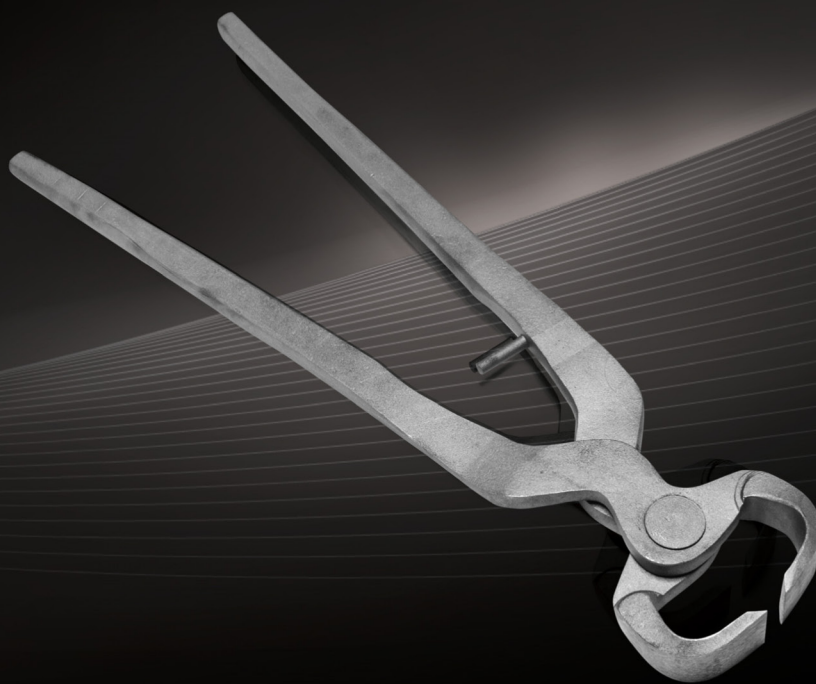


# Programming

Microsoft®

# ASP.NET 3.5



Dino Esposito

## Chapter 3

# Anatomy of an ASP.NET Page

### In this chapter:

Invoking a Page .....	89
The <i>Page</i> Class .....	112
The Page Life Cycle .....	132
Conclusion .....	138

ASP.NET pages are dynamically compiled on demand when first required in the context of a Web application. Dynamic compilation is not specific to ASP.NET pages (*.aspx* files); it also occurs with .NET Web Services (*.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—precisely, an instance of a class derived from the system’s *Page* class. 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.

## 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 invalidates the current page-specific assembly and forces the HTTP runtime to create a new assembly on the next request for page.

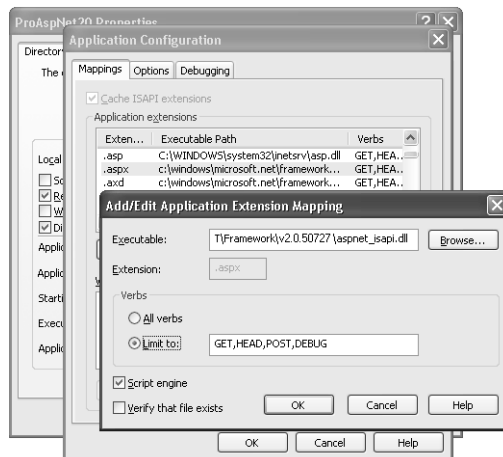


**Note** Editing files such as *web.config* and *global.asax* causes the whole application to restart. In this case, all the pages will be recompiled as soon as each page is requested. The same happens if a new assembly is copied or replaced in the application's *Bin* folder.

## The Runtime Machinery

All resources that you can access on an Internet Information Services (IIS)—based Web server are grouped by file extension. Any incoming request is then assigned to a particular run-time module for actual processing. Modules that can handle Web resources within the context of IIS are Internet Server Application Programming Interface (ISAPI) extensions—that is, plain old Win32 dynamic-link libraries (DLLs) that expose, much like an interface, a bunch of API functions with predefined names and prototypes. IIS and ISAPI extensions use these DLL entries as a sort of private communication protocol. When IIS needs an ISAPI extension to accomplish a certain task, it simply loads the DLL and calls the appropriate function with valid arguments. Although the ISAPI documentation doesn't mention an ISAPI extension as an interface, it is just that—a module that implements a well-known programming interface.

When the request for a resource arrives, IIS first verifies the type of the resource. Static resources such as images, text files, HTML pages, and scriptless ASP pages are resolved directly by IIS without the involvement of any external modules. IIS accesses the file on the local Web server and flushes its contents to the output console so that the requesting browser can get it. Resources that require server-side elaboration are passed on to the registered module. For example, ASP pages are processed by an ISAPI extension named *asp.dll*. In general, when the resource is associated with executable code, IIS hands the request to that executable for further processing. Files with an *.aspx* extension are assigned to an ISAPI extension named *aspnet\_isapi.dll*, as shown in Figure 3-1.



**FIGURE 3-1** The IIS application mappings for resources with an *.aspx* extension.

Resource mappings are stored in the IIS *metabase*, which is an IIS-specific configuration database. Upon installation, ASP.NET modifies the IIS metabase to make sure that *aspnet\_isapi.dll* can handle some typical ASP.NET resources. Table 3-1 lists some of these resources.

**TABLE 3-1 IIS Application Mappings for *aspnet\_isapi.dll***

Extension	Resource Type
.asax	ASP.NET application files such as <i>global.asax</i> . The mapping is there to ensure that <i>global.asax</i> 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 implement .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 ( <i>trace.axd</i> ) or script injection ( <i>webresource.axd</i> ).

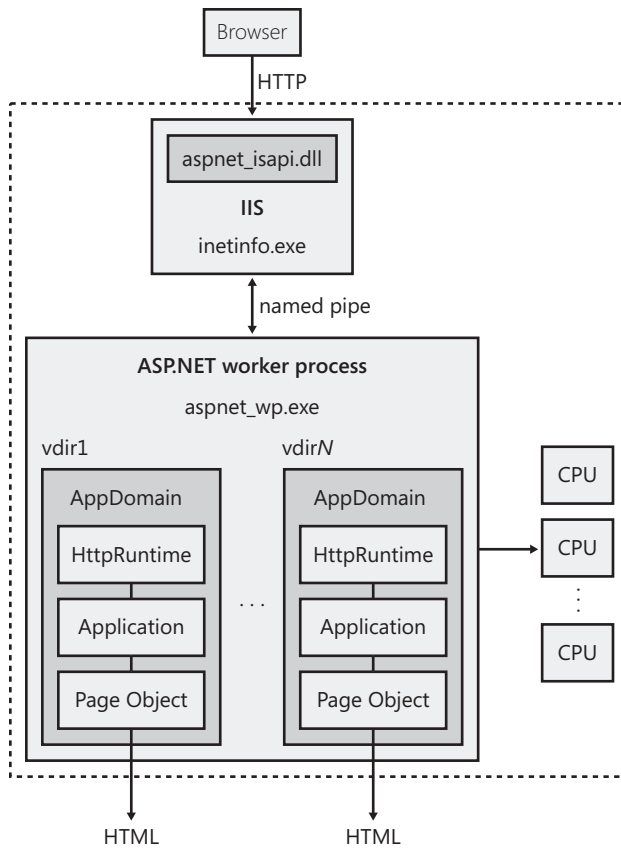
In addition, the *aspnet\_isapi.dll* extension handles other typical Microsoft Visual Studio extensions, such as *.cs*, *.csproj*, *.vb*, *.vbproj*, *.config*, and *.resx*.

As mentioned in Chapter 1, the exact behavior of the ASP.NET ISAPI extension depends on the process model selected for the application. There are two options, as described in the following sections.

## IIS 5.0 Process Model

The IIS 5.0 process model is the only option you have if you host your ASP.NET application on any version of Microsoft Windows prior to Windows 2003 Server. According to this processing model, *aspnet\_isapi.dll* doesn't process the *.aspx* file, but instead acts as a dispatcher. It collects all the information available about the invoked URL and the underlying resource, and then it routes the request toward another distinct process—the ASP.NET worker process named *aspnet\_wp.exe*. The communication between the ISAPI extension and worker process takes place through named pipes.

The whole model is illustrated in Figure 3-2.



**FIGURE 3-2** The ASP.NET runtime environment according to the IIS 5.0 process model.

A single copy of the worker process runs all the time and hosts all the active Web applications. The only exception to this situation is when you have a Web server with multiple CPUs. In this case, you can configure the ASP.NET runtime so that multiple worker processes run, one per each available CPU. A model in which multiple processes run on multiple CPUs in a single-server machine is known as a *Web garden* and is controlled by attributes on the `<processModel>` section in the *machine.config* file.

When a single worker process is used by all CPUs and controls all Web applications, it doesn't necessarily mean that no process isolation is achieved. Each Web application is, in fact, identified with its virtual directory and belongs to a distinct *application domain*, commonly referred to as an AppDomain. A new AppDomain is created within the ASP.NET worker process whenever a client addresses a virtual directory for the first time. After creating the new AppDomain, the ASP.NET runtime loads all the needed assemblies and passes control to the hosted HTTP pipeline to actually service the request.

If a client requests a page from an already running Web application, the ASP.NET runtime simply forwards the request to the existing AppDomain associated with that virtual directory. If the assembly needed to process the page is not loaded in the AppDomain, it will be created on the fly; otherwise, if it was already created upon the first call, it will be simply used.

## IIS 6.0 Process Model

The IIS 6.0 process model is the default option for ASP.NET when the Web server operating system is Windows 2003 Server or newer. As the name of the process model clearly suggests, this model requires IIS 6.0. However, on a Windows 2003 Server machine you can still have ASP.NET play by the rules of the IIS 5.0 process model. If this is what you want, explicitly enable the model by tweaking the `<processModel>` section of the *machine.config* file, as shown here:

```
<processModel enable="true">
```

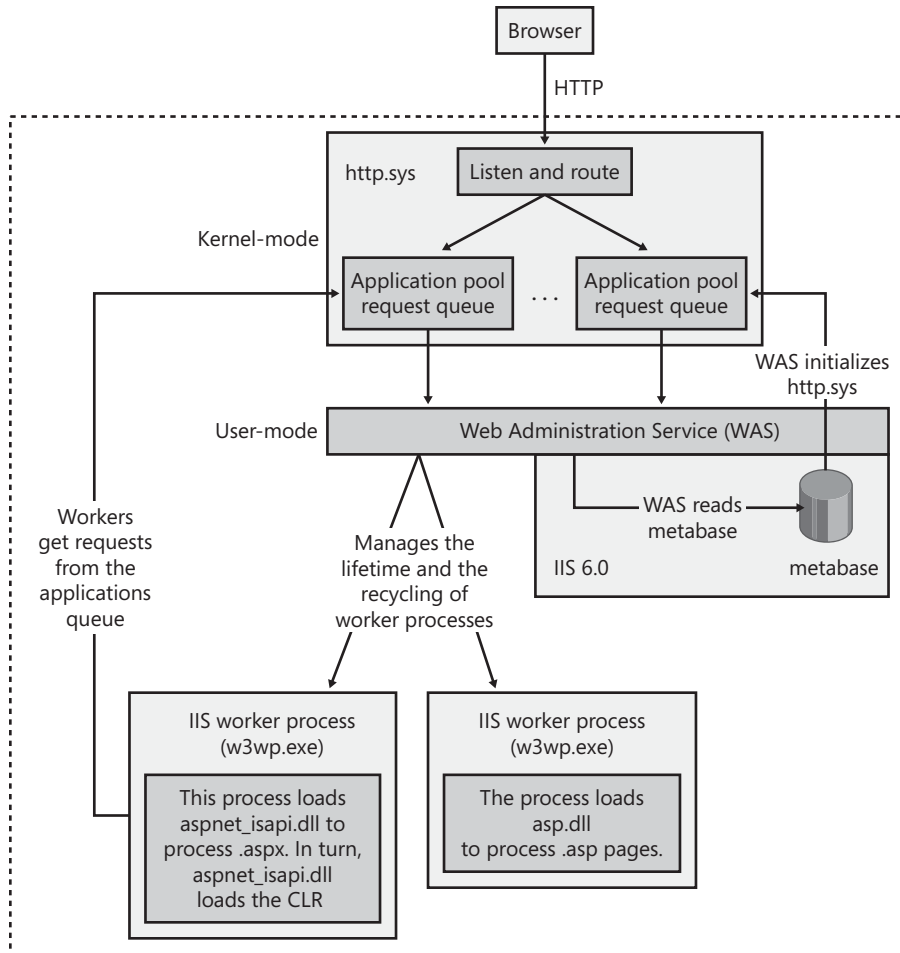
Be aware that switching back to the old IIS 5.0 process model is not a recommended practice, although it is perfectly legal. The main reason lies in the fact that IIS 6.0 employs a different pipeline of internal modules to process an inbound request and can mimic the behavior of IIS 5.0 only if running in emulation mode. The IIS 6.0 pipeline is centered around a generic worker process named *w3wp.exe*. A copy of this executable is shared by all Web applications assigned to the same application pool. In the IIS 6.0 jargon, an application pool is a group of Web applications that share the same copy of the worker process. IIS 6.0 lets you customize the application pools to achieve the degree of isolation that you need for the various applications hosted on a Web server.

The *w3wp.exe* worker process loads *aspnet\_isapi.dll*; the ISAPI extension, in turn, loads the common language runtime (CLR) and starts the ASP.NET runtime pipeline to process the request. When the IIS 6.0 process model is in use, the built-in ASP.NET worker process is disabled.



**Note** Only ASP.NET version 1.1 and later takes full advantage of the IIS 6.0 process model. If you install ASP.NET 1.0 on a Windows 2003 Server machine, the process model will default to the IIS 5.0 process model. This happens because only the version of *aspnet\_isapi.dll* that ships with ASP.NET 1.1 is smart enough to recognize its host and load the CLR if needed. The *aspnet\_isapi.dll* included in ASP.NET 1.0 is limited to forwarding requests to the ASP.NET worker process and never loads the CLR.

Figure 3-3 shows how ASP.NET applications and other Web applications are processed in IIS 6.0.



**FIGURE 3-3** How ASP.NET and Web applications are processed in IIS 6.0.

IIS 6.0 implements its HTTP listener as a kernel-level module. As a result, all incoming requests are first managed by a driver—*http.sys*. No third-party code ever interacts with the listener, and no user-mode crashes will ever affect the stability of IIS. The *http.sys* driver listens for requests and posts them to the request queue of the appropriate application pool. A module called the Web Administration Service (WAS) reads from the IIS metabase and instructs the *http.sys* driver to create as many request queues as there are application pools registered in the metabase.

In summary, in the IIS 6.0 process model, ASP.NET runs even faster because no interprocess communication between *inetinfo.exe* (the IIS executable) and the worker process is required. The HTTP request is delivered directly at the worker process that hosts the CLR. Furthermore, the ASP.NET worker process is not a special process but simply a copy of the IIS worker process. This fact shifts to IIS the burden of process recycling, page output caching, and health checks.

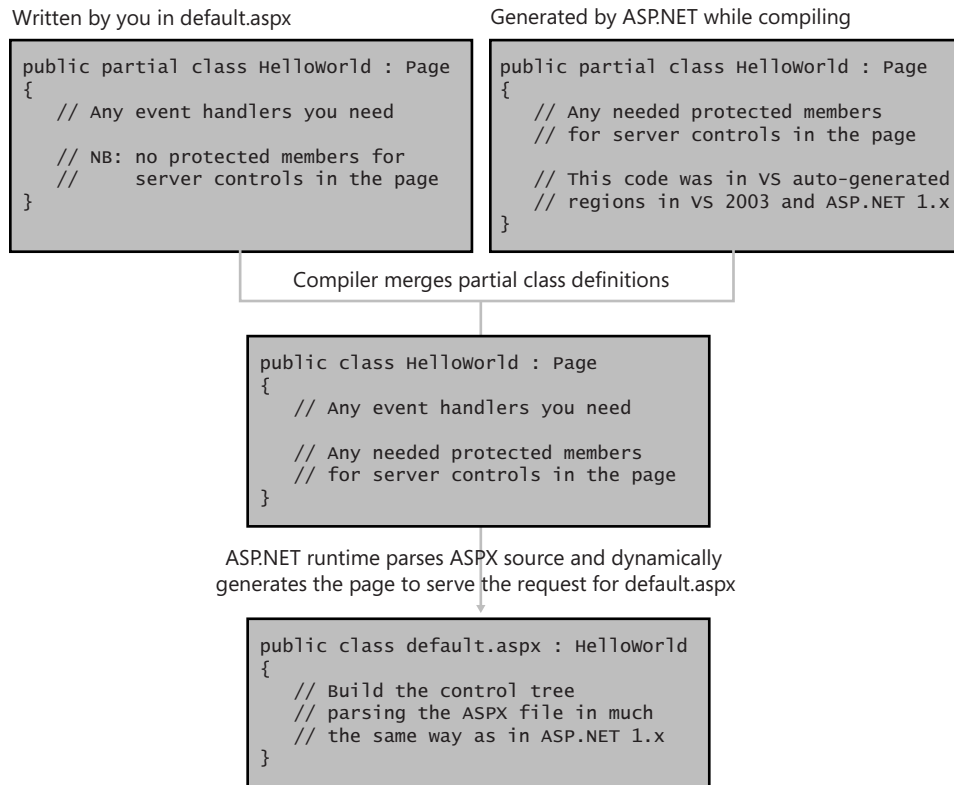
In the IIS 6.0 process model, ASP.NET ignores most of the contents of the `<processModel>` section from the *machine.config* file. Only thread and deadlock settings are read from that section of *machine.config*. Everything else goes through the metabase and can be configured only by using the IIS Manager. (Other configuration information continues to be read from *.config* files.)

## Representing the Requested Page

Each incoming request that refers to an *.aspx* resource is mapped to, and served through, a *Page*-derived class. 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*. If no class exists with that name in any of the assemblies currently loaded in the AppDomain, the HTTP runtime orders that the class be created and compiled. The source code for the class is created by parsing the source code of the *.aspx* resource, and it's temporarily saved in the ASP.NET temporary folder. Next, the 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.)

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 created through Visual Studio and a second partial class dynamically arranged by the ASP.NET HTTP runtime. Figure 3-4 provides a graphical demonstration of how the source code of the dynamic page class is built.





**FIGURE 3-4** ASP.NET generates the source code for the dynamic class that will serve a request.

Partial classes are a hot feature of the latest .NET compilers (version 2.0 and later). 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 new 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

Ideal for team development, partial classes simplify coding and avoid manual file synchronization in all situations in which a mix of user-defined and tool-generated code is used. Want an illustrious example? ASP.NET projects developed with Visual Studio 2003.

Partial classes are a compiler feature specifically designed to overcome the brittleness of tool-generated code in many Visual Studio 2003 projects, including ASP.NET projects. A savvy use of partial classes allows you to eliminate all those weird, auto-generated, semi-hidden regions of code that Visual Studio 2003 inserts to support page designers.

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. For example, 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 2003, these declarations are added by the integrated development environment (IDE) as you save your markup and stored in semi-hidden regions. In Visual Studio 2005, the code-behind class is a partial class that just lacks member 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.



**Note** In Visual Studio 2008 and the .NET Framework 3.5 partial classes are partnered with extension methods as a way to add new capabilities to existing .NET classes. By creating a class with extension methods you can extend, say, the `System.String` class with a `ToInt32` method that returns an integer if the content of the string can be converted to an integer. Once you added to the project the class with extension methods, any string in the project features the new methods. IntelliSense fully supports this feature.

## Processing the Request

To serve a request for a page named *default.aspx*, the ASP.NET runtime needs to get a reference to a class *ASP.default.aspx*. As you recall, if this class doesn't exist in any of the assemblies currently loaded in the AppDomain, it will be created. 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. Once 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 ASP.NET worker process—be it *w3wp.exe* or *aspnet\_wp.exe*—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*.

The *HttpRuntime* Object

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 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 3-2.

TABLE 3-2 Public Methods in the *HttpRuntime* Class

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

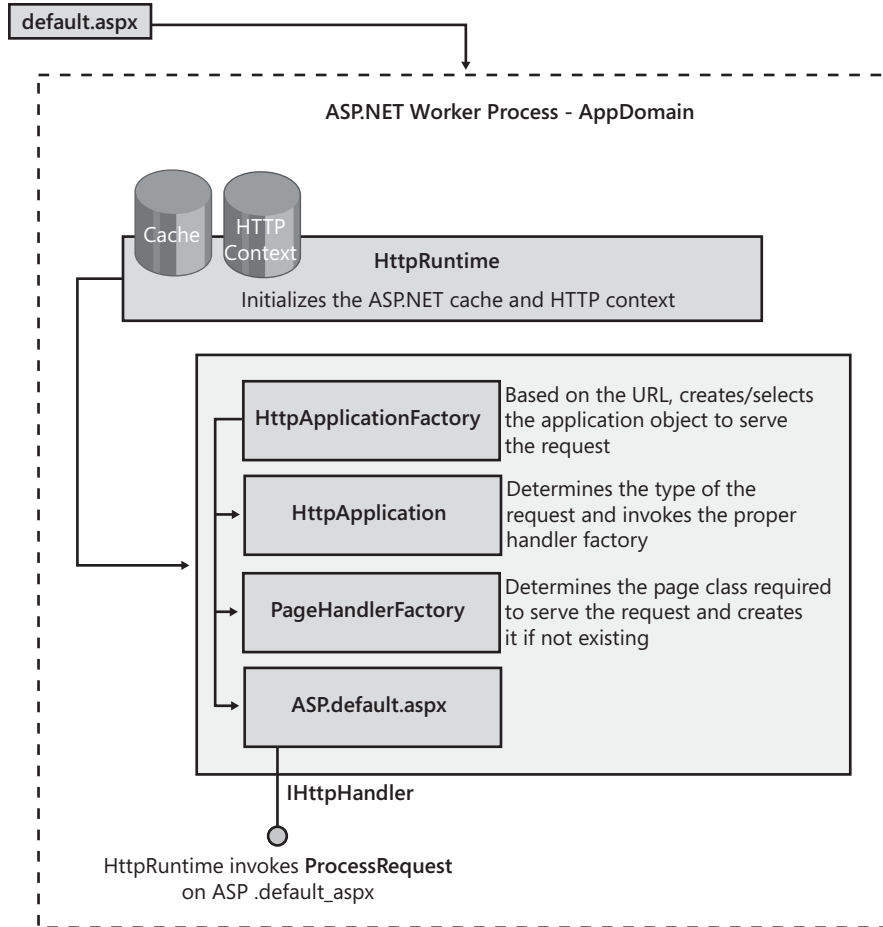
It is important to note that all the methods shown in Table 3-2 have limited applicability in user applications. In particular, you’re not supposed to use *ProcessRequest* in your own code, whereas *Close* is useful only if you’re hosting ASP.NET in a custom application. Of the three methods in Table 3-2, 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 3-5 is in order.



**FIGURE 3-5** 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 running ASP.NET application is represented by a dynamically created class that inherits from *HttpApplication*. The source code of the dynamically generated application class is created 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.

## The Page Factory

The *HttpApplication* class determines the type of object that must handle the request and delegates the type-specific handler factory to create an instance of that type. Let's see what happens when the resource requested is a page.

Once 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. Table 3-3 contains a brief list of the main handlers registered.

**TABLE 3-3 Handler Factory Classes in the .NET Framework**

Handler Factory	Type	Description
<i>HttpRemotingHandlerFactory</i>	*.rem; *.soap	Instantiates the object that will take care of a .NET Remoting request routed through IIS. Instantiates an object of type <i>HttpRemotingHandler</i> .
<i>PageHandlerFactory</i>	*.aspx	Compiles and instantiates the type that represents the page. The source code for the class is built while parsing the source code of the <i>.aspx</i> file. Instantiates an object of a type that derives from <i>Page</i> .
<i>SimpleHandlerFactory</i>	*.ashx	Compiles and instantiates the specified HTTP handler from the source code of the <i>.ashx</i> file. Instantiates an object that implements the <i>IHttpHandler</i> interface.
<i>WebServiceHandlerFactory</i>	*.asmx	Compiles the source code of a Web service, and translates the SOAP payload into a method invocation. Instantiates an object of the type specified in the Web service file.

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. (This bit of efficiency is the primary reason the factory pattern is followed in this case.)

So when the request is received, the page handler factory creates an instance of an object that represents the particular requested page. As mentioned, this 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 14, 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 that, created by the *HttpRuntime* class, is populated and 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 in the long run the overall behavior of the application. Another reason is that too many dynamic changes to ASPX pages may have caused too large a number of assemblies (typically, one per page) to be loaded in memory. Any application that consumes more than a certain share of virtual memory is killed and restarted. The ASP.NET runtime environment 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.)
- Changes occurred to settings that control the restart/shutdown of the ASP.NET worker process. These settings are read from *machine.config* if you don't use Windows 2003 Server with the IIS 6.0 process model. If you're taking full advantage of IIS 6.0, an application is restarted if you modify properties in the *Application Pools* node of the IIS manager.

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 runtime 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 3-4.

**TABLE 3-4 Directives Supported by ASP.NET Pages**

Directive	Description
<code>@ Assembly</code>	Links an assembly to the current page or user control.
<code>@ Control</code>	Defines control-specific attributes that guide the behavior of the control compiler.
<code>@ Implements</code>	Indicates that the page, or the user control, implements a specified .NET Framework interface.
<code>@ Import</code>	Indicates a namespace to import into a page or user control.
<code>@ Master</code>	Identifies an ASP.NET master page. (See Chapter 6.) <i>This directive is not available with ASP.NET 1.x.</i>
<code>@ MasterType</code>	Provides a way to create a strongly typed reference to the ASP.NET master page when the master page is accessed from the <i>Master</i> property. (See Chapter 6.) <i>This directive is not available with ASP.NET 1.x.</i>
<code>@ OutputCache</code>	Controls the output caching policies of a page or user control. (See Chapter 16.)
<code>@ Page</code>	Defines page-specific attributes that guide the behavior of the page compiler and the language parser that will preprocess the page.
<code>@ PreviousPageType</code>	Provides a way to get strong typing against the previous page, as accessed through the <i>PreviousPage</i> property.
<code>@ Reference</code>	Links a page or user control to the current page or user control.
<code>@ Register</code>	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 it generates a compile error if used with other types of ASP.NET pages, 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 about 30 attributes that can be logically grouped in three categories: compilation (defined in Table 3-5), overall page behavior (defined in Table 3-6), and page output (defined in Table 3-7). 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. Attributes listed in Table 3-5 let you fine-tune parameters for the compiler and choose the language to use.

TABLE 3-5 @Page Attributes for Page Compilation

Attribute	Description
<i>ClassName</i>	Specifies the name of the class name that will be dynamically compiled when the page is requested. Must be a class name without namespace information.
<i>CodeFile</i>	Indicates the path to the code-behind class for the current page. The source class file must be deployed to the Web server. <i>Not available with ASP.NET 1.x.</i>
<i>CodeBehind</i>	Attribute consumed by Visual Studio .NET 2003, indicates the path to the code-behind class for the current page. The source class file will be compiled to a deployable assembly. (Note that for ASP.NET version 2.0 and later, the <i>CodeFile</i> attribute should be used.)
<i>CodeFileBaseClass</i>	Specifies the type name of a base class for a page and its associated code-behind class. The attribute is optional, but when it is used the <i>CodeFile</i> attribute must also be present. <i>Not available with ASP.NET 1.x.</i>

Attribute	Description
<i>CompilationMode</i>	Indicates whether the page should be compiled at run time. <i>Not available with ASP.NET 1.x.</i>
<i>CompilerOptions</i>	A sequence of compiler command-line switches used to compile the page.
<i>Debug</i>	A Boolean value that indicates whether the page should be compiled with debug symbols.
<i>Explicit</i>	A Boolean value that determines whether the page is compiled with the Visual Basic <i>Option Explicit</i> mode set to <i>On</i> . <i>Option Explicit</i> forces the programmer to explicitly declare all variables. The attribute is ignored if the page language is not Visual Basic .NET.
<i>Inherits</i>	Defines the base class for the page to inherit. It can be any class derived from the <i>Page</i> class.
<i>Language</i>	Indicates the language to use when compiling inline code blocks (<% ... %>) and all the code that appears in the page <script> section. Supported languages include Visual Basic .NET, C#, JScript .NET, and J#. If not otherwise specified, the language defaults to Visual Basic .NET.
<i>LinePragmas</i>	Indicates whether the runtime should generate line pragmas in the source code
<i>MasterPageFile</i>	Indicates the master page for the current page. <i>Not available with ASP.NET 1.x.</i>
<i>Src</i>	Indicates the source file that contains the implementation of the base class specified with <i>Inherits</i> . The attribute is not used by Visual Studio and other rapid application development (RAD) designers.
<i>Strict</i>	A Boolean value that determines whether the page is compiled with the Visual Basic <i>Option Strict</i> mode set to <i>On</i> . When enabled, <i>Option Strict</i> permits only type-safe conversions and prohibits implicit conversions in which loss of data is possible. (In this case, the behavior is identical to that of C#.) The attribute is ignored if the page language is not Visual Basic .NET.
<i>Trace</i>	A Boolean value that indicates whether tracing is enabled. If tracing is enabled, extra information is appended to the page's output. The default is <i>false</i> .
<i>TraceMode</i>	Indicates how trace messages are to be displayed for the page when tracing is enabled. Feasible values are <i>SortByTime</i> and <i>SortByCategory</i> . The default, when tracing is enabled, is <i>SortByTime</i> .
<i>WarningLevel</i>	Indicates the compiler warning level at which you want the compiler to abort compilation for the page. Possible values are 0 through 4.

Notice that the default values of the *Explicit* and *Strict* attributes are read from the application's configuration settings. The configuration settings of an ASP.NET application are obtained by merging all machine-wide settings with application-wide and even folder-wide settings. This means you can also control what the default values for the *Explicit* and *Strict* attributes are. Unless you change the default configuration settings—the configuration files are created when the .NET Framework is installed—both *Explicit* and *Strict* default to *true*. Should the related settings be removed from the configuration files, both attributes would default to *false* instead.

Attributes listed in Table 3-6 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 the *@Page* is not as strict as for other directives. In particular, you can list as a *@Page* attribute, and initialize, any public properties defined on the page class.

TABLE 3-6 *@Page* Attributes for Page Behavior

Attribute	Description
<i>AspCompat</i>	A Boolean attribute that, when set to <i>true</i> , allows the page to be executed on a single-threaded apartment (STA) thread. The setting allows the page to call COM+ 1.0 components and components developed with Microsoft Visual Basic 6.0 that require access to the unmanaged ASP built-in objects. (I'll cover this topic in Chapter 14.)
<i>Async</i>	If set to <i>true</i> , the generated page class derives from <i>IHttpAsyncHandler</i> rather than having <i>IHttpHandler</i> add some built-in asynchronous capabilities to the page. <i>Not available with ASP.NET 1.x.</i>
<i>AsyncTimeout</i>	Defines the timeout in seconds used when processing asynchronous tasks. The default is 45 seconds. <i>Not available with ASP.NET 1.x.</i>
<i>AutoEventWireup</i>	A Boolean attribute that indicates whether page events are automatically enabled. Set to <i>true</i> by default. Pages developed with Visual Studio .NET have this attribute set to <i>false</i> , and page events are individually tied to handlers.
<i>Buffer</i>	A Boolean attribute that determines whether HTTP response buffering is enabled. Set to <i>true</i> by default.
<i>Description</i>	Provides a text description of the page. The ASP.NET page parser ignores the attribute, which subsequently has only a documentation purpose.
<i>EnableEventValidation</i>	A Boolean value that indicates whether the page will emit a hidden field to cache available values for input fields that support event data validation. Set to <i>true</i> by default. <i>Not available with ASP.NET 1.x.</i>

Attribute	Description
<i>EnableSessionState</i>	Defines how the page should treat session data. If set to <i>true</i> , the session state can be read and written. If set to <i>false</i> , session data is not available to the application. Finally, if set to <i>ReadOnly</i> , the session state can be read but not changed.
<i>EnableViewState</i>	A Boolean value that indicates whether the page <i>view state</i> is maintained across page requests. The view state is the page call context—a collection of values that retain the state of the page and are carried back and forth. View state is enabled by default. (I'll cover this topic in Chapter 15.)
<i>EnableTheming</i>	A Boolean value that indicates whether the page will support themes for embedded controls. Set to <i>true</i> by default. <i>Not available in ASP.NET 1.x.</i>
<i>EnableViewStateMac</i>	A Boolean value that indicates ASP.NET should calculate a machine-specific authentication code and append it to the view state of the page (in addition to Base64 encoding). The <i>Mac</i> in the attribute name stands for <i>machine authentication check</i> . When the attribute is <i>true</i> , upon postbacks ASP.NET will check the authentication code of the view state to make sure that it hasn't been tampered with on the client.
<i>ErrorPage</i>	Defines the target URL to which users will be automatically redirected in case of unhandled page exceptions.
<i>MaintainScrollPosition-OnPostBack</i>	Indicates whether to return the user to the same scrollbar position in the client browser after postback. The default is false.
<i>SmartNavigation</i>	A Boolean value that indicates whether the page supports the Microsoft Internet Explorer 5 or later smart navigation feature. Smart navigation allows a page to be refreshed without losing scroll position and element focus.
<i>Theme, StyleSheetTheme</i>	Indicates the name of the theme (or style-sheet theme) selected for the page. <i>Not available with ASP.NET 1.x.</i>
<i>Transaction</i>	Indicates whether the page supports or requires transactions. Feasible values are: <i>Disabled</i> , <i>NotSupported</i> , <i>Supported</i> , <i>Required</i> , and <i>RequiresNew</i> . Transaction support is disabled by default.
<i>ValidateRequest</i>	A Boolean value that indicates whether request validation should occur. If this value is set to <i>true</i> , ASP.NET checks all input data against a hard-coded list of potentially dangerous values. This functionality helps reduce the risk of cross-site scripting attacks for pages. The value is <i>true</i> by default. <i>This feature is not supported in ASP.NET 1.0.</i>
<i>ViewStateEncryption-Mode</i>	Indicates how view state is encrypted, with three possible enumerated values: <i>Auto</i> , <i>Always</i> , or <i>Never</i> . The default is <i>Auto</i> meaning that the viewstate is encrypted only if a control requests that. Note that using encryption over the viewstate adds some overhead to the processing of the page on the server for each request.

Attributes listed in Table 3-7 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 3-7 @Page Directives for Page Output**

Attribute	Description
<i>ClientTarget</i>	Indicates the target browser for which ASP.NET server controls should render content.
<i>CodePage</i>	Indicates the code page value for the response. Set this attribute only if you created the page using a code page other than the default code page of the Web server on which the page will run. In this case, set the attribute to the code page of your development machine. A code page is a character set that includes numbers, punctuation marks, and other glyphs. Code pages differ on a per-language basis.
<i>ContentType</i>	Defines the content type of the response as a standard MIME type. Supports any valid HTTP content type string.
<i>Culture</i>	Indicates the culture setting for the page. Culture information includes the writing and sorting system, calendar, and date and currency formats. The attribute must be set to a non-neutral culture name, which means it must contain both language and country information. For example, <i>en-US</i> is a valid value, unlike <i>en</i> alone, which is considered country-neutral.
<i>LCID</i>	A 32-bit value that defines the locale identifier for the page. By default, ASP.NET uses the locale of the Web server.
<i>ResponseEncoding</i>	Indicates the character encoding of the page. The value is used to set the <i>CharSet</i> attribute on the content type HTTP header. Internally, ASP.NET handles all strings as Unicode.
<i>Title</i>	Indicates the title of the page. Not really useful for regular pages which would likely use the <i>&lt;title&gt;</i> HTML tag, the attribute has been defined to help developers add a title to content pages where access to the <i>&lt;title&gt;</i> attribute may not be possible. (This actually depends on how the master page is structured.)
<i>UICulture</i>	Specifies the default culture name used by the Resource Manager to look up culture-specific resources at run time.

As you can see, many attributes discussed in Table 3-7 are related to page localization. Building multilanguage and international applications is a task that ASP.NET, and the .NET Framework in general, greatly simplify. In Chapter 5, we'll delve into the topic.

## The @Assembly Directive

The *@Assembly* directive links an assembly to the current page so that its classes and interfaces are available for use on the page. When ASP.NET compiles the page, a few assemblies are linked by default. So you should resort to the directive only if you need linkage to a non-default assembly. Table 3-8 lists the .NET assemblies that are automatically provided to the compiler.

**TABLE 3-8 Assemblies Linked by Default**

Assembly File Name	Description
Mscorlib.dll	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. <i>Not included in ASP.NET 1.x.</i>
System.Data.dll	Defines data container and data access classes, including the whole ADO.NET framework.
System.Drawing.dll	Implements the GDI+ features.
System.EnterpriseServices.dll	Provides the classes that allow for serviced components and COM+ interaction.
System.Web.dll	The assembly implements the core ASP.NET services, controls, and classes.
System.Web.Mobile.dll	The assembly implements the core ASP.NET mobile services, controls, and classes. <i>Not included if version 1.0 of the .NET Framework is installed.</i>
System.Web.Services.dll	Contains the core code that makes Web services run.
System.Xml.dll	Implements the .NET Framework XML features.
System.Runtime.Serialization	Defines the API for .NET serialization. This was one of the additional assemblies that was most frequently added by developers in ASP.NET 2.0 applications. <i>Only included in ASP.NET 3.5.</i>
System.ServiceModel	Defines classes and structure for Windows Communication Foundation (WCF) services. <i>Only included in ASP.NET 3.5.</i>
System.ServiceModel.Web	Defines the additional classes required by ASP.NET and AJAX to support WCF services. <i>Only included in ASP.NET 3.5.</i>
System.WorkflowServices	Defines classes for making workflows and WCF services interact. <i>Only included in ASP.NET 3.5.</i>

In addition to these assemblies, the ASP.NET runtime automatically links to the page all the assemblies that reside in the Web application *Bin* subdirectory. Note that you can modify, extend, or restrict the list of default assemblies by editing the global settings set in the global machine-level *web.config* file. In this case, changes apply to all ASP.NET applications run on that Web server. Alternately, you can modify the assembly list on a per-application basis by editing the application's specific *web.config* file. To prevent all assemblies found in the *Bin* directory from being linked to the page, remove the following line from the root configuration file:

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



**Warning** For an ASP.NET application, the whole set of configuration attributes is set at the machine level. Initially, all applications hosted on a given server machine share the same settings. Then individual applications can override some of those settings in their own *web.config* files. Each application can have a *web.config* file in the root virtual folder and other copies of specialized *web.config* files in application-specific subdirectories. Each page is subject to settings as determined by the configuration files found in the path from the machine to the containing folder. In ASP.NET 1.x, the *machine.config* file contains the complete tree of default settings. In ASP.NET 2.0, the configuration data that specifically refers to Web applications has been moved to a *web.config* file installed in the same system folder as *machine.config*. The folder is named CONFIG and located below the installation path of ASP.NET—that is, %WINDOWS%\Microsoft.Net\Framework\[version].

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*.

## 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();
```

Once 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.



**Caution** Notice that *@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 linked, the compiler will generate a type error. When an assembly has not been linked, 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. Bear in mind it only happens by chance and that assemblies and namespaces are radically different entities, each requiring the proper directive.

For example, to be able to connect to a 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.

## The *@Implements* Directive

The *@Implements* 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 and 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 consequences regarding the way in which 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 and 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. Notice that 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** Starting with ASP.NET 2.0, you are better off using cross-page posting to enable communication between pages.

## 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, we 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*, *Page* 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 also a very important feature for iterative data-bound controls, such as *DataGrid*, for user controls, and controls that fire server-side events.)

The *Page* class also implements the methods of the *IHttpHandler* interface, thus qualifying 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* object 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 3-9 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 3-9 ASP.NET Intrinsic Objects in the *Page* Class**

Property	Description
<i>Application</i>	Instance of the <i>HttpApplicationState</i> class; represents the state of the application. It is functionally equivalent to the ASP intrinsic <i>Application</i> object.
<i>Cache</i>	Instance of the <i>Cache</i> class; implements the cache for an ASP.NET application. More efficient and powerful than <i>Application</i> , it supports item priority and expiration.
<i>Profile</i>	Instance of the <i>ProfileCommon</i> class; represents the user-specific set of data associated with the request.
<i>Request</i>	Instance of the <i>HttpRequest</i> class; represents the current HTTP request. It is functionally equivalent to the ASP intrinsic <i>Request</i> object.
<i>Response</i>	Instance of the <i>HttpResponse</i> class; sends HTTP response data to the client. It is functionally equivalent to the ASP intrinsic <i>Response</i> object.
<i>Server</i>	Instance of the <i>HttpServerUtility</i> class; provides helper methods for processing Web requests. It is functionally equivalent to the ASP intrinsic <i>Server</i> object.
<i>Session</i>	Instance of the <i>HttpSessionState</i> class; manages user-specific data. It is functionally equivalent to the ASP intrinsic <i>Session</i> object.
<i>Trace</i>	Instance of the <i>TraceContext</i> class; performs tracing on the page.
<i>User</i>	An <i>IPrincipal</i> object that represents the user making the request.

We'll cover *Request*, *Response*, and *Server* in Chapter 14; *Application* and *Session* in Chapter 15; *Cache* will be the subject of Chapter 16. Finally, *User* and security will be the subject of Chapter 17.

## Worker Properties

Table 3-10 details page properties that are both informative and provide the grounds for functional capabilities. You can hardly write code in the page without most of these properties.

**TABLE 3-10 Worker Properties of the *Page* Class**

Property	Description
<i>ClientScript</i>	Gets a <i>ClientScriptManager</i> object that contains the client script used on the page. <i>Not available with ASP.NET 1.x.</i>
<i>Controls</i>	Returns the collection of all the child controls contained in the current page.
<i>ErrorPage</i>	Gets or sets the error page to which the requesting browser is redirected in case of an unhandled page exception.
<i>Form</i>	Returns the current <i>HtmlForm</i> object for the page. <i>Not available with ASP.NET 1.x.</i>

Property	Description
<i>Header</i>	Returns a reference to the object that represents the page's header. The object implements <i>IPageHeader</i> . <i>Not available with ASP.NET 1.x.</i>
<i>IsAsync</i>	Indicates whether the page is being invoked through an asynchronous handler. <i>Not available with ASP.NET 1.x.</i>
<i>IsCallback</i>	Indicates whether the page is being loaded in response to a client script callback. <i>Not available with ASP.NET 1.x.</i>
<i>IsCrossPagePostBack</i>	Indicates whether the page is being loaded in response to a postback made from within another page. <i>Not available with ASP.NET 1.x.</i>
<i>IsPostBack</i>	Indicates whether the page is being loaded in response to a client postback or whether it is being loaded for the first time.
<i>IsValid</i>	Indicates whether page validation succeeded.
<i>Master</i>	Instance of the <i>MasterPage</i> class; represents the master page that determines the appearance of the current page. <i>Not available with ASP.NET 1.x.</i>
<i>MasterPageFile</i>	Gets and sets the master file for the current page. <i>Not available with ASP.NET 1.x.</i>
<i>NamingContainer</i>	Returns <i>null</i> .
<i>Page</i>	Returns the current <i>Page</i> object.
<i>PageAdapter</i>	Returns the adapter object for the current <i>Page</i> object.
<i>Parent</i>	Returns <i>null</i> .
<i>PreviousPage</i>	Returns the reference to the caller page in case of a cross-page postback. <i>Not available with ASP.NET 1.x.</i>
<i>TemplateSourceDirectory</i>	Gets the virtual directory of the page.
<i>Validators</i>	Returns the collection of all validation controls contained in the page.
<i>ViewStateUserKey</i>	String property that represents a user-specific identifier used to hash the view-state contents. This trick is a line of defense against one-click attacks. <i>Not available with ASP.NET 1.0.</i>

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 that has been added with version 1.1 of the .NET Framework deserves a special mention. A common use for the user key is to stuff user-specific information that would then be used to hash the contents of the view state along with other information. (See Chapter 15.) 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 3-11 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 3-11 Page-Specific Properties of the *Page* Class**

Property	Description
<i>ClientID</i>	Always returns the empty string.
<i>ClientQueryString</i>	Gets the query string portion of the requested URL. <i>Not available with ASP.NET 1.x.</i>
<i>ClientTarget</i>	Set to the empty string by default; allows you to specify the type of the browser the HTML should comply with. Setting this property disables automatic detection of browser capabilities.
<i>EnableViewState</i>	Indicates whether the page has to manage view-state data. You can also enable or disable the view-state feature through the <i>EnableViewState</i> attribute of the <i>@Page</i> directive.
<i>EnableViewStateMac</i>	Indicates whether ASP.NET should calculate a machine-specific authentication code and append it to the page view state.
<i>EnableTheming</i>	Indicates whether the page supports themes. <i>Not available with ASP.NET 1.x.</i>
<i>ID</i>	Always returns the empty string.
<i>MaintainScrollPositionOnPostBack</i>	Indicates whether to return the user to the same position in the client browser after postback. <i>Not available with ASP.NET 1.x.</i>
<i>SmartNavigation</i>	Indicates whether smart navigation is enabled. Smart navigation exploits a bunch of browser-specific capabilities to enhance the user's experience with the page.
<i>StyleSheetTheme</i>	Gets or sets the name of the style sheet applied to this page. <i>Not available with ASP.NET 1.x.</i>

Property	Description
<i>Theme</i>	Gets and sets the theme for the page. Note that themes can be programmatically set only in the <i>PreInit</i> event. <i>Not available with ASP.NET 1.x.</i>
<i>Title</i>	Gets or sets the title for the page. <i>Not available with ASP.NET 1.x.</i>
<i>TraceEnabled</i>	Toggles page tracing on and off. <i>Not available with ASP.NET 1.x.</i>
<i>TraceModeValue</i>	Gets or sets the trace mode. <i>Not available with ASP.NET 1.x.</i>
<i>UniqueID</i>	Always returns the empty string.
<i>ViewStateEncryptionMode</i>	Indicates if and how the view state should be encrypted.
<i>Visible</i>	Indicates whether ASP.NET has to render the page. If you set <i>Visible</i> to <i>false</i> , ASP.NET doesn't generate any HTML code for the page. When <i>Visible</i> is <i>false</i> , only the text explicitly written using <i>Response.Write</i> hits the client.

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

## Methods of the *Page* Class

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

## Rendering Methods

Table 3-12 details the methods that are directly or indirectly involved with the generation of the markup code.

**TABLE 3-12 Methods for Markup Generation**

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

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

Not directly exposed by the `Page` class, but strictly related to it, is the `GetWebResourceUrl` method on the `ClientScriptManager` class in ASP.NET 2.0 and higher. The method provides a long-awaited feature to control developers. When you develop a control, you often need to embed static resources such as images or client script files. You can make these files be separate downloads but, even though it's effective, the solution looks poor and inelegant. Visual Studio .NET 2003 and newer versions allow 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 should the assembly be modified.

Controls-Related Methods

Table 3-13 details a bunch of helper methods on the `Page` class that are architected to let you manage and validate child controls and resolve URLs.

TABLE 3-13 Helper Methods of the `Page` Object

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

Method	Description
<i>GetTypeHashCode</i>	Retrieves the hash code generated by <i>ASP.xxx.aspx</i> page objects at run time. In the base <i>Page</i> class, the method implementation simply returns 0; significant numbers are returned by classes used for actual pages.
<i>GetValidators</i>	Returns a collection of control validators for a specified validation group. <i>Not available with ASP.NET 1.x.</i>
<i>HasControls</i>	Determines whether the page contains any child controls.
<i>LoadControl</i>	Compiles and loads a user control from an .ascx file, and returns a <i>Control</i> object. If the user control supports caching, the object returned is <i>PartialCachingControl</i> .
<i>LoadTemplate</i>	Compiles and loads a user control from an .ascx file, and returns it wrapped in an instance of an internal class that implements the <i>ITemplate</i> interface. The internal class is named <i>SimpleTemplate</i> .
<i>MapPath</i>	Retrieves the physical, fully qualified path that an absolute or relative virtual path maps to.
<i>ParseControl</i>	Parses a well-formed input string, and returns an instance of the control that corresponds to the specified markup text. If the string contains more controls, only the first is taken into account. The <i>runat</i> attribute can be omitted. The method returns an object of type <i>Control</i> and must be cast to a more specific type.
<i>RegisterRequiresControlState</i>	Registers a control as one that requires control state. <i>Not available with ASP.NET 1.x.</i>
<i>RegisterRequiresPostBack</i>	Registers the specified control to receive a postback handling notice, even if its ID doesn't match any ID in the collection of posted data. The control must implement the <i>IPostBackDataHandler</i> interface.
<i>RegisterRequiresRaiseEvent</i>	Registers the specified control to handle an incoming postback event. The control must implement the <i>IPostBackEventHandler</i> interface.
<i>RegisterViewStateHandler</i>	Mostly for internal use, the method sets an internal flag causing the page view state to be persisted. If this method is not called in the prerendering phase, no view state will ever be written. Typically, only the <i>HtmlForm</i> server control for the page calls this method. There's no need to call it from within user applications.
<i>ResolveUrl</i>	Resolves a relative URL into an absolute URL based on the value of the <i>TemplateSourceDirectory</i> property.
<i>Validate</i>	Instructs any validation controls included on the page to validate their assigned information. ASP.NET 2.0 supports 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 *ParseControl* 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 3-14 enumerates all the methods in the *Page* class that have to do with HTML and script code to be inserted in the client page.

TABLE 3-14 Script-Related Methods

Method	Description
<i>GetCallbackEventReference</i>	Obtains a reference to a client-side function that, when invoked, initiates a client call back to server-side events. <i>Not available with ASP.NET 1.x.</i>
<i>GetPostBackClientEvent</i>	Calls into <i>GetCallbackEventReference</i> .
<i>GetPostBackClientHyperlink</i>	Appends <i>javascript:</i> to the beginning of the return string received from <i>GetPostBackEventReference</i> .  <i>javascript:__doPostBack('CtlID', '')</i>
<i>GetPostBackEventReference</i>	Returns the prototype of the client-side script function that causes, when invoked, a postback. It takes a <i>Control</i> and an argument, and it returns a string like this:  <i>__doPostBack('CtlID', '')</i>
<i>IsClientScriptBlockRegistered</i>	Determines whether the specified client script is registered with the page. <i>Marked as obsolete.</i>
<i>IsStartupScriptRegistered</i>	Determines whether the specified client startup script is registered with the page. <i>Marked as obsolete.</i>

Method	Description
<i>RegisterArrayDeclaration</i>	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 <i>theArray</i> and <i>"a", "b"</i> , you get the following JavaScript code:  <pre>var theArray = new Array('a', 'b');</pre> <i>Marked as obsolete.</i>
<i>RegisterClientScriptBlock</i>	An ASP.NET page uses this method to emit client-side script blocks in the client page just after the opening tag of the HTML <i>&lt;form&gt;</i> element. <i>Marked as obsolete.</i>
<i>RegisterHiddenField</i>	Use this method to automatically register a hidden field on the page. <i>Marked as obsolete.</i>
<i>RegisterOnSubmitStatement</i>	Use this method to emit client script code that handles the client <i>OnSubmit</i> event. The script should be a JavaScript function call to client code registered elsewhere. <i>Marked as obsolete.</i>
<i>RegisterStartupScript</i>	An ASP.NET page uses this method to emit client-side script blocks in the client page just before closing the HTML <i>&lt;form&gt;</i> element. <i>Marked as obsolete.</i>
<i>SetFocus</i>	Sets the browser focus to the specified control. <i>Not available with ASP.NET 1.x.</i>

As you can see, some methods in Table 3-14, which are defined and usable in ASP.NET 1.x, are marked obsolete. In ASP.NET 3.5 applications, you should avoid calling them and resort to methods with the same name exposed out of the *ClientScript* property. (See Table 3-10.)

```
// Avoid this in ASP.NET 3.5
Page.RegisterArrayDeclaration(...);
// Use this in ASP.NET 3.5
Page.ClientScript.RegisterArrayDeclaration(...);
```

We'll return to *ClientScript* in Chapter 5.

Methods listed in Table 3-14 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 3-15 shows, some events are orthogonal to the typical life cycle of a page (initialization, postback, 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 on the page life cycle.

**TABLE 3-15 Events That a Page Can Fire**

Event	Description
<i>AbortTransaction</i>	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction aborts.
<i>CommitTransaction</i>	Occurs for ASP.NET pages marked to participate in an automatic transaction when a transaction commits.
<i>DataBinding</i>	Occurs when the <i>DataBind</i> method is called on the page to bind all the child controls to their respective data sources.
<i>Disposed</i>	Occurs when the page is released from memory, which is the last stage of the page life cycle.
<i>Error</i>	Occurs when an unhandled exception is thrown.
<i>Init</i>	Occurs when the page is initialized, which is the first step in the page life cycle.
<i>InitComplete</i>	Occurs when all child controls and the page have been initialized. <i>Not available in ASP.NET 1.x.</i>
<i>Load</i>	Occurs when the page loads up, after being initialized.
<i>LoadComplete</i>	Occurs when the loading of the page is completed and server events have been raised. <i>Not available in ASP.NET 1.x.</i>
<i>PreInit</i>	Occurs just before the initialization phase of the page begins. <i>Not available in ASP.NET 1.x.</i>
<i>PreLoad</i>	Occurs just before the loading phase of the page begins. <i>Not available in ASP.NET 1.x.</i>
<i>PreRender</i>	Occurs when the page is about to render.
<i>PreRenderComplete</i>	Occurs just before the pre-rendering phase begins. <i>Not available in ASP.NET 1.x.</i>
<i>SaveStateComplete</i>	Occurs when the view state of the page has been saved to the persistence medium. <i>Not available in ASP.NET 1.x.</i>
<i>Unload</i>	Occurs when the page is unloaded from memory but not yet disposed.

## 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 of 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 Linux-based systems (for example, LAMP), 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. Briefly, 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 once the page posts back.

## The Single Form Model

Admittedly, for programmers whose experience is with ASP or JSP, the single form model of ASP.NET can be difficult to make sense of at first. These programmers frequently ask questions on forums and newsgroups such as, “Where’s the *Action* property of the form?” and “Why can’t I redirect to a particular page when a form is submitted?”

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 as 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 5.

## 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 is mostly due to the fact that HTTP handlers, including page classes, work synchronously. To alleviate this issue, ASP.NET supports asynchronous handlers since version 1.0 through the *IHttpAsyncHandler* interface. Starting with ASP.NET 2.0, creating asynchronous pages is even easier thanks to specific support from the framework.

Two aspects characterize an asynchronous ASP.NET page: a new 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 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 *BeginEventHandler* and *EndEventHandler* are delegates defined as follows:

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

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

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



**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* and *End* step. Each request is always served by a pooled thread. Typically, the *Begin* step is served by a thread picked up from the ASP.NET worker thread pool. The *End* step is served by a thread selected from the completion thread pool.

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.

## The Significance of *PreRenderComplete*

So an asynchronous page executes up until the *PreRenderComplete* stage is reached and then blocks while waiting for the asynchronous operation to complete. 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;

    void Page_Load (object sender, EventArgs e)
    {
        AddOnPreRenderCompleteAsync (
            new BeginEventHandler(BeginTask),
            new EndEventHandler(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 pooled thread

        string text;
        using (WebResponse response = req.EndGetResponse(ar))
        {
            StreamReader reader;
            using (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());
    }
}
```



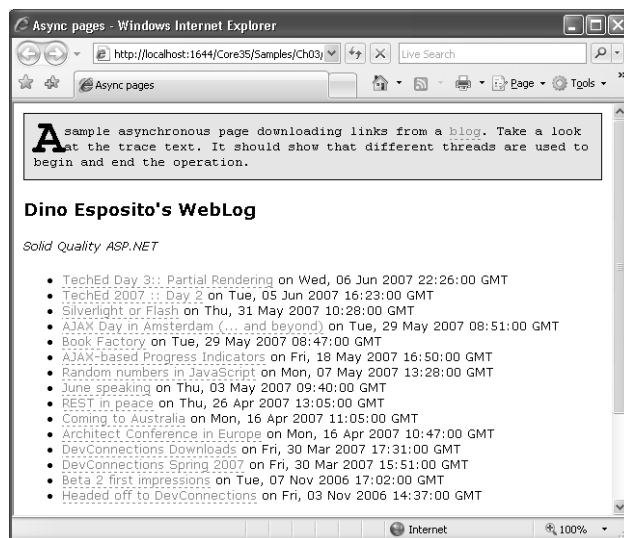
```

        // 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 3-6 shows the sample page in action.



**FIGURE 3-6** A sample asynchronous page downloading links from an RSS feed.

It would also be interesting to take a look at the messages traced by the page. Figure 3-7 provides visual clues of it. The Begin and End stages are served by different threads and take place at different times.

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.000342780995908698	0.000343
aspx.page	Begin Init	0.000513752446190787	0.000171
aspx.page	End Init	0.000705955645200717	0.000192
aspx.page	Begin InitComplete	0.000867149316463405	0.000161
aspx.page	End InitComplete	0.00102750489238157	0.000160
aspx.page	Begin PreLoad	0.00118450808692166	0.000157
aspx.page	End PreLoad	0.00134430493261015	0.000160
aspx.page	Begin Load	0.00150186685737992	0.000158
aspx.page	End Load	0.00167283830766201	0.000171
aspx.page	Begin LoadComplete	0.00183459070915438	0.000162
aspx.page	End LoadComplete	0.00199382882461318	0.000159
aspx.page	Begin PreRender	0.00215111138426811	0.000157
aspx.page	End PreRender	0.00231202569041596	0.000161
	Begin async: Thread=4	0.00248271777558321	0.000171
	End async: Thread=9	1.32697759072731	1.324495
aspx.page	Begin PreRenderComplete	1.32727008600255	0.000292
aspx.page	End PreRenderComplete	1.32745642253415	0.000186
aspx.page	Begin SaveState	1.33882490651745	0.011368
aspx.page	End SaveState	1.33928977006654	0.000465
aspx.page	Begin SaveStateComplete	1.33946437326532	0.000175
aspx.page	End SaveStateComplete	1.33962891931796	0.000165
aspx.page	Begin Render	1.33978731933807	0.000158
aspx.page	End Render	1.34042203687899	0.000635

**FIGURE 3-7** The traced request details clearly show the two steps needed to process a request asynchronously.

Note the time elapsed between the time we enter *BeginTask* and exit *EndTask* stages (indicated by the elapsed time between the “Begin async” and “End async” entries shown in Figure 3-7). 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.

## 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 we 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 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. Note that configuring the asynchronous timeout in *web.config* causes all asynchronous pages to use the same timeout value. Individual pages are still free to set their own timeout value in their *@Page* directive.

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. This 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 significantly easier the implementation of multiple calls to remote sources. 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.



**Note** For more information on asynchronous pages, check out Chapter 5 of my book *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press 2006).

## 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 to 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.

## 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 sub-stages, are simply not marked as public and are out of the developer's control. They 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. It is important to note that when the request processing begins, all child controls and page intrinsic objects such as HTTP context, request objects, and response objects are set.

The very first step in the page lifetime is determining why the runtime 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

Introduced with ASP.NET 2.0, this event is the entry point in the page life cycle. When the event fires, no master page and no theme have 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 property 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 and theme, if each exists, 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 sub-stage. 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. In ASP.NET 1.x, you must wait for the *Load* event to start writing safely to the page or any control view state.

## 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 are 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.

## The *PreLoad* Event

Introduced with ASP.NET 2.0, 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 that has to do with 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 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 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 that allows page authors to 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** Starting with ASP.NET 2.0, a new property, *UseSubmitBehavior*, 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. In ASP.NET 1.x, the *Button* control always outputs an `<input type="submit">` element. In ASP.NET 2.0 and beyond, by setting *UseSubmitBehavior* to *false*, you can 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.

## The *LoadComplete* Event

Introduced in ASP.NET 2.0, the page-only *LoadComplete* event signals the end of the page-preparation phase. It is important to 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 sub-stage 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, in ASP.NET 2.0 a new event has been added and 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. It is important to 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 14.)

Starting with ASP.NET 2.0, controls 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 view state, but the purpose of 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 control state, while user interface property data (such as the control's contents) should be kept in view state.

Introduced with ASP.NET 2.0, 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 15.

## 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 into a buffer. Several overridable methods allow control developers to intervene in various steps during the markup generation—begin tag, body, and end tag. No user event is associated with the rendering phase.

## The *Unload* Event

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

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

## Conclusion

ASP.NET is a complex technology built on top of a substantially simple—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.

There are two relevant aspects in the ASP.NET Web Forms model: the process model, including the Web server 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 sub-stages) 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, we 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 various server controls, which include Web controls, HTML controls, and validation controls.



## Just the Facts

- A pipeline of run-time modules receive from IIS an incoming HTTP packet and make it evolve from a protocol-specific payload up to an instance of a class derived from *Page*.
- The page class required to serve a given request is dynamically compiled on demand when first required in the context of a Web application.
- The page class compiled to an assembly remains in use as long as no changes occur to the linked *.aspx* source file or the whole application is restarted.
- Each page class is an HTTP handler—that is, a component that the run time uses to service requests of a certain type.
- The ASP.NET code-behind model employs partial classes to generate missing declarations for protected members that represent server controls. This code was auto-generated by old versions of Visual Studio and placed in hidden regions.
- ASP.NET pages always post to themselves and use the view state to restore the state of controls existing when the page was last generated on the server.
- The view state creates the illusion of a stateful programming model in a stateless environment.
- Processing the page on the server entails handling a bunch of events that collectively form the page life cycle. A deep understanding of the page life cycle is key to diagnosing possible problems and implementing advanced features quickly and efficiently.



## Chapter 18

# HTTP Handlers and Modules

### In this chapter:

Quick Overview of the IIS Extensibility API .....	868
Writing HTTP Handlers .....	873
Writing HTTP Modules .....	901
Conclusion .....	913

HTTP modules and HTTP handlers are fundamental pieces of the ASP.NET architecture. HTTP handlers and modules are truly the building blocks of the .NET Web platform. Any requests for an ASP.NET managed resource is always resolved by an HTTP handler and passes 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.

An HTTP handler is the component that actually takes care of serving the request. It is an instance of a class that implements the *IHttpHandler* interface. The *ProcessRequest* method of the interface is the central console that governs the processing of the request. For example, the *Page* class—the base class for all ASP.NET run-time pages—implements the *IHttpHandler* interface, and its *ProcessRequest* method is responsible for loading and saving the view state and for firing the well-known set of page events, including *Init*, *Load*, *PreRender*, and the like.

ASP.NET maps each incoming HTTP request to a particular HTTP handler. A special breed of component—named the *HTTP handler factory*—provides the infrastructure for creating the physical instance of the handler to service the request. For example, the *PageHandlerFactory* class parses the source code of the requested *.aspx* resource and returns a compiled instance of the class that represents the page. 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 machine-wide *web.config* file or a local *web.config* file, depending on the scope you desire.

HTTP modules are classes that implement the *IHttpModule* interface and 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.

HTTP handlers and HTTP modules have the same functionality as ISAPI extensions and ISAPI filters, respectively, but with a much simpler programming model. ASP.NET allows you to create custom handlers and custom modules. Before we get into this rather advanced aspect of Web programming, a review of the Internet Information Services (IIS) extensibility model is in order because this model determines what modules and handlers can do and what they cannot do.



**Note** ISAPI stands for Internet Server Application Programming Interface and represents the protocol by means of which IIS talks to external components. The ISAPI model is based on a Microsoft Win32 unmanaged dynamic-link library (DLL) that exports a couple of functions. This model is significantly expanded in IIS 7.0 and largely matches the ASP.NET extensibility model, which is based on HTTP handlers and modules. I'll return to this topic shortly.

## Quick Overview of the IIS Extensibility API

A Web server is primarily a server application that can be contacted using a bunch of Internet protocols, such as HTTP, File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP), and the Simple Mail Transfer Protocol (SMTP). IIS—the Web server included with the Microsoft Windows operating system—is no exception.

A Web server generally also provides a documented 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 in modern Web applications because they require a new process for each HTTP request. As you can easily understand, this approach is rather inadequate for high-volume Web sites and poses severe scalability issues. IIS supports CGI applications, but you will seldom use this feature unless you have serious backward-compatibility issues. More recent versions of Web servers supply an alternate and more efficient model to extend the capabilities of the module. In IIS, this alternative model takes the form of the ISAPI interface.

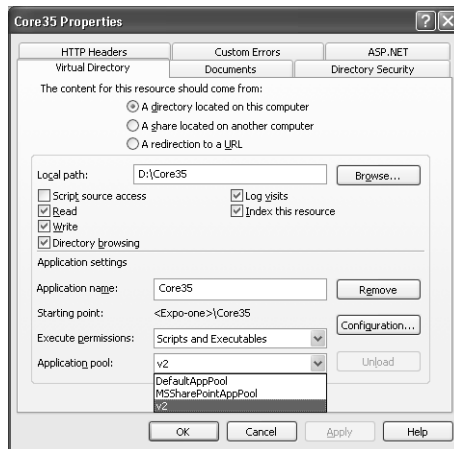
### The ISAPI Model

When the ISAPI model is used, instead of starting a new process for each request, IIS loads an ISAPI component—namely, a Win32 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. Starting with IIS 4.0, though, some countermeasures have been taken to address this problem. Before the advent of IIS 6.0, you were allowed to set the protection level of a newly installed application choosing from three options: low, medium, and high.

If you choose a low protection, the application (and its extensions) will be run within the Web server process (*inetinfo.exe*). If you choose medium protection, applications will be pooled together and hosted by an instance of a different worker process (*dllhost.exe*). If you choose high protection, each application set to High will be hosted in its own individual worker process (*dllhost.exe*).

Web applications running under IIS 6.0 are grouped in pools, and the choice you can make is whether you want to join an existing pool or create a new one. Figure 18-1 shows the dialog box picking the application pool of choice in IIS 6.0 and Microsoft Windows Server 2003.



**FIGURE 18-1** Configuring the protection level of Web applications in IIS 6.0 under Windows Server 2003.

All applications in a pool share the same run-time settings and the same worker process—*w3wp.exe*.

## Illustrious Children of the ISAPI Model

The ISAPI model has another key drawback—the programming model. An ISAPI component represents a compiled piece of code—a Win32 DLL—that retrieves data and writes HTML code to an output console. It has to be developed using C or C++, it should generate multi-threaded code, and it must be written with extreme care because of the impact that bugs or runtime failures can have on the application.



A while back, Microsoft attempted to encapsulate the ISAPI logic in the Microsoft Foundation Classes (MFC), but even though the effort was creditable, it didn't pay off very well. MFC tended to bring more code to the table than high-performance Web sites would perhaps like, and worse, the resulting ISAPI extension DLL suffered from a well-documented memory leak.

Active Server Pages (ASP), the predecessor of ASP.NET, is, on the other hand, an example of a well-done ISAPI application. ASP is implemented as an ISAPI DLL (named *asp.dll*) registered to handle HTTP requests with an *.asp* extension. The internal code of the ASP ISAPI extension DLL parses the code of the requested resource, executes any embedded script code, and builds the page for the browser.

As of IIS 6.0, any functionality built on top of IIS must be coded according to the guidelines set by the ISAPI model. ASP and ASP.NET are no exceptions. Today, the whole ASP.NET platform works closely with IIS, but it is not part of it. The *aspnet\_isapi.dll* core component is the link between IIS and the ASP.NET runtime environment. When a request for *.aspx* resources comes in, IIS passes the control to *aspnet\_isapi.dll*, which in turn hands the request to the ASP.NET pipeline inside an instance of the common language runtime (CLR).

As of this writing, to extend IIS you can write a Win32 DLL only with a well-known set of entry points. This requirement ceases to exist with IIS 7.0, which is scheduled to ship with Windows 2008 Server.



**Note** A good place to learn about IIS 7.0 and find good scripts and code snippets is <http://www.iis.net>. IIS 7.0 is also part of Windows Vista, but that is not particularly relevant here in the context of an ASP.NET book. Although you can certainly develop part of your Web site on a Windows Vista machine, it is simply out of question that you use Windows Vista as a Web server to host a site. Although fully functional, the IIS 7.0 that has shipped with Windows Vista can be seen as a live tool to experiment and test. The “real” IIS 7.0 for Web developers and administrators will ship in 2008 with Windows 2008 Server.

## Structure of ISAPI Components

An ISAPI extension is invoked through a URL that ends with the name of the DLL that implements the function, as shown in the following URL:

```
http://www.contoso.com/apps/he1lo.dll
```

The DLL must export a couple of functions—*GetExtensionVersion* and *HttpExtensionProc*. The *GetExtensionVersion* function sets the version and the name of the ISAPI server extension. When the extension is loaded, the *GetExtensionVersion* function is the first function to be called. *GetExtensionVersion* is invoked only once and can be used to initialize any needed variables. The function is expected to return *true* if everything goes fine. In the case of errors, the function should return *false* and the Web server will abort loading the DLL and put a message in the system log.

The core of the ISAPI component is represented by the *HttpExtensionProc* function. The function receives basic HTTP information regarding the request (for example, the query string and the headers), performs the expected action, and prepares the response to send back to the browser.



**Note** Certain handy programming facilities, such as the session state, are abstractions the ISAPI programming model lacks entirely. The ISAPI model is a lower level programming model than, say, ASP or ASP.NET.

The ISAPI programming model is made of two types of components—ISAPI extensions and ISAPI filters.

## ISAPI Extensions

ISAPI extensions are the IIS in-process counterpart of CGI applications. As mentioned, an ISAPI extension is a DLL that is loaded in the memory space occupied by IIS or another host application. Because it is a DLL, only one instance of the ISAPI extension needs to be loaded at a time. On the downside, the ISAPI extension must be thread-safe so that multiple client requests can be served simultaneously. ISAPI extensions work in much the same way as an ASP or ASP.NET page. It takes any information about the HTTP request and prepares a valid HTTP response.

Because the ISAPI extension is made of compiled code, it must be recompiled and reloaded at any change. If the DLL is loaded in the Web server's memory, the Web server must be stopped. If the DLL is loaded in the context of a separate process, only that process must be stopped. Of course, when an external process is used, the extension doesn't work as fast as it could when hosted in-process, but at least it doesn't jeopardize the stability of IIS.

## ISAPI Filters

ISAPI filters are components that intercept specific server events before the server itself handles them. Upon loading, the filter indicates what event notifications it will handle. If any of these events occur, the filter can process them or pass them on to other filters.

You can use ISAPI filters to provide custom authentication techniques or to automatically redirect requests based on HTTP headers sent by the client. Filters are a delicate gear in the IIS machinery. They can facilitate applications and let them take control of customizable aspects of the engine. For this same reason, though, ISAPI filters can also degrade performance if not written carefully. Filters, in fact, can run only in-process. Filters can be loaded for the Web server as a whole or for specific Web sites.

ISAPI filters can accomplish tasks such as implementing custom authentication schemes, compression, encryption, logging, and request analysis. The ability to examine, and if necessary modify, both incoming and outgoing streams of data makes ISAPI filters very

powerful and flexible. This last sentence shows the strength of ISAPI filters but also indicates their potential weakness, which is that they will hinder performance if not written well.

## Changes in IIS 7.0

ASP.NET 1.0 was originally a self-contained, brand new runtime environment bolted onto IIS 5.0. With the simultaneous release of ASP.NET 1.1 and IIS 6.0, the Web development and server platforms have gotten closer and started sharing some services, such as process recycling and output caching. The advent of ASP.NET 2.0 and newer versions hasn't changed anything, but the release of IIS 7.0 will.

### A Unified Runtime Environment

In a certain way, IIS 7.0 represents the unification of the ASP.NET and IIS platforms. HTTP handlers and modules, the runtime pipeline, and configuration files will become constituent elements of a common environment. The whole IIS internal pipeline has been componentized to originate a distinct and individually configurable component. A new section will be added to the *web.config* schema of ASP.NET applications to configure the IIS environment.

Put another way, it will be as if the ASP.NET runtime expanded to incorporate and replace the surrounding Web server environment. It's hard to say whether things really went this way or whether it was the other way around. As a matter of fact, the same concepts and instruments you know from ASP.NET are available in IIS 7.0 at the Web server level.

To illustrate, on Windows 2008 Server (and for testing purposes, also on a Windows Vista machine) you can use Forms authentication to protect access to any resources available on the server and not just ASP.NET-specific resources. You might already know that static resources such as HTML pages and JPG images are not served by ASP.NET by default; as such, they're not subject to the authentication rules you set for the application. Where IIS 7.0 is supported, you can now define a handler for some specific and static resources and be sure that IIS will use your code to serve those resources.

### Managed ISAPI Extensions and Filters

Today if you want to take control of an incoming request in any version of IIS older than version 7.0, you have no choice other than writing a C or C++ DLL, using either MFC or perhaps the ActiveX Template Library (ATL). More comfortable HTTP handlers and modules are an ASP.NET-only feature, and they can be applied only to ASP.NET-specific resources and only after the request has been authenticated by IIS and handed over to ASP.NET.

In IIS 7.0, you can write HTTP handlers and modules to filter *any* requests and implement any additional features using .NET code for whatever resources the Web server can serve. More precisely, you'll continue writing HTTP handlers and modules as you do today for ASP.NET,

except that you will be given the opportunity to register them for any file type. Needless to say, old-style ISAPI extensions will still be supported, but unmanaged extensions and filters will likely become a thing of the past. I'll demonstrate IIS 7.0 handlers later in the chapter.

## Writing HTTP Handlers

ASP.NET comes with a small set of built-in HTTP handlers. There is a handler to serve ASP.NET pages, one for Web services, and yet another to accommodate .NET Remoting requests for remote objects hosted by IIS. Other helper handlers are defined to view the tracing of individual pages in a Web application (*trace.axd*) and to block requests for prohibited resources such as *.config* or *.asax* files. Starting with ASP.NET 2.0, you also find a handler (*webresource.axd*) to inject assembly resources and script code into pages. In ASP.NET 3.5, the *scriptresource.axd* handler has been added as a more refined tool to inject script code and AJAX capabilities into Web pages.

You can write custom HTTP handlers whenever you need ASP.NET to process certain 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.



**Note** 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 are asynchronous pages. Later in this chapter, though, we'll take a look at the mechanics of asynchronous handlers, of which asynchronous pages are a special case.

Conventional ISAPI extensions and filters should be registered within the IIS metabase. In contrast, HTTP handlers are registered in the *web.config* file if you want the handler to participate in the HTTP pipeline processing of the Web request. In a manner similar to ISAPI extensions, you can also invoke the handler directly via the URL.

## 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. More specifically, 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 18-1. *ProcessRequest* is a method, whereas *IsReusable* is a Boolean property.

TABLE 18-1 Members of the *IHttpHandler* Interface

Member	Description
<i>IsReusable</i>	This property gets a Boolean value indicating whether the HTTP runtime can reuse the current instance of the HTTP handler while serving another request.
<i>ProcessRequest</i>	This method processes the HTTP request.

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

Again, an HTTP handler is simply a class that implements the *IHttpHandler* interface. The output for the request is built within the *ProcessRequest* method, as shown in the following code:

```
using System.Web;

namespace Core35.Components
{
    public class SimpleHandler : IHttpHandler
    {
        // Override the ProcessRequest method
        public void ProcessRequest(HttpContext context)
        {
            context.Response.Write("<H1>Hello, I'm an HTTP handler</H1>");
        }
    }
}
```

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

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.aspx"
          type="Core35.Components.SimpleHandler" />
    </httpHandlers>
  </system.web>
</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.aspx*. Note that the URL *hello.aspx* 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 contains the handler. It's canonical format is *type[,assembly]*. You omit the assembly information if the component is defined in the *App\_Code* or other reserved folders.



**Note** If you enter the settings shown earlier in the global *web.config* file, you will register the *SimpleHandler* component as callable from within all Web applications hosted by the server machine.

If you invoke the *hello.aspx* URL, you obtain the results shown in Figure 18-2.



**FIGURE 18-2** A sample HTTP handler that answers requests for *hello.aspx*.

The technique discussed here is the quickest and simplest way of putting an HTTP handler to work, but there is more to know about registration of HTTP handlers and there are many more options to take advantage of. Now let's consider a more complex example of an HTTP handler.

## An HTTP Handler for Quick Data Reports

With their relatively simple programming model, HTTP handlers give you a means of interacting with the low-level request and response services of IIS. In the previous example, we returned only constant text and made no use of the request information. In the next example, we'll configure the handler to intercept and process only requests of a particular type and generate the output based on the contents of the requested resource.

The idea is to build an HTTP handler for custom *.sqlx* resources. A SQLX file is an XML document that expresses the statements for one or more SQL queries. The handler grabs the information about the query, executes it, and finally returns the result set formatted as a grid. Figure 18-3 shows the expected outcome.

firstname	lastname	city	country
Nancy	Davolio	Seattle	USA
Andrew	Fuller	Tacoma	USA
Janet	Leverling	Kirkland	USA
Margaret	Peacock	Redmond	USA
Steven	Buchanan	London	UK
Michael	Suyama	London	UK
Robert	King	London	UK
Laura	Callahan	Seattle	USA
Anne	Dodsworth	London	UK
Jim	Foo		

companyname	contactname	city	country
Franchi S.p.A.	Paolo Accorti	Torino	Italy
Magazzini Alimentari Riuniti	Giovanni Rovelli	Bergamo	Italy
Reggiani Caseifici	Maurizio Moroni	Reggio Emilia	Italy

**FIGURE 18-3** A custom HTTP handler in action.

To start, let's examine the source code for the *IHttpHandler* class.



**Warning** Take this example for what it really is—merely a way to process a custom XML file with a custom extension doing something more significant than outputting a “hello world” message. *Do not* take this handler as a realistic prototype for exposing your Microsoft SQL Server databases over the Web.

## Building a Query Manager Tool

The HTTP handler should get into the game whenever the user requests an *.sqlx* resource. Assume for now that the system knows how to deal with such a weird extension, and focus on what's needed to execute the query and pack the results into a grid. To execute the query, at a minimum, we need the connection string and the command text. The following text illustrates the typical contents of an *.sqlx* file:

```
<queries>
  <query connString="DATABASE=northwind;SERVER=localhost;UID=...">
    SELECT firstname, lastname, country FROM employees
  </query>
  <query connString="DATABASE=northwind;SERVER=localhost;UID=...">
    SELECT companyname FROM customers WHERE country='Italy'
  </query>
</queries>
```

The XML document is formed by a collection of *<query>* nodes, each containing an attribute for the connection string and the text of the query.

The *ProcessRequest* method extracts this information before it can proceed with executing the query and generating the output:

```
class SqlxDData
{
    public string ConnectionString;
    public string QueryText;
}

public class QueryHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        // Parses the SQLX file
        SqlxDData[] data = ParseFile(context);

        // Create the output as HTML
        StringCollection htmlColl = CreateOutput(data);

        // Output the data
        context.Response.Write("<html><head><title>");
        context.Response.Write("QueryHandler Output");
        context.Response.Write("</title></head><body>");
        foreach (string html in htmlColl)
        {
            context.Response.Write(html);
            context.Response.Write("<hr />");
        }
        context.Response.Write("</body></html>");
    }
}
```



```

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

        ...
    }

```

The *ParseFile* helper function parses the source code of the *.sqlx* file and creates an instance of the *SqlxData* class for each query found:

```

private SqlxData[] ParseFile(HttpContext context)
{
    XmlDocument doc = new XmlDocument();
    string filePath = context.Request.Path;
    using (Stream fileStream = VirtualPathProvider.OpenFile(filePath)) {
        doc.Load(fileStream);
    }

    // Visit the <mapping> nodes
    XmlNodeList mappings = doc.SelectNodes("queries/query");
    SqlxData[] descriptors = new SqlxData[mappings.Count];
    for (int i=0; i < descriptors.Length; i++)
    {
        XmlNode mapping = mappings[i];
        SqlxData query = new SqlxData();
        descriptors[i] = query;

        try {
            query.ConnectionString =
                mapping.Attributes["connString"].Value;
            query.QueryText = mapping.InnerText;
        }
        catch {
            context.Response.Write("Error parsing the input file.");
            descriptors = new SqlxData[0];
            break;
        }
    }
    return descriptors;
}

```

The *SqlxData* internal class groups the connection string and the command text. The information is passed to the *CreateOutput* function, which will actually execute the query and generate the grid:

```

private StringCollection CreateOutput(SqlxData[] descriptors)
{
    StringCollection coll = new StringCollection();

    foreach (SqlxData data in descriptors)
    {

```

```

        // Run the query
        DataTable dt = new DataTable();
        SqlDataAdapter adapter = new SqlDataAdapter(data.QueryText,
            data.ConnectionString);
        adapter.Fill(dt);

        // Error handling
        ...

        // Prepare the grid
        DataGrid grid = new DataGrid();
        grid.DataSource = dt;
        grid.DataBind();

        // Get the HTML
        string html = Utils.RenderControlAsString(grid);
        coll.Add(html);
    }
    return coll;
}

```

After executing the query, the method populates a dynamically created *DataGrid* control. In ASP.NET pages, the *DataGrid* control, like any other control, is rendered to HTML. However, this happens through the care of the special HTTP handler that manages *.aspx* resources. For *.sqlx* resources, we need to provide that functionality ourselves. Obtaining the HTML for a Web control is as easy as calling the *RenderControl* method on an HTML text writer object. This is just what the helper method *RenderControlAsString* does:

```

static class Utils
{
    public static string RenderControlAsString(Control ctl)
    {
        StringWriter sw = new StringWriter();
        HtmlTextWriter writer = new HtmlTextWriter(sw);
        ctl.RenderControl(writer);
        return sw.ToString();
    }
}

```



**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.

## 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 register the handler in the configuration file:

```
<system.web>
  <httpHandlers>
    <add verb="*"
        path="*.sqlx"
        type= "Core35.Components.QueryHandler,Core35Lib" />
  </httpHandlers>
</system.web>
```

You add the new handler to the `<httpHandlers>` section of the local or global *web.config* file. The section supports three actions: `<add>`, `<remove>`, and `<clear>`. You use `<add>` to add a new HTTP handler to the scope of the *.config* file. You use `<remove>` to remove a particular handler. Finally, you use `<clear>` to get rid of all the registered handlers. To add a new handler, you need to set three attributes—*verb*, *path*, and *type*—as shown in Table 18-2.

**TABLE 18-2 Attributes Needed to Register an HTTP Handler**

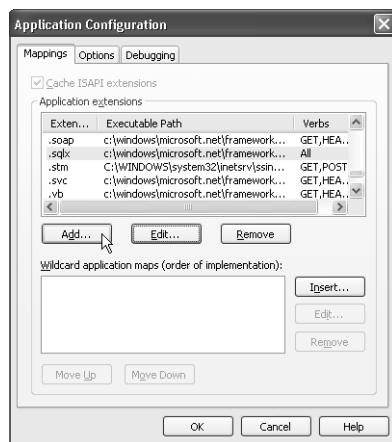
Attribute	Description
<i>Verb</i>	Indicates the list of the supported HTTP verbs—for example, <i>GET</i> , <i>PUT</i> , and <i>POST</i> . The wildcard character (*) is an acceptable value and denotes all verbs.
<i>Path</i>	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, <i>*.aspx</i> .
<i>Type</i>	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application’s private <i>Bin</i> directory and then in the system global assembly cache.

These attributes are mandatory. An optional attribute is also supported—*validate*. 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 any error or problem with it.

So far, you have correctly deployed and registered the HTTP handler, but if you try invoking an *.sqlx* resource, the results you produce are not what you’d expect. The problem lies in the fact that so far you configured ASP.NET to handle only *.sqlx* resources, but IIS still doesn’t know anything about them!

A request for an *.sqlx* resource is handled by IIS *before* it is handed to the ASP.NET ISAPI extension. If you don’t register *some* ISAPI extension to handle *..sqlx* resource requests, IIS will treat each request as a request for a static resource and serve the request by sending

back the source code of the .sqlx file. The extra step required is registering the .sqlx extension with the IIS 6.0 metabase such that requests for .sqlx resources are handed off to ASP.NET, as shown in Figure 18-4.



**FIGURE 18-4** Registering the .sqlx extension with the IIS 6.0 metabase.

The dialog box in the figure is obtained by clicking on the properties of the application in the IIS 6.0 manager and then the configuration of the site. To involve the HTTP handler, you must choose *aspnet\_isapi.dll* as the ISAPI extension. In this way, all .sqlx requests are handed out to ASP.NET and processed through the specified handler. Make sure you select *aspnet\_isapi.dll* from the folder of the ASP.NET version you plan to use.



**Caution** In Microsoft Visual Studio, if you test a sample .sqlx resource using the local embedded Web server, nothing happens that forces you to register the .sqlx 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.

## Registering the Handler with IIS 7.0

If you run IIS 7.0, you don't strictly need to change anything through the IIS Manager. You can add a new section to the *web.config* file and specify the HTTP handler also for static resources that would otherwise be served directly by IIS. Here's what you need to enter:

```
<system.webServer>
  <add verb="*"
    path="*.sqlx"
    type="Core35.Components.QueryHandler, Core35Lib" />
</system.webServer>
```

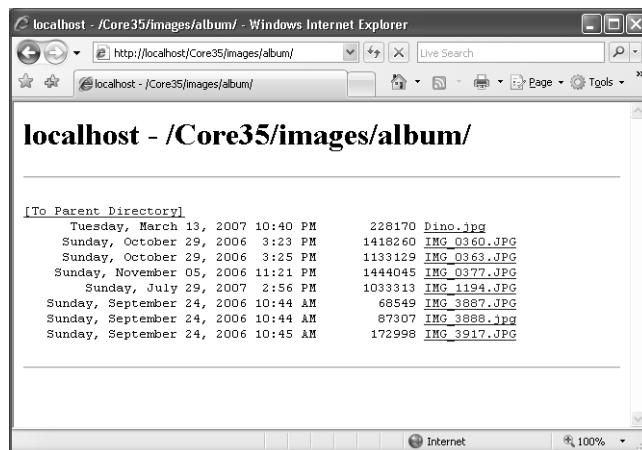
The new section is a direct child of the root tag `<configuration>`. Without this setting, IIS can't recognize the page and won't serve it up. The configuration script instructs IIS 7.0 to forward any `*.sqlx` requests to your application, which knows how to deal with it.

## The Picture Viewer Handler

Let's examine another scenario that involves custom HTTP handlers. Thus far, we have explored custom resources and realized how important it is to register any custom extensions with IIS.

To speed up processing, IIS claims the right of personally serving some resources that typically form a Web application without going down to a particular ISAPI extension. The list 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 18-5.



**FIGURE 18-5** 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 you would let IIS know about your wishes. You can use a particular endpoint that, 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 binding our 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 will process the URL, extract the folder name, and select 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 PictureBoxInfo
{
    public PictureBoxInfo() {
        DisplayWidth = 200;
        ColumnCount = 3;
    }
    public int DisplayWidth;
    public int ColumnCount;
    public string FolderName;
}

public class PictureBoxHandler : IHttpHandler
{
    // Override the ProcessRequest method
    public void ProcessRequest(HttpContext context)
    {
        PictureBoxInfo 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:

```

private ArrayList GetAllImages(string path)
{
    string[] fileTypes = { "*.bmp", "*.gif", "*.jpg", "*.png" };
    ArrayList images = new ArrayList();
    DirectoryInfo di = new DirectoryInfo(path);
    foreach (string ext in fileTypes)
    {
        FileInfo[] files = di.GetFiles(ext);
        if (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. The image is not downloaded as a thumbnail, but it is more simply rendered in a smaller area. 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 most modern 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.

```

string CreateOutputForFolder(PictureBoxInfo info)
{
    ArrayList images = GetAllImages(info.FolderName);
    Table t = new Table();

    int index = 0;
    bool moreImages = true;

```

```

while (moreImages)
{
    TableRow row = new TableRow();
    t.Rows.Add(row);
    for (int i = 0; i < info.ColumnCount; i++)
    {
        TableCell cell = new TableCell();
        row.Cells.Add(cell);

        // Create the image
        Image img = new Image();
        FileInfo fi = (FileInfo)images[index];
        img.ImageUrl = fi.Name;
        img.Width = Unit.Pixel(info.DisplayWidth);

        // Wrap the image in an anchor so that a larger image
        // is shown when the user clicks
        HtmlAnchor a = new HtmlAnchor();
        a.HRef = fi.Name;
        a.Controls.Add(img);
        cell.Controls.Add(a);

        // Check whether there are more images to show
        index++;
        moreImages = (index < images.Count);
        if (!moreImages)
            break;
    }
}
}

```

You might want to make the handler accept some optional query string parameters, such as width and 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:

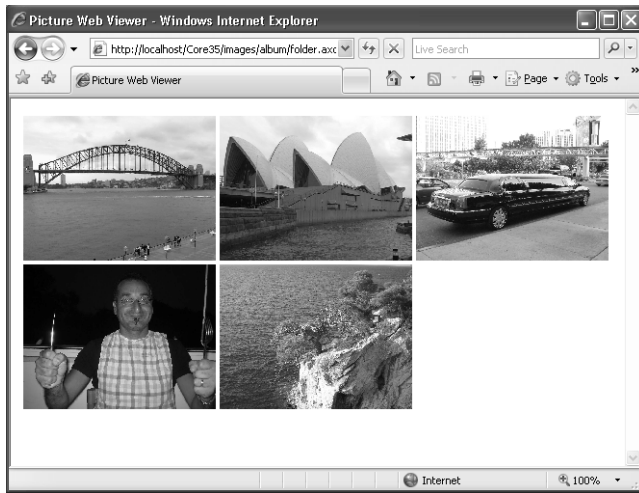
```

PictureViewerInfo info = new PictureViewerInfo();
object p1 = context.Request.Params["Width"];
object p2 = context.Request.Params["Cols"];
if (p1 != null)
    Int32.TryParse((string)p1, out info.DisplayWidth);
if (p2 != null)
    Int32.TryParse((string)p2, out info.ColumnCount);

```

Figure 18-6 shows the handler in action.





**FIGURE 18-6** The picture viewer handler in action with a given number of columns and width.

Registering the handler is easy too. You just add the following script to the *web.config* file:

```
<add verb="*" path="folder.aspx"
      type="Core35.Components.PictureViewerHandler,Core35Lib" />
```

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.

## Serving Images More Effectively

Any page we get from the Web today 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 seven or eight years 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? For example, you can load images from a database or you can generate or modify them 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 support in 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
        int id = -1;
        bool result = Int32.TryParse(ctx.Request.QueryString["id"], out id);
        if (!result)
            ctx.Response.End();

        string connString = "...";
        string cmdText = "SELECT photo FROM employees WHERE employeeid=@id";

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

```
// 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're assuming 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 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.



**Warning** If the database you're using is Northwind (as in the preceding example), an extra step might be required to ensure that the images are correctly managed. For some reason, the SQL Server version of the Northwind database stores the images in the *photo* column of the *Employees* table as OLE objects. This is probably because of the conversion that occurred when the database was upgraded from the Microsoft Access version. As a matter of fact, the array of bytes you receive contains a 78-byte prefix that has nothing to do with the image. Those bytes are just the header created when the image was added as an OLE object to the first version of Access. Although the preceding code works like a champ with regular BLOB fields, it must undergo the following modification to work with the *photo* field of the Northwind.Employees database:

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

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 18-7. 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.

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



**FIGURE 18-7** 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 *dbimage.axd* and the script to enter in the configuration file is shown next:

```
<httpHandlers>
  <add verb="*" path="dbimage.axd"
    type="Core35.Components.DbImageHandler,Core35Lib"/>
</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 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 out-

put 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 capable of providing an object model for image generation. This server-side system is GDI+, and contrary to what some people might have you believe, GDI+ is fair game for generating images on the fly for ASP.NET applications.

As its name suggests, GDI+ is the successor of GDI, the Graphics Device Interface included with versions of the Windows operating system that shipped before Windows XP. The .NET Framework encapsulates the key GDI+ functionalities in a handful of managed classes and makes those functions available to Web, Windows Forms, and Web service applications.

Most of the GDI+ services belong to the following categories: 2D vector graphics and imaging. 2D vector graphics involve drawing simple figures such as lines, curves, and polygons. Under the umbrella of imaging are functions to display, manipulate, save, and convert bitmap and vector images. Finally, a third category of functions can be identified—typography, which includes the display of text in a variety of fonts, sizes, and styles. Having the goal of creating images dynamically, we are most interested in drawing figures and text and in saving the work as JPEGs or GIFs.

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* lacks writing permissions on virtual directories—a pretty common situation.

ASP.NET and GDI+ provide 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

GDI+ supports quite a few image formats, including JPEG, GIF, BMP, and PNG. The whole collection of image formats is in the *ImageFormat* structure from 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 GDI+ 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 touched on. This example shows how to load an existing image, add some copyright notes, and serve the modified version to the user. In doing so, we'll load an image into a *Bitmap* object, obtain a *Graphics* for that bitmap, and use graphics primitives to write. When finished, we'll save the result to the page's output stream and indicate a particular MIME type.

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

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

The page contains no ASP.NET code and displays an image through a static HTML *<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)
{
    object o = context.Request["url"];
    if (o == null)
    {
        context.Response.Write("No image found.");
        context.Response.End();
        return;
    }

    string file = context.Server.MapPath((string)o);
    string 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
    Bitmap bmp = new Bitmap(file);
    Graphics g = Graphics.FromImage(bmp);

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

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

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

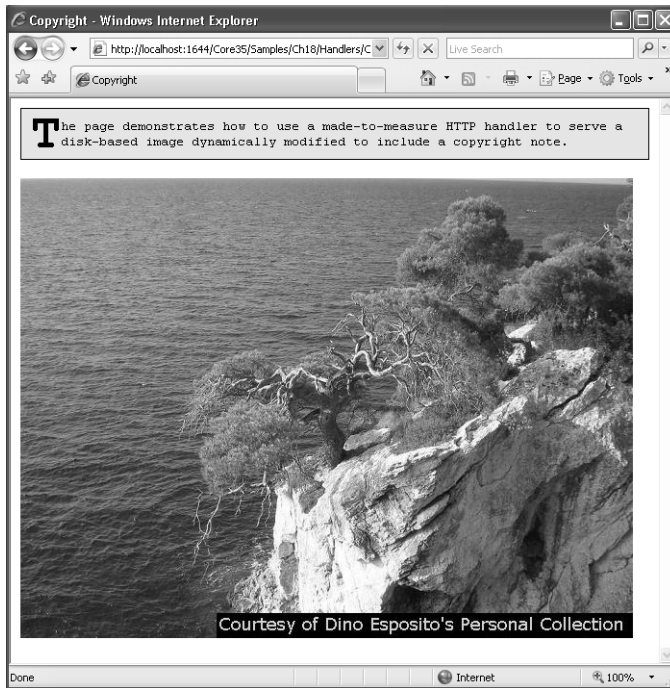
    // Calculate the output rectangle and fill
    float x = ((float) bmp.Width-textSize.Width-3);
    float y = ((float) bmp.Height-textSize.Height-3);
    float w = ((float) x + textSize.Width);
    float h = ((float) y + textSize.Height);
    RectangleF textArea = new RectangleF(x, y, w, h);
    g.FillRectangle(btmBackColor,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

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

    return bmp;
}
```

Figure 18-8 shows the results.

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.



**FIGURE 18-8** A server-resident image has been modified before being displayed.



**Note** 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? In these cases, the original image is served without any further modification. Why is it so? As mentioned, for performance reasons IIS serves static files, such as JPG images, directly without involving any external module, including the ASP.NET runtime. 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 6.0, you must register the JPG extension as an ASP.NET extension for a particular application using the IIS Manager as shown in Figure 18-4. In this case, each request for JPG resources is forwarded to your application and resolved through the HTTP handler.

In IIS 7.0, things are even simpler for developers. All that you have to do is add the following lines to the application's *web.config* file:

```
<system.webServer>
  <handlers>
    <add verb="*"
        path="*.jpg"
        type="Core35.Components.DynImageHandler,Core35Lib" />
  </handlers>
</system.webServer>
```

The *system.webServer* section is a direct child of the root *configuration* node.



## 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="Core35.Components.YourHandler" %>
```

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 once 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 them, 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 make 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? 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 is the right handler 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 represented by 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 18-3.

**TABLE 18-3 Members of the *IHttpHandlerFactory* Interface**

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

The *GetHandler* method has the following signature:

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

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

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

```
class MyHandlerFactory : IHttpHandlerFactory
{
    public IHttpHandler GetHandler(HttpContext context,
        string requestType, String url, String pathTranslated)
    {
        // Feel free to create a pool of HTTP handlers here
        if(context.Request.RequestType.ToLower() == "get")
            return (IHttpHandler) new MyGetHandler();
        else if(context.Request.RequestType.ToLower() == "post")
            return (IHttpHandler) new MyPostHandler();
        return null;
    }

    public void ReleaseHandler(IHttpHandler handler)
    {
        // Nothing to do
    }
}
```

When you use an HTTP handler factory, it's the factory, not the handler, that needs to be registered with 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 *IHttpAsyncHandler* interface are shown in Table 18-4.

TABLE 18-4 Members of the *IHttpAsyncHandler* Interface

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

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

```
IAAsyncResult 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 the pseudocode that the HTTP runtime employs when it deals with asynchronous handlers. The HTTP runtime invokes the *BeginProcessRequest* method as illustrated by the following pseudocode:

```
// Sets an internal member of the HttpContext class with
// the current instance of the asynchronous handler
context.AsyncHandler = 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.AsyncHandler;
    asyncHandler.EndProcessRequest(ar);

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

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

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

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

## Implementing Asynchronous Handlers

Asynchronous handlers essentially serve one particular scenario—when 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 the thread is a valuable member of the ASP.NET thread pool, lengthy tasks 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 *IAAsyncResult* 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 asynchronous pages than in asynchronous HTTP handlers—that is, the same mechanism but applied to *.aspx* resources. In this case, 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 might start I/O-bound and potentially lengthy operations.)

Starting with ASP.NET 2.0, you find ad hoc support for building asynchronous pages more easily and comfortably. An introductory but still practical chapter on asynchronous pages can be found in my book *Programming ASP.NET Applications—Advanced Topics* (Microsoft Press, 2006).



**Warning** I've seen several ASP.NET developers using 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.

## Writing HTTP Modules

So we've learned that any incoming requests for ASP.NET resources are handed over to the worker process for the actual processing within the context of the CLR. In IIS 6.0, the worker process is a distinct process from IIS, so if one ASP.NET application crashes, it doesn't bring down the whole server.

ASP.NET manages a pool of *HttpApplication* objects for each running application and picks up one of the pooled instances to serve a particular request. These objects are based on the class defined in your *global.asax* file, or on the base *HttpApplication* class if *global.asax* is missing. The ultimate goal of the *HttpApplication* object in charge of the request is getting an HTTP handler.

On the way to the final HTTP handler, the *HttpApplication* object makes the request pass through a pipeline of 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.

Generally speaking, 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 highly-configurable nature of ASP.NET makes it possible for you to also write and register your own HTTP modules and make them plug into the ASP.NET runtime pipeline, handle system events, and fire their own events.

### 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 18-5 lists the events that HTTP modules can listen to and handle.

**TABLE 18-5 *HttpApplication* Events**

Event	Description
<i>AcquireRequestState</i> , <i>PostAcquireRequestState</i>	Occurs when the handler that will actually serve the request acquires the state information associated with the request. <i>The post event is not available in ASP.NET 1.x.</i>
<i>AuthenticateRequest</i> , <i>PostAuthenticateRequest</i>	Occurs when a security module has established the identity of the user. <i>The post event is not available in ASP.NET 1.x.</i>
<i>AuthorizeRequest</i> , <i>PostAuthorizeRequest</i>	Occurs when a security module has verified user authorization. <i>The post event is not available in ASP.NET 1.x.</i>
<i>BeginRequest</i>	Occurs as soon as the HTTP pipeline begins to process the request.
<i>Disposed</i>	Occurs when the <i>HttpApplication</i> object is disposed of as a result of a call to <i>Dispose</i> .
<i>EndRequest</i>	Occurs as the last event in the HTTP pipeline chain of execution.
<i>Error</i>	Occurs when an unhandled exception is thrown.
<i>PostMapRequestHandler</i>	Occurs when the HTTP handler to serve the request has been found. <i>The event is not available in ASP.NET 1.x.</i>
<i>PostRequestHandlerExecute</i>	Occurs when the HTTP handler of choice finishes execution. The response text has been generated at this point.
<i>PreRequestHandlerExecute</i>	Occurs just before the HTTP handler of choice begins to work.
<i>PreSendRequestContent</i>	Occurs just before the ASP.NET runtime sends the response text to the client.
<i>PreSendRequestHeaders</i>	Occurs just before the ASP.NET runtime sends HTTP headers to the client.
<i>ReleaseRequestState</i> , <i>PostReleaseRequestState</i>	Occurs when the handler releases the state information associated with the current request. <i>The post event is not available in ASP.NET 1.x.</i>

Event	Description
<i>ResolveRequestCache,</i> <i>PostResolveRequestCache</i>	Occurs when the ASP.NET runtime resolves the request through the output cache. <i>The post event is not available in ASP.NET 1.x.</i>
<i>UpdateRequestCache,</i> <i>PostUpdateRequestCache</i>	Occurs when the ASP.NET runtime stores the response of the current request in the output cache to be used to serve subsequent requests. <i>The post event is not available in ASP.NET 1.x.</i>

All these events are exposed by the *HttpApplication* object that an HTTP module receives as an argument to the *Init* method.

## A Custom HTTP Module

Let's begin coming 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 Core35.Components
{
    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 += new EventHandler(OnBeginRequest);
    app.EndRequest += new EventHandler(OnEndRequest);
}

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)
{
    HttpApplication app = (HttpApplication) sender;
    HttpContext ctx = app.Context;

    // More code here
    ...

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

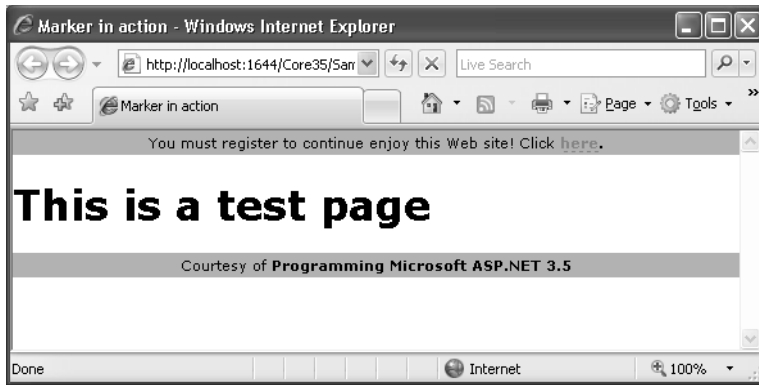
    // PageHeaderText is a constant string defined elsewhere
    ctx.Response.Write(PageHeaderText);
}

public void OnEndRequest(object sender, EventArgs e)
{
    // Get access to the HTTP context
    HttpApplication app = (HttpApplication) sender;
    HttpContext 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 18-9.



**FIGURE 18-9** 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="Core35.Components.MarkerModule,Core35Lib" />
  </httpModules>
</system.web>
```

The *type* attribute is the usual comma-separated string that contains the name of the class and the related assembly. The configuration settings can be entered into the application's configuration file as well as into the global *web.config* file. In the former case, only pages within the application are affected; in the latter case, all pages within all applications are processed by the specified module.

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 just 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:

```
SessionStateModule sess = app.Modules["Session"];
sess.Start += new EventHandler(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.

## The Page Refresh Feature

Let's examine a practical situation in which the ability to filter the request before it gets processed by an HTTP handler helps to implement a feature that would otherwise be impossible. The postback mechanism has a nasty drawback—if the user refreshes the currently displayed


page, the last action taken on the server is blindly repeated. If a new record was added as a result of a previous posting, for example, the application would attempt to insert an identical record upon another postback. Of course, this results in the insertion of identical records and should result in an exception. This snag has existed since the dawn of Web programming and was certainly not introduced by ASP.NET. To implement nonrepeatable actions, some countermeasures are required to essentially transform any critical server-side operation into an *idempotency*. In algebra, an operation is said to be *idempotent* if the result doesn't change regardless of how many times you execute it. For example, take a look at the following SQL command:

```
DELETE FROM employees WHERE employeeid=9
```

You can execute the command 1000 consecutive times, but only one record at most will ever be deleted—the one that satisfies the criteria set in the WHERE clause. Consider this command, instead:

```
INSERT INTO employees VALUES (...)
```

Each time you execute the command, a new record might be added to the table. This is especially true if you have auto-number key columns or nonunique columns. If the table design requires that the key be unique and specified explicitly, the second time you run the command a SQL exception would be thrown.

Although the particular scenario we considered is typically resolved in the data access layer (DAL), the underlying pattern represents a common issue for most Web applications. So the open question is, how can we detect whether the page is being posted as the result of an explicit user action or because the user simply hit F5 or the page refresh () toolbar button?

## The Rationale Behind Page Refresh Operations

The page refresh action is a sort of internal browser operation for which the browser doesn't provide any external notification in terms of events or callbacks. Technically speaking, the page refresh consists of the "simple" reiteration of the latest request. The browser caches the latest request it served and reissues it when the user hits the page refresh key or button. No browsers that I'm aware of provide any kind of notification for the page refresh event—and if there are any that do, it's certainly not a recognized standard.

In light of this, there's no way the server-side code (for example, ASP.NET, classic ASP, or ISAPI DLLs) can distinguish a refresh request from an ordinary submit or postback request. To help ASP.NET detect and handle page refreshes, you need to build surrounding machinery that makes two otherwise identical requests look different. All known browsers implement the refresh by resending the last HTTP payload sent; to make the copy look different from the original, any extra service we write must add more parameters and the ASP.NET page must be capable of catching them.

I considered some additional requirements. The solution should not rely on session state and should not tax the server memory too much. It should be relatively easy to deploy and as unobtrusive as possible.

Outline of the Solution

The solution is based on the idea that each request will be assigned a ticket number and the HTTP module will track the last-served ticket for each distinct page it processes. If the number carried by the page is lower than the last-served ticket for the page, it can only mean that the *same* request has been served already—namely, a page refresh. The solution consists of a couple of building blocks: an HTTP module to make preliminary checks on the ticket numbers, and a custom page class that automatically adds a progressive ticket number to each served page. Making the feature work is a two-step procedure: first, register the HTTP module; second, change the base code-behind class of each page in the relevant application to detect browser refreshes.

The HTTP module sits in the middle of the HTTP runtime environment and checks in every request for a resource in the application. The first time the page is requested (when not posting back), there will be no ticket assigned. The HTTP module will generate a new ticket number and store it in the *Items* collection of the *HttpContext* object. In addition, the module initializes the internal counter of the last-served ticket to 0. Each successive time the page is requested, the module compares the last-served ticket with the page ticket. If the page ticket is newer, the request is considered a regular postback; otherwise, it will be flagged as a page refresh. Table 18-6 summarizes the scenarios and related actions.

TABLE 18-6 Scenarios and Actions

Scenario	Action
Page has no ticket associated: ■ No refresh	Counter of the last ticket served is set to 0. The ticket to use for the next request of the current page is generated and stored in <i>Items</i> .
Page has a ticket associated: ■ Page refresh occurs if the ticket associated with the page is lower than the last served ticket	Counter of the last ticket served is set with the ticket associated with the page. The ticket to use for the next request of the current page is generated and stored in <i>Items</i> .

Some help from the page class is required to ensure that each request—except the first—comes with a proper ticket number. That’s why you need to set the code-behind class of each page that intends to support this feature to a particular class—a process that we’ll discuss in a moment. The page class will receive two distinct pieces of information from the HTTP module—the next ticket to store in a hidden field that travels with the page, and whether or not the request is a page refresh. As an added service to developers, the code-behind class

will expose an extra Boolean property—*IsRefreshed*—to let developers know whether or not the request is a page refresh or a regular postback.



**Important** The *Items* collection on the *HttpContext* class is a cargo collection purposely created to let HTTP modules pass information down to pages and HTTP handlers in charge of physically serving the request. The HTTP module we employ here sets two entries in the *Items* collection. One is to let the page know whether the request is a page refresh; another is to let the page know what the next ticket number is. Having the module pass the page the next ticket number serves the purpose of keeping the page class behavior as simple and linear as possible, moving most of the implementation and execution burden on to the HTTP module.

## Implementation of the Solution

There are a few open points with the solution I just outlined. First, some state is required. Where do you keep it? Second, an HTTP module will be called for each incoming request. How do you distinguish requests for the same page? How do you pass information to the page? How intelligent do you expect the page to be?

It's clear that each of these points might be designed and implemented in a different way than shown here. All design choices made to reach a working solution here should be considered arbitrary, and they can possibly be replaced with equivalent strategies if you want to rework the code to better suit your own purposes. Let me also add this disclaimer: I'm not aware of commercial products and libraries that fix this reposting problem. In the past couple of years, I've been writing articles on the subject of reposting and speaking at various user groups. The version of the code presented in this next example incorporates the most valuable suggestions I've collected along the way. One of these suggestions is to move as much code as possible into the HTTP module, as mentioned in the previous note.

The following code shows the implementation of the HTTP module:

```
public class RefreshModule : IHttpModule
{
    public void Init(HttpApplication app) {
        app.BeginRequest += new EventHandler(OnAcquireRequestState);
    }
    public void Dispose() {
    }
    void OnAcquireRequestState(object sender, EventArgs e) {
        HttpApplication app = (HttpApplication) sender;
        HttpContext ctx = app.Context;
        RefreshAction.Check(ctx);
        return;
    }
}
```



The module listens to the *BeginRequest* event and ends up calling the *Check* method on the helper *RefreshAction* class:

```
public class RefreshAction
{
    static Hashtable requestHistory = null;

    // Other string constants defined here
    ...

    public static void Check(HttpContext ctx) {
        // Initialize the ticket slot
        EnsureRefreshTicket(ctx);

        // Read the last ticket served in the session (from Session)
        int lastTicket = GetLastRefreshTicket(ctx);

        // Read the ticket of the current request (from a hidden field)
        int thisTicket = GetCurrentRefreshTicket(ctx, lastTicket);

        // Compare tickets
        if (thisTicket > lastTicket ||
            (thisTicket==lastTicket && thisTicket==0)) {
            UpdateLastRefreshTicket(ctx, thisTicket);
            ctx.Items[PageRefreshEntry] = false;
        }
        else
            ctx.Items[PageRefreshEntry] = true;
    }

    // Initialize the internal data store
    static void EnsureRefreshTicket(HttpContext ctx)
    {
        if (requestHistory == null)
            requestHistory = new Hashtable();
    }

    // Return the last-served ticket for the URL
    static int GetLastRefreshTicket(HttpContext ctx)
    {
        // Extract and return the last ticket
        if (!requestHistory.ContainsKey(ctx.Request.Path))
            return 0;
        else
            return (int) requestHistory[ctx.Request.Path];
    }

    // Return the ticket associated with the page
    static int GetCurrentRefreshTicket(HttpContext ctx, int lastTicket)
    {
        int ticket;
        object o = ctx.Request[CurrentRefreshTicketEntry];
        if (o == null)
            ticket = lastTicket;
        else
            ticket = Convert.ToInt32(o);
    }
}
```

```

        ctx.Items[RefreshAction.NextPageTicketEntry] = ticket + 1;
        return ticket;
    }

    // Store the last-served ticket for the URL
    static void UpdateLastRefreshTicket(HttpContext ctx, int ticket)
    {
        requestHistory[ctx.Request.Path] = ticket;
    }
}

```

The *Check* method performs the following actions. It compares the last-served ticket with the ticket (if any) provided by the page. The page stores the ticket number in a hidden field that is read through the *Request* object interface. The HTTP module maintains a hashtable with an entry for each distinct URL served. The value in the hashtable stores the last-served ticket for that URL.



**Note** The *Item* indexer property is used to set the last-served ticket instead of the *Add* method because *Item* overwrites existing items. The *Add* method just returns if the item already exists.

In addition to creating the HTTP module, you also need to arrange a page class to use as the base for pages wanting to detect browser refreshes. Here's the code:

```

// Assume to be in a custom namespace
public class Page : System.Web.UI.Page
{
    public bool IsRefreshed {
        get {
            HttpContext ctx = HttpContext.Current;
            object o = ctx.Items[RefreshAction.PageRefreshEntry];
            if (o == null)
                return false;
            return (bool) o;
        }
    }

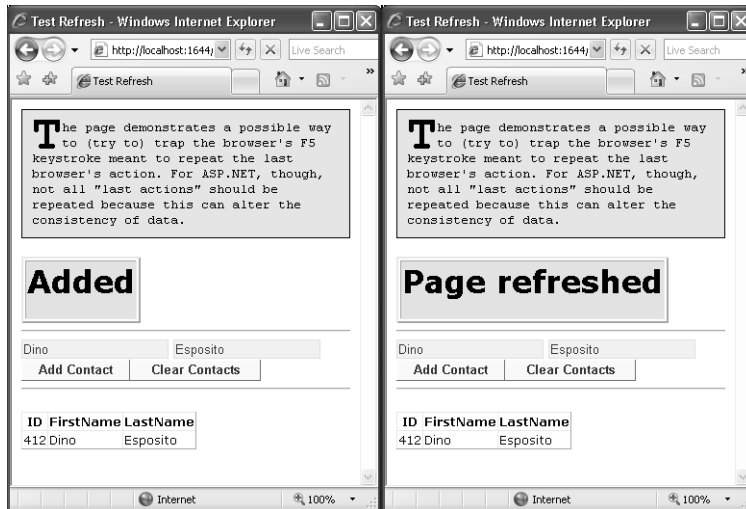
    // Handle the PreRenderComplete event
    protected override void OnPreRenderComplete(EventArgs e) {
        base.OnPreRenderComplete(e);
        SaveRefreshState();
    }

    // Create the hidden field to store the current request ticket
    private void SaveRefreshState() {
        HttpContext ctx = HttpContext.Current;
        int ticket = (int) ctx.Items[RefreshAction.NextPageTicketEntry];
        ClientScript.RegisterHiddenField(
            RefreshAction.CurrentRefreshTicketEntry,
            ticket.ToString());
    }
}

```

The sample page defines a new public Boolean property *IsRefreshed* that you can use in code in the same way you would use *IsPostBack* or *IsCallback*. It overrides *OnPreRenderComplete* to add the hidden field with the page ticket. As mentioned, the page ticket is received from the HTTP module through an ad hoc (and arbitrarily named) entry in the *Items* collection.

Figure 18-10 shows a sample page in action. Let's take a look at the source code of the page.



**FIGURE 18-10** The page doesn't repeat a sensitive action if the user refreshes the browser's view.

```
public partial class TestRefresh : Core35.Components.Page
{
    protected void AddContactButton_Click(object sender, EventArgs e)
    {
        Msg.InnerText = "Added";
        if (!this.IsRefreshed)
            AddRecord(FName.Text, LName.Text);
        else
            Msg.InnerText = "Page refreshed";

        BindData();
    }
    ...
}
```

The *IsRefreshed* property lets you decide what to do when a postback action is requested. In the preceding code, the *AddRecord* method is not invoked if the page is refreshing. Needless to say, *IsRefreshed* is available only with the custom page class presented here. The custom page class doesn't just add the property, it also adds the hidden field, which is essential for the machinery to work.

## Conclusion

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. At the highest level of abstraction, the behavior of an HTTP handler closely resembles that of an ISAPI extension. While the similarity makes sense, a key difference exists. HTTP handlers are managed and CLR-resident components. The CLR, in turn, is hosted by the worker process. An ISAPI extension, on the other hand, is a Win32 library that can live within the IIS process. In the ASP.NET process model, the *aspnet\_isapi* component is a true ISAPI extension that collects requests and dispatches them to the worker process. ASP.NET internally implements an ISAPI-like extensibility model in which HTTP handlers play the role of ISAPI extensions in the IIS world. This model changes in IIS 7.0, at which point managed HTTP modules and extensions will also be recognized within the IIS environment.

HTTP modules are to ISAPI filters what HTTP handlers are to ISAPI extensions. 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.



### Just the Facts

- HTTP handlers and modules are like classic ISAPI extensions and filters except that they are managed components and provide a much simpler, less error-prone programming model.
- An HTTP handler is the ASP.NET component in charge of handling a request. In the end, an ASP.NET page is just an instance of an HTTP handler.
- HTTP handlers are classes that implement the *IHttpHandler* interface and take care of processing the payload of the request.
- HTTP modules are classes that implement the *IHttpModule* interface and listen to application-level events.
- Custom HTTP handlers and modules must be registered with the application, or all applications in the server machine, through special sections in the *web.config* file.



# Programming Microsoft® ASP.NET 3.5

**Your expert guide to the technology for developing next-generation Web sites.**

Get the definitive guide to Microsoft ASP.NET—now updated for version 3.5. Led by well-known programming expert Dino Esposito, you'll delve into core features of ASP.NET as well as the latest capabilities—and build your proficiency creating innovative Web applications.

## Discover how to:

- Author rich, visually consistent pages with themes, wizards, and master pages
- Use the Dynamic Data feature to build and customize data-driven Web applications
- Integrate query operations into the Microsoft .NET platform with LINQ
- Perform state, application, and session management for optimal performance
- Use AJAX and Microsoft Silverlight™ to create rich, interactive Web applications
- Implement security strategies such as forms authentication and membership API
- Understand the internal mechanics of Web forms and the view state technique
- Employ HTTP handlers and modules to service Web requests
- Learn the three pillars of the ASP.NET data binding model

## Get code samples on the Web

For **system requirements**, see the Introduction.

ISBN-13: 978-0-7356-2527-3  
ISBN-10: 0-7356-2527-1



**U.S.A. \$59.99**  
[Recommended]

Web Development/  
ASP.NET



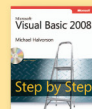
## About the Author

**Dino Esposito** is a well-known ASP.NET and AJAX expert. He speaks at industry events, including DevConnections and Microsoft TechEd, contributes to *MSDN® Magazine* and other publications, and is the author of several Microsoft Press® books, including *Introducing Microsoft ASP.NET AJAX*.

## RESOURCE ROADMAP

### Developer Step by Step

- Hands-on tutorial covering fundamental techniques and features
- Practice files on CD
- Prepares and informs new-to-topic programmers



### Developer Reference

- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology



### Focused Topics

- Deep coverage of advanced techniques and capabilities
- Extensive, adaptable coding examples
- Promotes full mastery of a Microsoft technology



See inside cover for more information



Microsoft®  
**Visual Studio® 2008**

**Microsoft®**