Chapter 7

ASP.NET Up Close

In the last six chapters, you have been introduced to a diverse C# CodeBase and some active server page (ASP) code sprinkled here and there. For example, in Chapter 4 we covered an application named *Managed Synergy*, primarily to show you how to apply C#. To do that, we had to throw in some ASP.NET in certain places. In this chapter (and Chapter 8), we discuss ASP.NET in more detail—what it is, how it works, and how to apply its more advanced features.

.NET is about to be released to the world at large. During the last few months of its beta period, its early adoption rate has been staggering. Judging from the feedback on the .NET lists and the size of the community that is growing up around it, .NET looks like it is going to be huge. As we have discussed so far, the .NET framework represents an implementation of several great modern software development ideas that have been a long time coming. These ideas include a Common Type System (CTS), a common runtime environment, the ability to really mix languages within a single application, and the ability to invoke methods over the Internet. These factors contribute to the new software paradigm coming down the road: people-oriented software. .NET comprises many facets, but one of the most important parts of .NET—the one that will almost certainly be a lynchpin in its success—is ASP.NET. ASP.NET ties together all three facets of people-oriented software: universalization, translation, and collaboration. Although a few of the modern Microsoft-based Web sites will probably continue using ISAPI extensions and classic ASP (and perhaps the Active Template Library server [ATL server]), most other Microsoft-based Web sites will be written using ASP.NET.

CONNECTIVE TISSUE

Most of us in the software industry these days agree that the computing world of the future will involve the Internet as the network of choice. How could it be any other way? The Internet is ubiquitous. The dream of the 1980s and 1990s was to get computers within a single office talking to one another. Now, local area networks (LANs) and companywide intranets are commonplace. The vision for the next decade is to get all the computers in the world talking to one another. Now that nearly everyone agrees on the protocol (Hypertext Transfer Protocol [HTTP] or HTTPS) and the bits to send across (Extensible Markup Language [XML] that is formatted using Simple Object Access Protocol [SOAP]), connecting Web sites programmatically can actually happen.

However, although the communication standards and protocols are currently being set up and agreed to, what is really going to kick the .NET vision into high gear is a decent way to implement the standards. At the heart of it all, Web sites are still going to be nodes on the Internet managing HTTP requests and responses. As developers, we need a structured way to intercept and respond to HTTP requests. This is what ASP.NET provides. ASP.NET is going to be one of the most important parts of .NET for bringing people-oriented software to life.

The Road to ASP

So that you can really understand ASP.NET, let's take a quick look at the metamorphosis the Web has gone through during the past few years. The Web really began to be popular around 1995 or so, and in that short time it has been transformed from a set of fancy marketing brochures into sites with full-fledged order-entry systems and other advanced features.

Classic ASP evolved as a way to more easily develop "dynamic" content—that is, to have the content of your Web site change at runtime. The first Web pages up on the Web were just hypertext markup language (HTML) pages that never changed—kind of like electronic brochures. People would sign on to a uniform resource locater (URL) somewhere, and their browsers would simply zoom in on some files in some directory on the remote computer. The browser on the client machine would download the HTML using HTTP and interpret it. Obviously, the next step was to make the content change based on various conditions (such as who is signed on, the content that is available at runtime, and the security context of the client). This is the origin of dynamic content.

The first dynamic Web sites were the result of the common gateway interface (CGI). CGI provides dynamic content by directly processing incoming HTTP requests and issuing responses in a custom process. The CGI process spits out customized HTML based on the requests coming in. One of the problems of CGI was the fact that

each incoming HTTP request got a new process, creating a burden on the server. Creating a new process for each request was pretty expensive.

To reduce the burden of creating a new process for every request (on the Microsoft platform), Microsoft implemented a programming interface they named the Internet Services API (ISAPI). Instead of firing up a new process for each request, ISAPI fires up a new instance of an ISAPI dynamic link library (DLL), customized to spit out specific HTML. Of course, ISAPI DLLs were not the easiest things in the world to write. The next step in the evolution was ASPs (active server pages).

You can think of classic ASP as one big ISAPI DLL that Microsoft has already written for you. Classic ASP accepts and posts responses to HTTP requests. ASP mixes presentation (HTML) with scripting blocks so that you can control the content coming from an ASP page. ASP parses files with *.asp* extensions and does its best to emit the right kind of HTML. HTTP requests and responses are available as well-established objects you can get to easily from within the script blocks. (They are part of the component object model [COM] runtime context.)

CLASSIC ASP VERSUS ASP.NET

Classic ASP goes a long way toward simplifying Web programming. It is often much easier to write some HTML and mingle it with script than it is to write a new DLL from scratch. Still, classic ASP is not without its issues. First, ASP pages are often just an unstructured mass of code. Classic ASP is very much like the early days of Basic programming, when you could get something done quickly but the resulting code was often difficult to follow. The ASP object model has numerous intrinsic, or global, objects. For example, when writing script code to generate the content of an HTTP request, you send the content out to the client using the intrinsic `Response` object. For simple applications in which you can guarantee that only one client is involved in talking to your Web application, this isn't much of a problem. But how many Web applications can guarantee this? (The number is probably very close to zero—if it is not actually zero.) Because of the way ASP is structured with these intrinsic objects, managing the state of clients is a nightmare.

ASP.NET evolves classic ASP. For example, you still have the same intrinsic objects in ASP.NET and you can add scripting wherever you want it on the page. In fact, most ASP pages can easily be brought over and run as ASP.NET pages by renaming them with the *.aspx* extension.

ASP.NET brings a lot of new features to the table. First, ASP.NET pages are compiled into assemblies just like the other components within .NET, providing both performance and security benefits. Second, ASP.NET supports using any .NET language within it. No longer are you confined to using JavaScript or VB Script on your Web pages. You can now use more structured languages. ASP.NET lets you write the executable parts of your pages in C# or full-fledged Visual Basic.

ASP.NET brings with it a whole new programming model incorporating Web forms, server-side controls, data binding, and Web services. Web forms and server-side controls work by tailoring the markup language they spit out to match the client browser attached. Data binding formalizes exchanging data between controls at runtime (such as edit boxes and combo boxes) and data variables within the Web site program. Finally, Web services formalize the process of getting multiple computers talking to each other automatically using XML and HTTP (SOAP). ASP.NET is ripe for creating Web services in which a machine's software can reveal itself to the rest of the world as a SOAP server.

DEEMPHASIZING ISAPI

At the end of the day, writing a Web server is all about processing HTTP requests and delivering responses. ASP.NET is no different; its main purpose in life is to service HTTP requests and responses. In this vein, ASP.NET deemphasizes the ISAPI architecture that has been around for the last six years or so. ISAPI has not disappeared; it is just that ASP.NET prefers the common language runtime (CLR) to Internet information server's (IIS's) ISAPI/ASP architecture wherever possible.

ASP.NET works by dispatching HTTP requests to handler objects. That is, ASP.NET dispatches HTTP requests to those objects implementing a CLR interface named `IHttpHandler`. You can bind uniform resource identifier (URI) paths to classes for handling specific requests (see Chapter 8). InternetBaton (see Chapter 2) also uses this technique. For example, if someone submits a query for some information, you can map that URI to a specific class for handling that query. Incoming URIs that are not mapped to a specific handler class are swallowed by ASP.NET's default handler. Most of the time, the file specified in the URI is a basic ASPX file. Let's take a look at the core of an ASP.NET page.

ASP.NET: A COMMON LANGUAGE RUNTIME CITIZEN

ASP.NET improves on classic ASP by being a full-fledged CLR language. That is, ASP.NET files are compiled rather than simply interpreted like classic ASP. ASP.NET files are compiled on demand based on source code dependencies. ASP.NET compiles ASPX files once and caches the DLL in a well-established directory. If ASP.NET finds source code that is newer than the DLL, it compiles the new source code into a DLL and caches the DLL. ASP.NET is also superior as a deployment environment because you do not need to shut down your site to update the code. When new source code is added to the application, ASP.NET shadow-copies the old DLL to process existing requests using the preexisting DLL so that running processes may continue—you may upgrade the site without shutting it down.

System.Web.UI.Page

The core of every ASP.NET page is the `System.Web.UI.Page` class. Most ASP.NET pages work by extending this class. Nearly the whole user interface package for ASP.NET is wrapped up into `System.Web.UI.Page`. The following listing shows the `System.Web.UI.Page` class:

```
class Page : IHttpHandler, TemplateControl {
class Page : TemplateControl, IHttpHandler
{
    public Page(); // constructor
    public HttpApplicationState Application {get;}

    public Cache Cache {get;}
    public virtual string ClientID {get;}
    public string ClientTarget {get; set;}
    public virtual ControlCollection Controls {get;}
    public virtual bool EnableViewState {get; set;}
    public string ErrorPage {get; set;}
    public virtual string ID {get; set;}
    public bool IsPostBack {get;}
    public bool IsValid {get;}
    public virtual Control NamingContainer {get;}
    public virtual Page Page {get;}
    public virtual Control Parent {get;}
    public HttpRequest Request {get;}
    public HttpResponse Response {get;}
    public HttpServerUtility Server {get;}
    public HttpSessionState Session {virtual get;}
    public ISite Site {get; set;}
    public virtual string TemplateSourceDirectory {get;}
    public TraceContext Trace {get;}
    public virtual string UniqueID {get;}
    public IPrincipal User {get;}
    public ValidatorCollection Validators {get;}
    public virtual bool Visible {get; set;}
    public virtual void DataBind();
    public void DesignerInitialize();
    public virtual void Dispose();
    public virtual bool Equals(
        object obj
    ;
    public virtual Control FindControl(
        string id
    );
    public virtual int GetHashCode();
    public string GetPostBackClientEvent(
        Control control,
        string argument
    );
    public string GetPostBackClientHyperlink(
        Control control,
```

```
    string argument
);
public string GetPostBackEventReference(
    Control control
);
public Type GetType();
public virtual int GetTypeHashCode();
public virtual bool HasControls();
public virtual void InstantiateIn(
    Control control
);
public bool IsClientScriptBlockRegistered(
    string key
);
public bool IsStartupScriptRegistered(
    string key
);
public UserControl LoadControl(
    string virtualPath
);
public ITemplate LoadTemplate(
    string virtualPath
);
public string MapPath(
    string virtualPath
);
public Control ParseControl(
    string content
);
public void RegisterArrayDeclaration(
    string arrayName,
    string arrayValue
);
public virtual void RegisterClientScriptBlock(
    string key,
    string script
);
public virtual void RegisterClientScriptFile(
    string key,
    string language,
    string filename
);
public virtual void RegisterHiddenField(
    string hiddenFieldName,
    string hiddenFieldInitialValue
);
public void RegisterOnSubmitStatement(
    string key,
    string script
);
public void RegisterRequiresPostBack(
    Control control
;
```

```
public virtual void RegisterRequiresRaiseEvent(
   IPostBackEventHandler control
);
public virtual void RegisterStartupScript(
    string key,
    string script
);
public void RenderControl(
    HtmlTextWriter writer
);
public string ResolveUrl(
    string relativeUrl
);
public void SetIntrinsics(
    HttpContext context
);
public void SetRenderMethodDelegate(
    RenderMethod renderMethod
);
public virtual string ToString();
public event EventHandler AbortTransaction;
public event EventHandler CommitTransaction;
public event EventHandler DataBinding;
public event EventHandler Disposed;
public event EventHandler Error;
public event EventHandler Init;
public event EventHandler Load;
public event EventHandler PreRender;
public event EventHandler Unload;
bool Buffer {set;}
protected bool ChildControlsCreated {get; set;}
int CodePage {set;}
string ContentType {set;}
protected HttpContext Context {get;}
string Culture {set;}
protected bool EnableViewStateMac {get; set;}
protected EventHandlerList Events {get;}
ArrayList FileDependencies {set;}
protected bool HasChildViewState {get;}
protected bool IsTrackingViewState {get;}
int LCID {set;}
string ResponseEncoding {set;}
protected bool SmartNavigation {get; set;}
protected virtual bool SupportAutoEvents {get;}
bool TraceEnabled {set;}
TraceMode TraceModeValue {set;}
TransactionOption TransactionMode {set;}
string UICulture {set;}
protected virtual StateBag ViewState {get;}
protected virtual bool ViewStateIgnoresCase {get;}
protected virtual void AddParsedSubObject(
    object obj
);
```

```
protected void AspCompatEndProcessRequest(
    IAsyncResult result
);
protected void BuildProfileTree(
    string parentId,
    bool calcViewState
);
protected void ClearChildViewState();
protected virtual void CreateChildControls();
protected virtual ControlCollection CreateControlCollection();
protected virtual HtmlTextWriter CreateHtmlTextWriter(
   TextWriter tw
);
protected LiteralControl CreateResourceBasedLiteralControl(
   int offset,
   int size,
   bool fAsciiOnly
);
protected virtual NameValueCollection DeterminePostBackMode();
protected virtual void EnsureChildControls();
protected virtual void Finalize();
public virtual Control FindControl(
   string id
);
protected virtual void FrameworkInitialize();
protected virtual void InitOutputCache(
  int duration,
  string varyByHeader,
  string varyByCustom,
  OutputCacheLocation location,
  string varyByParam
);
protected bool IsLiteralContent();
protected virtual object LoadPageStateFromPersistenceMedium();
protected virtual void LoadViewState(
   object savedState
);
protected object MemberwiseClone();
protected virtual void OnAbortTransaction(
   EventArgs e
);
protected virtual bool OnBubbleEvent(
   object source,
   EventArgs args
);
protected virtual void OnCommitTransaction(
   EventArgs e
);
protected virtual void OnDataBinding(
   EventArgs e
);
protected virtual void OnError(
   EventArgs e
```

```
    );
    protected virtual void OnInit(
       EventArgs e
    );
    protected virtual void OnLoad(
       EventArgs e
    );
    protected virtual void OnPreRender(
       EventArgs e
    );
    protected virtual void OnUnload(
       EventArgs e
    );
    protected void RaiseBubbleEvent(
       object source,
       EventArgs args
    );
    protected virtual void RaisePostBackEvent(
       IPostBackEventHandler sourceControl,
       string eventArgument
    );
    protected virtual void Render(
       HtmlTextWriter writer
    );
    protected virtual void RenderChildren(
       HtmlTextWriter writer
    );
    protected virtual void SavePageStateToPersistenceMedium(
       object viewState
    );
    protected virtual object SaveViewState();
    protected void SetStringResourcePointer(
        IntPtr stringResourcePointer,
        int maxResourceOffset
    );
    protected virtual void TrackViewState();
    protected virtual void Validate();
}
```

We discuss some of these properties and methods later in the chapter. For now, we are going to discuss how the class works. It is fairly large and contains pretty much everything needed to manage the HTTP protocol between the client browser and the Web server.

System.Web.UI.Page Fundamentals

`System.Web.UI.Page` implements the core ASP.NET interface named `IHttpHandler`, enabling the `Page` class to receive HTTP requests and deliver responses. The `Page` class is from the `Control` class used for writing server-side controls. (We discuss the `Control` class later in this chapter.) This gives the `Page` class the ability to

manage standard controls (like edit boxes and list boxes) and other user interface elements. The following listing shows an ASP.NET file that spits a bit of HTML out to the browser advertising the CLR type of the page. If the syntax looks a bit bizarre at first, do not worry. Later in this chapter, we discuss the specifics of mixing C# and ASP.NET code.

```
<%@ page language="C#" %>
<%
  // finding out the base type of the Web page
  string s;

  s = this.GetType().BaseType.ToString();

  // Send some text out to the browser
  this.Response.Write("<html>");
  this.Response.Write("<body>");
  this.Response.Write("Based on the following CLR Class:");
  this.Response.Write(s);
  this.Response.Write("</body>");
  this.Response.Write("</html>");
%>
```

Notice that the code in the previous listing is written with C# (indicated by the language directive at the top). When the page shows the type `this`, notice that the page reports the entire namespace: `System.Web.UI.Page`. Figure 7-1 shows the output to a browser.

The `<%  %>` block markers specify a block of code and the language to use to execute the block. Notice that the structured C# code fits in with the rest of the page. In

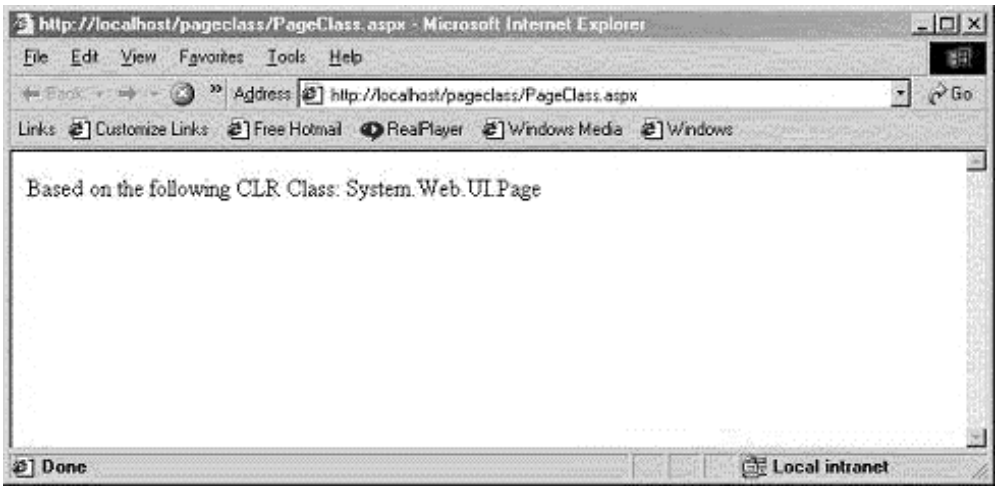

FIGURE 7-1 Output of the .ASPX File after Executing.

addition, notice that the base class is `System.Web.UI.Page`. The `Page` class includes a `Response` object suitable for spitting text out to the browser. The C# code that executes simply asks the page for its CLR type and spits that string out to the browser.

In classic ASP, the `Response` object was called an "intrinsic object"—a global variable. In ASP.NET, the `Response` object is moved into the CLR `Page` class, making it a bit easier to manage. You do not need to push all your markup language through the `Response` object. ASP.NET lets you include markup text as part of the page itself (just as in classic ASP). However, the ASP.NET model makes it easier to customize your output through the `Response` class as shown previously.

ASP.NET Connection Object Model

HTTP is basically a connectionless protocol. That is, the connection between the client and the Web server is not held. Unlike a connection-intensive protocol like distributed component object model (DCOM), clients connect to the server only for the duration of an HTTP request, meaning there needs to be some way of managing the connections within an application. ASP.NET's connection model is based on the `System.Web.Http.Context` class. ASP.NET generates one `HttpContext` object for each request serviced and passes it to HttpHandlers (of which `System.Web.UI.Page` is one). You can always get to the current `HttpContext` object because it is exposed as a static property of `HttpContext: HttpContext.Current`. You can use the `HttpContext` object to get information about the request and its relationship to your application. For example, `HttpContext` is useful for getting information about the HTTP request, as shown in the following listing. In addition, `HttpContext` manages the session state; you can get to the session state information through the `HttpContext` object via the `Session` property.

```
<%@ page language="C#" %>
<%
  string s;

  // Get The current context . . .
  HttpContext httpc = HttpContext.Current;

  // spit out some text . . .
  httpc.Response.Write("URL: ");
  // find out the URL of the current connection
  s = httpc.Request.RawUrl;
  httpc.Response.Output.WriteLine(s);
%>
```

The previous listing shows how to get the current connection state and query it for the current HTTP request and the URL used to get to the page. Figure 7-2 shows the output to the browser.

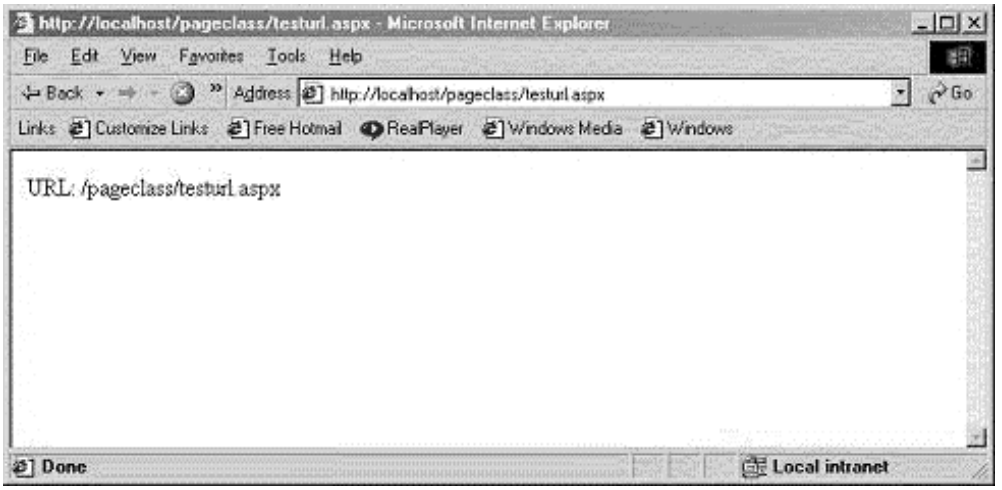

FIGURE 7-2 Output Advertising the Raw URL Used to Surf to the Page.

Mixing ASP.NET and C#

Take another peek at the previous listing. You may notice that the language directive at the top of the file immediately indicates the language of choice, in this case C#. (It could just as easily be Visual Basic.NET or some other CLR-compliant language.) The executable code is mixed in, bracketed by the `<%` and `%>` markers. This is often a convenient way to quickly mix executable CLR code with your HTML (or other markup language). Unfortunately, this technique can also create messy spaghetti code in your Web pages. ASP.NET supports *code behind the page*—mixing ASP code with executable code coming from another file. Basically, you write executable code using C# or Visual Basic (or some other CLR language) and insert it into the ASP page using a special directive.

To separate your executable code from the page layout, write a source code file containing your executable code and place it alongside your ASPX page. The following listing shows a C# class whose job it is to print out the date and time to any client connected to the server.

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

namespace AppliedDotNet
{
   public class CodeBehindPage : Page
   {
      public void PrintDate()
      {
```

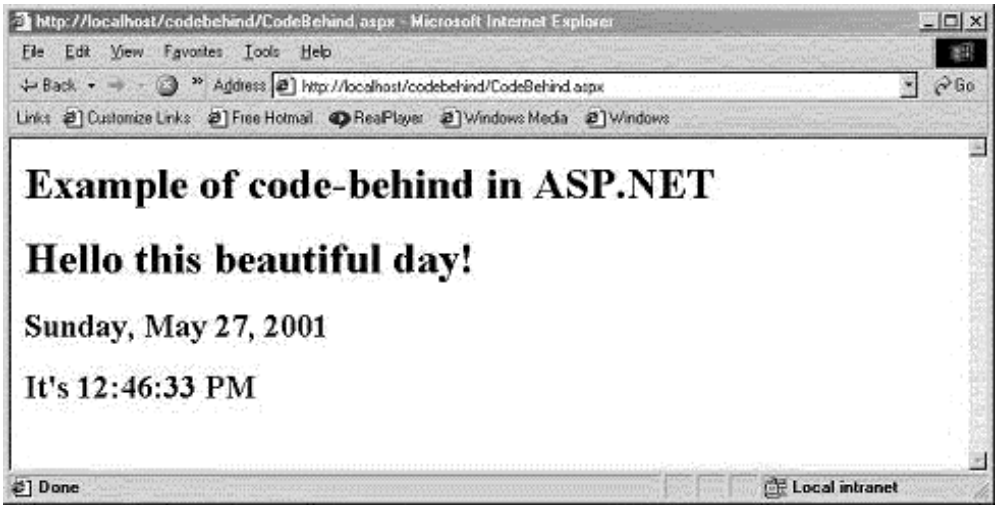

FIGURE 7-3 Output of Listings (pp. 258–260).

```
         String strDate;
         DateTime dateTime = DateTime.Today;
         strDate = dateTime.ToString( "D" );
         Response.Write(strDate);
      }

      public void PrintTime()
      {
         String strTime;
         DateTime dateTime = DateTime.Now;
         strTime = dateTime.ToString( "T" );
         Response.Write(strTime);
    }
   }
}
```

Figure 7-3 shows the output of the previous listing and the next listing (pp. 258–260). Notice that the `CodeBehindPage` class derives from `System.Web.UI.Page`. The CLR `Page` class is the master class of ASP.NET that wraps up all the HTTP request/response functionality. Deriving from the `Page` class brings your ASP.NET page into the runtime. You can use all the functionality that comes with the runtime. The `Page` class also includes a `Response` object that can print formatted text out to the client. These two functions simply figure out the date and time using the CLR-sponsored classes and push the text stream out to the browser. The following listing shows the ASP page that uses the class:

```
 <%@ Page language=C# src=CodeBehindPage.cs
Inherits=AppliedDotNet.CodeBehindPage %>
```

```
<html>
<body>
<h1>Example of code-behind in ASP.NET</h1>
<h1>Hello this beautiful day!</h1>
<h2><% PrintDate(); %></h2>
<h2>It's <%PrintTime();%></h2>
</body> </html>
```

One of the great advantages of ASP.NET is the ability to segregate the executable code from the presentation. The ASP.NET code-behind technique encourages this separation.

ASP.NET Configuration Files

Although ASP.NET still has a few ties to IIS, it tries to keep them to a minimum. ASP.NET adds the ability to configure Web applications through configuration files. After ASP.NET configures an application using the IIS metabase, ASP.NET looks to an application's configuration file to configure the application. (We already looked at configuration files in Chapters 5 and 6.) The ASP.NET configuration files include configuration information specific to ASP.NET.

ASP.NET configuration files are simply XML files named WEB.CONFIG. They can do a number of things, including mapping file extensions to custom handlers, registering custom extension modules, specifying how session state is implemented, and supporting user-defined element types (and corresponding parser objects).

Like normal CLR configuration files, ASP.NET configuration files include sections. The WEB.CONFIG file may contain several sections above and beyond the normal CLR configuration files:

- `<authentication>`: This section manages ASP.NET's authentication settings, including the default authentication code, the HTTP cookie to use for authentication, the password format, and the log-on user name and password.
- `<authorization>`: This section manages ASP.NET authorization settings, including allowing or disallowing access to a specific resource.
- `<httpModules>`: This section adds, removes, or clears HTTP modules within an application.
- `<customErrors>`: This section specifies custom error messages for an ASP.NET application, including specifying the default URL to direct a browser if an error occurs and whether custom errors are enabled, disabled, or shown only to remote clients.
- `<identity>`: This section controls the application identity of the Web application. Settings include specifying whether client impersonation is used on

each request and specifying the user name and password if client impersonation is set to *false*.

- `<pages>`: This section identifies page-specific configuration settings, such as whether the URL resource uses response buffering, whether session state is enabled, whether view state is enabled, whether page events are enabled, the code-behind class the page inherits, and the virtual root location of the client-side script libraries installed by ASP.NET.
- `<processModel>`: This section configures the ASP.NET process model settings on IIS Web server systems. Settings include whether the process model is enabled, the number of minutes until ASP.NET launches a new worker process to take the place of the current one, the idle time-out before ASP.NET shuts down the worker process, the number of minutes allowed for the worker process to shut down gracefully, the number of requests allowed before ASP.NET automatically launches a new worker process to take the place of the current one, the number of requests allowed in the queue before ASP.NET launches a new worker process and reassigns the requests, the maximum allowed memory size, which processors on a multiprocessor server are eligible to run ASP.NET processes, and managing central processing unit (CPU) affinity.
- `<httpHandlers>`: This section maps incoming URL requests to classes implementing `IHttpHandler`. The section adds, removes, or otherwise clears HttpHandlers.
- `<sessionState>`: This section configures the session state `HttpModule`. This includes setting the mode (session state turned off, session state stored in process, session state stored in a remote server, or session state stored on a box running SQL Server), whether to support clients without cookies, time-out period for an idle session, the server name, and the SQL connection string.
- `<globalization>`: This section configures the globalization settings of an application, including such items as the encoding of incoming requests, the encoding of outgoing responses, ASPX file encoding, the default culture to process incoming requests, and the default culture for user interface resources.
- `<compilation>`: This section contains all the compilation settings used by ASP.NET. Settings include enabling debugging, the default language being used on the page, the Visual Basic *Explicit* option, a time-out period for batch compilation, whether to recompile resources before an application restarts, and enabling the Visual Basic *Strict* compile option.
- `<trace>`: This section configures the ASP.NET trace service. Settings in this section include enabling tracing, limiting the number of trace requests to store on the server, and enabling trace output at the end of the page.
- `<browserCaps>`: This section configures the settings for the browser capabilities component.

Each section entry is associated with a specific value within the WEB.CONFIG file. (See Chapters 2 and 8 for examples of using configuration files.)

WEB FORMS

The new coding model for ASP.NET makes it easy to manage presentation (using HTML) and executable code—by either mixing the code into the Web page or writing separate code-behind pages. In addition, ASP.NET provides new server-side control abstractions for managing the right kind of HTML depending on the browser on the other end of the connection (so that you do not have to fiddle with the raw control tags). Combining all of this is the basis of Web forms. The Web forms model of programming makes it feel like you are building a local user interface (as you might in Visual Basic). However, in reality, it is a widely distributed user interface generated almost entirely by pushing HTML from the server to the client browser. We see an example of a Web forms application in the next chapter. The Web forms programming model brings with it the notion of custom server-side controls.

CUSTOM SERVER-SIDE CONTROLS

During the last five years, Web usage has increased phenomenally—both through regular Web surfing and e-commerce. Because of this, one of the most valuable assets consumers can provide for a company is their attention to the company's Web site. Naturally, companies want to create compelling, useful Web sites. Therefore Web users need sophisticated controls with which they can interact with the site.

Most browsers support standard controls such as push buttons and list boxes. However, the standard controls can take you only so far. Over the past few years, the primary way to enrich a Web page was to extend the browser. Several folks have taken a stab at that—notably with Java applets and ActiveX controls (and we focus on ActiveX controls here).

When the user hits a page containing an ActiveX control, the browser proceeds to download a binary file representing the ActiveX control's executable code. The browser calls `CoCreateInstance` on the object, negotiates some interfaces, and renders the control in the browser. Java applets work in a somewhat similar fashion.

Both of these approaches have some specific advantages. The most significant advantage is that you can provide much more natural and intuitive ways for the user and Web site to talk to one another.

A great example of extending the browser is a calendar control. Imagine you are signing on to a Web site set up to handle airline ticket bookings. When it comes down to actually selecting the date for travel, clients needs to give their travel dates to the Web site. It is possible the Web site could expose a text box so that the client can type

in a complete date, such as *October 22, 2001*. A better way would be to have standard combo boxes for selecting the month, day, and year of travel. Imagine that the users can also pull up a calendar to select their dates. Ahhh! That is much more intuitive and even more convenient. They can just point at the date they want and click to select it.

Many sites out there display a calendar for selecting dates. However, invariably, the old standby controls—usually some combo boxes—are often right beside the calendar. What is going on? Why do users need both sets of controls?

Extending the Browser

The problem with extending the browser to enrich the user interface is that the browser has to support a technology to extend it. For example, if you want the browser to use an ActiveX control to interact with the site, that browser needs the infrastructure to support ActiveX controls. The browser requires a handful of COM interfaces for this. Likewise, any Web site using Java applets expects browsers to support the Java runtime.

Unfortunately, it is impossible to make sure every Web surfer out there uses the best browser available. It is difficult even to get people to agree on which browser is the best. It is also unrealistic to expect everybody to always use the very latest version of a particular browser. Most of us do not mind upgrading software every now and then, but there are many people out there who just hate it. The point is, there are diverse client browsers out there, and nobody seems to be able to enforce a standard.

It would not be so bad if the problem ended with browsers. However, during the last couple of years, numerous different devices—such as Internet cell phones, Windows CE machines, and handheld devices—are now able to browse Web sites. Now it is even more difficult to count on a specific kind of UI on the client end. If you make Java applets or ActiveX controls integral to your Web site, you cannot reach certain clients.

ASP.NET's answer to this dilemma is to move to the server the responsibility of rendering and managing custom controls.

Server-Side Rendering

In the Web client world, HTML is the common denominator. Almost every browser understands how to parse and display HTML. If you cannot enrich the user interface by extending the browser, why not have the server generate the correct kind of HTML to give a customized appearance to the user? These controls are called *Web forms* controls. Microsoft provides an established framework and protocol for generating a sophisticated user interface from the server.

Web forms controls are reusable in that they are independent of the Web pages using them and can be created, modified, and maintained separately from the pages.

The public fields, properties, and methods of a Web forms control can be accessed programmatically by the containing control or page. In addition, Web forms controls are hierarchical and can inherit from, contain, and extend other controls. In short, Web forms help "componentize" Web development (i.e., break apart Web user interfaces into small modularized pieces) while simultaneously making it easier to reach a much more diverse range of client devices.

Basically, a Web forms control is a component designed to work within the ASP.NET page framework. It's a DLL written in a .NET CLR language that has the proper hooks expected by the ASP.NET runtime. As ASP.NET parses a Web page, the ASP.NET runtime may encounter one of these controls. When ASP.NET does encounter a control, the ASP.NET runtime creates and initializes it. Like ActiveX controls, ASP.NET server-side custom controls expose properties, methods, and events. However, remember that these components now live on the server (and are not downloaded to the client on demand).

Control Life Cycle

A custom server-side control is much like a finite state machine. It has an established life cycle and set of states in which it can exist. For example, the control can be in a loading state, may be responding to a postback, or may be shutting down. The `Control` class provides the methods and properties for managing a page execution life cycle, including viewing state management, postback data handling, postback event handling, and output rendering. Implementing a server-side control means deriving from the `Control` class and overriding certain methods.

Knowing a page's execution life cycle is very useful for understanding how server-side controls work. As a page is loaded, the ASP.NET runtime parses the page and generates a `Page` class to execute. The `Page` class (which inherits from the `Control` class) instantiates and populates a tree of server control instances. Once the tree is created, the `Page` class begins an execution sequence that enables both ASP.NET page code and server controls to participate in the request processing and rendering of the page.

Following is a rundown of the individual control life cycle. The communication protocol for a server-side control is divided between calls to established well-known methods and various postback processing.

- Responding to a page request, the ASP.NET page first calls the control's `Init` function. Here's where any initial setup happens.
- The control now has an opportunity to set up its view state. View state information is not available during the control's `Init` call, so the control provides a function named `LoadViewState` explicitly for this purpose. A control

maintains its view state in a `Control.State` collection. The State methods allow you to programmatically restore internal state settings between page views.

- A page experiences something called a postback every once in a while. In addition to being activated as a result of a navigation request, ASP.NET pages can be activated as a result of a postback from a previously rendered instance of the same page on the client (for example, a registration form with multiple text fields that must be resubmitted and validated). Think of a postback as a chance to refresh. Controls have the opportunity to process data during a postback. That is, a control has the opportunity to process any incoming form data and update its object models and internal state appropriately. The visual and internal states of the control can automatically remain in sync. Postbacks are handled by the `IPostBackDataHandler` interface.
- Once a page has been activated, a control wants to know when it is being loaded. Controls have the opportunity to override the `OnLoad` method to perform any actions common to each incoming page request (such as setting up a database query or updating a timer on a page).
- A control may have to notify the client that data has changed. This is done during the postback change notification phase of the control. Controls fire appropriate change notification events during this phase in response to form and control value changes that occurred on the client between the previous and current postbacks to a page.
- Controls handle client actions that cause a postback during the postback event processing phase of the control. For example, a button server control could handle a postback from the client in response to being clicked by a user and then raise an appropriate `OnClick` event on the server.
- Controls have the opportunity to perform any last-minute update operations that must take place immediately before page/control state is saved and output rendered by overriding the control's `PreRender` function.
- A control has the opportunity to save its view state by overriding its `SaveViewState` method. A control's state information is automatically transferred from the `Control.State` collection into a string object immediately after the `SaveViewState` stage of execution. To prepare for this, controls can override the `SaveViewState` event to modify the state collection.
- Probably the most important function within a control is `Render`. Controls use the `Render` class to generate HTML output to the browser client. When it is time for an ASP.NET page to render itself, ASP.NET walks the entire list of controls instantiated on the pages, asking each one to render itself through the `Render` method.
- Controls need a chance to clean up after a page is unloaded. Controls override the `Dispose` method to perform any final cleanup work.

Reasons to Use a Custom Server-Side Control

We considered a specific scenario for using a custom server-side control. However, there are several other compelling reasons for using one. In addition to making the sophisticated user interface of your Web application workable for any number of client situations, you may want to use controls to partition your application into code and content. ASP.NET's programming model lets you separate HTML from executable code. By separating the presentation part and the logical part of your application, both parts can be developed in parallel and independently.

Custom server-side controls represent a great opportunity for encapsulating reusable code. Server-side controls can be designed to be completely self-contained so that a page developer can just drop an HTML tag onto a page.

Finally, custom server-side controls simplify the application programming model. By encapsulating functionality in a custom control, page developers can use that functionality by setting control properties and responding to events. ASP.NET custom server-side controls effectively handle the issue of getting a sophisticated user interface to work in a variety of settings.

WEB SERVICES AND ASP.NET

The coupe de grace brought by .NET (and ASP.NET especially) is support for programmable Web sites. In addition to simplifying the user interface aspect of Web development, ASP.NET's other huge contribution to the .NET platform is making it easy to produce Web services (which we have been discussing throughout this book).

During the short history of the Internet, there has always been a human interacting with Web sites, which has been a big boon to commerce in general. The Internet and the Web provide a whole new medium for research, information finding, advertising, and sales. However, this is only the tip of the iceberg.

There is a whole world of untapped resources, meaning that there is a whole world of problems to be solved—particularly those related to getting computers on the Web to talk to one another without human intervention and regardless of the type of computer platform being used. Sites with programmable services are called *Web services* in .NET parlance.

A Web service provides remote access to server functionality. For a number of years, those in both the COM camp and the Common Object Request Broker Architecture (CORBA) camp felt their particular component technology would revolutionize the Internet. The catch is that neither COM nor CORBA mixes very well with firewalls. The other problem is that neither protocol is ubiquitous between any two enterprises. Therefore although both these models are great for building software systems for the enterprise, they do not work between enterprises. Being able to pro-

grammatically contact another business and talk to it through the Internet has been stymied until now.

Although COM and CORBA are not viable Internet protocols, there is another protocol that everyone agrees on and is ubiquitous on the Internet—HTTP. In addition, XML has emerged as the de facto format for transmission. Together, XML and HTTP make up SOAP. SOAP makes writing programmatic Web sites possible.

In the .NET world, businesses will use Web services to expose programmatic interfaces to their data and business logic. Other businesses can then use those interfaces to communicate programmatically with the original business. Web services use HTTP and XML messaging to move data across firewalls—something that could not be done until recently. Because Web services are not tied to a particular component technology or object-calling convention, it does not matter which languages are used to write the programs and components. Microsoft puts the focus on interoperability, which will enable businesses of all natures to hook up through the Internet.

The idea of programmable Web sites is straightforward enough. Obviously, people have been getting information into Web sites (using browser interfaces). Web services allow computers to do the same thing. You just have to write your Web page correctly.

Web Methods and ASP.NET

When writing a Web forms application, the class you use derives from `System.Web.UI.Page`. When you write a Web service, the class you use derives from `System.Web.Services.WebService`. To add functionality to your Web service, your class includes methods marked with the `WebMethod` attribute. The beauty of ASP.NET is that it takes care of correctly mapping incoming HTTP requests onto calls on your object. (Technically, you can get by without using the `WebService` class. However, using `WebService` does give your application access to all of ASP.NET.)

Calling Web services from the client side is simple as well because the runtime provides proxy classes that wrap up the calls on the client side. A client-side proxy manages generating the right HTTP request to the server and parsing the response from the server.

Service Description Language and ASP.NET

Clients know the shape of the methods available on a Web service by reading its service description language (SDL). ASP.NET clients get a copy of a Web service's SDL by requesting the service's file (FOO.ASMX) and passing the SDL query string. At that point, ASP.NET uses reflection to generate an SDL file. The client code then uses the SDL to create a client-side proxy.

Invoking Web Methods

In addition to supporting Web service methods, ASP.NET supports invoking Web methods by one of three means: HTTP GET requests with parameters encoded within a query string, HTTP POST requests with parameters from input controls, and through a SOAP proxy (generated from a Web page's SDL).

OPTIMIZATIONS: ASP.NET CACHING

ASP.NET supports caching to reduce round trips to the server. ASP supports two forms of caching: output caching and data caching. Output caching improves server throughput. Data caching improves the performance on the client end.

Output Caching

When a Web page is accessed and a request is made, the server has to spend a bit of time producing the output. When a Web page's content does not change very much, it makes sense to cache the output so that the server does not have to regenerate it. This output caching is turned off by default, and Web pages need to turn it on specifically using the `OutputCache` directive. The `OutputCache` directive also controls the duration for which the output is cached in seconds.

Whenever an initial HTTP GET request is made for a cached page, the output for that page is cached. If an HTTP GET or HEAD request is made for the same page, ASP.NET satisfies the request from the contents of the cache (until the cached output expires). Very often, Web applications use query strings to look up data in a database and generate output. ASP.NET stores the query string, and the output cache compares query strings to verify a request's identity. HTTP POST requests are never cached—they are always generated explicitly.

Data Caching

In any distributed system, one of the most important optimizations you can make is to reduce the number of round trips between the client and the server. When developing Web pages, the fewer round trips your browser clients need to make back to the server, the better.

ASP.NET provides a full-featured cache engine that can be used by pages to store objects across HTTP requests. The CLR class `System.Web.Caching.Cache` provides this facility. The granddaddy of all Web page classes, `System.Web.UI.Page`, includes a property of type cache.

The cache is a simple database supporting a simple dictionary interface—very much like COM's property bags. When you want to cache values over a certain period of time, you put them in the cache. You can put almost whatever you want into the cache and, in addition, you have control over expiration of values.

MANAGING SESSION STATE

Building Web sites and Web services using HTTP is inherently different than building distributed systems using protocols such as DCOM. The DCOM is a connection-intensive protocol. It often makes sense to pay attention to state for the duration of a connection. That is, if a client connects to some server somewhere using DCOM, the server may usually expect that calls coming in over the same connection are always from the same client. Therefore the client can modify the state of the server several times over the course of a connection and may know that it is modifying the same state—this is not so with HTTP.

HTTP is light in terms of connections. That is, it does not maintain connections between requests. When you surf to a URL, your Web browser connects to the server and sends the request, and the server returns a response. That is the end of the connection—it gets torn down right away. The next time you surf to the same URL (e.g., by clicking on a link), the browser sets up a new connection, sends a new request, gets a new response, and then tears down that connection.

Because of its lightweight connection nature, HTTP is a stateless protocol, meaning the server has no automatic guarantee that a string of requests is from the same client (or even the same single browser instance). This is an issue for Web applications that need to maintain state across connections. Common types of state that are often shared include the contents of shopping carts and page scrolling (so that the previous page shows up in the same scrolling position when you hit the Back button). This sort of state management can be incredibly cumbersome to program by yourself, which is why transconnection state management is built into ASP.NET.

Web servers are constantly bombarded by incoming HTTP requests. Quite often an HTTP request will come in from one client and immediately be followed by a request from a completely different client.

ASP.NET makes managing state much easier. The state management facilities come for free. Classic ASP provides two facilities for maintaining state across requests, and they are based on application scope and session scope. Session-scoped objects are bound to a particular client Web application manager (WAM) and live for 20 minutes (by default) after the last request is received. Application-scoped objects are shared by all sessions in a WAM and live as long as at least one session is alive. Each WAM directory can have a `global.asa` file. ASP.NET provides session-management infrastructure with built-in session-state functionality. This built-in functionality serves four main functions:

- It helps you identify and classify requests coming from the same browser client into a logical application "session" on the server.
- It provides a place to store session-scoped data on the server for use across multiple HTTP requests.
- It raises session-lifetime management events (such as `OnSessionStart` and `OnSessionEnd`) so that you can manage the session in your application code.

- It automatically cleans up the session data if the same browser fails to revisit your application after a specified time-out period.

We discuss using ASP.NET's session-state management in more detail in Chapter 8.

CONCLUSION

You now have an overview of ASP.NET. Overall, the new .NET platform has a lot to offer the software developer, including an ecumenical type system, better type information, and an efficient compilation model, all of which truly tighten up the boundaries between components. Taking charge of all these great features is ASP.NET, which will most likely be .NET's "Killer App."

As we mentioned during our discussion of people-oriented software, the future of computing will undoubtedly involve getting machines to talk to one another over the Internet. The Internet is a ubiquitous network to which scores of people have access. Until now, there has not really been a way to use the Internet as a software platform. However, the technology is now in place to make that happen, and ASP.NET makes it that much easier to manage Web programming. ASP.NET keeps all the good features of classic ASP (in process performance, a well-established syntax, the ability to add executable blocks to your Web page) and improves on them (e.g., by providing a more granular HTTP request handling model, providing a compilation model for Web pages, and organizing the parts of a Web page into classes and making those classes available through the CLR type system). ASP.NET will undoubtedly be the tool of choice for most Web developers for the next five to ten years.