming containers, 221
 - programming, 7
 - RAD designer data bindings, 218
 - role of, 217
 - security-related, 825–835
 - skins for, 220, 340–342
 - Static* option, 224
 - template definitions for, 340
 - themes, 220, 235, 337, 340–341
 - validation of, 381–382
 - view state, 227–228
 - view state, enabling or disabling for, 715–717
 - visibility of, 228
 - Web controls, 217, 253–268
- server forms, 365
- Server* object, 660–663
- server processes, memory limits, 94
- server transfers, 378–379
- server variables, 673
- ServerChange* event, 245, 248–249
- ServerClick* event, 247
- servers
 - machinewide settings, 70
 - view state, storing on, 719–720
- server-side controls
 - runat=server* attribute, 200
 - view state information, 200
- server-side events, 212–213

server-side expressions, syntax

server-side expressions, syntax, 690

server-side forms, 240

server-side *<form>* tags, single, 200

server-side handlers, 6

server-side programming, 839

server-side redirection, 663

server-side validation, 387–388
in wizards, 405

Server.Transfer method, 275, 378

ServerValidate event, 384

service layer
defined, 602
methods in, 604
presenter, connecting to, 629–630

Service Layer pattern, 602

Service Locator pattern, 582

services, scriptable, 880–889

session cookies, 687–688. *See also* cookies

session hijacking, 690, 780

session ID, 687–692
custom, 708–710
default generation behavior, 708
encrypting, 690
generating, 687

session ID managers, 708–710

Session object, 680–681
behavior and implementation, 98
removal of values from, 694–695

session providers, out-of-process, 753

session state, 680–704. *See also* *HttpSessionState* class
access to, 680, 699
best practices for, 710
concurrent access, 684
configuring, 98–100, 691–692
customizing management of, 704–710
errors on page and, 695
expiration of, 706
extensibility model for, 680
HTTP handler access to, 141
InProc mode issues, 694–695
lifetime of, 693–695
loss of, 694–695
management timeline, 683
persisting data to remote servers, 695–699
persisting data to SQL Server, 699–704

remote, 695–699

serialization and deserialization, 695–697

Session_End event, 693–694

session ID, assigning, 687–692

Session_Start event, 693

session-state HTTP module, 680–684

state client manager, 681–682

synchronizing access to, 683–686

Web farm/garden scenarios, 703

session state store, 705. *See also* state providers

Session_End event, 686, 693–694

SessionIDManager class, 708
deriving from, 709

sessions
abandoning, 686
cookieless, 688–691
identifying, 687–692
lifetime of, 693
out-of-process, 695–697

Session_Start event, 693

<sessionState> section, 98–100, 691–692
attributes of, 692
SQL Server, setting as state provider, 700–701

SessionStateModule, 119, 680–684

SessionStateStoreData class, 707

SessionStateStoreProviderBase class, 705–706

SessionStateUtility class, 707

SetAuthCookie method, 798

SetCacheability method, 666

SetExpires method, 666

SetPropertyValue property, 290

settings
inheritance, 63, 90
packaging, 43–51

SetVaryByCustom method, 768

shadow-copy feature, 84

shadowCopyBinAssemblies attribute, 84

sharding, 612

Shared attribute, 773

SharedCache, 754

ShouldHook helper function, 153

show function, 915–917

shutdownTimeout attribute, 84

SignOut method, 795

sign-outs, 795–796

Silverlight
compatibility with other applications, 632

WCF service configuration in, 885

SimpleGaugeBar control, 522–527
color support, 526–527
extending, 533–543
object model definition, 523
object model implementation, 523–526
output of, 529
properties of, 523
rendering, 527–533
ruler, 526, 530–531
using, 532–533

SimpleHandlerFactory class, 142, 144

Single Responsibility Principle (SRP), 573–574

single-form model, 365–366, 368.
See also input forms

site map providers, 100, 354–355
default, 352

<siteMap> section, 100

site maps
configuring, 357–360
defining, 352–353
localizing, 359–361
multiple, 357–358
securing access with roles, 358

site navigation API, 352–358
configuration settings, 100

site precompilation, 52–55
benefits of, 52
deployment precompilation, 53–55
in-place precompilation, 53
target directory support, 53, 54

site replication tools, advantages of, 42

site-level configuration settings, 108–110

SiteMap class, 355–356

<siteMapNode> elements, 353

SiteMapPath controls, 356–358

SiteMapProvider class, 354

SkinID property, 342

skins, 338–341
applying, 341–342
for server controls, 220

sliding expirations, 723, 726, 731–732

SoC, 10, 571–573
in ASP.NET MVC, 23
favoring, 14
layers and, 593

MVC pattern and, 617

Socket class, 102

software, rigid, 567–568

- software dependencies, 568
- software design, 565
 - abstraction, 575–576
 - big ball of mud, 566–567
 - cohesion and coupling, 569–571
 - maintainability, 565
 - methodologies, 595
 - object-oriented design, 599
 - principles of, 569–572
 - requirements churn, 567
 - security planning, 779
 - separation of concerns, 571–573
 - SOLID principles, 573–583
 - structured writing, 615
 - symptoms of deterioration, 567–569
 - test-driven development, 638
 - three-tiered architecture, 593–595
 - from use-case, 624
 - software design team
 - limited skills, 566–567
 - member turnover, 567
 - software modules
 - cohesion of, 570
 - coupling of, 570–571
 - low coupling between, 575
 - software reuse, 568
 - software workarounds, 568–569
 - SOLID principles, 573–583
 - Dependency Inversion principle, 580–583
 - Interface Segregation principle, 579–580
 - Liskov's principle, 576–578
 - Open/Closed Principle, 575–576
 - Single Responsibility Principle, 573–574
 - Sort buttons, 511
 - sorting
 - data, 453–454
 - expressions, 453
 - lists, 511
 - source code
 - of content pages, 325
 - deploying, 40
 - for derived classes, generating, 172
 - parsing, 170
 - source files
 - dynamic compilation of, 189
 - generating, 611
 - tags, 388
 - sprites, 315–316
 - SQL Azure, 613
 - SQL Express, 286
 - SQL injection, 780
 - SQL Server
 - cache dependency, 743–745, 762
 - hosting identity access, 703
 - persisting session state to, 699–704
 - session state data store, creating, 701–703
 - SqlCacheDependency class, 80, 743–745
 - SqlCommand object, 744
 - SqlDependency attribute, 762
 - SqlRoleProvider, 821
 - SQLServer mode, 99
 - SQLServer provider, 695, 705
 - src attribute, 858
 - SRP, 573–574
 - canonical example, 574
 - SSL, 782
 - authentication tickets, securing with, 803–804
 - StackOverflow site, 267
 - startMode attribute, 60
 - state client managers, 681–682
 - state information. *See also* view state
 - detecting changes, 212–213, 248–249
 - persisting, 33
 - releasing, 33
 - retrieving, 32
 - state management. *See also* view state
 - application state, 676–679
 - best practices, 710
 - cookies, 675
 - levels of, 675
 - session state, 680–710
 - view state, 710–720
 - state providers
 - ASP.NET, 697–699
 - custom, 704–708
 - expiration callback support, 706
 - expiration mechanisms, 706
 - locking mechanisms, 706
 - out-of-process, 695–699
 - partition resolvers, 704
 - registering, 707
 - SQL Server, 700–704
 - writing, 707
 - StateBag class, 711–712
 - methods of, 711–712
 - properties of, 711
 - stateful behavior
 - postbacks for, 6
 - view state and, 6–7
 - StateServer mode, 100
 - StateServer provider, 695, 705
 - static files, IIS serving of, 128
 - Static option, 224
 - static properties in *global.asax* file, 655
 - static requests, processing, 29
 - static resources
 - behavior of, 757–758
 - images, 133
 - StaticSiteMapProvider class, 354
 - statusCode attribute, 81
 - S3, 613
 - StopRoutingHandler class, 164
 - storage
 - of HTTP requests, 673
 - intermediate, 721
 - local, 923
 - of output caching, 776–777
 - schema-less storage, 614
 - store providers, 692
 - for session-state settings, 99
 - stored procedures, 612
 - stream classes, creating, 666–667
 - strings, lengths of, 696
 - stubs, 640
 - Style class, 254–255
 - style information
 - adding to pages, 337–345
 - themes, 337
 - style properties, 357
 - of Web controls, 254–255
 - style sheet files, external, linking to, 242
 - style sheets, 339. *See also* CSS defined, 338
 - style sheet themes, 338, 340
 - StyleSheetTheme attribute, 340, 341
 - styling pages, 336–344
 - submit buttons, 213, 247
 - SubSonic, 600
 - Substitution control, 775–776
 - .svc resources, 881–882
 - swapText function, 921
 - synchronization
 - of application state operations, 678–679
 - of cache, 736
 - with Copy Web Site feature, 42
 - synchronous handlers, 121–127. *See also* HTTP handlers
 - SYSTEM account, 781
 - system classes, 12
 - System.ApplicationException class, 272

System.Configuration namespace, 63
 configuration management classes in, 110
 <system.diagnostics> section, 101
 System.Drawing namespace, use of classes in, 140
 <system.serviceModel> section, 67
 <system.web.extensions> section, 107–108
 <system.web> section, 71–105
 <Caching> subgroup, 73
 HTTP handlers, registering in, 124
 important sections in, 71–73
 <system.webServer> section, 108–110
 HTTP handlers, registering in, 125
 reading and writing in, 112
 System.Web.UI.HtmlControls namespace, 237
 System.Web.UI.Page class, 12, 36, 172
 ProcessRequest method, 36
 System.Web.UI.WebControls namespace, 253

T

T4 templates, 600
 Table Module pattern, 597, 598
 DAL and, 606
 <table> tag, 232
 table-based interfaces, 480–485
 tables, for multicolumn layouts, 485
 tabular layouts, 480–485
 alternate item rendering, 483–484
 HTML tables, 484–485
 item template, 481–483
 layout template, 480–481
 tag-based selectors, 910
 <tagMapping> section, 91–92
 tasks, asynchronous execution, 201
 TDD, 23, 638
 Telerik JustCode, 270
 template containers, defining, 558–559
 template definitions, for controls, 340
 template properties
 attributes, 557–558
 defining, 557–558
 setting, 559–560
 TemplateControl class, 190
 Eval method, 438
 templated fields, 450–451
 templates
 for custom controls, 556–561
 defined, 434
 insulating in separate file, 557
 ListView support of, 473
 login templates, 831
 rendering, 560–561
 role-based, 831–832
 T4 templates, 600
 for wizards, 400
 temporary ASP.NET files folder, 786
 permissions on, 784
 test doubles, 640
 test harnesses, 638
 testability of Web Forms, 636–642
 test-driven development (TDD), 638
 with ASP.NET MVC, 23
 testing
 CacheDependency objects, 742
 code-behind classes, 361
 DAL interfacing and, 609
 presenter classes, 639–642
 test names, 639
 unit testing, 637–638
 for usability, 361–364
 writing tests, 637
 text
 inserting as literals, 260
 localized text, 306–307
 text boxes, multiline, 245
 text controls, 260–261
 TextBox class, interfaces of, 260
 theme attribute, 340
 Theme attribute, 341
 Themeable attribute, 235
 ThemeList controls, 343
 themes, 319, 337–341
 applying, 340–341
 changing, 209
 vs. CSS, 357
 customization themes, 338
 defined, 337
 enabling and disabling, 342–343
 loading dynamically, 343
 precedence of, 341
 for server controls, 220, 235
 skins, 341–342
 structure of, 339–340
 style sheet themes, 338

thread pool, free threads in, 86
 threads
 asynchronous handlers and, 147–148
 impersonation and, 784–785
 minimum settings for, 94–95
 three-tiered architecture, 593–595
 business logic layer, 596–605
 design model, 595
 tickets, authentication, 792–793
 getting and setting, 798
 securing, 803–804
 storage in cookies, 799–800
 tiled layouts, 487–493
 grouping items, 487–489
 group item count, 489–491
 group separators, 489
 populating dynamically, 491–493
 Timer control, 878–879
 ToInt32 method, 131
 topology of distributed cache, 746
 <trace> section, 100–101
 Trace.axd handler, 129
 tracing, 100–101
 Transaction Script (TS) pattern, 597–598
 Transfer method, 663
 Transform attribute, 50
 transformation files, 50–51
 transition events, defined, 404
 TransmitFile method, 669
 tree of controls
 building, 209
 unique names in, 190
 trigger function, 918
 Triggers collection, 869
 triggers of postbacks, 868–869
 trust levels, 786–789
 configuration settings, 101–104
 and policy files, mappings between, 97–98
 <trust> section, 101–104
 code access security permissions, 787
 <trustLevel> elements, 97
 try/catch/finally blocks, 270
 wrapping code in, 278
 typed attributes, 180–181
 TypeName attribute, 376

U

UICulture property, 315
 unbind function, 918–919

- UniqueID* property, 211, 220
- unit testing, 637–638
 - base classes and, 656
 - unit test classes, 639
- Unity, 587–592
 - declarative configuration, 589–590
 - dependencies, resolving, 588–589
 - lifetime managers, 590–592
 - types and instances, registering, 587–588
- Unload* event, 215
- unobtrusive JavaScript, 918
- UnprotectSection* method, 114
- update callbacks, 726
- Update* method
 - exceptions thrown in, 868
 - signature, 868
- update operations, 466–468
 - in *ListView* control, 499–501
 - modifying objects, 468
 - parameters for, 466
- UpdateMode* property, 866–867
- UpdatePanel* control, 851, 860–865
 - conditional refreshes, 866–870
 - contents of, 865
 - example of, 861
 - feedback during updates, 870–876
 - full postbacks from, 869–870
 - in master pages, 864–865
 - vs. panel controls, 860–861
 - populating, 863–864
 - postbacks, triggering, 868–869
 - properties of, 862
 - UpdateProgress* control for, 871
 - UpdateProgress* control, 870–872
 - events of, 872–873
 - UpdateRequestCache* event, 33, 651
- updating
 - concurrent calls, 877–878
 - conditional refreshes, 866–870
 - pending operations, aborting, 876
 - progress screen, 871–872
 - refresh conditions, 866
 - user interface, disabling, 874–875
- Updating* event, 468
- uploading files control, 261–262
- Uri* class, 673
- url* attribute, 104
- URL authorization, 791
- URL encoding, 661
- URL Rewrite Module, 37
- URL rewriting, 155, 157–158, 349, 658–659
 - drawback of, 158
 - vs. URL routing, 159
- URL routing, 155–165
 - constraints on, 162, 164
 - vs. HTTP handlers, 165
 - preventing for defined URLs, 164–165
 - vs. URL rewriting, 159
 - in Web Forms, 36, 160–165
- URL routing engine, 119–120, 155, 157–159
- URLAuthorizationModule* HTTP module, 791
- <urlMappings>* section, 104
- urlMetadataSlidingExpiration* attribute, 84
- UrlRoutingModule* class, 155
- URLs
 - for advertisements, 262–263
 - derived classes, linking to, 172
 - for embedded resources, 195
 - and endpoints, mappings between, 104
 - expressive URLs, 156–157
 - for hyperlinks, 447
 - for images, 133, 259
 - logic and parameters in, 156
 - mangling, 690
 - mapping to ASPX pages, 36
 - navigating to, 243–244
 - preventing routing for, 164–165
 - resolving, methods for, 195–197
 - route data in, 156
- usability, 344–364
 - cross-browser rendering, 344–348
 - navigation system, 351–357
 - SEO, 348–351
 - site map configuration, 357–360
 - testing for, 361–364
- UseDeviceProfile*, 691, 801
- useHostingIdentity* attribute, 703
- user account management, 806
- user authentication, 784, 794–795
 - configuration settings, 74–76
- user controls
 - cacheability of, 770
 - caching in cacheable pages, 773–774
 - caching output of, 770
 - vs. custom controls, 513
 - description of, 768–769
 - dynamically loading, 557
 - inserting into pages, 769
 - master pages, 329. *See also* master pages
- pages, linking to, 189–190
- sharing output of, 772–773
- Static* option, 224
- strongly typed instances of, 189
- user credentials, collecting, 794
- user input
 - filtering, 135
 - validation of, 379–396
- user interface
 - disabling, 874–875
 - dynamic, 18–19
 - iterative controls for, 427–432
 - table-based interfaces, 480–485
 - for Web pages, 3
- user profiles
 - in action, 296–298
 - for anonymous users, 294–295, 299–300
 - automatically saving, 97
 - configuration settings, 96–97
 - creating, 285–292
 - grouping properties, 290
 - interaction with page, 292–300
 - profile database, creating, 292–294
 - profile providers, 300–303
 - properties, accessing, 295–296
 - storage of data, 286
 - user-specific information in, 299
 - for Web Application Projects, defining, 286
 - for Web site projects, 285
- UserControl* class, 321
- user-defined code, invoking, 245
- usersOnlineTimeWindow* attribute, 88
- UsersOnlineTimeWindow* property, 808
- user-mode caching, 58
- users
 - adding and creating, 806, 809–810, 834–835
 - anonymous identification feature and, 73–74
 - authenticating, 793, 810–811. *See also* authentication
 - authentication state, 829
 - authorization of, 76–77
 - feedback for, 870–876
 - information about, storing, 106
 - managing, 811–812
 - reauthorization of, 663
 - roles, 817
- UseSubmitBehavior* property, 213, 258

V

val function, 923

Validate method, 379, 388

ValidateRequest attribute, 674

ValidateUser function, 810–811

validation

of input fields, 248

of new records, 504–505

of cached pages, 764–765

validation attribute, 87

validation controls, 379–396

BaseValidator class, 380–381

client-side validation, 393–394

CompareValidator control, 382–383

cross-page posting and, 395–396

CustomValidator control, 383–385

error information, displaying, 388–389

ForeColor property, 381

generalities of, 379–382

and input controls, linking, 381–382

multiple controls, 380

multiple validators, 390–391

properties of, 380–381

RangeValidator control, 386

RegularExpressionValidator

control, 385

RequiredFieldValidator control, 386–387

server-side validation, 387–388

validation groups, 394–395

validation summary, 391–393

<*validation*> element, 109

ValidationGroup property, 394–395

validationKey attribute, 87

[*ValidationProperty*] attribute, 385

ValidationSummary control, 380, 391–393

Validators collection, 379

value, cached item, 725

VaryByControl attribute, 770–772

VaryByCustom attribute, 767

VaryByHeader attribute, 767

VaryByHeaders property, 767

VaryByParam attribute, 759–760, 765

VaryByParams property, 765

verbs attribute, 76

VerifyRenderingInServerForm method, 195, 365

view

abstracting, 624–626

in ASP.NET MVC, 21–22

autonomous views, 616

defined, 616

role in MVC, 618

role in MVP, 620

XAML-based, 623

view controls, 266–268, 411, 432–434

DataKeyNames property, 421

DetailsView control, 432

FormView control, 433

GridView control, 433

ListView control, 433–434

programmatic control in, 476

view state, 4–7, 710–720

authentication checks for, 713

of controls, 227–228

control state, 718–719

cross-page posting and, 374–375

disabling, 715–717

encrypting, 712–713

encryption and decryption keys, 87

functionality of, 716

information stored in, 710

issues with, 712–715

methods of, 711–712

page performance and, 713–715

persisting, 200

programming, 715–720

properties of, 711

restoring, 210

saving to storage medium, 214

security of, 192–193, 712–713

SEO and, 350

on server, 719–720

size of, 7, 10, 227, 713–715

StateBag class, 711–712

tracking, 210

truncation of, 90–91

when to use, 717

writing, 711

ViewState class, 676

ViewState container

classes, saving in, 539

control properties in, 536–538

property values, storing, 539

__VIEWSTATE hidden field, 215, 712

restoring contents of, 210

ViewState property, 710–712

ViewStateEncryptionMode

property, 713

ViewStateMode property, 227–228, 716

ViewStateUserKey property, 192–193, 713

Virtual Accounts, 39

virtual directories, configuring properties, 43

virtual folders for running applications, 645

virtual members, safe use of, 578

VirtualPath attribute, 376

viscosity, 569

visibility operators, 915–917

Visual Basic, 3

Visual Studio

Add Config Transform option, 51

adding content pages to projects, 324

Build|ASP.NET Configuration menu item, 302

deploy packages, building, 45–46

designers, 615

exception handling, 270

Mark As IIS Application On Destination check box, 47

MSUnit, 638

Package/Publish SQL tab, 46

Package/Publish Web tab, 45–46

resources files, adding, 304

site precompilation, 52, 55

T4 templates, 600

Table Module pattern and, 598

web.config transformations, 49–51

Web Deployment Tool, 44–45

Web project deployment, 40

Web setup applications, creating, 42–43

Web Site Administration Tool (WSAT), 809

XCopy capabilities, 40–41

Visual Studio Development Server, 48

Visual Studio Publish Wizard, 48

Visual Studio 2010, 20

Visual Studio 2010 Coded UI Tests, 363

Visual Studio 2010 Ultimate, 615

Vlissides, John, 575

VSDOC file, 904

W

WAPs, 40

data model definition, 290–292
personalization data in, 295–296

- WAPs (*continued*)
 - user profiles, building, 286
 - web.config* transformations, 49
- warm-up. *See* application warm-up
- WatiN, 362–364
- WCF services, 882
 - in AJAX pages, 881–885
 - ASP.NET compatibility mode, 887, 889
 - DataContract* attribute, 891–892
 - method execution requests, 887
- WDT, 44–45
 - building, 45–47
 - capabilities of, 45
 - contents of, 47
 - installing, 44
- Web application folders, configuring, 43
- Web Application Projects. *See* WAPs
- Web applications. *See also* applications
 - autostarting, 38–39
 - grouping, 29
 - IIS settings, specifying, 48
 - initialization tasks, 38
 - installing, 39
 - machinewide settings, 69, 70
 - per-directory settings, 68
 - presentation layer, 269. *See also* pages
 - publishing in Visual Studio, 46–47
 - responsiveness of, 8
 - root *web.config* file, 69
- Web attacks
 - common types of, 779–780
 - fending off, 779. *See also* security
- Web browsers. *See* browsers
- Web cache, 755–756
- Web controls, 217, 253–268. *See also* controls
 - AdRotator* controls, 262–263
 - AJAX and, 267–268
 - base class, 253
 - button controls, 257–258
 - Calendar* controls, 263–264
 - check box controls, 259–260
 - core controls, 256–257
 - correct rendering of, 195
 - file upload control, 261–262
 - hidden field control, 261–262
 - hyperlink controls, 258–259
 - image button controls, 259
 - image controls, 259
 - methods of, 255–256
 - PlaceHolder* control, 265–266
 - properties of, 253–254
 - radio button controls, 259–260
 - runat="server"* attribute for, 253
 - styling, 254–255
 - text controls, 260–261
 - user interface, 527
 - view controls, 266–267
 - Xml* control, 264–265
- Web deployment, 40. *See also* application deployment
- Web Deployment Tool or Web Deploy (WDT), 44–47
- Web development
 - ASP.NET for, 3
 - ASP.NET MVC for, 4
 - tools for, 19
- Web farms/gardens, session state and, 703
- Web Forms, 3–14
 - in action, 5
 - alternatives to, 21–26
 - base class, 36
 - code testability, 636–642
 - vs. Data-for-Data model, 17
 - effectiveness of, 11, 14
 - HTTP handlers, determining, 35
 - moving away from, 15–19
 - MVC pattern and, 618
 - MVP pattern and, 621
 - MVVM pattern and, 622
 - navigation in, 634–636
 - opting out of built-in features, 25
 - Page Controller pattern, 11–14, 618
 - page postbacks, 4–5
 - page weights, 10
 - postback events, 5
 - presentation layer patterns, 615–623
 - abstraction layer, 14
 - runtime environment, 27
 - runtime stack, 23
 - Selective Updates model, 20
 - server controls, 4–5
 - strengths of, 4–8
 - testability of, 10
 - UI focus, 26,
 - usability of, 11
 - view state, 4–5
 - weaknesses of, 8–10
- Web frameworks, 18–19
 - AJAX built into, 19–20
- Web methods, defining, 896
- Web pages. *See also* ASP.NET pages
 - image references, 133
 - markup mix, 3
- Web Platform Installer, 44
- Web servers, 27. *See also* IIS
 - extensions of, 120
 - functionality of, 27
 - redesign of, 29
 - uploading files to, 249–251
- Web Setup Projects
 - creating, 42–43
 - Web application folders, 43
- Web Site Administration Tool (WSAT), 292–293, 809
 - for role management, 818
- Web site projects (WSPs), 40
 - Copy Web Site function, 40–41
 - data model definition, 286–287
 - personalization data in, 295–296
 - user profiles, defining, 285
- Web site root folder, 786
- Web sites
 - development skill set for, 3
 - integration testing, 49
 - interface elements, 319
 - JSONP-enabled, 930
 - navigation system, 351–357
 - page composition, 319–345
 - rich client sides, 839
 - root *web.config* file, 69
 - testing for usability, 361–364
 - usability, 344–364
 - visual idea for, 319
- Web user controls, use of, 557
- web.config* file. *See also individual section names*
 - additional files, 64
 - assemblies, editing, 186
 - centralized files, 69
 - for classic and integrated IIS 7 working modes, 109
 - for configuration, 63
 - current environment settings in, 49
 - custom build configurations, 51
 - debug, release, and test versions, 49–51
 - editing, 50, 116–117, 170
 - global settings, replicating in, 70
 - <*identity*> section, encrypting, 87
 - <*location*> section, 68
 - numRecompilesBeforeAppRestart* attribute, 56
 - <*outputCacheProfiles*> section, 774
 - processing of, 64–65

web.config file (continued)
 remote session state, enabling in, 698
 root file, 64
 sections in, declaring, 68
 writing to, 65

WebConfigurationManager class, 110, 111

WebControl class, 253, 514
 vs. *Control* class, 519
 deriving controls from, 513

<*webControls*> section, 104–105

web.debug.config file, 49–50

WebGet attribute, 882

@*WebHandler* directive, 141–142

webHttpBinding model, 883

WebInvoke attribute, 883

WebMatrix IDE, 25

WebMethod attribute, 886, 895

web.release.config file, 49–50, 51

WebRequest class, 102

WebResource.axd handler, 859

web.sitemap file, 352

WIF, 76
 claims and, 822
 downloading, 824

Windows authentication, 76, 782, 790–791
 limitations of, 791

Windows CardSpace, 791

Windows Communication Foundation (WCF), 603

Windows event log, logging exceptions in, 277

Windows Identity Foundation, (WIF), 76, 822, 824

Windows Server AppFabric, 747–753

Windows service always running, 38

Windows System folder, 786

WindowsTokenRoleProvider, 821

Wizard control, 266, 374, 397–402
 events of, 401
 main properties, 400
 style properties, 399–400
 suffixes, 400–401
 templates for, 400

WizardNavigationEventArgs structure, 406, 407

WizardNavigationEventHandler delegate, 406

wizards, 397–409
 adding steps to, 402–405
 canceling navigation events, 407–408
 finalizing, 408–409
 headers, 398
 input steps, 403–404
 input validation, 404
 jumping to steps, 401
 navigating through, 405–409
 navigation bar, 398
 programming interface, 400–402
 server-side validation, 405
 sidebar, 398, 404–405
 steps, types of, 402–403
 structure of, 397–399
 style of, 399–400
 templates, 400

view, 398

WizardStep class, 402

WizardStepType enumeration, 402

workarounds, 568–569

worker process
 ASP.NET standalone, 28–29
 identity of, 781, 783
 identity of, changing, 784–786
 IIS native, 29
 incoming requests, fielding, 149
 recycling, 55, 59

worker properties, of *Page* class, 191–193

worker threads, number of, 94–95

World Wide Web Consortium (W3C), 339
 proxy component standard, 842
 updatable DOM standard, 842

wrapped sets, 905, 908–914
 CSS classes, working with, 917
 enumerating content, 908–909
 operating on, 908–909, 915–919
 visibility operators, 915–917

WriteFile method, 669

WriteSubstitution method, 776

WSPs, 40–41, 286–287, 295–296

w3wp.exe, 29

WWW publishing service, 29

X

XCopy, 40–43
 Visual Studio capabilities, 40–41

xdt elements, 50

XHTML, ASP.NET support for, 3

XHTML rendering mode, designating, 105

<*xhtmlConformance*> section, 105

XML
 advertisement files, 262–263
 data, cache dependency for, 739–742
 vs. JSON, 892–893
 as serialization format, 890

Xml controls, 264

XML documents, embedding in pages, 264–265

XML encryption, 107
 for <*identity*> section, 87

XmlDataCacheDependency class, 739–740
 implementing, 740–741

XmlHttpRequest object, 16, 840–843
 Same Origin Policy, 850
 using, 844–845

XmlSiteMapProvider class, 352, 358

XslTransform class, 264, 265

Y

Yooder, Joseph, 566

YSlow, 317

YSOD (yellow screen of death), 272

About the Author



Dino Esposito is a software architect and trainer living near Rome and working all around the world. Having started as a C/C++ developer, Dino has embraced the ASP.NET world since its beginning and has contributed many books and articles on the subject, helping a generation of developers and architects to grow and thrive.

More recently, Dino shifted his main focus to principles and patterns of software design as the typical level of complexity of applications—most of which were, are, and will be Web applications—increased beyond a critical threshold. Developers and architects won't go far today without creating rock-solid designs and architectures that span from the browser presentation all the way down to the data store, through layers and tiers of services and workflows. Another area of growing interest for Dino is mobile software, specifically cross-platform mobile software that can accommodate Android and iPhone, as well as Microsoft Windows Phone 7.

Every month, at least five different magazines and Web sites in one part of the world or another publish Dino's articles, which cover topics ranging from Web development to data access and from software best practices to Android, Ajax, Silverlight, and JavaScript. A prolific author, Dino writes the monthly "Cutting Edge" column for MSDN Magazine, the "CoreCoder" columns for DevConnectionsPro Magazine, and the Windows newsletter for Dr.Dobb's Journal. He also regularly contributes to popular Web sites such as DotNetSlackers—<http://www.dotnetslackers.com>.

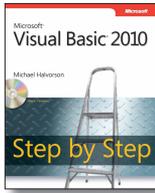
Dino has written an array of books, most of which are considered state-of-the-art in their respective areas. His more recent books are *Programming ASP.NET MVC 3* (Microsoft Press, 2011) and *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008), which is slated for an update in 2011.

Dino regularly speaks at industry conferences worldwide (such as Microsoft TechEd, Microsoft DevDays, DevConnections, DevWeek, and Basta) and local technical conferences and meetings in Europe and the United States.

In his spare time (so to speak), Dino manages software development and training activities at **Crionet** and is the brains behind some software applications for live scores and sporting clubs.

If you would like to get in touch with Dino for whatever reason (for example, you're running a user group, company, community, portal, or play tennis), you can tweet him at **@despos** or reach him via Facebook.

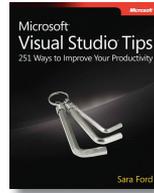
For Visual Basic Developers



Microsoft® Visual Basic® 2010 Step by Step

Michael Halvorson
ISBN 9780735626690

Teach yourself the essential tools and techniques for Visual Basic 2010—one step at a time. No matter what your skill level, you'll find the practical guidance and examples you need to start building applications for Windows and the Web.



Microsoft Visual Studio® Tips 251 Ways to Improve Your Productivity

Sara Ford
ISBN 9780735626409

This book packs proven tips that any developer, regardless of skill or preferred development language, can use to help shave hours off everyday development activities with Visual Studio.



Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build, Second Edition

Sayed Ibrahim Hashimi,
William Bartholomew
ISBN 9780735645240

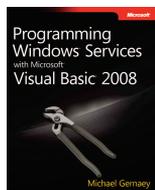
Your practical guide to using, customizing, and extending the build engine in Visual Studio 2010.



Parallel Programming with Microsoft Visual Studio 2010

Donis Marshall
ISBN 9780735640603

The roadmap for developers wanting to maximize their applications for multicore architecture using Visual Studio 2010.

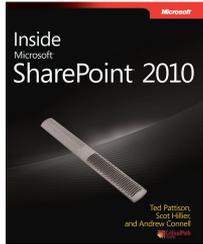


Programming Windows® Services with Microsoft Visual Basic 2008

Michael Germaey
ISBN 9780735624337

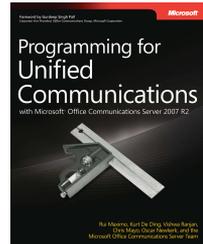
The essential guide for developing powerful, customized Windows services with Visual Basic 2008. Whether you're looking to perform network monitoring or design a complex enterprise solution, you'll find the expert advice and practical examples to accelerate your productivity.

Collaborative Technologies— Resources for Developers



**Inside Microsoft®
SharePoint® 2010**
Ted Pattison, Andrew Connell,
and Scot Hillier
ISBN 9780735627468

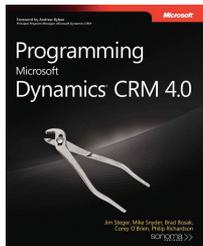
Get the in-depth architectural insights, task-oriented guidance, and extensive code samples you need to build robust, enterprise content-management solutions.



**Programming for
Unified Communications
with Microsoft Office
Communications
Server 2007 R2**

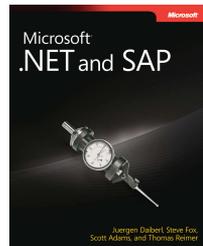
Rui Maximo, Kurt De Ding,
Vishwa Ranjan, Chris Mayo,
Oscar Newkerk, and the
Microsoft OCS Team
ISBN 9780735626232

Direct from the Microsoft Office Communications Server product team, get the hands-on guidance you need to streamline your organization's real-time, remote communication and collaboration solutions across the enterprise and across time zones.



**Programming
Microsoft
Dynamics® CRM 4.0**
Jim Steger, Mike Snyder,
Brad Bosak, Corey O'Brien,
and Philip Richardson
ISBN 9780735625945

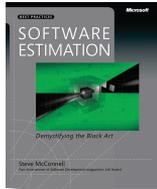
Apply the design and coding practices that leading CRM consultants use to customize, integrate, and extend Microsoft Dynamics CRM 4.0 for specific business needs.



**Microsoft
.NET and SAP**
Juergen Daiberl,
Steve Fox, Scott Adams,
and Thomas Reimer
ISBN 9780735625686

Develop integrated, .NET-SAP solutions—and deliver better connectivity, collaboration, and business intelligence.

Best Practices for Software Engineering



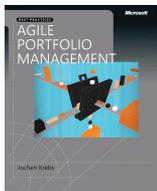
**Software Estimation:
Demystifying the Black Art**
Steve McConnell
ISBN 9780735605350

Amazon.com's pick for "Best Computer Book of 2006"! Generating accurate software estimates is fairly straightforward—once you understand the art of creating them. Acclaimed author Steve McConnell demystifies the process—illuminating the practical procedures, formulas, and heuristics you can apply right away.



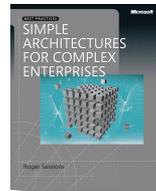
**Code Complete,
Second Edition**
Steve McConnell
ISBN 9780735619678

Widely considered one of the best practical guides to programming—fully updated. Drawing from research, academia, and everyday commercial practice, McConnell synthesizes must-know principles and techniques into clear, pragmatic guidance. Rethink your approach—and deliver the highest quality code.



Agile Portfolio Management
Jochen Krebs
ISBN 9780735625679

Agile processes foster better collaboration, innovation, and results. So why limit their use to software projects—when you can transform your entire business? This book illuminates the opportunities—and rewards—of applying agile processes to your overall IT portfolio, with best practices for optimizing results.



**Simple Architectures for
Complex Enterprises**
Roger Sessions
ISBN 9780735625785

Why do so many IT projects fail? Enterprise consultant Roger Sessions believes complex problems require simple solutions. And in this book, he shows how to make simplicity a core architectural requirement—as critical as performance, reliability, or security—to achieve better, more reliable results for your organization.



The Enterprise and Scrum
Ken Schwaber
ISBN 9780735623378

Extend Scrum's benefits—greater agility, higher-quality products, and lower costs—beyond individual teams to the entire enterprise. Scrum cofounder Ken Schwaber describes proven practices for adopting Scrum principles across your organization, including that all-critical component—managing change.

ALSO SEE

**Software Requirements,
Second Edition**
Karl E. Wiegand
ISBN 9780735618794

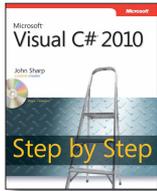
**More About Software
Requirements:
Thorny Issues and
Practical Advice**
Karl E. Wiegand
ISBN 9780735622678

**Software Requirement
Patterns**
Stephen Withall
ISBN 9780735623989

**Agile Project
Management
with Scrum**
Ken Schwaber
ISBN 9780735619937

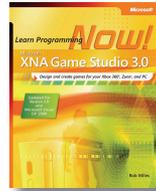
Solid Code
Donis Marshall, John Bruno
ISBN 9780735625921

For C# Developers



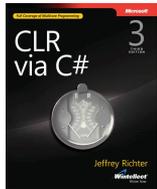
**Microsoft®
Visual C#® 2010
Step by Step**
John Sharp
ISBN 9780735626706

Teach yourself Visual C# 2010—one step at a time. Ideal for developers with fundamental programming skills, this practical tutorial delivers hands-on guidance for creating C# components and Windows-based applications. CD features practice exercises, code samples, and a fully searchable eBook.



**Microsoft
XNA® Game Studio 3.0:
Learn Programming Now!**
Rob Miles
ISBN 9780735626584

Now you can create your own games for Xbox 360® and Windows—as you learn the underlying skills and concepts for computer programming. Dive right into your first project, adding new tools and tricks to your arsenal as you go. Master the fundamentals of XNA Game Studio and Visual C#—no experience required!



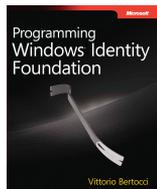
**CLR via C#,
Third Edition**
Jeffrey Richter
ISBN 9780735627048

Dig deep and master the intricacies of the common language runtime (CLR) and the .NET Framework. Written by programming expert Jeffrey Richter, this guide is ideal for developers building any kind of application—ASP.NET, Windows Forms, Microsoft SQL Server®, Web services, console apps—and features extensive C# code samples.



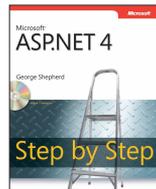
**Windows via C/C++,
Fifth Edition**
Jeffrey Richter, Christophe Nasarre
ISBN 9780735624245

Get the classic book for programming Windows at the API level in Microsoft Visual C++®—now in its fifth edition and covering Windows Vista®.



**Programming Windows®
Identity Foundation**
Vittorio Bertocci
ISBN 9780735627185

Get practical, hands-on guidance for using WIF to solve authentication, authorization, and customization issues in Web applications and services.



**Microsoft® ASP.NET 4
Step by Step**
George Shepherd
ISBN 9780735627017

Ideal for developers with fundamental programming skills—but new to ASP.NET—who want hands-on guidance for developing Web applications in the Microsoft Visual Studio® 2010 environment.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft
Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter