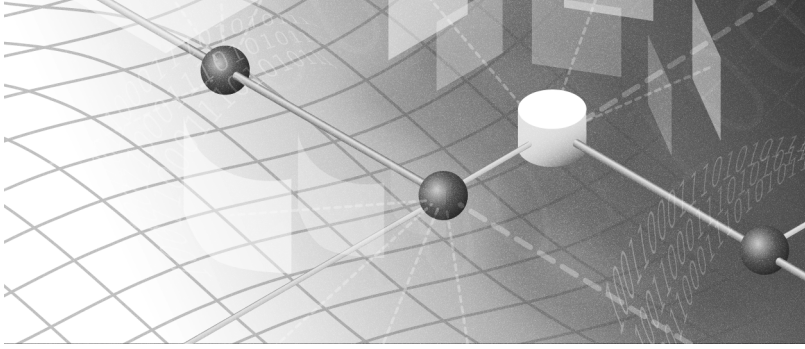

2

Web Forms Internals

Few things are harder to put up with than the annoyance of a good example.
—Mark Twain

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 Web Services (.asmx files), Web user controls (.ascx files), HTTP handlers (.ashx files), and ASP.NET application files such as the global.asax file. But what does it mean exactly that an ASP.NET page is compiled? How does the ASP.NET runtime turn the source code of an .aspx file into a .NET Framework compilable class? And what becomes of the dynamically created assembly when the associated source file gets updated? And finally, what happens once a compiled assembly has been associated with the requested .aspx URL?

Don't be too surprised to find all these questions crowding your mind at this time. Their presence indicates you're on the right track and ready to learn more about the underpinnings of the Web Forms programming model.

Executing ASP.NET Pages

The expression *compiled page* is at once precise as well as vague and generic. It is *precise* because it tells you exactly what happens when a URL with an .aspx extension is requested. It is *vague* because it doesn't specify which module launches and controls the compiler and what actual input the compiler receives on the command line. Finally, it is *generic* because it omits a fair number of details.

In this chapter, we simply aim to unveil all the mysteries fluttering around the dynamic compilation of ASP.NET pages. We'll do this by considering the actions performed on the Web server, and which modules perform them, when a request arrives for an .aspx page.

> **Note** Much of the content of this chapter is based on the behavior of the ASP.NET runtime version 1.0 and version 1.1 with Internet Information Services (IIS) 5.0 serving as the Web server. Some key differences apply when using Windows Server 2003 and IIS 6.0. Any significant differences that affect the ASP.NET way of working are noted. Throughout this chapter, IIS is always IIS 5.0 unless another version is explicitly mentioned.
>
> The first part of this chapter discusses under-the-hood details that might not interest you, as they aren't strictly concerned with the development of ASP.NET applications. Reading this chapter in its entirety is not essential to understanding fundamental techniques of ASP.NET programming. So, if you want, you can jump directly to the "The Event Model" section, which is the section in which we discuss what happens once an ASP.NET page has been requested and starts being processed.

The IIS Resource Mappings

All resources you can access on an IIS-based Web server are grouped by their 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 ISAPI extensions—that is, plain-old Win32 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 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 2-1.

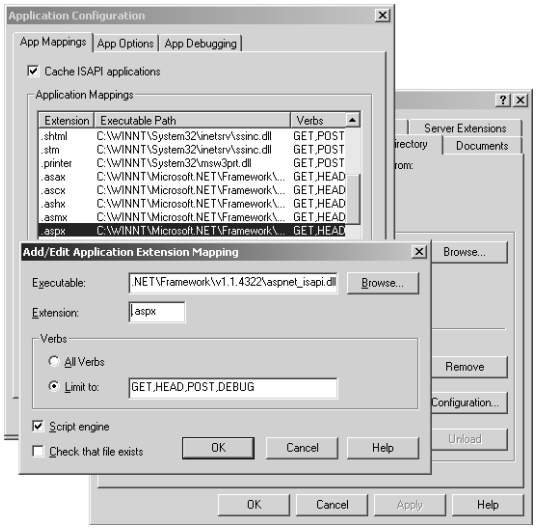

Figure 2-1 The IIS application mappings for resources with an .aspx extension.

Just like any other ISAPI extension, aspnet_isapi.dll is hosted by the IIS 5.0 process—the executable named inetinfo.exe. Resource mappings are stored in the IIS metabase. Upon installation, ASP.NET modifies the IIS metabase to make sure that aspnet_isapi.dll can handle all the resources identified by the extensions listed in Table 2-1.

Table 2-1 IIS Application Mappings for aspnet_isapi.dll

Extension	Resource Type
.asax	ASP.NET application files. The typical example is global.asax.
.ascx	Web user control files used to embed pagelets in ASP.NET pages.
.ashx	HTTP handlers, namely managed modules that interact with the low-level request and response services of IIS. (See Chapter 23.)
.asmx	Files that implement XML Web services.
.aspx	Files that represent ASP.NET pages.

(continued)

Table 2-1 IIS Application Mappings for aspnet_isapi.dll *(continued)*

Extension	Resource Type
.axd	Extension that identifies the ASP.NET trace-viewer application (trace.axd). When invoked in a virtual folder, the trace viewer displays trace information for every page in the application. (See Chapter 4.)
.rem	Fake resource used to qualify the Uniform Resource Identifier (URI) of a .NET Remoting object hosted by IIS.
.soap	Same as .rem.

In addition, the aspnet_isapi.dll extension handles other typical Microsoft Visual Studio .NET extensions such as .cs, .csproj, .vb, .vbproj, .licx, .config, .resx, .webinfo, and .vsdisco. Other extensions added with Visual Studio .NET 2003 for J# projects are .java, .jsl, .resources, .vjsproj.

The ASP.NET ISAPI extension doesn't process the .aspx file but acts as a dispatcher. It collects all the information available about the invoked URL and the underlying resource, and it routes the request toward another distinct process—the ASP.NET worker process.

The ASP.NET Worker Process

The ASP.NET worker process represents the ASP.NET runtime environment. It consists of a Win32 unmanaged executable named aspnet_wp.exe, which hosts the .NET common language runtime (CLR). This process is the executable you need to attach to in order to debug ASP.NET applications. The ASP.NET worker process activates the HTTP pipeline that will actually process the page request. The HTTP pipeline is a collection of .NET Framework classes that take care of compiling the page assembly and instantiating the related page class.

The connection between aspnet_isapi.dll and aspnet_wp.exe is established through a *named pipe*—a Win32 mechanism for transferring data over process boundaries. As the name suggests, a named pipe works like a pipe: you enter data in one end, and the same data comes out the other end. Pipes can be established both to connect local processes and processes running on remote machines. Figure 2-2 illustrates the ASP.NET layer built on top of IIS.

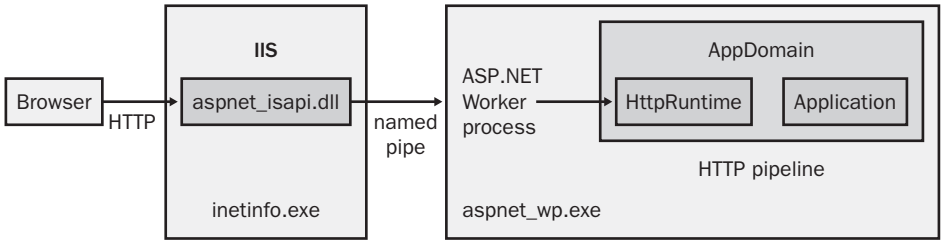

Figure 2-2 IIS receives page requests and forwards them to the ASP.NET runtime.

How the ASP.NET Runtime Works

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 for 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. (I'll cover ASP.NET configuration files in Chapter 12.)

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 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. All the assemblies needed to process the page are now ready to use because they were compiled upon the first call. Figure 2-3 provides a more general view of the ASP.NET runtime.

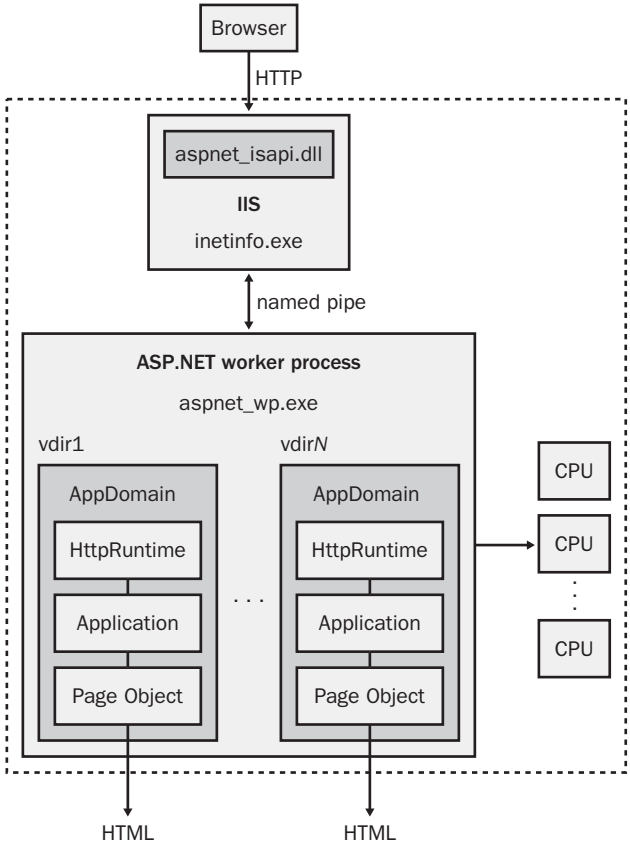

Figure 2-3 The ASP.NET runtime and the various AppDomains.

Tip To configure the ASP.NET runtime to work as a Web garden—that is, to have more worker processes running on multiple CPUs in the same physical server—open the machine.config file and locate the *<processModel>* section. Next, you set the *webGarden* attribute to *true* (because it is *false* by default) and the *cpuMask* attribute to a bit mask in which each 1 identifies an available and affined CPU. If ASP.NET is running with IIS 6.0, you must use the IIS Manager to configure Web gardens. In this case, the settings in machine.config are ignored. If *webGarden* is *false*, the *cpuMask* setting is ignored and only one process is scheduled regardless of how many CPUs you have.

Note that the documentation available with version 1.0 of the .NET Framework is a bit confusing on this point. Documentation in version 1.1 is clear and correct.

Processes, AppDomains, and Threads

In .NET, executable code must be loaded into the CLR to be *managed* while running. To manage the application's code, the CLR must first obtain a pointer to an AppDomain. AppDomains are separate units of processing that the CLR recognizes in a running process. All .NET processes run at least one AppDomain—known as the default AppDomain—that gets created during the CLR initialization. An application can have additional AppDomains. Each AppDomain is independently configured and given personal settings, such as security settings, reference paths, and configuration files.

AppDomains are separated and isolated from one another in a way that resembles process separation in Win32. The CLR enforces isolation by preventing direct calls between objects living in different AppDomains. From the CPU perspective, AppDomains are much more lightweight than Win32 processes. The certainty that AppDomains run type-safe code allows the CLR to provide a level of isolation that's as strong as the process boundaries but more cost effective. Type-safe code cannot cause memory faults, which in Win32 were one of the reasons to have a physical separation between process-memory contexts. An AppDomain is a logical process and, as such, is more lightweight than a true process.

Managed code running in an AppDomain is carried out by a particular thread. However, threads and AppDomains are orthogonal entities in the sense that you can have several threads active during the execution of the AppDomain code, but a single thread is in no way tied to run only within the context of a given AppDomain.

The .NET Remoting API is as a tailor-made set of system services for accessing an object in an external AppDomain.

Process Recycling

The behavior and performance of the ASP.NET worker process is constantly monitored to catch any form of decay as soon as possible. Parameters used to evaluate the performance include the number of requests served and queued, the total life of the process, and the percentage of physical memory (60% by default) it can use.

The *<processModel>* element of the machine.config file defines threshold values for all these parameters. The aspnet_isapi.dll checks the overall state of the current worker process before forwarding any request to it. If the process breaks one of these measures of good performance, a new worker process is started to serve the next request. The old process continues running as long as

there are requests pending in its own queue. After that, when it ceases to be invoked, it goes into idle mode and is then shut down.

This automatic scavenging mechanism is known as *process recycling* and is one of the aspects that improve the overall robustness and efficiency of the ASP.NET platform. In this way, in fact, memory leaks and run-time anomalies are promptly detected and overcome.

Process Recycling in IIS 6.0

Process recycling is also a built-in feature of IIS 6.0 that all types of Web applications, including ASP.NET and ASP applications, automatically take advantage of. More often than not and in spite of the best efforts to properly build them, Web applications leak memory, are poorly coded, or have other run-time problems. For this reason, administrators will periodically encounter situations that necessitate rebooting or restarting a Web server.

Up to the release of IIS 6.0, restarting a Web site required interrupting the entire Web server. In IIS 6.0, all user code is handled by worker processes, which are completely isolated from the core Web server. Worker processes are periodically recycled according to the number of requests they served, the memory occupied, and the time elapsed since activation. Worker processes are also automatically shut down if they appear to hang or respond too slowly. An ad hoc module in IIS 6.0 takes care of replacing faulty processes with fresh new ones.

> **Note** In IIS 6.0, you'll find many design and implementation features of ASP.NET that are enhanced and extended to all resources. Historically, Microsoft was originally developing IIS 6.0 and ASP.NET together. Microsoft split them into separate projects when a decision was made to ship an initial version of ASP.NET prior to shipping IIS with a new version of Windows. ASP.NET clearly needed to support older versions of IIS, so a parallel IIS 5.0 model for ASP.NET was also built. In that sense, the ASP.NET model for IIS 5.0 matured much more quickly and inspired a lot of features in the newest IIS 6.0 model. As a significantly different product, IIS 6.0 takes the essence of the ASP.NET innovations and re-architects them in a wider and more general context. As a result, specific features of ASP.NET (for example, output caching and process recycling) become features of the whole Web server infrastructure with IIS 6.0. Those features are available to all Web applications hosted by IIS 6.0, including ASP.NET applications. ASP.NET is designed to detect the version of IIS and adjust its way of working.

Configuring the ASP.NET Worker Process

The aspnet_isapi module controls the behavior of the ASP.NET worker process through a few parameters. Table 2-2 details the information that gets passed between the ASP.NET ISAPI extension and the ASP.NET worker process.

Table 2-2 Parameters of the ASP.NET Process

Parameter	Description
IIS-Process-ID	The process ID number of the parent IIS process.
This-Process-Unique-ID	A unique process ID used to identify the worker process in a Web garden configuration.
Number-of-Sync-Pipes	Number of pipes to listen to for information.
RPC_C_AUTHN_LEVEL_XXX	Indicates the required level of authentication for DCOM security. Default is *Connect*.
RPC_C_IMP_LEVEL_XXX	Indicates the authentication level required for COM security. Default is *Impersonate*.
CPU-Mask	Bit mask indicating which CPUs are available for ASP.NET processes if the run time is configured to work as a Web garden.
Max-Worker-Threads	Maximum number of worker threads per CPU in the thread pool.
Max-IO-Threads	Maximum number of IO threads per CPU in the thread pool.

Default values for the arguments in Table 2-2 can be set by editing the attributes of the <*processModel*> section in the machine.config file. (I'll cover the machine.config file in more detail in Chapter 12.)

These parameters instruct the process how to perform tasks that need to happen before the CLR is loaded. Setting COM security is just one such task, and that's why authentication-level values need to be passed to the ASP.NET worker process. What does ASP.NET have to do with COM security? Well, the CLR is actually exposed as a COM object. (Note that the CLR itself is not made of COM code, but the interface to the CLR is a COM object.)

Other parameters are the information needed to hook up the named pipes between the ISAPI extension and the worker process. The names for the pipes are generated randomly and have to be communicated. The worker process retrieves the names of the pipes by using the parent process ID (that is, the IIS process) and the number of pipes created.

> **Note** All the system information needed to set up the ASP.NET worker process (that is, the contents of the machine.config file) is read by the aspnet_isapi.dll unmanaged code prior to spawning any instance of the worker process.

About the Web Garden Model

The *This-Process-Unique-ID* parameter is associated with Web garden support. When multiple worker processes are used in a Web garden scenario, the aspnet_isapi.dll needs to know which process it's dealing with. Any HTTP request posted to the pipe must address a precise target process, and this information must be written into the packet sent through the pipe. The typical way of identifying a process is by means of its process ID.

Unfortunately, though, aspnet_isapi.dll can't know the actual ID of the worker process being spawned because the ID won't be determined until the kernel is done with the *CreateProcess* API function. The following pseudocode demonstrates that the *[process_id]* argument of aspnet_wp.exe can't be the process ID of the same process being created!

```
// aspnet_isapi.dll uses this code to create a worker process
CreateProcess("aspnet_wp.exe", "[iis_id] [process_id] ...", ...);
```

For this reason, aspnet_isapi.dll generates a unique but fake process ID and uses that ID to uniquely identify each worker process running on a multiprocessor machine configured as a Web garden. In this way, the call we just saw is rewritten as follows:

```
// [This-Process-Unique-ID] is a unique GUID
// generated by aspnet_isapi.dll
CreateProcess("aspnet_wp.exe", "[iis_id] [This-Process-Unique-ID] ...", ...);
```

The worker process caches the *This-Process-Unique-ID* argument and uses it to recognize which named-pipe messages it has to serve.

ASP.NET and the IIS 6.0 Process Model

IIS 6.0, which ships as a part of Windows Server 2003, implements its HTTP listener as a kernel-level module. As a result, all incoming requests are first managed by such a driver—http.sys—and in kernel mode. 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*. An application pool is a blanket term that identifies a worker process and a virtual directory. A module, known as 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.

So when a request arrives, the driver looks at the URL and queues the request to the corresponding application pool. The WAS is responsible for creating and administering the worker processes for the various pools. The IIS worker process is an executable named w3wp.exe, whose main purpose is extracting HTTP requests from the kernel-mode queue. The worker process hosts a core application handler DLL to actually process the request and load ISAPI extensions and filters as appropriate.

Looking at the diagram of ASP.NET applications in Figure 2-4, you can see the IIS 6.0 process model eliminates the need for aspnet_wp.exe. The w3wp.exe loads the aspnet_isapi.dll, and in turn, the ISAPI extension loads the CLR in the worker process and launches the pipeline. With IIS 6.0, ASP.NET is managed by IIS and no longer concerns itself with things like process recycling, Web gardening, and isolation from the Web server.

(continued)

ASP.NET and the IIS 6.0 Process Model (continued)

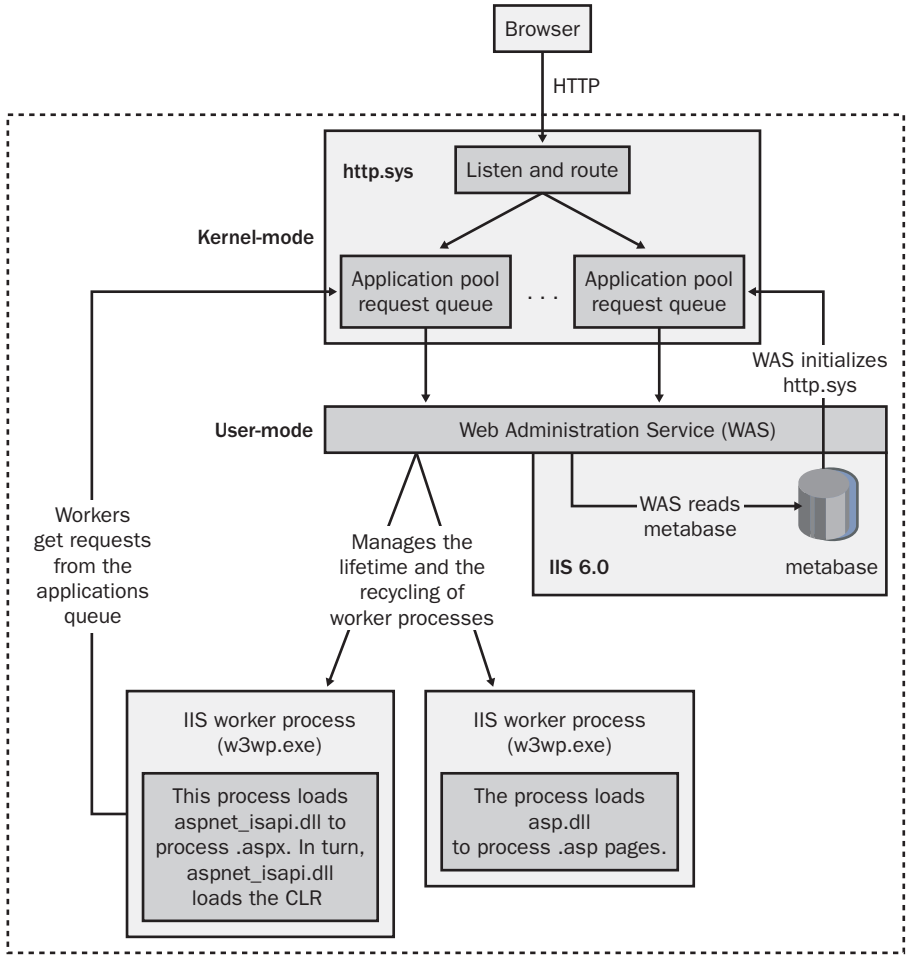

Figure 2-4 How Web applications are processed in IIS 6.0.

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 aspnet_wp.exe is required. The HTTP request arrives 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 and health checks.

ASP.NET and the IIS 6.0 Process Model

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

The ASP.NET HTTP Pipeline

The ASP.NET worker process is responsible for running the Web application that lives behind the requested URL. It passes any incoming HTTP requests to the so-called HTTP pipeline—that is, the fully extensible chain of managed objects that works according to the classic concept of a pipeline. Unlike ASP pages, ASP.NET pages are not simply parsed and served to the user. While serving pages is the ultimate goal of ASP.NET, the way in which the resultant HTML code is generated is much more sophisticated than in ASP and involves many more objects.

A page request passes through a pipeline of objects that process the HTTP content and, at the end of the chain, produce some HTML code for the browser. The entry point in this pipeline is the *HttpRuntime* class. The ASP.NET runtime activates the HTTP pipeline by creating a new instance of the *HttpRuntime* class and then calling the method *ProcessRequest*.

The *HttpRuntime* Object

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 context for the request and initializes a specialized text writer object in which the HTML code will be accumulated. A context is given by an instance of the *HttpContext* class, which encapsulates all HTTP-specific information about the request. The text writer is an instance of the *HttpWriter* class and is the object that actually buffers text sent out through the *Response* object.

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.

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, 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 any is present, and the creation of the assembly that represents the actual page requested. An *HttpApplication* object is used to process a single page request at a time; multiple objects are used to serve simultaneous requests for the same page.

> **Note** ASP.NET global.asax files are dynamically compiled the first time any page or Web service is requested in the virtual directory. This happens even before the target page or Web service is compiled. ASP.NET pages and Web services within that Web application are subsequently linked to the resulting global.asax compiled class when they are in turn compiled.

The *HttpApplication* Object

HttpApplication is a global.asax-derived object that the ASP.NET worker process uses to handle HTTP requests that hit a particular virtual directory. A particular *HttpApplication* instance is responsible for managing the entire lifetime of the request it is assigned to, and the instance of *HttpApplication* can be reused only after the request has been completed. The *HttpApplication* class defines the methods, properties, and events common to all application objects—.aspx pages, user controls, Web services, and HTTP handlers—within an ASP.NET application.

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.

HTTP modules represent the managed counterpart of ISAPI filters and will be covered in greater detail in Chapter 23.

The *HttpApplication* object determines the type of object that represents the resource being requested—typically, an ASP.NET page. It then uses a handler factory object to either instantiate the type from an existing assembly or dynamically create the assembly and then an instance of the type. 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.

> **Caution** Although the name sounds vaguely evocative of the intrinsic ASP *Application* object, the ASP.NET *HttpApplication* has nothing to do with it. The ASP *Application* object is fully supported in ASP.NET, but it maps to an object of type *HttpApplicationState*. However, the *HttpApplication* object has a property named *Application*, which returns just the ASP.NET counterpart of the ASP intrinsic application-state object.

The Handler Factory

The *HttpApplication* determines the type of object that must handle the request, and it 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 an undocumented class named *PageHandlerFactory*.

> **Note** The *HttpApplication* object determines the proper handler for the request and creates an instance of that class. To find the appropriate handler, it uses the information in the *<httpHandlers>* section of the machine.config file. The section lists all the currently registered handlers for the application.

The page handler factory is responsible for either finding the assembly that contains the page class or dynamically creating an ad hoc assembly. The *System.Web* namespace defines a few handler factory classes. These are listed in Table 2-3.

Table 2-3 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 a directory on the Web server and used until the corresponding resource file is modified.

So the page handler factory creates an instance of an object that represents the particular page requested. This object inherits from the *System.Web.UI.Page* class, which in turn implements the *IHttpHandler* interface. The page object is built as an instance of a dynamically generated class based on the source code embedded in the .aspx file. 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 *ProcessRequest* method on the page object. This call causes the page to execute the user-defined code and generate the HTML text for the browser.

Figure 2-5 illustrates the overall HTTP pipeline architecture.

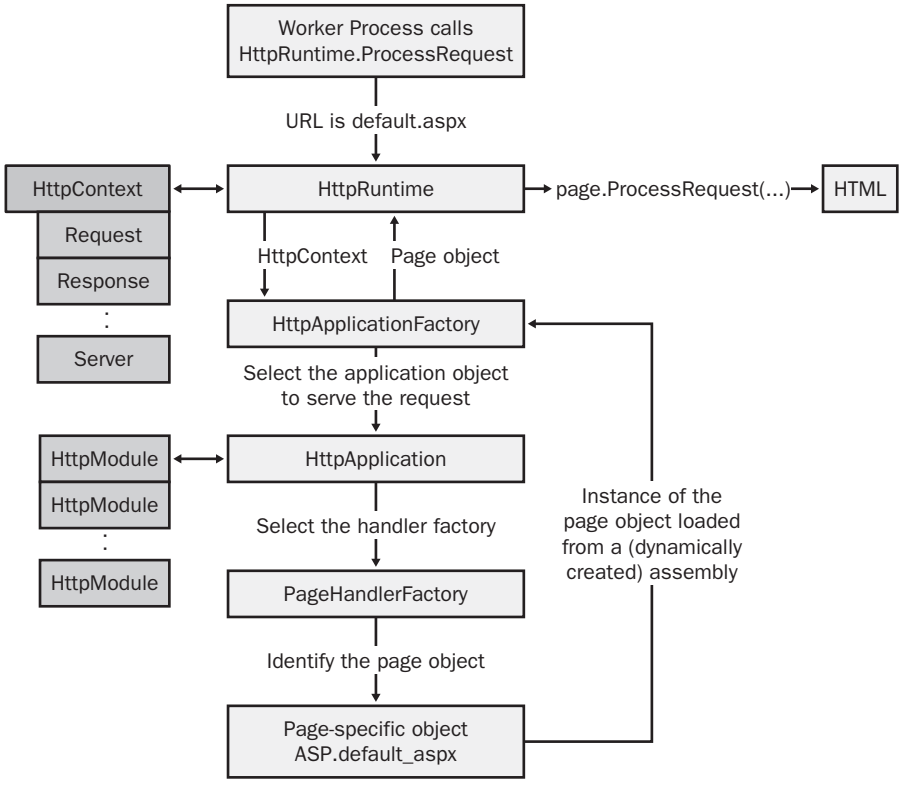

Figure 2-5 The HTTP pipeline processing for a page.

The ASP.NET Page Factory Object

Let's examine in detail how the .aspx page is converted into a class and compiled into an assembly. Generating an assembly for a particular .aspx resource is a two-step process. First, the source code for the class is created by merging the content of the <*script*> section with the code-behind file, if any. Second, the dynamically generated class is compiled into an assembly and cached in a well-known directory.

Locating the Assembly for the Page

Assemblies generated for ASP.NET pages are cached in the Temporary ASP.NET Files folder. The path for version 1.1 of the .NET Framework is as follows.

```
%SystemRoot%\Microsoft.NET\Framework\v1.1.4322\Temporary ASP.NET Files
```

Of course, the directory depends on the version of the .NET Framework you installed. The directory path for version 1.0 of the .NET Framework includes a subdirectory named v1.0.3705. The Temporary ASP.NET Files folder

has one child directory for each application ever executed. The name of the subfolder matches the name of the virtual directory of the application. Pages that run from the Web server's root folder are grouped under the Root subfolder.

Page-specific assemblies are cached in a subdirectory placed a couple levels down the virtual directory folder. The names of these child directories are fairly hard to make sense of. Names are the result of a hash algorithm based on some randomized factor along with the application name. A typical path is shown in the following listing. The last two directories (in boldface) have fake but realistic names.

```
\Framework
    \v1.1.4322
        \Temporary ASP.NET Files
            \MyWebApp
                \3678b103
                    \e60405c7
```

Regardless of the actual algorithm implemented to determine the folder names, from within an ASP.NET application the full folder path is retrieved using the following, pretty simple, code:

```
string tempAspNetDir = HttpRuntime.CodegenDir;
```

So much for the location of the dynamic assembly. So how does the ASP.NET runtime determine the assembly name for a particular .aspx page? The assembly folder contains a few XML files with a particular naming convention:

```
[filename].[hashcode].xml
```

If the page is named, say, default.aspx, the corresponding XML file can be named like this:

```
default.aspx.2cf84ad4.xml
```

The XML file is created when the page is compiled. This is the typical content of this XML file:

```
<preserve assem="c5gaxkyh" type="ASP.Default_aspx"
    hash="fffffeda266fd5f7">
    <filedep name="C:\Inetpub\wwwroot\MyWebApp\Default.aspx" />
</preserve>
```

I'll say more about the schema of the file in a moment. For now, it will suffice to look at the *assem* attribute. The attribute value is just the name of the assembly (without extension) created to execute the default.aspx page. Figure 2-6 provides a view of the folder.

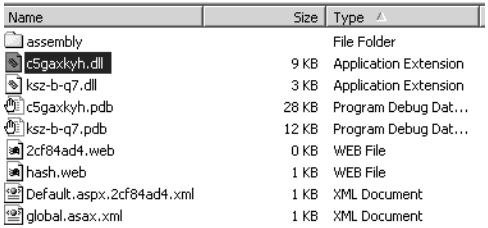

Figure 2-6 Temporary ASP.NET Files: a view of interiors.

The file c5gaxkyh.dll is the assembly that represents the default.aspx page. The other assembly is the compiled version of the global.asax file. (If not specified, a standard global.asax file is used.) The objects defined in these assemblies can be viewed with any class browser tool, including Microsoft IL Disassembler, ILDASM.exe.

> **Important** If the *Debug* attribute of the *@Page* directive is set to *true*, in the same folder as the assembly, you'll also find the source code of the page class. The source code is a Microsoft Visual Basic .NET or C# file according to the value of the *Language* attribute. The name of the source file is *assembly_name.0.ext*, where *assembly_name* denotes the name (without extension) of the assembly and *ext* denotes the language extension. For example, c5gaxkyh.0.cs is the C# source file for default.aspx.

Detecting Page Changes

As mentioned earlier, the dynamically compiled assembly is cached and used to serve any future request for the page. However, changes made to an .aspx file will automatically invalidate the assembly, which will be recompiled to serve the next request. The link between the assembly and the source .aspx file is kept in the XML file we mentioned a bit earlier. Let's recall it:

```
<preserve assem="c5gaxkyh" type="ASP.Default_aspx" hash="fffffeda266fd5f7">
    <filedep name="C:\Inetpub\wwwroot\MyWebApp\Default.aspx" />
</preserve>
```

The *name* attribute of the <*filedep*> node contains just the full path of the file associated with the assembly whose name is stored in the *assem* attribute of the <*preserve*> node. The *type* attribute, on the other hand, contains the name of the class that renders the .aspx file in the assembly. The actual object running when, say, default.aspx is served is an instance of a class named *ASP.Default_aspx*.

Based on the Win32 file notification change system, this ASP.NET feature enables developers to quickly build applications with a minimum of process overhead. Users, in fact, can "just hit Save" to cause code changes to immediately take effect within the application. In addition to this development-oriented benefit, deployment of applications is greatly enhanced by this feature, as you can simply deploy a new version of the page that overwrites the old one.

When a page is changed, it's recompiled as a single assembly, or as part of an existing assembly, and reloaded. ASP.NET ensures that the next user will be served the new page outfit by the new assembly. Current users, on the other hand, will continue viewing the old page served by the old assembly. The two assemblies are given different (because they are randomly generated) names and therefore can happily live side by side in the same folder as well as be loaded in the same AppDomain. Because that was so much fun, let's drill down a little more into this topic.

How ASP.NET Replaces Page Assemblies

When a new assembly is created for a page as the effect of an update, ASP.NET verifies whether the old assembly can be deleted. If the assembly contains only that page class, ASP.NET attempts to delete the assembly. Often, though, it finds the file loaded and locked, and the deletion fails. In this case, the old assembly is renamed by adding a .DELETE extension. (All executables loaded in Windows can be renamed at any time, but they cannot be deleted until they are released.) Renaming an assembly in use is no big deal in this case because the image of the executable is already loaded in memory and there will be no need to reload it later. The file, in fact, is destined for deletion. Note that .DELETE files are cleaned up when the directory is next accessed in *sweep* mode, so to speak. The directory, in fact, is not scavenged each time it is accessed but only when the application is restarted or an application file (global.asax or web.config) changes.

Each ASP.NET application is allowed a maximum number of recompiles (with 15 as the default) before the whole application is restarted. The threshold value is set in the machine.config file. If the latest compilation exceeds the threshold, the AppDomain is unloaded and the application is restarted. Bear in mind that the atomic unit of code you can unload in the CLR is the AppDomain, not the assembly. Put another way, you can't unload a single assembly without unloading the whole AppDomain. As a result, when a page is recompiled, the old version stays in memory until the AppDomain is unloaded because either the Web application exceeded its limit of recompiles or the ASP.NET worker process is taking up too much memory.

Getting ASP.NET Runtime Information

The page runtimeinfo.aspx in the book samples displays some run-time information about the running application and AppDomains. Obtained from properties of the *HttpRuntime* class, the information includes the ID, path, and virtual path of the current AppDomain, plus useful paths such as the directory in which ASP.NET generates dynamic assemblies (*CodegenDir*), the machine.config path, and the Bin directory of the application (*BinDirectory*).

The runtimeinfo.aspx page also lists all the assemblies currently loaded in the AppDomain. The sample page needs 12 system assemblies, including those specific to the application—global.asax and the page class. This number increases each time you save the .aspx file because after a page update, a new assembly is loaded but the old one is not unloaded until the whole AppDomain is unloaded. If you save the .aspx file several times (by just opening the file and hitting Ctrl+S), you see that after 15 recompiles the AppDomain ID changes and the number of loaded assemblies reverts back to 12 (or whatever it was). Figure 2-7 shows the result of this exercise.

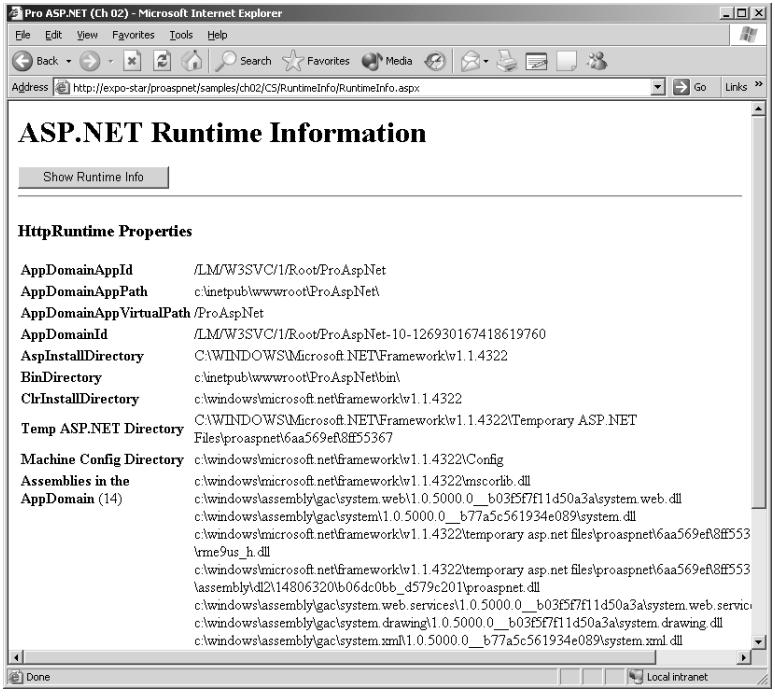

Figure 2-7 Runtimeinfo.aspx shows ASP.NET runtime information.

Batch Compilation

Compiling an ASP.NET page takes a while. So even though you pay this price only once, you might encounter situations in which you decide it's best to happily avoid that. Unfortunately, as of version 1.1, ASP.NET lacks a tool (or a built-in mechanism) to scan the entire tree of a Web application and do a precompilation of all pages. However, you can always request each page of a site before the site goes live or, better yet, have an ad hoc application do it.

In effect, since version 1.0, ASP.NET has supported batch compilation, but this support takes place only at run time. ASP.NET attempts to batch into a single compiled assembly as many pages as possible without exceeding the configured maximum batch size. Furthermore, batch compilation groups pages by language, and it doesn't group in the same assembly pages that reside in different directories.

Just as with many other aspects of ASP.NET, batch compilation is highly configurable and is a critical factor for overall site performance. Fine-tuning the related parameters in the <*compilation*> section of the machine.config file is important and should save you from having and loading 1000 different assemblies for as many pages or from having a single huge assembly with 1000 classes inside. Notice, though, that the problem here is not only with the size and the number of the assemblies but also with the time needed to recompile the assemblies in case of updates.

> **Note** The next version of ASP.NET should fully support precompilation of Web sites and provide an offline tool to precompile an entire Web site as well as build manager classes to programmatically control the process. In addition, ASP.NET 2.0 will extend dynamic compilation to class files (typically, code-behind .cs or .vb files) that currently must be compiled either manually or through Visual Studio.

How ASP.NET Creates a Class for the Page

An ASP.NET page executes as an instance of a type that, by default, inherits from *System.Web.UI.Page*. The page handler factory creates the source code for this class by putting a parser to work on the content of the physical .aspx file. The parser produces a class written with the language the developer specified. The class belongs to the *ASP* namespace and is given a file-specific name. Typically, it is the name and the extension of the file with the dot (.) replaced by an underscore (_). If the page is default.aspx, the class name will be *ASP.Default_aspx*. You can check the truthfulness of this statement with the following simple code:

```
void Page_Load(object sender, EventArgs e)
{
    Response.Write(sender.ToString());
}
```

As mentioned earlier, when the page runs with the *Debug* attribute set to *true*, the ASP.NET runtime does not delete the source code used to create the assembly. Let's have a quick look at the key parts of the source code generated. (Complete sources are included in this book's sample code.)

> **Important** As hinted earlier, the base page class is *System.Web.UI.Page* only by default. In either machine.config or web.config, you can change the base class by using the *pageBaseType* attribute of the *<pages>* element. The same thing can be done for user controls. We'll return to this point later in this chapter and in Chapter 12.

Reviewing the Class Source Code

For a better understanding of the code generated by ASP.NET, let's first quickly review the starting point—the .aspx source code:

```
<%@ Page Language="C#" Debug="true" %>
<%@ Import Namespace="System.IO" %>

<script runat="server">
private void Page_Load(object sender, EventArgs e)  {
    TheSourceFile.Text = HttpRuntime.CodegenDir;
}
private void MakeUpper(object sender, EventArgs e)  {
        string buf = TheString.Value;
        TheResult.InnerText = buf.ToUpper();
}
</script>

<html>
<head><title>Pro ASP.NET (Ch 02)</title></head>
<body>
    <h1>Sample Page</h1>
    <form runat="server">
        <asp:Label runat="server" id="TheSourceFile" /><hr>
        <input runat="server" type="text" id="TheString" />
        <input runat="server" type="submit" id="TheButton"
            value="Uppercase..." onserverclick="MakeUpper" /><br>
        <span id="TheResult" runat="server"></span>
    </form>
</body>
</html>
```

The following listing shows the source code that ASP.NET generates to process the preceding page. The text in boldface type indicates code extracted from the .aspx file:

```
namespace ASP
{
    using System;
    ⋮
    using ASP;
    using System.IO;

    public class Default_aspx : Page, IRequiresSessionState
    {
        private static int __autoHandlers;
        protected Label TheSourceFile;
        protected HtmlInputText TheString;
        protected HtmlInputButton TheButton;
        protected HtmlGenericControl TheResult;
        protected HtmlForm TheAppForm;
        private static bool __initialized = false;
        private static ArrayList __fileDependencies;

        private void Page_Load(object sender, EventArgs e)
        {
            TheSourceFile.Text = HttpRuntime.CodegenDir;
        }
        private void MakeUpper(object sender, EventArgs e)
        {
            string buf = TheString.Value;
            TheResult.InnerText = buf.ToUpper();
        }

        public Default_aspx()
        {
            ArrayList dependencies;
            if (__initialized == false)
            {
                dependencies = new ArrayList();
                dependencies.Add(
                  "c:\\inetpub\\wwwroot\\vdir\\Default.aspx");
                __fileDependencies = dependencies;
                __initialized = true;
            }
            this.Server.ScriptTimeout = 30000000;
        }

        protected override int AutoHandlers {
            get {return __autoHandlers;}
            set {__autoHandlers = value;}
        }

        protected Global_asax ApplicationInstance {
```

```
    get {return (Global_asax)(this.Context.ApplicationInstance));}
}

public override string TemplateSourceDirectory {
    get {return "/vdir";}
}

private Control __BuildControlTheSourceFile() {
    Label __ctrl = new Label();
    this.TheSourceFile = __ctrl;
    __ctrl.ID = "TheSourceFile";
    return __ctrl;
}

private Control __BuildControlTheString() {
    // initialize the TheString control
}

private Control __BuildControlTheButton() {
    // initialize the TheButton control
}

private Control __BuildControlTheResult() {
    // initialize the TheResult control
}

private Control __BuildControlTheAppForm() {
    HtmlForm __ctrl = new HtmlForm();
    this.TheAppForm = __ctrl;
    __ctrl.ID = "TheAppForm";
    IParserAccessor __parser = (IParserAccessor) __ctrl;
    this.__BuildControlTheSourceFile();
    __parser.AddParsedSubObject(this.TheSourceFile);
    __parser.AddParsedSubObject(new LiteralControl("<hr>"));
    this.__BuildControlTheString();
    __parser.AddParsedSubObject(this.TheString);
    this.__BuildControlTheButton();
    __parser.AddParsedSubObject(this.TheButton);
    __parser.AddParsedSubObject(new LiteralControl("<br>"));
    this.__BuildControlTheResult();
    __parser.AddParsedSubObject(this.TheResult);
    return __ctrl;
}

private void __BuildControlTree(Control __ctrl)
{
    IParserAccessor __parser = (IParserAccessor)__ctrl;
    __parser.AddParsedSubObject(
        new LiteralControl("<html>…</h1>"));
    this.__BuildControlTheAppForm();
    __parser.AddParsedSubObject(this.TheAppForm);
    __parser.AddParsedSubObject(new LiteralControl("…</html>"));
}
```

```
        protected override void FrameworkInitialize() {
            this.__BuildControlTree(this);
            this.FileDependencies = __fileDependencies;
            this.EnableViewStateMac = true;
this.Request.ValidateInput();
        }

        public override int GetTypeHashCode() {
            return 2003216705;
        }
    }
}
```

Important As you can see, portions of the source code in the .aspx file are used to generate a new class in the specified language. Just because the inline code in a page will be glued together in a class doesn't mean you can use multiple languages to develop an ASP.NET page. The .NET platform is language-neutral but, unfortunately, .NET compilers are not capable of cross-language compilation!

In addition, for ASP.NET pages, the language declared in the *@Page* directive must match the language of the inline code. The *Language* attribute, in fact, is used to determine the language in which the class is to be created. Finally, the source code is generated using the classes of the language's Code Document Object Model (CodeDOM). CodeDOM can be used to create and retrieve instances of code generators and code compilers. Code generators can be used to generate code in a particular language, and code compilers can be used to compile code into assemblies. Not all .NET languages provide such classes, and this is why not all languages can be used to develop ASP.NET applications. For example, the CodeDOM for J# has been added only in version 1.1 of the .NET Framework, but there is a J# redistributable that adds this functionality to version 1.0.

All the controls in the page marked with the *runat* attribute are rendered as protected properties of the type that corresponds to the tag. Those controls are instantiated and initialized in the various *__BuildControlXXX* methods. The initialization is done using the attributes specified in the .aspx page. The build method for the form adds child-parsed objects to the *HtmlForm* instance. This means that all the parent-child relationships between the controls within the form are registered. The *__BuildControlTree* method ensures that all controls in the whole page are correctly registered with the page object.

All the members defined in the *<script>* block are copied verbatim as members of the new class with the same level of visibility you declared. The base class for the dynamically generated source is *Page* unless the code-behind approach is used. In that case, the base class is just the code-behind class. We'll return to this later in "The Code-Behind Technique" section.

Processing the Page Request

The *HttpRuntime* object governs the HTTP pipeline in which a page request is transformed into a living instance of a *Page*-derived class. The *HttpRuntime* object causes the page to generate its HTML output by calling the *ProcessRequest* method on the *Page*-derived class that comes out of the pipeline. *ProcessRequest* is a method defined on the *IHttpHandler* interface that the *Page* class implements.

The Page Life Cycle

Within the base implementation of *ProcessRequest*, the *Page* class first calls the *FrameworkInitialize* method, which, as seen in the source code examined a moment ago, builds the controls tree for the page. Next, *ProcessRequest* makes the page go through various phases: initialization, loading of view-state information and postback data, loading of the page's user code, and execution of postback server-side events. After that, the page enters rendering mode: the updated view state is collected, and the HTML code is generated and sent to the output console. Finally, the page is unloaded and the request is considered completely served.

During the various phases, the page fires a few events that Web controls and user-defined code can intercept and handle. Some of these events are specific to controls and can't be handled at the level of the .aspx code. In theory, a page that wants to handle a certain event should explicitly register an appropriate handler. However, for backward compatibility with the Visual Basic programming style, ASP.NET also supports a form of implicit event hooking. By default, the page tries to match method names and events and considers the method a handler for the event. For example, a method named *Page_Load* is the handler for the page's *Load* event. This behavior is controlled by the *AutoEventWireup* attribute on the *@Page* directive. If the attribute is set to *false*, any applications that want to handle an event need to connect explicitly to the page event. The following code shows how to proceed from within a page class:

```
// C# code
this.Load += new EventHandler(this.MyPageLoad);
' VB code
AddHandler Load, AddressOf Me.MyPageLoad
```

By proceeding this way, you will enable the page to get a slight performance boost by not having to do the extra work of matching names and events. Visual Studio .NET disables the *AutoEventWireup* attribute.

Page-Related Events

To handle a page-related event, an ASP.NET page can either hook up the event (for example, *Load*) or, in a derived class, override the corresponding method—for example, *OnLoad*. The second approach provides for greater flexibility because you can decide whether and when to call the base method, which, in the end, fires the event to the user-code. Let's review in detail the various phases in the page life cycle:

- **Page initialization** At this stage, the page is called to initialize all the settings needed during the lifetime of the incoming page request. The event you need to hook up from an ASP.NET page is *Init*. The method to use to override in a derived class (for example, code-behind) is *OnInit*.
- **Loading the view state** During this phase, the previously saved state of the page—the *ViewState* property—is restored. The restored value for *ViewState* comes from a hidden field that ASP.NET spookily inserts in the HTML code (more on this later). There is no event associated with this phase. You can override the default implementation of the *LoadViewState* method of the *Page* only by deriving a new class.
- **Loading postback data** The page loads all incoming <*form*> data cached in *Request* and updates page and control properties accordingly. For example, suppose a page contained a list of check boxes. In *ViewState*, the page stores the selected items when the page is generated. Next, the user works with check boxes on the client, selects a different set of items, and posts back. In the previous phase, the server-side list of check boxes has been restored to the state it was in the last time it was processed on the server. In this stage, the list of check boxes is updated to reflect client-side changes. No user event is associated with this stage.
- **Loading the user code** At this point, server controls in the page tree are created and initialized, the state is restored, and form controls reflect client-side data. The page is ready to execute any initialization code that has to do with the logic and behavior of the page. The event you need to hook up from an ASP.NET page is *Load*. The method to use to override in a derived class (for example, code-behind) is *OnLoad*.

- **Send postback change notifications** Raise change events in response to state changes between the current and previous postbacks. Any further change entered by the *Load* event handler is also taken into account to determine any difference. This notification is sent if, for example, the aforementioned list of check boxes is changed on the client.
- **Handle postback events** Execute the .aspx code associated with the client-side event that caused the postback. For example, if you click a Submit button, the page posts back and, at this stage, executes the code bound to the *onclick* event of the button. This phase represents the core of the event model of ASP.NET.
- **Prerendering** Perform any updates before the output is rendered. Any changes made to the state of the control in the prerender phase can be saved, while changes made in the rendering phase are lost. The event you need to hook up from an ASP.NET page is *PreRender*. The method to use to override in a derived class (for example, code-behind) is *OnPreRender*.
- **Saving the view state** During this phase, the page serializes the content of the *ViewState* property to a string, which will then be appended to the HTML page as a hidden field. No user event is associated with this phase. You can override the default implementation of the *SaveViewState* method of the *Page* only by deriving a new class.
- **Page rendering** Generate the HTML output to be rendered to the client. The default implementation can be customized by overriding the *Render* method. No user event is associated with this phase.
- **Page unload** Perform any final cleanup before the *Page* object is released. Typical operations are closing files and database connections. The event you need to hook up from an ASP.NET page is *Unload*. The method to use to override in a derived class (for example, code-behind) is *OnUnload*. The page unload notification arrives when the page is unloaded. The *Dispose* method of the *Page* provides the last possibility for the page to perform final clean up before it is released from memory. To customize the behavior of *Dispose*, you need to derive a new class. *Dispose* is called immediately after the *OnUnload* method returns.

The Event 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 (JSP) and LAMP (Linux plus Apache plus MySQL plus either Perl, Python, or PHP as the programming language), 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.

The illusion of continuity is created by the view-state feature of ASP.NET pages and is based on some assumptions on how the page is designed and works. Also server-side Web controls play a remarkable role. In brief, before rendering its contents to HTML, the page encodes and stuffs into a 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* attribute.

The Single Form Model

Admittedly, for programmers whose experience is with ASP, the single form model of ASP.NET can be difficult to understand. Where's the *Action* property of the form? Why can't I redirect to a particular page when a form is submitted? These are common questions, frequently asked on forums and newsgroups.

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.

> **Note** 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.

Auto-Reentrant Web Pages

Let's consider a simple ASP.NET page and see the HTML code that ASP.NET generates for it:

```
<% @Page Language="C#" %>
<script runat="server">
private void MakeUpper(object sender, EventArgs e)
{
    string buf = TheString.Text;
    TheResult.Text = buf.ToUpper();
}
</script>

<html>
<body>
<form runat="server" id="TheForm">
    <asp:textbox runat="server" id="TheString" />
    <asp:button runat="server" text="Convert..." onclick="MakeUpper" /><hr>
    <asp:label runat="server" id="TheResult" />
</form>
</body>
</html>
```

As shown in Figure 2-8, the page lets the user type some text in an input field and then posts all the data to the server for further processing. If you point your browser to this page, the actual HTML code being displayed is the following. The text in boldface is the page's view state:

```
<html>
<body>
<form name="TheForm" method="post" action="MyForm.aspx" id="TheForm">
    <input type="hidden"
        name="__VIEWSTATE"
        value="dDwtMTM3NjQ2NjY2NTs7PrH3U/xuqPTNI63IlLw5THHvPFUf" />
    <input name="TheString" type="text" id="TheString" />
    <input type="submit" name="_ctl0" value="Convert..." /><hr>
    <span id="TheResult"></span>
</form>
</body>
</html>
```

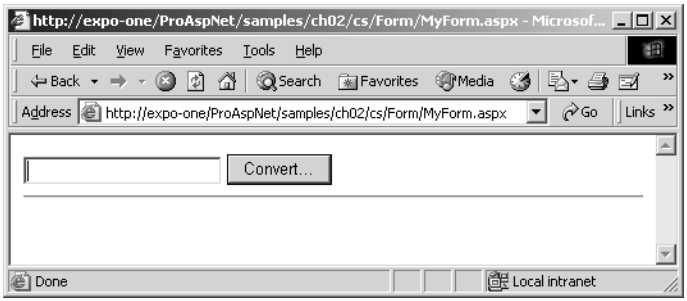

Figure 2-8 The sample ASP.NET page in action.

Note A question I often get at conferences and classes (but rarely a publicly asked question) concerns the use of hidden fields in ASP.NET. The question is typically, "Should I really use hidden fields also in ASP.NET?" Hidden fields have a bad reputation among ASP developers because they appear to be a quick fix and a sort of dirty trick. In some way, developers tend to think that they use hidden fields because they're unable to find a better solution. A similar feeling was common among Windows SDK programmers regarding the use of global variables or temporary files.

Developers seem to fear using hidden fields in the dazzling new object-oriented world of ASP.NET. Well, nothing really prohibits the use of hidden fields in ASP.NET applications and using them is in no way shameful, as long as HTTP remains the underlying protocol. The real point is even more positive.

Using hidden fields in ASP.NET allows you to create more useful Web pages—for example, the view-state mechanism. In light of this, hidden fields are especially recommended when you need to pass information to be consumed through client-side scripts.

The server-side <*form*> tag is rendered as a plain old HTML form in which the *Name* attribute matches the ID property of the *HtmlForm* control, the *Method* attribute defaults to POST and *Action* is automatically set with the URL of the same page. The form contains as many HTML elements as there are server controls in the .aspx source. The form also includes any static text or HTML tags declared in the page layout. The view state is rendered as a hidden field named __VIEWSTATE and filled with Base64-encoded text.

The View State of the Page

The contents of the view state are determined by the contents of the *ViewState* property of the page and its controls. The *ViewState* property is a collection of name/value pairs that both pages and controls fill with data they determine should be preserved across page requests. The various collections are merged together to form a unique object graph, which is then encoded as a Base64 string. Seen from the client side, the __VIEWSTATE hidden field is just a more or less overwhelming sequence of characters. The more state information you save in the *ViewState* property, the bigger your page becomes. Other security and performance concerns apply to the page view state, but we'll cover them in greater detail in Chapter 14. Note that for a real-world page, the size of the view state can easily be more than 10 KB of data, which is an extra burden of 10 KB transmitted over the network and simply ignored on the client.

Note The implementation of the view state in ASP.NET raises hidden fields from the rank of a dirty trick to the higher and more noble role of an internal system feature. Today, lots of ASP developers use hidden fields to persist small pieces of page-specific information. This information is, in most cases, written as plain text and serialized with homemade schemes. The view state implementation revamps that old trick and, more, exposes persisted information in an extremely handy, object-based fashion.

Some developers are afraid of having their own state information at the mercy of the client. That this is the actual situation cannot be denied; but that this automatically qualifies as a security hole has yet to be proven. However, if you were among the many who used to cache plain text in hidden fields, the ASP.NET view state represents a quantum leap! In addition, there are some extra security enforcements that can be applied to the view state. We'll review those in Chapter 14.

Postback Events

A postback event is a page request that results from a client action. Typically, when the user clicks on a submit button, like the Convert button in Figure 2-8, a postback event occurs. In this case, the HTML form posts to the same .aspx page and, as normally happens with HTTP POST commands, the contents of the various input fields in the form are packed in the body of the request.

On the server, the code in charge of processing the request first gets an instance of the class that represents the .aspx page and then goes through the steps described earlier in the "Page-Related Events" section.

State Restoration

As the first step of the page processing, the HTTP runtime builds the tree of controls for the page and sets the *IsPostBack* property. *IsPostBack* is a Boolean property that equals *true* if the page is being processed after a post from the same page—a postback event.

After initialization, the HTTP runtime instructs the page to deserialize the view-state string and transform it into an instance of the *StateBag* class. *StateBag* implements a dictionary that is populated with the name/value pairs stored and encoded in the view-state string. Once the *StateBag* object has been completely set up, user code can start playing with its contents. From a user-code perspective, however, this timing means it can't do anything before the *OnLoad* event fires. Before *OnLoad* is fired, though, another step needs to be accomplished.

At this point, the various controls active on the page have been restored to the state stored in the view state. Unless the view state has been sapiently and maliciously corrupted on the client (an extremely remote possibility, if not completely impossible), the state of the objects is the state that existed when that page was rendered to the client. It is not yet the state the user set with client-side input actions such as checking radio buttons or typing text. The next step updates the state of the various controls to reflect any client action. This final step ends the state restoration phase, and the page is now ready to process user code. Fire the *OnLoad* event.

Handling the Server-Side Event

The goal of a postback event is executing some server-side code to respond to user input. How can the page know what method it has to execute? The post is always the result of a form submission. However, the submit action can be accomplished in two ways: if the user clicks a control or if the user takes some action that causes some script to run. Let's examine the first case.

The HTML syntax dictates that the ID of the button clicked be inserted in the post data. On the server, the ASP.NET page digs out this information in its attempt to locate an element whose name matches one of its server controls. If a match is found, ASP.NET verifies that the control implements *IPostBackEventHandler*. This is enough to guarantee that the button clicked on the server was a submit button. Generally, it means the control has some code to execute upon postback. At this point, the page raises the postback event on the control and has the control execute some code. To raise the postback event on the control, the ASP.NET page invokes the control's *RaisePostBackEvent* method—one of the members of the *IPostBackEventHandler* interface. For a submit button, this causes the invocation of the method associated with the *onclick* property.

If some script code can post the page back to the server, the author of the script is responsible for letting ASP.NET know about the sender of the event. To understand how to accomplish this, let's see what happens when you use a Web control such as the *LinkButton*—a button rendered as a hyperlink:

```
<asp:linkbutton runat="server" id="TheLink"
    text="Convert..." onclick="MakeUpper" />
```

The HTML code for the page changes slightly. A JavaScript function and a couple of extra hidden fields are added. The two hidden fields are __EVENTTARGET and __EVENTARGUMENT. They are empty when the page is rendered to the browser and contain event data when the page is posted back. In particular, __EVENTTARGET contains the name of the control that caused the postback. The __EVENTARGUMENT field contains any argument that might be useful to carry out the call. A short piece of JavaScript code sets the two fields and submits the form:

```
<!--
<script language="javascript">
function __doPostBack(eventTarget, eventArgument) {
var theform;
if (window.navigator.appName.toLowerCase().indexOf("netscape") > -1) {
theform = document.forms["TheForm"];
}
else {
theform = document.TheForm;
}
theform.__EVENTTARGET.value = eventTarget.split("$").join(":");
theform.__EVENTARGUMENT.value = eventArgument;
theform.submit();
}
// -->
</script>
```

But what's the link between the *__doPostBack* function and the *LinkButton* object? Let's look at the HTML generated for the *LinkButton*:

```
<a href="javascript:__doPostBack('TheLink','')">Convert...</a>
```

Quite simply, any click on the anchor is resolved by executing the *__doPostBack* function.

If you want to provide a completely custom mechanism to post the page back, just borrow the preceding code and make sure the form is submitted with the __EVENTTARGET field set to the name of the sender of the event. (For example, you could have the page post back when the mouse hovers over a certain image.)

To find the sender control that will handle the server-side event, ASP.NET looks at the __EVENTTARGET hidden field if no match is found between its controls and the content of the *Request.Form* collection—that is, if the previous case (the submit button) does not apply.

Rendering Back the Page

After executing the server-side code, the page begins its rendering phase. At first, the page fires the *PreRender* event; it then serializes the content of the *ViewState* object—that is, the current state of the various controls active on the page—to a Base64 string. Next, the HTML code is generated and that Base64 string becomes the value of the __VIEWSTATE hidden field. And so on for the next round-trip.

The *Page* Class

In the .NET Framework, the *Page* class provides the basic behavior for all objects that an ASP.NET application builds 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. It's useful to also look at the functionalities of the *Page* class from an interface perspective, as in Figure 2-9.

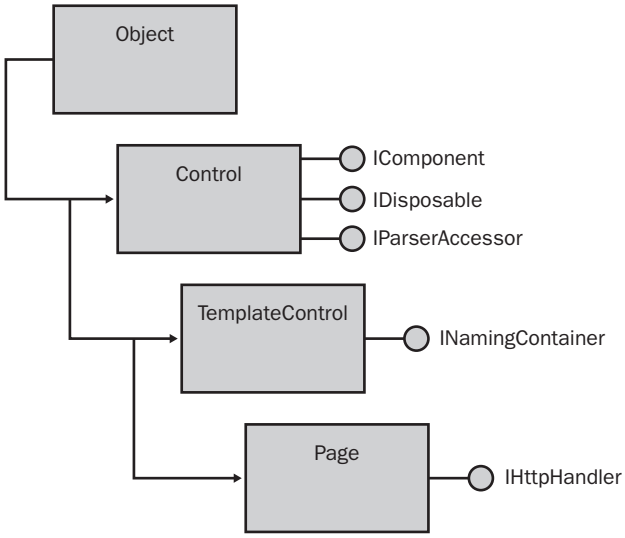

Figure 2-9 The hierarchy of classes from which *Page* inherits.

From the parent *Control* class, *Page* inherits the base behavior for interacting with all other components in the CLR—the *IComponent* interface—and for releasing unmanaged resources (the *IDisposable* interface). In addition, the *IParserAccessor* interface groups the methods that ASP.NET controls must implement to recognize parsed child elements. The *IParserAccessor* interface plays a key role in the implementation of the control tree in the dynamically generated source code for requested ASP.NET pages. (Actually, the *IParserAccessor* interface consists of the single *AddParsedSubObject* method we saw largely used in the code shown in "Reviewing the Class Source Code" section.)

> **Note** The *Control* class also implements the *IDataBindingsAccessor* interface. This interface, though, is not involved with the run-time behavior of server controls. The interface allows access to the collection of data-binding expressions on a control at design time.

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*, and for user controls.)

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* Object

The properties of the *Page* object can be classified in three distinct groups: intrinsic objects, worker properties, and page-specific properties. Tables 2-4, 2-5, and 2-6 enumerate and describe them.

Table 2-4 ASP.NET Intrinsic Objects in the *Page* Class

Property	Description
Application	Instance of the *HttpApplicationState* class, represents the state of the application. 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 decay and expiration.
Request	Instance of the *HttpRequest* class, represents the current HTTP request. Functionally equivalent to the ASP intrinsic *Request* object.
Response	Instance of the *HttpResponse* class, sends HTTP response data to the client. Functionally equivalent to the ASP intrinsic *Response* object.

Table 2-4 ASP.NET Intrinsic Objects in the *Page* Class

Property	Description
Server	Instance of the *HttpServerUtility* class, provides helper methods for processing Web requests. Functionally equivalent to the ASP intrinsic *Server* object.
Session	Instance of the *HttpSessionState* class, manages user-specific data. 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 13; *Application*, *Cache*, and *Session* in Chapter 14; and *User* and security will be the subject of Chapter 15.

Table 2-5 Worker Properties of the *Page* Class

Property	Description
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
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
NamingContainer	Returns *null*
Page	Returns the current *Page* object
Parent	Returns *null*
TemplateSourceDirectory	Gets the virtual directory of the page
Validators	Returns the collection of all validation controls contained in the page
ViewStateUserKey	String property used to assign an identifier to the view state variable for individual users. This trick is a line of defense against one-click attacks. *The property is 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).

A special note deserves the *ViewStateUserKey* property that has been added with version 1.1 of the .NET Framework. A common use for the user key would be to stuff user-specific information that will then be used to hash the contents of the view state along with other information. (See Chapter 14.) 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 couldn't construct a valid view state for your user account unless he could also authenticate as you. That way, you have another barrier against on-click attacks. This technique, though, might not be really 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.

Table 2-6 Page-Specific Properties of the *Page* Class

Property	Description
ClientID	Always returns the empty string.
ClientTarget	Set to the empty string by default, allows you to specify the type of browser the HTML should comply with. Setting this property disables automatic detection of browser capabilities.
EnableViewState	Gets or sets 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.
ID	Always returns the empty string.
SmartNavigation	Gets or sets a value indicating whether smart navigation is enabled. Smart navigation exploits a bunch of browser-specific capabilities to enhance the user's experience with the page. The feature works only with Internet Explorer 5.5 and newer versions. You can also enable or disable this feature through the *SmartNavigation* attribute of the *@Page* directive.
UniqueID	Always returns the empty string.
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* Object

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 HTML for the page (as shown in Table 2-7); others are helper methods to build the page and manage the constituent controls (as shown in Table 2-8). Finally, a third group collects all the methods that have to do with client-side scripting (as shown in Table 2-9).

Table 2-7 Methods for HTML 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.
RegisterRequiresPostBack	Registers the specified control with the page so that the control will receive a post-back handling notice. In other words, the page will call the *LoadPostData* method of registered controls. *LoadPostData* requires the implementation of the *IPostBackDataHandler* interface.
RegisterRequiresRaiseEvent	Registers the specified control to handle an incoming postback event. The control must implement the *IPostBackEventHandler* interface.
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 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 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.

Table 2-8 Worker 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 like 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.
GetTypeHashCode	Retrieves the hash code generated by *ASP.xxx_aspx* page objects at run time. (See the source code of the sample page class we discussed earlier in the "Reviewing the Class Source Code" section.) In the base *Page* class, the method implementation simply returns 0; significant numbers are returned by classes used for actual pages.
HasControls	Determines whether the page contains any child controls.
LoadControl	Compiles and loads a user control from a .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 a .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.
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.

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 can be omitted from the strings you pass directly to *ParseControl*.

Table 2-9 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 2-9 Script-Related Methods

Method	Description
GetPostBackClientEvent	Calls into *GetPostBackEventReference*.
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.
IsStartupScriptRegistered	Determines whether the specified client startup script is registered with the page.
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');`

(continued)

Table 2-9 Script-Related Methods *(continued)*

Method	Description
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.
RegisterHiddenField	Use this method to automatically register a hidden field on the page.
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.
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.

Many methods listed in Table 2-9 let you emit script in the client page—either JavaScript or VBScript. 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. The same pattern applies to hidden fields and *ECMAScript* arrays.

> **Note** JavaScript is the script language that virtually any available browser supports. For this reason, some of the methods in Table 2-9 default to JavaScript. However, when you register a script block, nothing really prevents you from using VBScript as long as you set the language attribute of the *<script>* tag accordingly. On the other hand, methods such as *RegisterOnSubmitStatement* and *RegisterArrayDeclaration* can be used only with JavaScript code.

Events of the *Page* Object

In addition to the events we mentioned earlier, the *Page* class fires a few extra events that are notified during the page life cycle. As Table 2-10 shows, some events are orthogonal to the typical life cycle of a page and are fired as extra-page situations evolve.

Table 2-10 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.
Load	Occurs when the page loads up.
PreRender	Occurs when the page is about to render.
Unload	Occurs when the page is unloaded from memory but not yet disposed.

The Code-Behind Technique

At the highest level of abstraction, an ASP.NET page is made of page layout information and code. The code for the page can be developed in two ways: inline through the <*script*> tag or by using a code-behind class. All inline code must be written in the same language declared for the page and will be copied verbatim in the source code of the dynamically generated class that represents the .aspx resource in execution.

A code-behind class inherits from *Page*, either directly or indirectly. As such, the code-behind class encapsulates in an object-oriented fashion all the functionalities you're building in the page—event handlers, method and property overrides, and new members. Visual Studio .NET takes advantage of code-behind classes to implement an efficient and effective code editor. As mentioned in Chapter 1, though, the code-behind syntax used by Visual Studio .NET is a bit unusual and to some extent undocumented. As a matter of fact, though, the resultant page architecture is even more efficient in practice.

In this section, we'll start with a review of the code-behind scheme of Visual Studio .NET, and then we'll move on to more general considerations of code-behind.

Code and Layout Separation

Code-behind is an architectural feature designed to clearly separate code and layout in an ASP.NET page. To achieve this, though, it exploits one of the cornerstones of object-oriented programming (OOP)—class inheritance. Just the capability to inherit pages from existing pages leads to even more attractive and compelling uses of the code-behind feature, such as visual inheritance and Web site master pages.

Code-Behind in Visual Studio .NET

When you create a new page in an ASP.NET project, Visual Studio .NET creates an .aspx file and a ghost file with the same name as the .aspx plus a .cs or .vb extension, depending on the project language. The .aspx file contains control declarations and other layout information. The .cs (or .vb) file contains all the code for the page. The page is seen as a class derived from *Page* with overrides and new members as appropriate.

When you build the Visual Studio .NET project, the page class is compiled into the project's assembly and the run-time link between the .aspx file and its base class is given by the value of the *Inherits* attribute in the *@Page* directive. The design-time link between the .aspx file and its code-behind class is given by the value of the *CodeBehind* attribute in the *@Page* directive. The *CodeBehind* attribute is blissfully ignored by the ASP.NET runtime and makes sense only to Visual Studio .NET.

The separation between code and layout is achieved by using two distinct files—the .aspx for the layout and the .cs (or whatever language you use) for the code—that the IDE manages transparently.

Code-Behind in Action at Run Time

For example, if the ASP.NET runtime receives a request for a page.aspx resource, the runtime creates an instance of a dynamically generated class named *ASP.Page_aspx*. By default, this class inherits from *Page* or from the class you specify in the configuration. The base class can be changed on a per-page basis also through the *Inherits* attribute in the *@Page* directive. The *Inherits* attribute contains only the fully qualified name of the class from which *ASP.Page_aspx* must derive. It doesn't say anything about the actual location of the class.

The ASP.NET runtime locates the base class assembly using the usual probing algorithm for missing assemblies. One of the directories scanned is the virtual folder's Bin directory, which is the location in which Visual Studio .NET places the project assemblies at development time.

For the code-behind feature to work, the *Inherits* attribute is always necessary, as it is the only supported way to inform the ASP.NET runtime to derive *ASP.Page_aspx* from a different and more specific page class. (The code-behind class itself must necessarily derive from *Page* or from a class that, in turn, inherits from *Page*.)

You can also deploy a code-behind class as plain source code. In this case, you use *Inherits* to indicate the name of the class and the *Src* attribute to specify the URL from which the class can be loaded and compiled, as shown in the following line of code:

```
<%@ Page Language="C#" Inherits="MyBasePageClass" Src="MyBasePage.cs" %>
```

The file pointed to by the *Src* attribute is automatically compiled on demand and recompiled in case of updates. It goes without saying that the class file used must match the language of the page.

> **Tip** Code-behind classes and inline code can happily cohabit within the same Web Forms page. Interestingly, this fact alone opens up the possibility of using multiple languages to develop the page. In fact, you could use one language to develop the code-behind class and then link it to the page using the *Inherits* attribute. You're responsible for compiling the class and deploying the class through an assembly. Next, you'd use the other language to set the *Language* attribute and develop inline code for the page. Finally, you should note that this programming style is not supported by Visual Studio .NET. If you like it, use Web Matrix or other third-party commercial products.

If you decide to use the *Src* attribute, you have to deploy the source code of the class, thus making the logic of your solution available on the Web server. On the other hand, this approach simplifies the maintenance of the Web site because you could modify the file (or better yet, a copy of it) on the Web server. Otherwise, you have to modify the source, rebuild the project yourself, and then deploy it again to the Web server.

> **Note** Visual Studio .NET does not support code-behind if it is implemented through the *Src* attribute. In this case, you can either resort to your own code editor or, better yet, download and install the free Web Matrix editor.

Moving to Code-Behind

A practical advantage of the code-behind approach is that it allows two programmers or programming teams to simultaneously work on the development of a Web page. After the structure of a page is defined in terms of the number, names, and types of constituent controls, HTML designers can work on the appearance of the page, determining the position, colors, and fonts of the controls; they can even determine the necessity for client-side scripting. At the same time, programmers can develop the code and event handlers required for the page to work properly and fulfill users' expectations.

Any existing ASP.NET page can migrate to code-behind with very few changes. The first step is writing a class that inherits, either directly or indirectly, from *Page*. Next, you add to the class, as a protected or public member, any control declared in the page layout that has the *runat* attribute set to *server*. Also, convert any global variable or procedure in the <*script*> section of the original ASP.NET page into a new member of the class with the scope and visibility you want.

Be sure to choose the right control type that will lie behind an ASP.NET or HTML tag. For example, an <*asp:textbox runat="server"*> tag maps to a *TextBox* control, whereas an equivalent <*input type=text runat="server"*> tag maps to *HtmlInputText*.

> **Caution** Because the actual page class being processed will inherit from the code-behind class, make sure you don't declare any control members as *private*. In this case, in fact, the actual derived class won't be able to access the control because of the protection level. Use the *public* modifier—or better yet, the *protected* modifier—instead.

Once you've accomplished this, you're almost done. A subtle design issue you might want to consider is the following. Typically, your Web page will have controls that fire server-side events—for example, a submit button:

```
<asp:button runat="server" id="TheButton" text="Submit" />
```

To make the control usable, you need to bind it to some code. You can do this in two ways. You could either add an *onclick* attribute to the tag or programmatically define an event handler in the code-behind class. In the former case, you end up having the following code:

```
<asp:button runat="server" id="TheButton" text="Submit"
    onclick="OnSubmitDocument" />
```

There's nothing bad about this code, at least in terms of overall functionality. However, it makes the ideal of neat separation between code and layout a bit less neat. The page layout, in fact, contains a reference to a piece of code. Among other things, this means HTML designers must remember to change the attribute should they decide to rename the handler.

From a design standpoint, a much better approach is to go the programmatic route and register the event handler with code. In C#, you would accomplish this with the following code:

```
override protected void OnInit(EventArgs e)
{
    this.TheButton.Click += new System.EventHandler(this.OnSubmitDocument);
}
```

In both cases, the *OnSubmitDocument* method should be defined in the code-behind class.

Page Inheritance

It's hard to say whether code-behind inspired the separation between code and layout or whether things happened the other way around—that is, code-behind was invented as a way to implement code and layout separation. Rather than concern ourselves with how it all developed, let's look at code-behind from another, more architectural, perspective.

At the end of the day, code-behind is just a fancy name for the ASP.NET feature that lets you explicitly set the base class for the dynamically created page class. And I think this feature alone is much more important to developers than the separation between code and layout. To achieve code separation, you don't need to know about the architecture of code-behind—just use Visual Studio .NET and you're done.

The capability of changing the base class of dynamic pages opens a new world of possibilities that can be described as the Web counterpart of the visual form inheritance we have in Windows Forms applications. For Windows Forms, visual inheritance indicates the .NET Framework ability to let you create new forms by inheriting from previously created forms. The .NET Framework provides you with a base *Form* class that you extend in user projects by adding child controls. Each application form is definitely an instance of some user-defined, *Form*-based class.

You normally inherit your custom forms always from the base class *Form*, but a smarter approach is possible. For example, to make all your applications share the same look and feel, you could create a form class inheriting from *Form*, add some features and controls, and compile the resultant class into a new assembly. Suppose you call this form class *AcmeForm*. Next, whenever you need to create a new Windows Forms application with

the same characteristics, you simply derive the main window from *AcmeForm* and get a ready-to-use and feature-rich form window. Although it's impressive to show, this feature involves nothing that qualifies as rocket science; it's just the offspring of class inheritance. As such, it can be applied to ASP.NET pages as well. Code-behind and the *Inherits* attribute of the *@Page* directive are the underpinnings of this important feature.

A Server-Side Abstraction for Client-Side Events

To demonstrate a real-world application of the code-behind technique, let's consider the level of server-side abstraction that ASP.NET provides for client-side events such as setting the input focus. ASP.NET doesn't provide, and it is not supposed to provide, a rich object model for client-side scripting and page decoration. In other words, you have to write any JavaScript code you might need and attach it to the page using the *RegisterXXX* methods of the *Page* class.

That said, though, consider that almost every Web page displays some HTML control that would work much better if you could set the input focus. For example, a page that contains a list of text boxes for data entry would look much more user-friendly if the input focus was automatically placed on the first field. After all, it's not such a hard task; a few lines of JavaScript code would do the job quickly and effectively. You need to write a small script to set the focus and call it from the body's *onload* method. The following HTML page shows just that:

```
<html>
<script language="javascript">
function SetInputFocus() {
    if(document.__theForm["fname"] != null)
        document.__theForm["fname"].focus();
}
</script>

<body onload="SetInputFocus()">
<form name="__theForm">
    <input type="text" value="Joe" name="fname">
    <input type="text" value="Users" name="lname">
</form>
</body>
</html>
```

Although it's easy to understand and implement, the solution is still boring to code (or even to cut and paste) for each page of the application. The code-behind technique comes to the rescue. The idea is that you derive a new class from *Page* and make sure it contains the needed script code and runs it upon startup.

A Better Page Class

The following listing demonstrates an alternative *Page* class. It inherits from the *System.Web.UI.Page* class and exposes a *Focus* property that clients can set with the name of the HTML control with the input focus:

```
using System;
using System.Web.UI;
using System.Text;

namespace ProAspNet.CS.Ch02
{
    public class Page : System.Web.UI.Page
    {
        public Page()
        {
        }

        // Internals
        private const string FUNCTION_NAME = "__setFocus";
        private const string SCRIPT_NAME = "__inputFocusHandler";

        // Gets/Sets the name of the HTML control with the input focus
        public string Focus;

        // Registers client-side script
        protected override void OnPreRender(EventArgs e)
        {
            base.OnPreRender(e);
            AddStartupScript();
        }

        // Create the script strings
        private void AddStartupScript()
        {
            // Find the ID of the ASP.NET form
            string formName = "";
            foreach(Control ctl in this.Controls)
                if (ctl is HtmlForm)  {
                    formName = "\"" + ctl.UniqueID + "\"";
                    break;
                }

            // Add the script to declare the function
            StringBuilder sb = new StringBuilder("");
            sb.Append("\n<script language=javascript>\n");
            sb.Append("function ");
            sb.Append(FUNCTION_NAME);
            sb.Append("(ctl) {");
            sb.Append("\n  if (document.forms[");
            sb.Append(formName);
            sb.Append("][ctl] != null)  {");
            sb.Append("\n      document.forms[");
```

```
            sb.Append(formName);
            sb.Append("][ctl].focus();");
            sb.Append("\n  }");
            sb.Append("\n}\n");

            // Add the script to call the function
            sb.Append(FUNCTION_NAME);
            sb.Append("('");
            sb.Append(Focus);
            sb.Append("');\n<");
            sb.Append("/");
            sb.Append("script>");

            // Register the script (names are CASE-SENSITIVE)
            if (!IsStartupScriptRegistered(SCRIPT_NAME))
                RegisterStartupScript(SCRIPT_NAME, sb.ToString());
        }
    }
}
```

In the *PreRender* event, the page registers a startup script. If the server form is named *theForm* and the *Focus* property is set to *user*, the following script will be generated:

```
<script language=javascript>
function __setFocus(ctl) {
    if (document.forms["theForm"][ctl] != null)  {
        document.forms["theForm"][ctl].focus();
    }
}
__setFocus('user');
</script>
```

The *__setFocus* JavaScript function is first declared and then invoked with a particular argument. The script is registered using the *RegisterStartupScript* method. A startup script is placed exactly before closing the *<form>* tag and executed when the browser has finished with the rest of the form—in this sense, it is called a startup script and is functionally equivalent to setting the *onload* attribute of the page body.

Programmatically Set the Input Focus

Because the *ProAspNet.CS.Ch02.Page* class inherits from *Page*, you can use it wherever a page class is acceptable. To make sure that ASP.NET builds an .aspx page just from the *ProAspNet.CS.Ch02.Page* class, use this type with the *Inherits* attribute of the page directive. Of course, you need to have an assembly for the class in one of the paths the ASP.NET runtime can reach—typically the virtual folder's Bin directory. The following page automatically sets the input focus to the field named *user*. The code in boldface type is all you have to add to an existing page to support the feature, which sets a new base class and the input focus:

```
<%@ Page Language="C#" Inherits="ProAspNet.CS.Ch02.Page" %>

<script runat="server">
    public void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
            this.Focus = "user";
    }
</script>

<html>
<head><title>Pro ASP.NET (Ch 02)</title></head>
<body>
    <h2>Log on to the System</h2>
    <form runat="server">
        <b>UserName</b><br>
        <asp:textbox runat="server" id="user" /><br>
        <b>Password</b><br>
        <asp:textbox runat="server" id="pswd" /><br>
        <asp:button id="go" runat="server" text="Connect" />
    </form>
</body>
</html>
```

The *IsPostBack* property is set to *true* if the page is being displayed after a postback. If you don't check against *IsPostBack*, the input focus will be set to the control named *user* each time the page is accessed; otherwise, it will be set only the first time the page is accessed in the session.

Changing the Default Page Base Type

As mentioned earlier, by default a page inherits from the *Page* class or from any other code-behind class specified in the *@Page* directive. If you need all pages in an application (or in a certain folder) to inherit from a common and user-defined class, you can override the base page type from the *System.Web.UI.Page* class to whatever class you want. Obviously, the new page class must in turn inherit from *System.Web.UI.Page*.

For example, to make all the pages in a Web application inherit the focus feature we discussed earlier, create a web.config file in the application's root folder and give it the following content:

```
<configuration>
    <system.web>
        <pages pageBaseType="ProAspNet.CS.Ch02.Page, Page" />
    </system.web>
</configuration>
```

The *pageBaseType* attribute must be set to a valid type name—that is, a comma-separated string formed by a type name and the assembly name. In the preceding code, the specified class is implemented in the page.dll assembly.

Note that the base page could also be changed for all the applications running on a server machine. In this case, you must edit the machine.config file, locate the <*pages*> section, and add the *pageBaseType* attribute. By default, the attribute is initialized implicitly and is not listed in the configuration file. To edit both web.config and machine.config, you need administrator privileges.

> **Note** In the next version of ASP.NET, the *WebControl* class and the *Page* class will provide a nearly identical feature through a method named *SetFocus*. The method takes the ID of a control and caches it internally. When the page is rendered to HTML, ad hoc script code will be generated to set the focus to the control.

Master Pages in ASP.NET

A large number of Web sites these days contain similar-looking pages that share the same header, footer, and perhaps some navigational menus or search forms. What's the recommended approach for reusing code across pages? One possibility is wrapping these user-interface elements in user controls and referencing them in each page. Although the model is extremely powerful and produces highly modular code, when you have hundreds of pages to work with, it soon becomes unmanageable.

An alternative approach entails using code-behind to aim at a kind of visual inheritance akin to that of Windows Forms. In this case, the custom base class programmatically generates user-interface elements such as the page header. In addition, the page class could expose custom objects as programmable entities—for example, a collection for the links to display on the header. To build a new page, you simply set the *Inherits* attribute with the class name. When part of the user interface (UI) is programmatically generated, merging the base elements and the actual content of the content page can be problematic unless absolute positioning is used. In the sample code of this chapter (located at ch02\Inherit), you find an application that solves this issue by forcing the client page to include an <*asp:placeholder*> control with a particular ID just where a particular parent UI element should appear.

In the next version of ASP.NET, the concept of the *master page* will be introduced. A master page is a distinct file referenced at the application level, as well as at the page level, that contains the static layout of the page. Regions that each *derived* page can customize are referenced in the master page with a special placeholder control. A derived page is simply a collection of blocks the run time will use to fill the holes in the master. Master pages in the next version of

ASP.NET are orthogonal to true visual inheritance. The contents of the master, in fact, are merged into the derived page rather than serving as a base class for the derived page.

Conclusion

ASP.NET is a complex technology built on top of a simple and, fortunately, solid and stable Web infrastructure. To provide highly improved performance and a richer programming tool set, 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. ASP.NET anticipates some of the features of IIS 6.0—the new and revolutionary version of the Microsoft Web information services. ASP.NET applications run in a separate worker process (as all applications will do in IIS 6.0); ASP.NET applications support output caching (as IIS 6.0 will do on behalf of all types of Web applications); and finally, the ASP.NET runtime automatically recycles processes to guarantee excellent performance in spite of run-time anomalies, memory leaks, and programming errors. The same feature becomes a system feature in IIS 6.0.

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. The code-behind technique allows you to programmatically change the base class for a given .aspx resource. The net effect of code-behind is the possibility to implement very cool features such as code and layout separation and page inheritance.

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, validation controls, and data-bound controls.

Resources

- HTTP Architecture (*http://www.develop.com/conferences/conferencedotnet/materials/A4.pdf*)
- HTTP Pipeline (*http://www.develop.com/summercamp/conferencedotnet/materials/W2.pdf*)
- Securely Implement Request Processing, Filtering, and Content Redirection with HTTP Pipelines in ASP.NET (*http://msdn.microsoft.com/msdnmag/issues/02/09/HTTPPipelines*)
- Viewstate Optimization Strategies in ASP.NET (*http://www.webreference.com/programming/asp/viewstate*)
- KB306005 Repair IIS Mapping After You Remove and Reinstall IIS
- KB315158 ASP.NET Does Not Work with the Default ASPNET Account on a Domain Controller