Microsoft

Programming Microsoft® ASP.NET 4

Dino Esposito

Microsoft®

Programming Microsoft® ASP.NET 4

Dino Esposito

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

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

Printed and bound in the United States of America.

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

Table of Contents

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
IsReusable	This property provides a Boolean value indicating whether the HTTP runtime can reuse the current instance of the HTTP handler while serving another request.
ProcessRequest	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; }
        }
    }
}
```

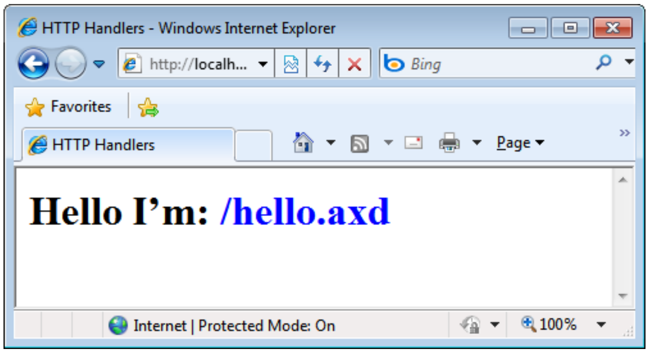
You need an entry point to be able to call the handler. In this context, an entry point into the handler's code is nothing more than an HTTP endpoint—that is, a public URL. The URL must be a unique name that IIS and the ASP.NET runtime can map to this code. When registered, the mapping between an HTTP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


HTTP Handlers - Windows Internet Explorer
http://localh...
Bing
Favorites
HTTP Handlers
Page
Internet | Protected Mode: On
100%

Hello I'm: /hello.axd

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
path	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, **.aspx*.
type	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application's private *Bin* directory and then in the system global assembly cache.
validate	If this attribute is set to *false*, ASP.NET loads the assembly with the handler on demand. The default value is *true*.
verb	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
allowPathInfo	If this attribute is set to *true*, the handler processes full path information in the URL or just the last section. It is set to *false* by default.
modules	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 *ManagedPipelineHandler* module.
name	Unique name of the handler.
path	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, **.aspx*.
preCondition	Specifies conditions under which the handler will run. (More information appears later in this section.)
requireAccess	Indicates the type of access that a handler requires to the resource, either read, write, script, execute, or none. The default is script.
resourceType	Indicates the type of resource to which the handler mapping applies: file, directory, or both. The default option, however, is *Unspecified*, meaning that the handler can handle requests for resources that map to physical entries in the file system as well as to plain commands.
responseBufferLimit	Specifies the maximum size, in bytes, of the response buffer. The default value is 4 MB.
scriptProcessor	Specifies the physical path of the ISAPI extension or CGI executable that processes the request. It is not requested for managed handlers.
type	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application's private *Bin* directory and then in the system global assembly cache.
verb	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
bitness32	The handler is 32-bit code and should be loaded only in 64-bit worker processes running in 32-bit emulation.
bitness64	The handler is 64-bit and should be loaded only in native 64-bit worker processes.
integratedMode	The handler should respond only to requests in application pools configured in integrated mode.
ISAPIMode	The handler should respond only to requests in application pools configured in classic mode.
runtimeVersionv1.1	The handler should respond only to requests in application pools configured for version 1.1 of the ASP.NET runtime.
runtimeVersionv2.0	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

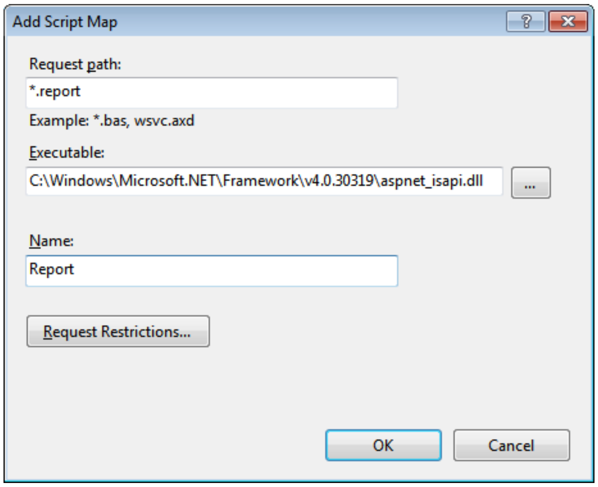
In ASP.NET applications, a common scenario when you want to use custom HTTP handlers is that you want to loosen yourself from the ties of ASPX files. Sometimes you want to place a request for a nonstandard ASP.NET resource (for example, a custom XML file) and expect the handler to process the content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


Add Script Map

Request path: *.report

Example: *.bas, wsvc.axd

Executable: C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_isapi.dll

Name: Report

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.)

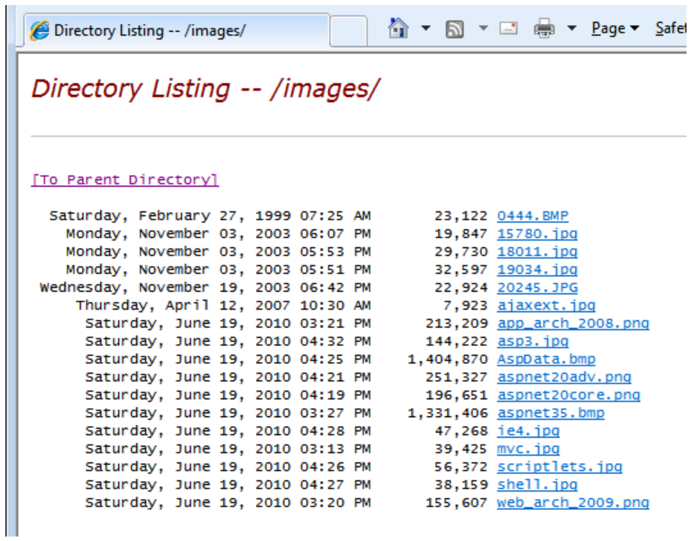
Now let's consider a more complex example of an HTTP handler.

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.



Directory Listing -- /images/

[To Parent Directory]

```
  Saturday, February 27, 1999 07:25 AM       23,122 0444.BMP
    Monday, November 03, 2003 06:07 PM       19,847 15780.jpg
    Monday, November 03, 2003 05:53 PM       29,730 18011.jpg
    Monday, November 03, 2003 05:51 PM       32,597 19034.jpg
 Wednesday, November 19, 2003 06:42 PM       22,924 20245.JPG
     Thursday, April 12, 2007 10:30 AM        7,923 ajaxext.jpg
      Saturday, June 19, 2010 03:21 PM      213,209 app_arch_2008.png
      Saturday, June 19, 2010 04:32 PM      144,222 asp3.jpg
      Saturday, June 19, 2010 04:25 PM    1,404,870 AspData.bmp
      Saturday, June 19, 2010 04:21 PM      251,327 aspnet20adv.png
      Saturday, June 19, 2010 04:19 PM      196,651 aspnet20core.png
      Saturday, June 19, 2010 03:27 PM    1,331,406 aspnet35.bmp
      Saturday, June 19, 2010 04:28 PM       47,268 ie4.jpg
      Saturday, June 19, 2010 03:13 PM       39,425 mvc.jpg
      Saturday, June 19, 2010 04:26 PM       56,372 scriptlets.jpg
      Saturday, June 19, 2010 04:27 PM       38,159 shell.jpg
      Saturday, June 19, 2010 03:20 PM      155,607 web_arch_2009.png
```

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 *<img>* 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.

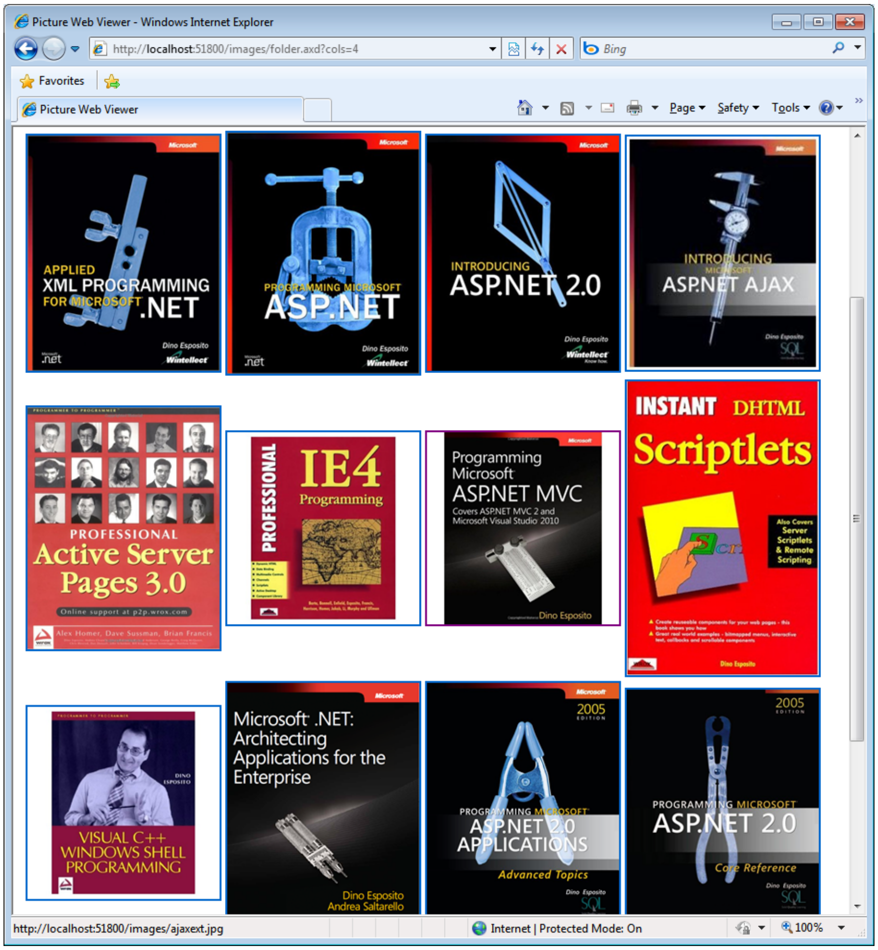

Picture Web Viewer - Windows Internet Explorer
http://localhost:51800/images/folder.axd?cols=4
Bing
Favorites
Picture Web Viewer
Page
Safety
Tools
http://localhost:51800/images/ajaxext.jpg
Internet | Protected Mode: On
100%

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 *<img>* 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 *<img>* 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 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.

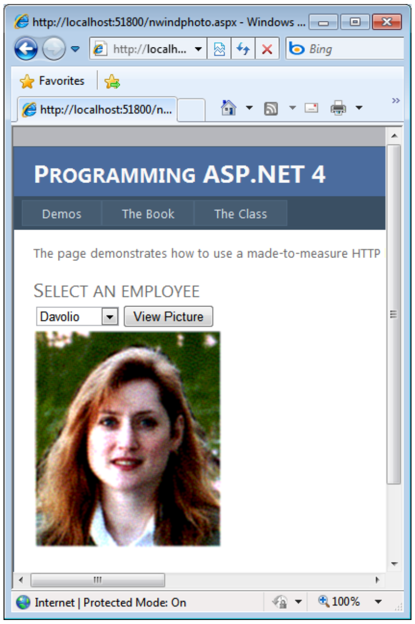
Although the preceding code works like a champ with regular BLOB fields, it must undergo the following modification to work with the *photo* field of the Northwind.Employees database:

```
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;
```


http://localhost:51800/nwindphoto.aspx - Windows ...
Favorites
PROGRAMMING ASP.NET 4
Demos
The Book
The Class
Davolio
View Picture
Internet | Protected Mode: On
100%

The page demonstrates how to use a made-to-measure HTTP

SELECT AN EMPLOYEE

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>
    <img id="picture" src="dynimage.axd?url=images/pic1.jpg" />
</body>
</html>
```

The page contains no ASP.NET code and displays an image through a static HTML *<img>* 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.

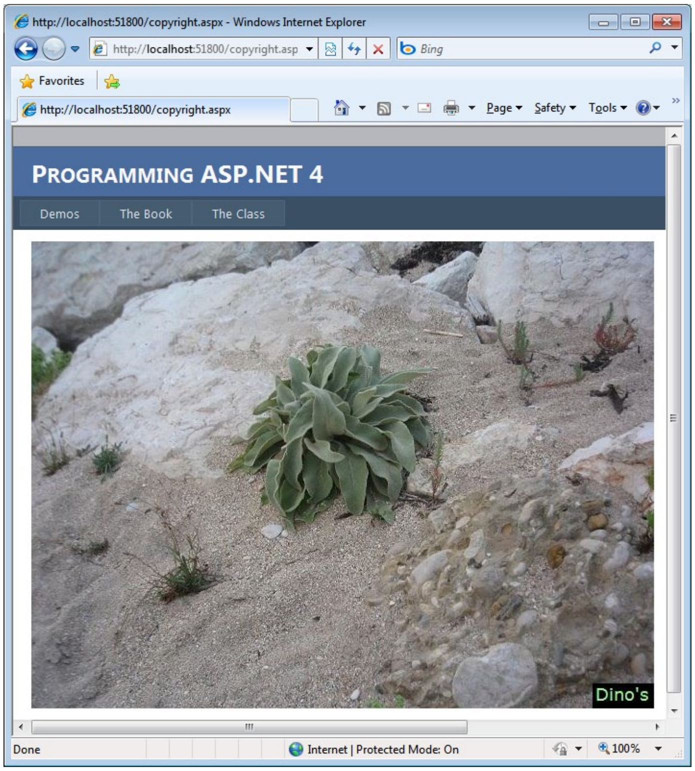

http://localhost:51800/copyright.aspx - Windows Internet Explorer
http://localhost:51800/copyright.asp
Bing
Favorites
http://localhost:51800/copyright.aspx
Page
Safety
Tools
PROGRAMMING ASP.NET 4
Demos
The Book
The Class
Dino's
Done
Internet | Protected Mode: On
100%

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
GetHandler	Returns an instance of an HTTP handler to serve the request.
ReleaseHandler	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
BeginProcessRequest	Initiates an asynchronous call to the specified HTTP handler
EndProcessRequest	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 *IAsyncResult* interface. The *IAsyncResult* 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(IAsyncResult 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 *IAsyncResult* 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(IAsyncResult result);
```

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

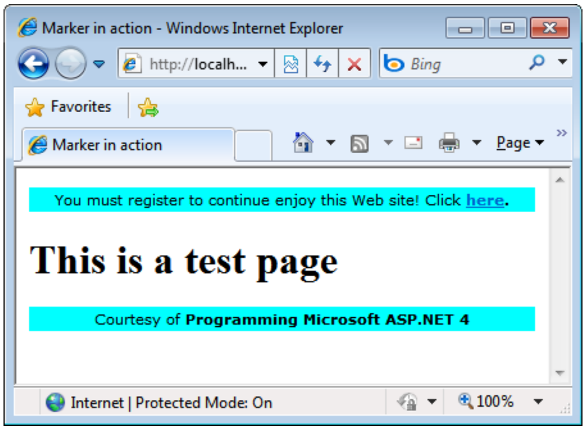
The following listing shows the implementation of the *Init* and *Dispose* methods for the sample module:

```
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.


Marker in action - Windows Internet Explorer
http://localh...
Bing
Favorites
Marker in action
Page
Internet | Protected Mode: On
100%

You must register to continue enjoy this Web site! Click here.

This is a test page

Courtesy of **Programming Microsoft ASP.NET 4**

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
AnonymousIdentificationModule	Manages anonymous identifiers for the ASP.NET application
DefaultAuthenticationModule	Ensures that the *User* object is always bound to some identity
FileAuthorizationModule	Verifies that the user has permission to access the given file.
FormsAuthenticationModule	Manages Forms authentication
OutputCacheModule	Implements output page caching
ProfileModule	Implements the data retrieval for profile data
RoleManagerModule	Manages the retrieval of role information
ScriptModule	Manages script requests placed through ASP.NET AJAX
SessionStateModule	Manages session state
UrlAuthorizationModule	Verifies that the user has permission to access the given URL
UrlRoutingModule	Implements URL routing
WindowsAuthenticationModule	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 cur- rent 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
DataTokens	List of additional custom values that are passed to the route handler
GetRequiredString	Method that takes the name of a route parameter and returns its value
Route	Returns the current *Route* object
RouteHandler	Returns the handler for the current route
Values	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
Constraints	List of additional constraints the URL should fulfill to match the route.
DataTokens	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.
Defaults	List of default values to be used for route parameters.
RouteHandler	The object responsible for retrieving the HTTP handler to serve the request.
Url	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

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.

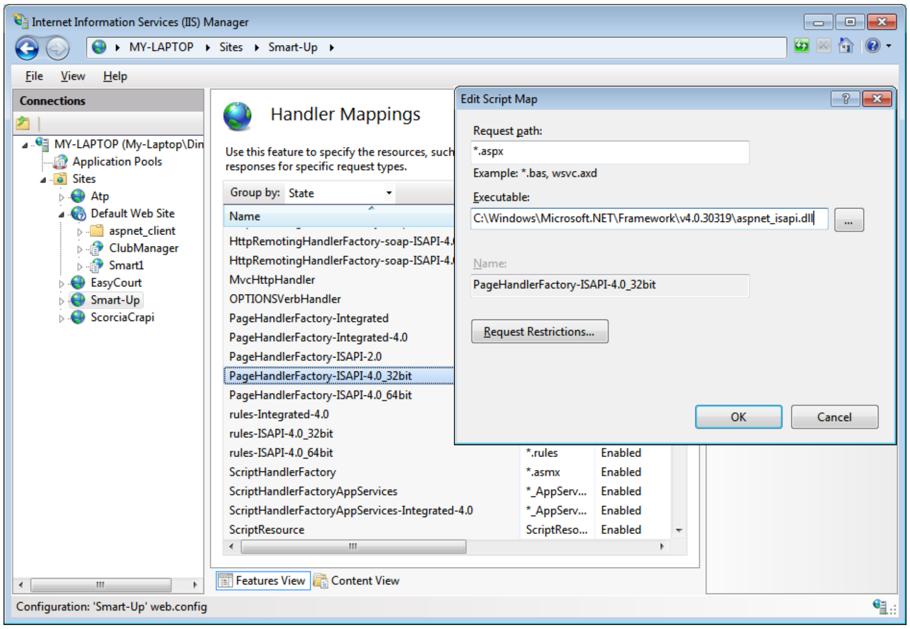
The compiled page remains in use as long as no changes occur to the linked *.aspx* source file or the whole application is restarted. Any changes to the linked *.aspx* file invalidate the current page-specific assembly and force the HTTP runtime to create a new assembly on the next request for the page.

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.)



Use this feature to specify the resources, such responses for specific request types.

Name		
HttpRemotingHandlerFactory-soap-ISAPI-4.		
HttpRemotingHandlerFactory-soap-ISAPI-4.		
MvcHttpHandler		
OPTIONSVerbHandler		
PageHandlerFactory-Integrated		
PageHandlerFactory-Integrated-4.0		
PageHandlerFactory-ISAPI-2.0		
PageHandlerFactory-ISAPI-4.0_32bit		
PageHandlerFactory-ISAPI-4.0_64bit		
rules-Integrated-4.0		
rules-ISAPI-4.0_32bit		
rules-ISAPI-4.0_64bit	*.rules	Enabled
ScriptHandlerFactory	*.asmx	Enabled
ScriptHandlerFactoryAppServices	*_AppServ...	Enabled
ScriptHandlerFactoryAppServices-Integrated-4.0	*_AppServ...	Enabled
ScriptResource	ScriptReso...	Enabled

Request path: *.aspx
Example: *.bas, wsvc.axd
Executable: C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_isapi.dll
Name: PageHandlerFactory-ISAPI-4.0_32bit

Configuration: 'Smart-Up' web.config

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.dl***

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

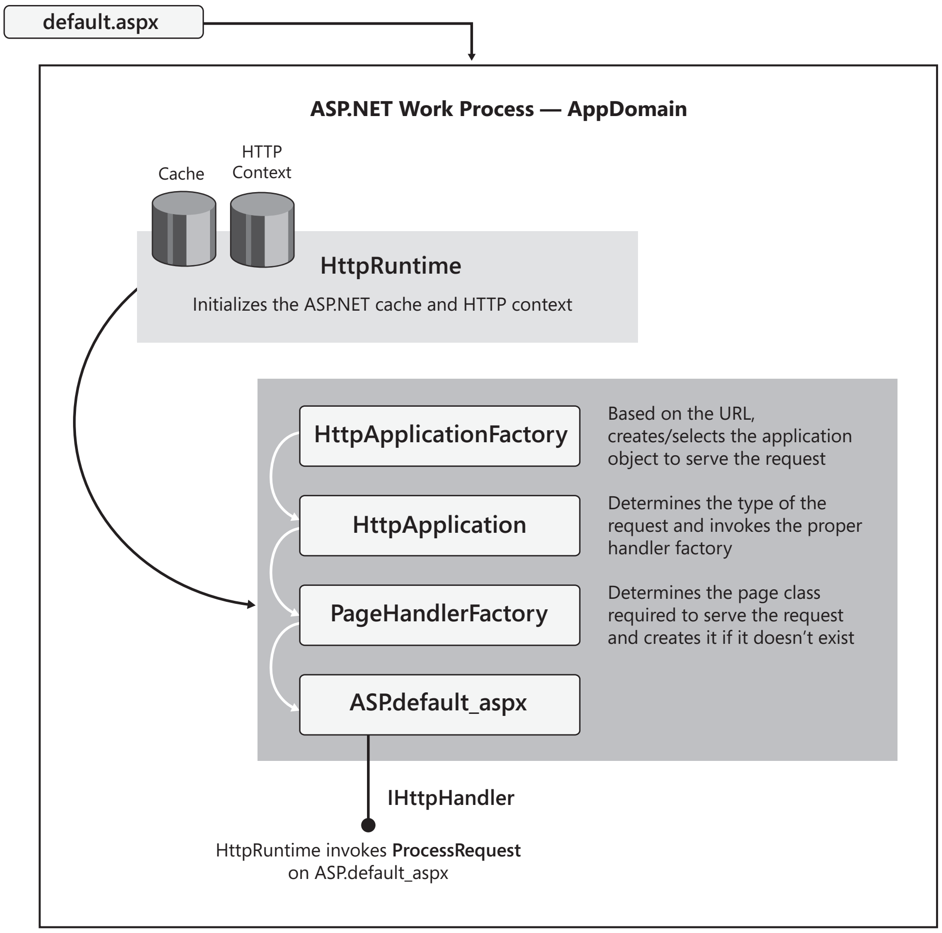

default.aspx
ASP.NET Work Process — AppDomain
Cache
HTTP Context
HttpRuntime
Initializes the ASP.NET cache and HTTP context
HttpApplicationFactory
Based on the URL, creates/selects the application object to serve the request
HttpApplication
Determines the type of the request and invokes the proper handler factory
PageHandlerFactory
Determines the page class required to serve the request and creates it if it doesn't exist
ASP.default_aspx
IHttpHandler
HttpRuntime invokes ProcessRequest on ASP.default_aspx

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.

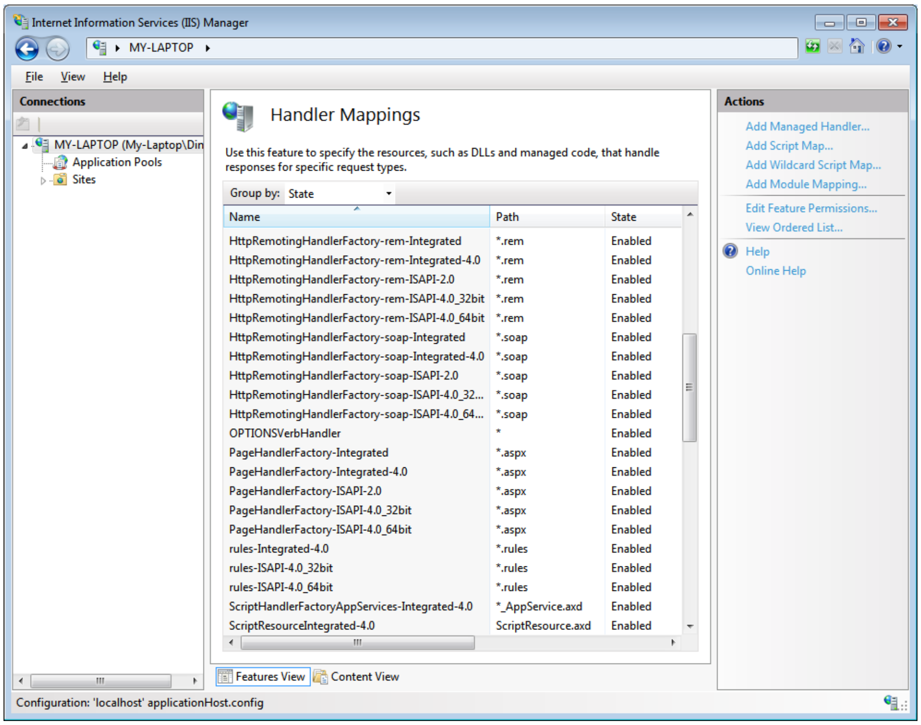
An instance of an *HttpApplication*-derived class is responsible for managing the entire lifetime of the request it is assigned to. The same instance can be reused only after the request has been completed. The *HttpApplication* maintains a list of HTTP module objects that can filter and even modify 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.



Handler Mappings

Use this feature to specify the resources, such as DLLs and managed code, that handle responses for specific request types.

Group by: State

Name	Path	State
HttpRemotingHandlerFactory-rem-Integrated	*.rem	Enabled
HttpRemotingHandlerFactory-rem-Integrated-4.0	*.rem	Enabled
HttpRemotingHandlerFactory-rem-ISAPI-2.0	*.rem	Enabled
HttpRemotingHandlerFactory-rem-ISAPI-4.0_32bit	*.rem	Enabled
HttpRemotingHandlerFactory-rem-ISAPI-4.0_64bit	*.rem	Enabled
HttpRemotingHandlerFactory-soap-Integrated	*.soap	Enabled
HttpRemotingHandlerFactory-soap-Integrated-4.0	*.soap	Enabled
HttpRemotingHandlerFactory-soap-ISAPI-2.0	*.soap	Enabled
HttpRemotingHandlerFactory-soap-ISAPI-4.0_32...	*.soap	Enabled
HttpRemotingHandlerFactory-soap-ISAPI-4.0_64...	*.soap	Enabled
OPTIONSVerbHandler	*	Enabled
PageHandlerFactory-Integrated	*.aspx	Enabled
PageHandlerFactory-Integrated-4.0	*.aspx	Enabled
PageHandlerFactory-ISAPI-2.0	*.aspx	Enabled
PageHandlerFactory-ISAPI-4.0_32bit	*.aspx	Enabled
PageHandlerFactory-ISAPI-4.0_64bit	*.aspx	Enabled
rules-Integrated-4.0	*.rules	Enabled
rules-ISAPI-4.0_32bit	*.rules	Enabled
rules-ISAPI-4.0_64bit	*.rules	Enabled
ScriptHandlerFactoryAppServices-Integrated-4.0	*_AppService.axd	Enabled
ScriptResourceIntegrated-4.0	ScriptResource.axd	Enabled

Configuration: 'localhost' applicationHost.config

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
@ Assembly	Links an assembly to the current page or user control.
@ Control	Defines control-specific attributes that guide the behavior of the control compiler.
@ Implements	Indicates that the page, or the user control, implements a specified .NET Framework interface.
@ Import	Indicates a namespace to import into a page or user control.
@ Master	Identifies an ASP.NET master page. (See Chapter 8, "Page Composition and Usability.")
@ MasterType	Provides a way to create a strongly typed reference to the ASP.NET master page when the master page is accessed from the *Master* property. (See Chapter 8.)
@ OutputCache	Controls the output caching policies of a page or user control. (See Chapter 18, "ASP.NET Caching.")
@ Page	Defines page-specific attributes that guide the behavior of the page compiler and the language parser that will preprocess the page.
@ PreviousPageType	Provides a way to get strong typing against the previous page, as accessed through the *PreviousPage* property.
@ Reference	Links a page or user control to the current page or user control.
@ Register	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
ClassName	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.
CodeFile	Indicates the path to the code-behind class for the current page. The source class file must be deployed to the Web server.
CodeBehind	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.
CodeFileBaseClass	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 *CodeFile* attribute must also be present.
CompilationMode	Indicates whether the page should be compiled at run time.
CompilerOptions	A sequence of compiler command-line switches used to compile the page.
Debug	A Boolean value that indicates whether the page should be compiled with debug symbols.
Explicit	A Boolean value that determines whether the page is compiled with the Visual Basic *Option Explicit* mode set to *On*. *Option Explicit* forces the programmer to explicitly declare all variables. The attribute is ignored if the page language is not Visual Basic .NET.

Attribute	Description
Inherits	Defines the base class for the page to inherit. It can be any class derived from the *Page* class.
Language	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.
LinePragmas	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.
MasterPageFile	Indicates the master page for the current page.
Src	Indicates the source file that contains the implementation of the base class specified with *Inherits*. The attribute is not used by Visual Studio and other Rapid Application Development (RAD) designers.
Strict	A Boolean value that determines whether the page is compiled with the Visual Basic *Option Strict* mode set to *On*. When this attribute is enabled, *Option Strict* 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.
Trace	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 *false*.
TraceMode	Indicates how trace messages are to be displayed for the page when tracing is enabled. Feasible values are *SortByTime* and *SortByCategory*. The default, when tracing is enabled, is *SortByTime*.
WarningLevel	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
AspCompat	A Boolean attribute that, when set to *true*, 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.")
Async	If this attribute is set to *true*, the generated page class derives from *IHttpAsyncHandler* rather than having *IHttpHandler* add some built-in asynchronous capabilities to the page.

Attribute	Description
AsyncTimeOut	Defines the timeout in seconds used when processing asynchronous tasks. The default is 45 seconds.
AutoEventWireup	A Boolean attribute that indicates whether page events are automatically enabled. It's set to *true* by default. Pages developed with Visual Studio .NET have this attribute set to *false*, and page events for these pages are individually tied to handlers.
Buffer	A Boolean attribute that determines whether HTTP response buffering is enabled. It's set to *true* by default.
Description	Provides a text description of the page. The ASP.NET page parser ignores the attribute, which subsequently has only a documentation purpose.
EnableEventValidation	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 *true* by default.
EnableSessionState	Defines how the page should treat session data. If this attribute is set to *true*, the session state can be read and written to. If it's set to *false*, session data is not available to the application. Finally, if this attribute is set to *ReadOnly*, the session state can be read but not changed.
EnableViewState	A Boolean value that indicates whether the page *view state* 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.")
EnableTheming	A Boolean value that indicates whether the page will support themes for embedded controls. It's set to *true* by default.
EnableViewStateMac	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 *Mac* in the attribute name stands for *machine authentication check*. When the attribute is *true*, 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.
ErrorPage	Defines the target URL to which users will be automatically redirected in case of unhandled page exceptions.
MaintainScrollPositionOnPostback	A Boolean value that indicates whether to return the user to the same position in the client browser after postback.
SmartNavigation	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.
Theme, StylesheetTheme	Indicates the name of the theme (or style-sheet theme) selected for the page.

Attribute	Description
Transaction	Indicates whether the page supports or requires transactions. Feasible values are *Disabled*, *NotSupported*, *Supported*, *Required*, and *RequiresNew*. Transaction support is disabled by default.
ValidateRequest	A Boolean value that indicates whether request validation should occur. If this attribute is set to *true*, 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 *true* 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
ClientTarget	Indicates the target browser for which ASP.NET server controls should render content.
ClientIDMode	Specifies the algorithm to use to generate client ID values for server controls. *This attribute requires ASP.NET 4.*
CodePage	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.
ContentType	Defines the content type of the response as a standard MIME type. Supports any valid HTTP content type string.
Culture	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, *en-US* is a valid value, unlike *en* alone, which is considered country/region neutral.
LCID	A 32-bit value that defines the locale identifier for the page. By default, ASP.NET uses the locale of the Web server.
MetaDescription	Sets the "description" meta element for the page. The value set through the *@Page* directive overrides any similar values you might have specified as literal text in the markup. *This attribute requires ASP.NET 4.*
MetaKeywords	Sets the "keywords" meta element for the page. The value set through the *@Page* directive overrides any similar values you might have specified as literal text in the markup. *This attribute requires ASP.NET 4.*
ResponseEncoding	Indicates the character encoding of the page. The value is used to set the *CharSet* attribute on the content type HTTP header. Internally, ASP.NET handles all strings as Unicode.

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

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

System.Data.dll	Defines data container and data access classes, including the whole ADO.NET framework
System.Data.DataSetExtensions.dll	Defines additional functions built over the ADO.NET *DataSet* object
System.Drawing.dll	Implements the GDI+ features
System.EnterpriseServices.dll	Provides the classes that allow for serviced components and COM+ interaction
System.Web.dll	Indicates the assembly implements the core ASP.NET services, controls, and classes
System.Web.ApplicationServices.dll	Provides classes that enable you to access ASP.NET authentication, roles, and profile functions via a bunch of built-in WCF services
System.Web.DynamicData.dll	Provides classes behind the ASP.NET Dynamic Data framework
System.Web.Entity.dll	Contains the code for the *EntityDataSource* component that supports Entity Framework
System.Web.Extensions.dll	Contains the code for AJAX extensions to ASP.NET
System.Web.Services.dll	Contains the core code that makes Web services run
System.Xml.dll	Implements the .NET Framework XML features
System.Xml.Linq.dll	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
Application	Instance of the *HttpApplicationState* class; represents the state of the application. It is functionally equivalent to the ASP intrinsic *Application* object.
Cache	Instance of the *Cache* class; implements the cache for an ASP.NET application. More efficient and powerful than *Application*, it supports item priority and expiration.
Profile	Instance of the *ProfileCommon* class; represents the user-specific set of data associated with the request.
Request	Instance of the *HttpRequest* class; represents the current HTTP request.
Response	Instance of the *HttpResponse* class; sends HTTP response data to the client.
RouteData	Instance of the *RouteData* class; groups information about the selected route (if any) and its values and tokens. (Routing in Web Forms is covered in Chapter 4, "xxx.") *The object is supported only in ASP.NET 4*.
Server	Instance of the *HttpServerUtility* class; provides helper methods for processing Web requests.
Session	Instance of the *HttpSessionState* class; manages user-specific data.
Trace	Instance of the *TraceContext* class; performs tracing on the page.
User	An *IPrincipal* 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
AutoPostBackControl	Gets a reference to the control within the page that caused the postback event.
ClientScript	Gets a *ClientScriptManager* object that contains the client script used on the page.
Controls	Returns the collection of all the child controls contained in the current page.
ErrorPage	Gets or sets the error page to which the requesting browser is redirected in case of an unhandled page exception.
Form	Returns the current *HtmlForm* object for the page.
Header	Returns a reference to the object that represents the page's header. The object implements *IPageHeader*.
IsAsync	Indicates whether the page is being invoked through an asynchronous handler.
IsCallback	Indicates whether the page is being loaded in response to a client script callback.
IsCrossPagePostBack	Indicates whether the page is being loaded in response to a postback made from within another page.
IsPostBack	Indicates whether the page is being loaded in response to a client postback or whether it is being loaded for the first time.
IsValid	Indicates whether page validation succeeded.
Master	Instance of the *MasterPage* class; represents the master page that determines the appearance of the current page.
MasterPageFile	Gets and sets the master file for the current page.
NamingContainer	Returns *null*.
Page	Returns the current *Page* object.
PageAdapter	Returns the adapter object for the current *Page* object.
Parent	Returns *null*.
PreviousPage	Returns the reference to the caller page in case of a cross-page postback.
TemplateSourceDirectory	Gets the virtual directory of the page.
Validators	Returns the collection of all validation controls contained in the page.
ViewStateUserKey	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
ClientID	Always returns the empty string.
ClientIDMode	Determines the algorithm to use to generate the ID of HTML elements being output as part of a control's markup. *This property requires ASP.NET 4.*
ClientQueryString	Gets the query string portion of the requested URL.
ClientTarget	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.
EnableViewState	Indicates whether the page has to manage view-state data. You can also enable or disable the view-state feature through the *EnableViewState* attribute of the *@Page* directive.
EnableViewStateMac	Indicates whether ASP.NET should calculate a machine-specific authentication code and append it to the page view state.
EnableTheming	Indicates whether the page supports themes.
ID	Always returns the empty string.
MetaDescription	Gets and sets the content of the *description* meta tag. *This property requires ASP.NET 4.*
MetaKeywords	Gets and sets the content of the *keywords* meta tag. *This property requires ASP.NET 4.*
MaintainScrollPositionOnPostback	Indicates whether to return the user to the same position in the client browser after postback.
SmartNavigation	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
StyleSheetTheme	Gets or sets the name of the style sheet applied to this page.
Theme	Gets and sets the theme for the page. Note that themes can be programmatically set only in the *PreInit* event.
Title	Gets or sets the title for the page.
TraceEnabled	Toggles page tracing on and off.
TraceModeValue	Gets or sets the trace mode.
UniqueID	Always returns the empty string.
ViewStateEncryptionMode	Indicates if and how the view state should be encrypted.
ViewStateMode	Enables the view state for an individual control even if the view state is disabled for the page. *This property requires ASP.NET 4.*
Visible	Indicates whether ASP.NET has to render the page. If you set *Visible* to *false*, ASP.NET doesn't generate any HTML code for the page. When *Visible* is *false*, only the text explicitly written using *Response.Write* 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
DataBind	Binds all the data-bound controls contained in the page to their data sources. The *DataBind* method doesn't generate code itself but prepares the ground for the forthcoming rendering.
RenderControl	Outputs the HTML text for the page, including tracing information if tracing is enabled.
VerifyRenderingInServerForm	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 <IMG> 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
DesignerInitialize	Initializes the instance of the *Page* class at design time, when the page is being hosted by RAD designers such as Visual Studio.
FindControl	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
GetTypeHashCode	Retrieves the hash code generated by *ASP.xxx_aspx* page objects at run time. In the base *Page* class, the method implementation simply returns *0*; significant numbers are returned by classes used for actual pages.
GetValidators	Returns a collection of control validators for a specified validation group.
HasControls	Determines whether the page contains any child controls.
LoadControl	Compiles and loads a user control from an *.ascx* file, and returns a *Control* object. If the user control supports caching, the object returned is *PartialCachingControl*.
LoadTemplate	Compiles and loads a user control from an *.ascx* file, and returns it wrapped in an instance of an internal class that implements the *ITemplate* interface. The internal class is named *SimpleTemplate*.
MapPath	Retrieves the physical, fully qualified path that an absolute or relative virtual path maps to.
ParseControl	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 *runat* attribute can be omitted. The method returns an object of type *Control* and must be cast to a more specific type.
RegisterRequiresControlState	Registers a control as one that requires control state.
RegisterRequiresPostBack	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 *IPostBackDataHandler* interface.
RegisterRequiresRaiseEvent	Registers the specified control to handle an incoming postback event. The control must implement the *IPostBackEventHandler* interface.
RegisterViewStateHandler	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 *HtmlForm* server control for the page calls this method. There's no need to call it from within user applications.
ResolveUrl	Resolves a relative URL into an absolute URL based on the value of the *TemplateSourceDirectory* property.
Validate	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
GetCallbackEventReference	Obtains a reference to a client-side function that, when invoked, initiates a client callback to server-side events.
GetPostBackClientEvent	Calls into *GetCallbackEventReference*.
GetPostBackClientHyperlink	Appends *javascript:* to the beginning of the return string received from *GetPostBackEventReference*. For example: *javascript:__doPostBack('CtlID','')*
GetPostBackEventReference	Returns the prototype of the client-side script function that causes, when invoked, a postback. It takes a *Control* and an argument, and it returns a string like this: *__doPostBack('CtlID','')*
IsClientScriptBlockRegistered	Determines whether the specified client script is registered with the page. *It's marked as obsolete.*
IsStartupScriptRegistered	Determines whether the specified client startup script is registered with the page. *It's marked as obsolete.*

Method	Description
RegisterArrayDeclaration	Use this method to add an *ECMAScript* 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 *theArray* and *"'a', 'b'"*, you get the following JavaScript code: *var theArray = new Array('a', 'b');* *It's marked as obsolete.*
RegisterClientScriptBlock	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 *<form>* element. *It's marked as obsolete.*
RegisterHiddenField	Use this method to automatically register a hidden field on the page. *It's marked as obsolete.*
RegisterOnSubmitStatement	Use this method to emit client script code that handles the client *OnSubmit* event. The script should be a JavaScript function call to client code registered elsewhere. *It's marked as obsolete.*
RegisterStartupScript	An ASP.NET page uses this method to emit client-side script blocks in the client page just before closing the HTML *<form>* element. *It's marked as obsolete.*
SetFocus	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
AbortTransaction	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction aborts
CommitTransaction	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction commits
DataBinding	Occurs when the *DataBind* method is called on the page to bind all the child controls to their respective data sources
Disposed	Occurs when the page is released from memory, which is the last stage of the page life cycle
Error	Occurs when an unhandled exception is thrown.
Init	Occurs when the page is initialized, which is the first step in the page life cycle
InitComplete	Occurs when all child controls and the page have been initialized
Load	Occurs when the page loads up, after being initialized
LoadComplete	Occurs when the loading of the page is completed and server events have been raised
PreInit	Occurs just before the initialization phase of the page begins
PreLoad	Occurs just before the loading phase of the page begins
PreRender	Occurs when the page is about to render
PreRenderComplete	Occurs just before the pre-rendering phase begins
SaveStateComplete	Occurs when the view state of the page has been saved to the persistence medium
Unload	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:

```
IAsyncResult BeginEventHandler(
    object sender,
    EventArgs e,
    AsyncCallback cb,
    object state)
void EndEventHandler(
    IAsyncResult 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 *IAsyncResult* 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 *IAsyncResult* interface. In most cases, in fact, to start lengthy operations you just use built-in classes that already implement the asynchronous pattern and provide *IAsyncResult* 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

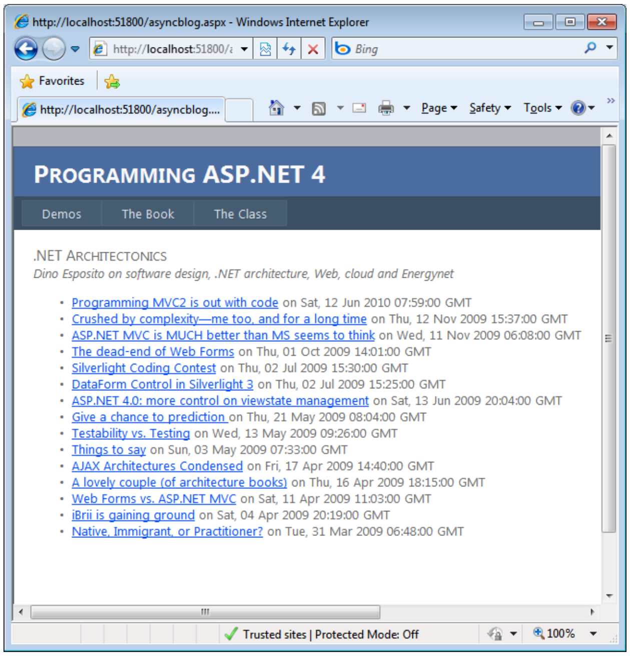
Let's roll a first asynchronous test page to download and process some RSS feeds. The page markup is quite simple indeed:

```
<%@ 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.


http://localhost:51800/asyncblog.aspx - Windows Internet Explorer
http://localhost:51800/
Bing
Favorites
http://localhost:51800/asyncblog....
Page
Safety
Tools
Trusted sites | Protected Mode: Off
100%

PROGRAMMING ASP.NET 4

Demos The Book The Class

.NET ARCHITECTONICS

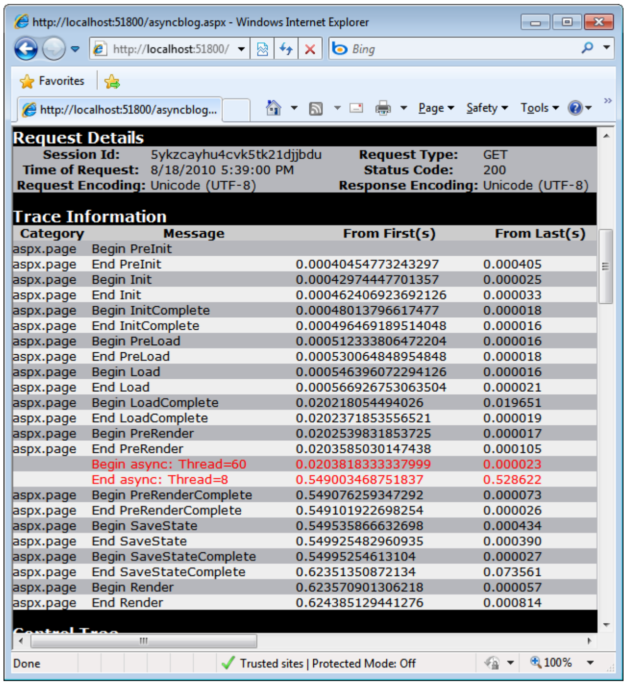
Dino Esposito on software design, .NET architecture, Web, cloud and Energynet

- Programming MVC2 is out with code on Sat, 12 Jun 2010 07:59:00 GMT
- Crushed by complexity—me too, and for a long time on Thu, 12 Nov 2009 15:37:00 GMT
- ASP.NET MVC is MUCH better than MS seems to think on Wed, 11 Nov 2009 06:08:00 GMT
- The dead-end of Web Forms on Thu, 01 Oct 2009 14:01:00 GMT
- Silverlight Coding Contest on Thu, 02 Jul 2009 15:30:00 GMT
- DataForm Control in Silverlight 3 on Thu, 02 Jul 2009 15:25:00 GMT
- ASP.NET 4.0: more control on viewstate management on Sat, 13 Jun 2009 20:04:00 GMT
- Give a chance to prediction on Thu, 21 May 2009 08:04:00 GMT
- Testability vs. Testing on Wed, 13 May 2009 09:26:00 GMT
- Things to say on Sun, 03 May 2009 07:33:00 GMT
- AJAX Architectures Condensed on Fri, 17 Apr 2009 14:40:00 GMT
- A lovely couple (of architecture books) on Thu, 16 Apr 2009 18:15:00 GMT
- Web Forms vs. ASP.NET MVC on Sat, 11 Apr 2009 11:03:00 GMT
- iBrii is gaining ground on Sat, 04 Apr 2009 20:19:00 GMT
- Native, Immigrant, or Practitioner? on Tue, 31 Mar 2009 06:48:00 GMT

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.


http://localhost:51800/asyncblog.aspx - Windows Internet Explorer
http://localhost:51800/
Bing
Favorites
http://localhost:51800/asyncblog...
Page
Safety
Tools
Done
Trusted sites | Protected Mode: Off
100%

Request Details

Session Id:	5ykzcayhu4cvk5tk21djjbdu	**Request Type:**	GET
Time of Request:	8/18/2010 5:39:00 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.00040454773243297	0.000405
aspx.page	Begin Init	0.00042974447701357	0.000025
aspx.page	End Init	0.000462406923692126	0.000033
aspx.page	Begin InitComplete	0.00048013796617477	0.000018
aspx.page	End InitComplete	0.000496469189514048	0.000016
aspx.page	Begin PreLoad	0.000512333806472204	0.000016
aspx.page	End PreLoad	0.000530064848954848	0.000018
aspx.page	Begin Load	0.000546396072294126	0.000016
aspx.page	End Load	0.000566926753063504	0.000021
aspx.page	Begin LoadComplete	0.020218054494026	0.019651
aspx.page	End LoadComplete	0.0202371853556521	0.000019
aspx.page	Begin PreRender	0.0202539831853725	0.000017
aspx.page	End PreRender	0.0203585030147438	0.000105
	Begin async: Thread=60	0.0203818333337999	0.000023
	End async: Thread=8	0.549003468751837	0.528622
aspx.page	Begin PreRenderComplete	0.549076259347292	0.000073
aspx.page	End PreRenderComplete	0.549101922698254	0.000026
aspx.page	Begin SaveState	0.549535866632698	0.000434
aspx.page	End SaveState	0.549925482960935	0.000390
aspx.page	Begin SaveStateComplete	0.54995254613104	0.000027
aspx.page	End SaveStateComplete	0.62351350872134	0.073561
aspx.page	Begin Render	0.623570901306218	0.000057
aspx.page	End Render	0.624385129441276	0.000814

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 *IAsyncResult* 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 cleanup 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

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.

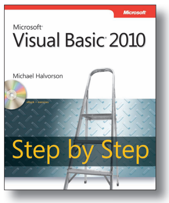
Every month, at least five different magazines and Web sites in one part of the world or another publish Dino's articles, which cover topics ranging from Web development to data access and from software best practices to Android, Ajax, Silverlight, and JavaScript. A prolific author, Dino writes the monthly "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.

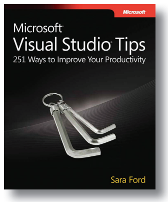
Dino has written an array of books, most of which are considered state-of-the-art in their respective areas. His more recent books are Programming ASP.NET MVC 3 (Microsoft Press, 2011) and Microsoft .NET: Architecting Applications for the Enterprise (Microsoft Press, 2008), which is slated for an update 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.

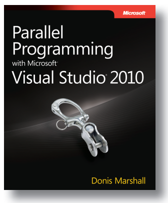
In his spare time (so to speak), Dino manages software development and training activities at **Crionet** and is the brains behind some software applications for live scores and sporting clubs.

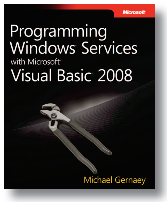
If you would like to get in touch with Dino for whatever reason (for example, you're running a user group, company, community, portal, or play tennis), you can tweet him at **@despos** or reach him via Facebook.


Microsoft
Microsoft Visual Basic 2010
Michael Halvorson
Step by Step


Microsoft
Microsoft Visual Studio Tips
251 Ways to Improve Your Productivity
Sara Ford

Microsoft
Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build
2 Second Edition
Sayed Ibrahim Hashimi
William Bartholomew


Microsoft
Parallel Programming with Microsoft Visual Studio 2010
Donis Marshall


Microsoft
Programming Windows Services with Microsoft Visual Basic 2008
Michael Gernaey