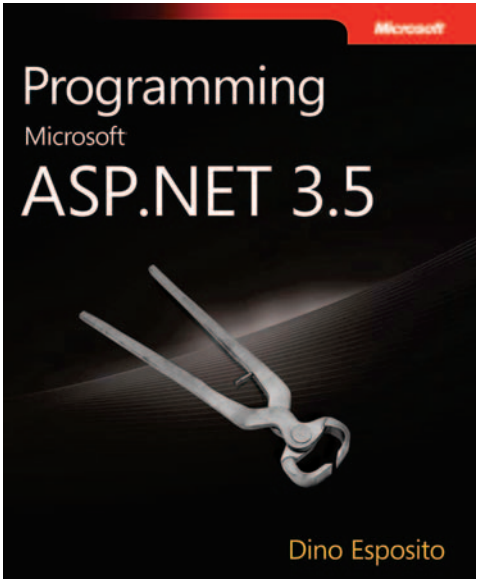


Programming Microsoft® ASP.NET 3.5

Dino Esposito

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12001.aspx

9780735625273

Table of Contents

Chapter 3

Anatomy of an ASP.NET Page

In this chapter:

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.

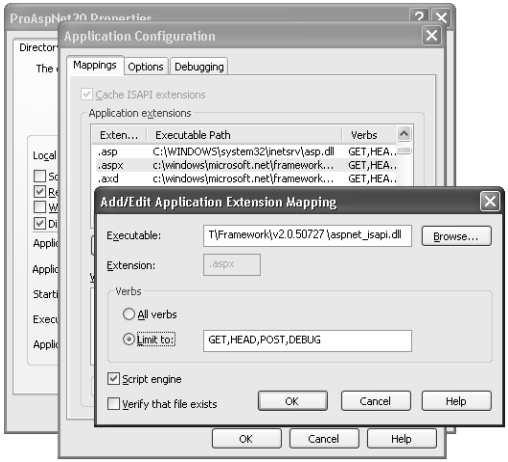


FIGURE 3-1 The IIS application mappings for resources with an *.aspx* extension.

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 *global.asax*.. The mapping is there to ensure that *global.asax* can't be requested directly.
.ascx	ASP.NET user control files.
.ashx	HTTP handlers, namely managed modules that interact with the low-level request and response services of IIS.
.asmx	Files that 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 (*trace.axd*) or script injection (*webresource.axd*).

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.

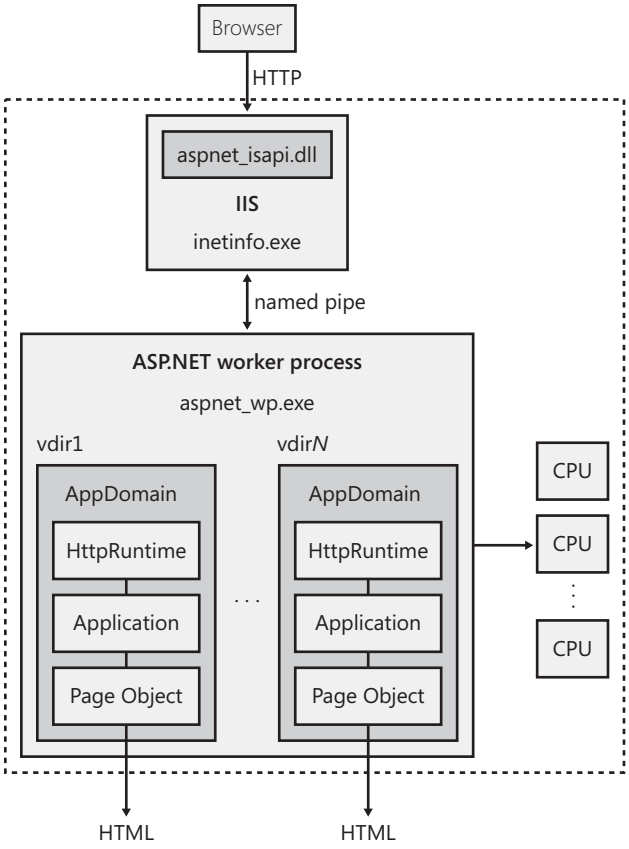


FIGURE 3-2 The ASP.NET runtime environment according to the IIS 5.0 process model.

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.

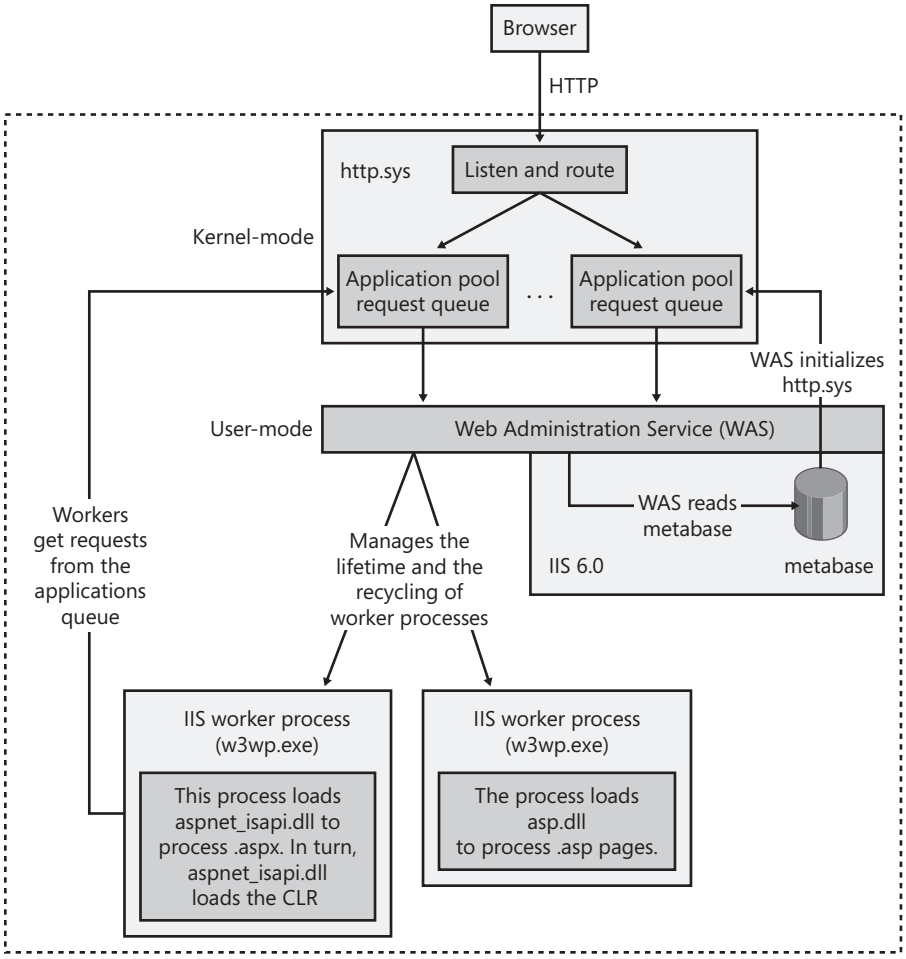


FIGURE 3-3 How ASP.NET and Web applications are processed in IIS 6.0.

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 recreated 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.

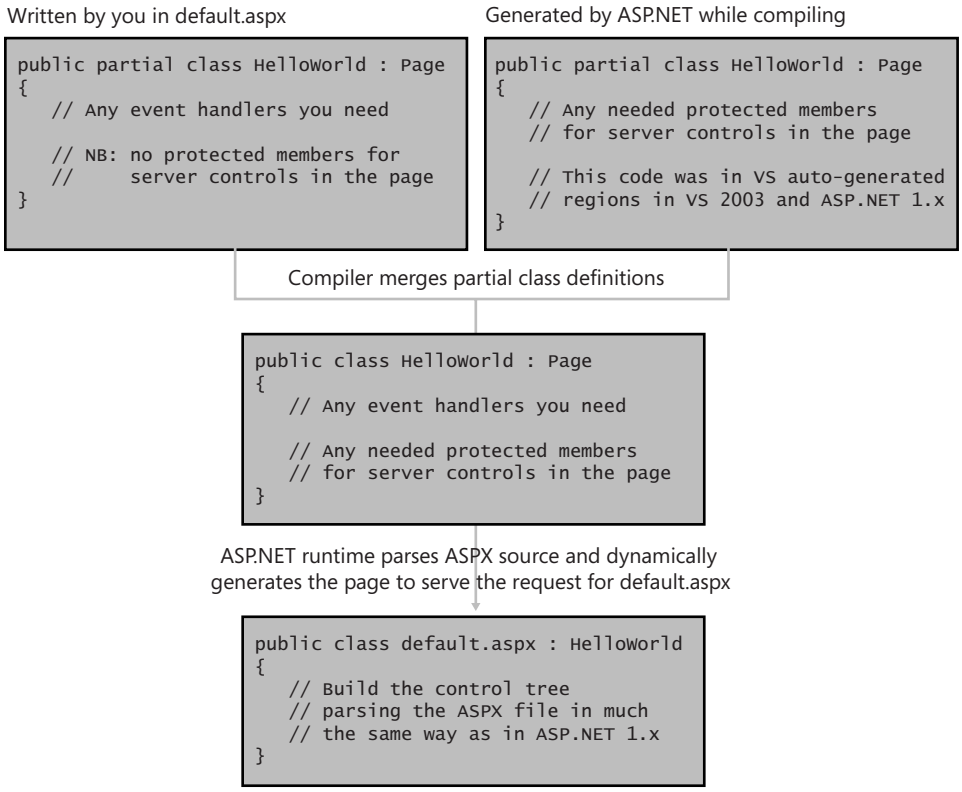


FIGURE 3-4 ASP.NET generates the source code for the dynamic class that will serve a request.

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

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.

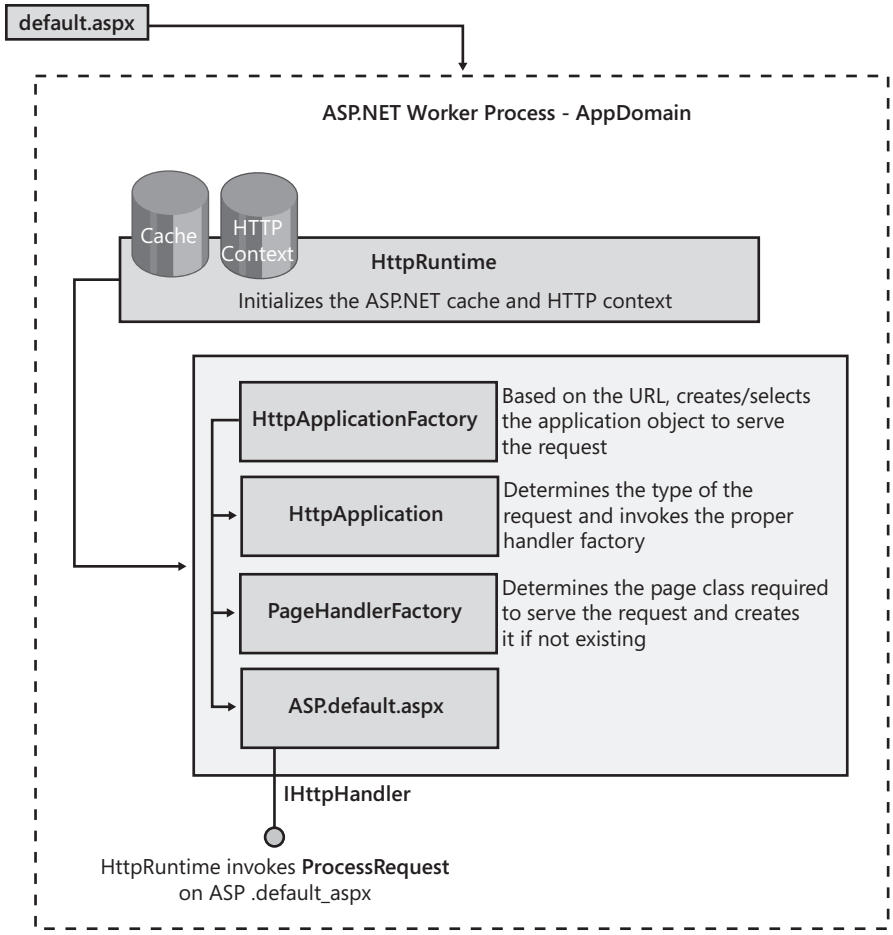


FIGURE 3-5 The HTTP pipeline processing for a page.

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
HttpRemotingHandlerFactory	*.rem; *.soap	Instantiates the object that will take care of a .NET Remoting request routed through IIS. Instantiates an object of type *HttpRemotingHandler*.
PageHandlerFactory	*.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 *.aspx* file. Instantiates an object of a type that derives from *Page*.
SimpleHandlerFactory	*.ashx	Compiles and instantiates the specified HTTP handler from the source code of the .ashx file. Instantiates an object that implements the *IHttpHandler* interface.
WebServiceHandlerFactory	*.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
@ Assembly	Links an assembly to the current page or user control.
@ Control	Defines control-specific attributes that guide the behavior of the control compiler.
@ Implements	Indicates that the page, or the user control, implements a specified .NET Framework interface.
@ Import	Indicates a namespace to import into a page or user control.
@ Master	Identifies an ASP.NET master page. (See Chapter 6.) *This directive is not available with ASP.NET 1.x*.
@ MasterType	Provides a way to create a strongly typed reference to the ASP.NET master page when the master page is accessed from the *Master* property. (See Chapter 6.) *This directive is not available with ASP.NET 1.x*.
@ OutputCache	Controls the output caching policies of a page or user control. (See Chapter 16.)
@ Page	Defines page-specific attributes that guide the behavior of the page compiler and the language parser that will preprocess the page.
@ PreviousPageType	Provides a way to get strong typing against the previous page, as accessed through the *PreviousPage* property.
@ Reference	Links a page or user control to the current page or user control.
@ Register	Creates a custom tag in the page or the control. The new tag (prefix and name) is associated with the namespace and the code of a user-defined control.

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

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

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

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

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

The *@Page* Directive

The *@Page* directive can be used only in *.aspx* pages, and 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
ClassName	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.
CodeFile	Indicates the path to the code-behind class for the current page. The source class file must be deployed to the Web server. *Not available with ASP.NET 1.x.*
CodeBehind	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 *CodeFile* attribute should be used.)
CodeFileBaseClass	Specifies the type name of a base class for a page and its associated code-behind class. The attribute is optional, but when it is used the *CodeFile* attribute must also be present. *Not available with ASP.NET 1.x.*

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

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

Attribute	Description
EnableSessionState	Defines how the page should treat session data. If set to *true*, the session state can be read and written. If set to *false*, session data is not available to the application. Finally, if set to *ReadOnly*, the session state can be read but not changed.
EnableViewState	A Boolean value that indicates whether the page *view state* is maintained across page requests. The view state is the page call context—a collection of values that retain the state of the page and are carried back and forth. View state is enabled by default. (I'll cover this topic in Chapter 15.)
EnableTheming	A Boolean value that indicates whether the page will support themes for embedded controls. Set to *true* by default. *Not available in ASP.NET 1.x*.
EnableViewStateMac	A Boolean value that indicates ASP.NET should calculate a machine-specific authentication code and append it to the view state of the page (in addition to Base64 encoding). The *Mac* in the attribute name stands for *machine authentication check*. When the attribute is *true*, upon postbacks ASP.NET will check the authentication code of the view state to make sure that it hasn't been tampered with on the client.
ErrorPage	Defines the target URL to which users will be automatically redirected in case of unhandled page exceptions.
MaintainScrollPosition-OnPostback	Indicates whether to return the user to the same scrollbar position in the client browser after postback. The default is false.
SmartNavigation	A Boolean value that indicates whether the page supports the Microsoft Internet Explorer 5 or later smart navigation feature. Smart navigation allows a page to be refreshed without losing scroll position and element focus.
Theme, StyleSheetTheme	Indicates the name of the theme (or style-sheet theme) selected for the page. *Not available with ASP.NET 1.x*.
Transaction	Indicates whether the page supports or requires transactions. Feasible values are: *Disabled*, *NotSupported*, *Supported*, *Required*, and *RequiresNew*. Transaction support is disabled by default.
ValidateRequest	A Boolean value that indicates whether request validation should occur. If this value is set to *true*, ASP.NET checks all input data against a hard-coded list of potentially dangerous values. This functionality helps reduce the risk of cross-site scripting attacks for pages. The value is *true* by default. *This feature is not supported in ASP.NET 1.0*.
ViewStateEncryption-Mode	Indicates how view state is encrypted, with three possible enumerated values: Auto, Always, or Never. The default is Auto 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
ClientTarget	Indicates the target browser for which ASP.NET server controls should render content.
CodePage	Indicates the code page value for the response. Set this attribute only if you created the page using a code page other than the default code page of the Web server on which the page will run. In this case, set the attribute to the code page of your development machine. A code page is a character set that includes numbers, punctuation marks, and other glyphs. Code pages differ on a per-language basis.
ContentType	Defines the content type of the response as a standard MIME type. Supports any valid HTTP content type string.
Culture	Indicates the culture setting for the page. Culture information includes the writing and sorting system, calendar, and date and currency formats. The attribute must be set to a non-neutral culture name, which means it must contain both language and country information. For example, *en-US* is a valid value, unlike *en* alone, which is considered country-neutral.
LCID	A 32-bit value that defines the locale identifier for the page. By default, ASP.NET uses the locale of the Web server.
ResponseEncoding	Indicates the character encoding of the page. The value is used to set the *CharSet* attribute on the content type HTTP header. Internally, ASP.NET handles all strings as Unicode.
Title	Indicates the title of the page. Not really useful for regular pages which would likely use the <*title*> HTML tag, the attribute has been defined to help developers add a title to content pages where access to the <*title*> attribute may not be possible. (This actually depends on how the master page is structured.)
UICulture	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. *Not included in ASP.NET 1.x.*
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. *Not included if version 1.0 of the .NET Framework is installed.*
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. *Only included in ASP.NET 3.5.*
System.ServiceModel	Defines classes and structure for Windows Communication Foundation (WCF) services. *Only included in ASP.NET 3.5.*
System.ServiceModel.Web	Defines the additional classes required by ASP.NET and AJAX to support WCF services. *Only included in ASP.NET 3.5.*
System.WorkflowServices	Defines classes for making workflows and WCF services interact. *Only included in ASP.NET 3.5.*

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
Application	Instance of the *HttpApplicationState* class; represents the state of the application. It is functionally equivalent to the ASP intrinsic *Application* object.
Cache	Instance of the *Cache* class; implements the cache for an ASP.NET application. More efficient and powerful than *Application*, it supports item priority and expiration.
Profile	Instance of the *ProfileCommon* class; represents the user-specific set of data associated with the request.
Request	Instance of the *HttpRequest* class; represents the current HTTP request. It is functionally equivalent to the ASP intrinsic *Request* object.
Response	Instance of the *HttpResponse* class; sends HTTP response data to the client. It is functionally equivalent to the ASP intrinsic *Response* object.
Server	Instance of the *HttpServerUtility* class; provides helper methods for processing Web requests. It is functionally equivalent to the ASP intrinsic *Server* object.
Session	Instance of the *HttpSessionState* class; manages user-specific data. It is functionally equivalent to the ASP intrinsic *Session* object.
Trace	Instance of the *TraceContext* class; performs tracing on the page.
User	An *IPrincipal* 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
ClientScript	Gets a *ClientScriptManager* object that contains the client script used on the page. *Not available with ASP.NET 1.x.*
Controls	Returns the collection of all the child controls contained in the current page.
ErrorPage	Gets or sets the error page to which the requesting browser is redirected in case of an unhandled page exception.
Form	Returns the current *HtmlForm* object for the page. *Not available with ASP.NET 1.x.*

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

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
ClientID	Always returns the empty string.
ClientQueryString	Gets the query string portion of the requested URL. *Not available with ASP.NET 1.x.*
ClientTarget	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.
EnableViewState	Indicates whether the page has to manage view-state data. You can also enable or disable the view-state feature through the *EnableViewState* attribute of the *@Page* directive.
EnableViewStateMac	Indicates whether ASP.NET should calculate a machine-specific authentication code and append it to the page view state.
EnableTheming	Indicates whether the page supports themes. *Not available with ASP.NET 1.x.*
ID	Always returns the empty string.
MaintainScrollPositionOnPostback	Indicates whether to return the user to the same position in the client browser after postback. *Not available with ASP.NET 1.x.*
SmartNavigation	Indicates whether smart navigation is enabled. Smart navigation exploits a bunch of browser-specific capabilities to enhance the user's experience with the page.
StyleSheetTheme	Gets or sets the name of the style sheet applied to this page. *Not available with ASP.NET 1.x.*

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

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

Methods of the *Page* Class

The whole range of *Page* methods can be classified in a few categories based on the tasks each method accomplishes. A few methods are involved with the generation of the markup for the page; others are helper methods to build the page and manage the constituent controls. Finally, a third group collects all the methods 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
DataBind	Binds all the data-bound controls contained in the page to their data sources. The *DataBind* method doesn't generate code itself but prepares the ground for the forthcoming rendering.
RenderControl	Outputs the HTML text for the page, including tracing information if tracing is enabled.
VerifyRenderingInServerForm	Controls call this method when they render to ensure that they are included in the body of a server form. The method does not return a value, but it throws an exception in case of error.

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

Not directly exposed by the *Page* class, but strictly related to it, is the *GetWebResourceUrl* method on the *ClientScriptManager* class 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
DesignerInitialize	Initializes the instance of the *Page* class at design time, when the page is being hosted by RAD designers such as Visual Studio.
FindControl	Takes a control's ID and searches for it in the page's naming container. The search doesn't dig out child controls that are naming containers themselves.

Method	Description
GetTypeHashCode	Retrieves the hash code generated by *ASP.xxx_aspx* page objects at run time. In the base *Page* class, the method implementation simply returns *0*; significant numbers are returned by classes used for actual pages.
GetValidators	Returns a collection of control validators for a specified validation group. *Not available with ASP.NET 1.x.*
HasControls	Determines whether the page contains any child controls.
LoadControl	Compiles and loads a user control from an .ascx file, and returns a *Control* object. If the user control supports caching, the object returned is *PartialCachingControl*.
LoadTemplate	Compiles and loads a user control from an .ascx file, and returns it wrapped in an instance of an internal class that implements the *ITemplate* interface. The internal class is named *SimpleTemplate*.
MapPath	Retrieves the physical, fully qualified path that an absolute or relative virtual path maps to.
ParseControl	Parses a well-formed input string, and returns an instance of the control that corresponds to the specified markup text. If the string contains more controls, only the first is taken into account. The *runat* attribute can be omitted. The method returns an object of type *Control* and must be cast to a more specific type.
RegisterRequiresControlState	Registers a control as one that requires control state. *Not available with ASP.NET 1.x.*
RegisterRequiresPostBack	Registers the specified control to receive a postback handling notice, even if its ID doesn't match any ID in the collection of posted data. The control must implement the *IPostBackDataHandler* interface.
RegisterRequiresRaiseEvent	Registers the specified control to handle an incoming postback event. The control must implement the *IPostBackEventHandler* interface.
RegisterViewStateHandler	Mostly for internal use, the method sets an internal flag 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 *HtmlForm* server control for the page calls this method. There's no need to call it from within user applications.
ResolveUrl	Resolves a relative URL into an absolute URL based on the value of the *TemplateSourceDirectory* property.
Validate	Instructs any validation controls included 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
GetCallbackEventReference	Obtains a reference to a client-side function that, when invoked, initiates a client call back to server-side events. *Not available with ASP.NET 1.x.*
GetPostBackClientEvent	Calls into *GetCallbackEventReference*.
GetPostBackClientHyperlink	Appends *javascript:* to the beginning of the return string received from *GetPostBackEventReference*. `javascript:__doPostBack('CtlID','')`
GetPostBackEventReference	Returns the prototype of the client-side script function that causes, when invoked, a postback. It takes a *Control* and an argument, and it returns a string like this: `__doPostBack('CtlID','')`
IsClientScriptBlockRegistered	Determines whether the specified client script is registered with the page. *Marked as obsolete.*
IsStartupScriptRegistered	Determines whether the specified client startup script is registered with the page. *Marked as obsolete.*

Method	Description
RegisterArrayDeclaration	Use this method to add an ECMAScript array to the client page. This method accepts the name of the array and a string that will be used verbatim as the body of the array. For example, if you call the method with arguments such as *theArray* and *"'a', 'b'"*, you get the following JavaScript code: `var theArray = new Array('a', 'b');` *Marked as obsolete.*
RegisterClientScriptBlock	An ASP.NET page uses this method to emit client-side script blocks in the client page just after the opening tag of the HTML *<form>* element. *Marked as obsolete.*
RegisterHiddenField	Use this method to automatically register a hidden field on the page. *Marked as obsolete.*
RegisterOnSubmitStatement	Use this method to emit client script code that handles the client *OnSubmit* event. The script should be a JavaScript function call to client code registered elsewhere. *Marked as obsolete.*
RegisterStartupScript	An ASP.NET page uses this method to emit client-side script blocks in the client page just before closing the HTML *<form>* element. *Marked as obsolete.*
SetFocus	Sets the browser focus to the specified control. *Not available with ASP.NET 1.x.*

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

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

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

The *Begin* handler is responsible for starting any operation you fear can block the underlying thread for too long. The handler is expected to return an *IAsyncResult* object to describe the state of the asynchronous task. 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 *IAsyncResult* interface. In most cases, in fact, to start lengthy operations you just use built-in classes that already implement the asynchronous pattern and provide *IAsyncResult* ready-made objects.

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.

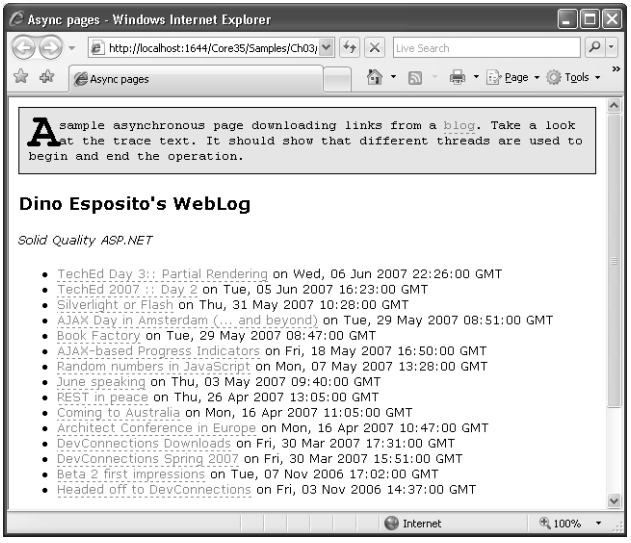
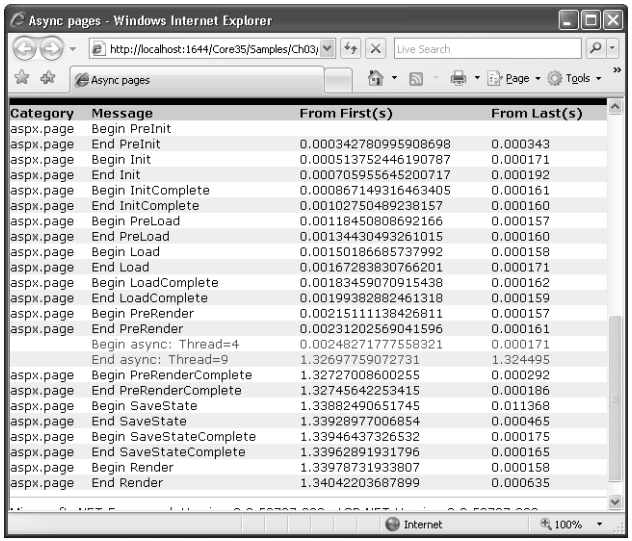


FIGURE 3-6 A sample asynchronous page downloading links from an RSS feed.

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	1.33928977006854	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 substage. For a page, only one operation takes place in between the *Init* and *InitComplete* events: tracking of view-state changes is turned on. Tracking view state is the operation that ultimately enables controls to *really* persist in the storage medium any values that are programmatically added to the *ViewState* collection. Simply put, for controls not tracking their view state, any values added to their *ViewState* are lost across postbacks.

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

Important In light of the previous statement, note that any value written to the *ViewState* collection before *InitComplete* won't be available on the next postback. 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 substages) a page passes through on the way to generate the markup for the browser. A deep understanding of the page life cycle and eventing model is key to diagnosing possible problems and implementing advanced features quickly and efficiently.

In this chapter, 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.