# Table of Contents

## Chapter 1. The ASP.NET Programming Model

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What’s ASP.NET, Anyway?</td>
<td>2</td>
</tr>
<tr>
<td>The ASP.NET Component Model</td>
<td>13</td>
</tr>
<tr>
<td>The ASP.NET Development Stack</td>
<td>19</td>
</tr>
<tr>
<td>The ASP.NET Provider Model</td>
<td>26</td>
</tr>
<tr>
<td>Conclusion</td>
<td>35</td>
</tr>
</tbody>
</table>
Chapter 1
The ASP.NET Programming Model

In this chapter:

- What’s ASP.NET Anyway? ................................................ 4
- The ASP.NET Component Model ........................................ 15
- The ASP.NET Development Stack ........................................ 21
- The ASP.NET Provider Model ........................................... 28
- Conclusion ........................................................... 37

ASP.NET is a Web development platform that provides services, a programming model, and the software infrastructure necessary to build enterprise-class applications. As part of the Microsoft .NET platform, ASP.NET provides a component-based, largely extensible, and easy-to-use way to build, deploy, and run Web applications that target any browser or mobile device.

ASP.NET is the culmination of Web development technologies that have rapidly followed one another in the past ten years—one building on another, and each filling the gaps of its predecessor. As a result, ASP.NET is currently the most technologically advanced, feature-rich, and powerful platform for building distributed applications transported by the HTTP protocol. ASP.NET simplifies a number of tasks and is sort of a programming paradise, especially for developers coming from classic ASP, Internet Server Application Programming Interface (ISAPI) programming, or other Web platforms.

Released in late 2005, ASP.NET 2.0 introduced significant changes for application architects as well as for developers. Many of the constituent classes have been reworked, and some underwent face-lift operations. Several new controls have been added for the sake of productivity, and a bunch of new and enhanced system modules made the run-time pipeline more customizable, flexible, robust, and secure. As a result, new practices have emerged as best practices, new programming techniques became available to architects and lead developers, and new system features provided native solutions to known issues with earlier versions.

ASP.NET 3.5 is the next milestone in the evolution of the ASP.NET platform. In this version, ASP.NET features Asynchronous JavaScript and XML (AJAX) capabilities, integration with the Windows Communication Foundation (WCF) service platform, Language INtegrated Query (LINQ) support and a few new server controls to fill existing functional gaps, such as in the area of rich graphical layout.
To maximize the benefits of using ASP.NET, you should first look at the overall model—the components, programmability, and infrastructure. And a close look at the overall model is exactly what this chapter provides. To start out, let’s examine some basic concepts of the ASP.NET platform and its programming model.

What’s ASP.NET, Anyway?

Prior to the advent of ASP.NET, three main technologies and platforms were available to develop Web applications: ASP, Java Server Pages (JSP), and the open source Web platform commonly referred to as LAMP (Linux plus Apache plus MySQL plus either Perl, Python, or PHP as the programming language).

Although each has language-specific and architecture-specific features, all these Web development platforms are designed to create interactive pages as part of a Web-based application. To some extent, all enable developers to separate programming logic from the page layout through the use of components that the page itself is responsible for calling and rendering. Aside from this common ultimate goal, significant differences exist among those platforms, most of which relate to the programming model and languages they promote and support. For example, JSP exploits the Java framework of classes and, with JavaBeans, provides an effective extensibility model for reusing components. In addition, JSP supports tag customization and lets developers associate code with a custom tag definition. Finally, because it’s a key element of the Java Enterprise Edition 5.0 (J2EE) platform, JSP relies on the Java language—a first-class, compiled language—as opposed to the scripting languages used by both ASP and LAMP platforms.

So how does ASP.NET fit in exactly?

Like other Web development environments, ASP.NET also works on top of the HTTP protocol and takes advantage of HTTP commands and policies to set up two-way, browser-to-server communication and cooperation. What really differentiates ASP.NET from the plethora of other Web development technologies is the abstract programming model it propounds, the Web Forms model. In addition, the whole ASP.NET platform comes as a native part of the Microsoft .NET Framework. To be sure you grasp the importance of this last point, let me explain. ASP.NET applications are compiled pieces of code, are made of reusable and
extensible components, can be authored with first-class languages (including C#, Microsoft Visual Basic .NET, Microsoft JScript .NET, and J#), and can access the entire hierarchy of classes in the .NET Framework.

Note In the future, there’s a distinct possibility that you can also use dynamic languages to build ASP.NET pages such as Python and Ruby. Dynamic languages should be considered just one more option you have to build Web pages, and they are far away from replacing statically compiled languages such as C#. The Dynamic Language Runtime (DLR) environment—where dynamically languages are hosted—is currently under development and should be integrated in the next version of Microsoft Silverlight, which is a plug-in for supporting a subset of the Windows Presentation Foundation (WPF) in a browser environment. Chances are that the DLR will be integrated in the ASP.NET runtime in a future release. Should this happen, developers will be allowed to also use Python or Ruby to create ASP.NET applications. For more information on the background technology, read the white paper at http://www.asp.net/IronPython/WhitePaper.aspx.

In short, ASP.NET delivers a wealth of goodies, tools, and powerful system features that can be effectively grouped within the blanket expression tools for abstracting the HTTP programming model. Lots of programmer-friendly classes let you develop pages using typical desktop methods. The Web Forms model promotes an overall event-driven approach, but it is deployed over the Web. In addition, AJAX capabilities make the platform even more powerful and dramatically improve the user’s experience.

Note ASP.NET is supported on a variety of server platforms, including Microsoft Windows 2000 with at least Service Pack 2, Windows Server 2003, and newer operating system versions. To develop ASP.NET server applications, Internet Information Services (IIS) version 5.0 or later is also required. Other software you need—for example, Microsoft Data Access Components (MDAC) 2.7—is automatically installed when you set up the .NET Framework. In terms of performance, robustness, and security, the ideal combination of current system software for hosting ASP.NET applications appears to be Windows Server 2003 (preferably with at least Service Pack 1 applied) and IIS 6.0. Windows Server 2008 and IIS 7.0 are just around the corner, though.

Programming in the Age of Web Forms

The rationale behind the ASP.NET Web Forms model is directly related to the search for a better strategy to deal with the growing demand for cheap but powerful Web interaction. As a matter of fact, the HTTP protocol represents the major strength and weakness of Web applications. The stateless nature of the HTTP protocol introduces vastly different programming concepts that are foreign to many desktop developers—first and foremost among these concepts is session state management. On the other hand, the inherent simplicity and scalability of HTTP is the key to its worldwide adoption and effectiveness—in short, we probably couldn’t have the Internet as we know it without a protocol like HTTP. Yet, as demand for rich
and powerful applications increases, programmers have to devise better ways of setting up easy and effective communication from the client to the server and vice versa. The advent of AJAX is just a clear sign of this need.

The Web Forms model is the ASP.NET implementation of the typical Web paradigm where the browser submits a form to the Web server and receives a full markup page in return. The growing complexity of today’s Web pages, full of multimedia, graphic contents and sophisticated layouts, makes a paradigm shift necessary. AJAX is the new paradigm that ASP.NET 3.5 fully supports. When AJAX is enabled, an ASP.NET page uses script code under the control of the page developer to send a request for data to the Web server and receive other data as a response. Unlike the Web Forms, the AJAX model doesn’t cause full replacement of the current page and results in smoother page transitions and flicker-free page updates. I’ll cover AJAX programming in ASP.NET in Chapter 19 and Chapter 20.

### Event-Driven Programming over HTTP

ASP.NET Web Forms bring the event-driven model of interaction to the Web. Implementing an event model over the Web requires any data related to the client-side user’s activity to be forwarded to the server for corresponding and stateful processing. The server processes the output of client actions and triggers reactions. The state of the application contains two types of information: the state of the client and the state of the session. The state of the client—mostly the contents of form input fields collectively referred to as the page state—is easily accessible through the server-side collections that store posted values. But what about the overall state of the session? The client expects that sending information to the server through one page is naturally related to any other page he or she might view later, such as when adding items to a shopping cart. Who remembers what a particular user has in the shopping cart? By itself, HTTP is incapable of keeping track of this information; that’s where session state and a proper server-side infrastructure surrounding and integrating HTTP fit in.

I can’t emphasize enough the importance of understanding the concepts involved with stateless programming when developing Web applications. As mentioned, HTTP is a stateless protocol, which means two successive requests across the same session have no knowledge of each other. They are resolved by newly instantiated environments in which no session-specific information is automatically maintained, except all the information the application itself might have stored in some of its own global objects.

The ASP.NET runtime carries the page state back and forth across page requests. When generating HTML code for a given page, ASP.NET encodes and stuffs the state of server-side objects into a few hidden, and transparently created, fields. When the page is requested, the same ASP.NET runtime engine checks for embedded state information—the hidden fields—and uses any decoded information to set up newly created instances of server-side objects. The net effect of such a mechanism is not unlike the Windows Forms model on the desktop and is summarized in Figure 1-1.
The Windows Forms model stems from the typical event-driven desktop programming style. No matter what connectivity exists between the client and server components, the server always works in reaction to the client’s input. The server is aware of the overall application state and operates in a two-tier, connected manner. The Web Forms model needs some machinery to support the same event-driven programming model. In Figure 1-1, the needed machinery is represented by the state deserialization that occurs when the page is requested and the state serialization performed when the HTML response is being generated.

In charge of this filtering work is the ASP.NET HTTP runtime—a piece of code that extends and specializes the overall capabilities of the hosting Web server. Hidden fields are the low-level tools used to perform the trick. Such a model wouldn’t be as effective without a server-side, rich object model spanning the entire content of the server page. This component model is crucial to the building and effective working of the ASP.NET development platform.

The ASP.NET component model identifies and describes the building blocks of ASP.NET pages. It is implemented through an object model that provides a server-side counterpart to virtually any HTML page element, such as HTML tags like `<form>` and `<input>`. In addition, the ASP.NET object model includes numerous components (called server controls or Web controls) that represent more complex elements of the user interface (UI). Some of these controls have no direct mapping with individual HTML elements but are implemented by combining multiple HTML tags. Typical examples of complex UI elements are the Calendar control and the GridView control.

In the end, an ASP.NET page is made of any number of server controls mixed with verbatim text, markup, and images. Sensitive data excerpted from the page and controls state is unobtrusively stored in hidden fields, and it forms the context of that page request.
association between an instance of the page and its state is unambiguous, not programmatically modifiable, and controlled by the ASP.NET HTTP runtime.

The ASP.NET component model is the first stop on the way to the full understanding of the ASP.NET platform. The component model supports you through the whole development cycle, including the phase of page authoring and run-time system configuration, as shown in Figure 1-2.

Before we dive into the various elements shown in Figure 1-2, let's briefly review the basics of the HTTP protocol, which remains the foundation of Web interaction. After that, we'll move on to describe the structure of an ASP.NET page and how to write and deploy ASP.NET applications.

The HTTP Protocol

This section provides a quick overview of the way Web applications operate. If you already have a working knowledge of the Web underpinnings, feel free to jump ahead to the section "Structure of an ASP.NET Page."

The acronym HTTP has become so familiar to us developers that we sometimes don't remember exactly what it stands for. Actually, HTTP stands for Hypertext Transfer Protocol.
HTTP is a text-based protocol that defines how Web browsers and Web servers communicate. The format of HTTP packets is fully described in RFC 2068 and is available for download from http://www.w3.org/Protocols/rfc2068/rfc2068.txt. HTTP packets travel over a Transmission Control Protocol (TCP) connection directed toward default port 80 at the target Internet Protocol (IP) address.

The HTTP Request

When you point the browser to a URL, it uses the available Domain Name System (DNS) to translate the server name you provided with the URL into an IP address. Next, the browser opens a socket and connects to port 80 at that address. For example, the packet with the download request for http://www.contoso.com/default.aspx can take the following simple form:

```
GET /default.aspx HTTP/1.1
Host: www.contoso.com
```

The first line of text in a request is the start line of the request. It must contain the name of the HTTP command to execute (GET in this case), the URL of the resource, plus the version of the HTTP protocol you want to use.

An HTTP request can contain, and usually does contain, a number of headers. An HTTP header is a line of text that provides additional information about the request. In the HTTP request just shown, the line beginning with “Host:” is an HTTP header. Headers that can be found in an HTTP request include the following:

- **User-Agent** Identifies the type of browser that originated the request
- **Connection** Closes a connection or keeps a connection alive
- **If-Modified-Since** Provides client-side cache validation

GET and POST are the most commonly used HTTP commands or verbs. The GET verb means retrieve whatever information is identified by the request URL. The POST verb is used to request that the origin server accept the content enclosed in the request and process it. Typically, the POST verb is used to provide a block of data (that is, the result of submitting a form) to a data-handling process.

**Note** The HTTP protocol is not limited to GET and POST verbs to express commands. Other options are available, such as PUT and DELETE to update and delete Web resources, respectively. These verbs, though, are never used in a typical browser session. The set of verbs supported by the HTTP protocol is a topic being revamped these days as the AJAX paradigm takes root and architects explore the most effective ways to build script-callable, server-based services for pages.
Part I Building an ASP.NET Page

The HTTP Response

The server's response includes a status line made from the message's protocol version and an exit code (indicating success or that an error has occurred). The status line is followed by a bunch of headers—typically the page content type and length—and the body content. A blank line separates the body content from the rest of the message, as shown in the following response:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/6.0
Content-Type: text/html
Content-Length: 55

<html><body><h1>ASP.NET 3.5 is cool!</h1></body></html>
```

The preceding code illustrates the simple HTML output returned by the Web server. Requests and responses are strings formatted according to the HTTP schema, and they travel over a TCP connection. The code 200 means that all went OK with the request. The specified Web server processes the request and returns content of a certain length expressed in the given Multipurpose Internet Mail Extensions (MIME) type (text/html). HTTP codes that could be returned are listed in the HTTP specification, available at the aforementioned URL. In addition, it should be noted that the blank line between the last header and the content of the HTTP response is not just formatting—the carriage-return and line-feed pair are required and are very much a part of the standard.

What happens next mostly depends on the MIME type and the local browser's capabilities. As long as the MIME type is text/html, the browser displays the content as HTML. If the MIME type is, say, text/xml, some browsers will render the content as plain text, while others (for example, Microsoft Internet Explorer 6.0 and newer versions) will apply a built-in style sheet.

Building a Server-Side Abstraction Layer

Every conversation between browsers and Web servers consists of an exchange of packets similar to the ones we have just examined. If the requested URL is an HTML page, the Web server typically reads the contents of the .html file and flushes it into the body of the response packet. If the URL is an ASP.NET page, a special IIS module gets involved. The module is an IIS ISAPI plug-in, called an ISAPI extension.

An ISAPI extension is a dynamic-link library (DLL) registered on a per-file extension basis. An ISAPI extension registered to handle .aspx files gets involved whenever a request comes in for this type of resource. The ISAPI extension analyzes the request and configures the server-side environment that will actually process the source of the page. When the state for the request has been successfully retrieved and restored completely, the page is allowed to run and produce the expected output.
Submitting Forms

The HTML `<form>` tag is the only element authorized to transmit client-side data to the server. When the user clicks on a button of type “submit,” by design the browser stuffs the current content of all the controls that belong to the form into a string. The string is then passed to the server as part of the `GET` or `POST` command, depending on the attribute set on the `<form>` tag.

The following HTML snippet illustrates a simple form containing a text box and submit button. As you can see, the form is associated with the `POST` command and the default.aspx URL:

```html
<form method="post" action="default.aspx">
  <input type="text" name="EmpCode" />
  <input type="submit" value="Send" />
</form>
```

The following request shows the `POST` command that hits the Web server when the user enters 1001 for the employee code and then clicks the submit button:

```
POST /default.aspx HTTP/1.1
Host: www.contoso.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 12

EmpCode=1001
```

While processing the page request, the ISAPI extension parses the body of the request and exposes any information found through a more programmer-friendly object model. For example, instead of remaining a simple name/value string, the `EmpCode` variable is moved to an application-wide collection—the `Request.Form` collection. This represents a first level of abstraction built over the raw HTTP programming model. Objects such as Request, Response, and Server form the HTTP context for the call and, as such, represent the minimum set of objects you find in most Web development platforms, including JSP and ASP. In ASP.NET, though, you find much more.

Structure of an ASP.NET Page

An ASP.NET page is a server-side text file saved with the `.aspx` extension. The internal structure of the page is extremely modular and comprises three distinct sections—page directives, code, and page layout:

- **Page directives** Page directives set up the environment in which the page will run, specify how the HTTP runtime should process the page, and determine which assumptions about the page are safe to make. Directives also let you import namespaces to simplify coding, load assemblies not currently in the global assembly cache (GAC), and register new controls with custom tag names and namespace prefixes.
Part I Building an ASP.NET Page

- **Code section** The code section contains handlers for page and control events, plus optional helper routines. Any source code pertinent to the page can be inserted inline or attached to the page through a separate file. If inserted inline, the code goes into a tag with the misleading name of `<script>`. (The name `<script>` has been chosen for backward-compatibility reasons.) Server-side `<script>` tags are distinguished from client-side `<script>` tags by the use of the runat=server attribute. (More on this in a moment.) Any page code is always compiled before execution. Starting with version 2.0 of ASP.NET, it can also be precompiled and deployed in the form of a binary assembly.

- **Page layout** The page layout represents the skeleton of the page. It includes server controls, literal text, inline JavaScript, and HTML tags. The user interface of the server controls can be fleshed out a bit using declared attributes and control properties.

For the page to work, you don't need to specify all sections. Although real-world pages include all the sections mentioned, perfectly valid and functional pages can include only the code section or page layout. In some special cases, you can even have an ASP.NET page made of a single directive.

In Chapter 2, and even more in Chapter 3, we'll delve deep into the features of a page and its building blocks.

A Sample ASP.NET Page

It is about time we see what an ASP.NET page looks like. To start, a simple text editor will suffice; so let’s open Notepad and let the sleeping giant (Microsoft Visual Studio) lie. The following code implements a simple ASP.NET page that lets you enter a string and then changes it to uppercase letters after you click a button. For the sake of simplicity, we use inline code. (As you'll learn later in the book, this is not what you'll be doing in real-world applications or in pages with any complexity.)

```csharp
<!-- Directives -->
<%@ Page Language="C#" %>

<!-- Code Section -->
<script runat="server">
private void MakeUpper(object sender, EventArgs e)
{
  string buf = TheString.Value;
  TheResult.InnerHtml = buf.ToUpper();
}
</script>

<!-- Layout -->
<html>
<head><title>UpperCase</title></head>
<body>
<h1>Make It Upper</h1>
</body>
</html>
```
Blank lines and comments in the preceding listing separate the three sections—directives, code, and page layout. Notice the unsparing use of the `runat` attribute—it’s one of the most important pieces of the whole ASP.NET jigsaw puzzle. In the next section, we’ll discuss `runat` in more detail. For now, it suffices to say that the `runat` attribute promotes an otherwise lifeless server-side tag to the rank of a component instance.

The page layout is made of literals and HTML tags, some of which contain the aforementioned `runat` attribute. Everything flagged this way, despite appearances, is not really an HTML element. More precisely, it is the markup placeholder of a server-side component—an ASP.NET control—that is actually responsible for the final markup served to the browser. In an ASP.NET source, every tag marked with the `runat` attribute is not output as-is, but undergoes a transformation process on the server at the end of which the real markup is generated. The ASP.NET runtime is in charge of mapping tags to control instances. Let’s quickly review the code.

**Quick Review of the Code**

Thanks to the `runat` attribute the input text field becomes an instance of the `HtmlInputControl` class when the page is processed on the server. The `Value` property of the class determines the default text to assign to the input field. When the user clicks the submit button, the page automatically posts back to itself. The magic is performed by the `runat` attribute set for the `<form>` tag. Once on the server, the posted value of the input field is read and automatically assigned to the `Value` property of a newly created instance of the `HtmlInputControl`. Next, the code associated with the `OnServerClick` event runs. This code takes the current content of the text box—the posted string—and converts it to uppercase letters. Finally, the uppercase string is assigned it to the `InnerHTML` property of the server-side control bound to the HTML `<span>` tag. When the `MakeUpper` event handler completes, the page is ready for rendering. At this point, updated HTML code is sent to the browser.

To test the page, copy the `.aspx` file to your Web server’s root directory. Normally, this is `c:\inetpub\wwwroot`. If you want, create an ad hoc virtual directory using the built-in IIS administration tools. Let’s assume the page is named `upper.aspx`. Next, point the browser to the page. Figure 1-3 shows what you get.
It would be useful to take a look at the HTML source of the page when it is first displayed to the user—that is, before the user clicks to make the text uppercase.

```html
<!-- Directives -->
<!-- Code Section -->
<!-- Layout -->
<html>
<head><title>UpperCase</title></head>
<body>
<h1>Make It Upper</h1>
<form method="post" action="upper.aspx" id="Form1">
<div>
<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" value="/wEPdUJx2MAN..=" />
</div>
<script type="text/javascript">
//
var theForm = document.forms["Form1"]; 
if ((theForm) { 
    theForm = document.Form1;
    function __doPostBack(eventTarget, eventArgument) { 
        if ((theForm.onsubmit || (theForm.onSubmit()) != false)) { 
            theForm.__EVENTTARGET.value = eventTarget;
            theForm.__EVENTARGUMENT.value = eventArgument;
            theForm.submit();
        }
    } // -->
</script>
</form>
</body>
</html>
```
Within the <form> tag, a hard-coded action attribute has been added to force posting to the same page. This is by design and is very characteristic of ASP.NET. The various hidden fields you see are essential to the implementation of the postback mechanism and are generated automatically. The same can be said for the embedded script code. The <input> tags are nearly identical to their counterpart in the .aspx source—only the runat attribute disappeared.

Now that we've dirtied our hands with some ASP.NET code, let's step back and review the layers that actually make ASP.NET pages work in the context of an application.

## The ASP.NET Component Model

ASP.NET is the key enabling technology for all Web-related functionality provided by the .NET Framework. The .NET Framework is made entirely of an object-oriented hierarchy of classes that span all programming topics for Windows operating systems. Generally speaking, a Web application is made of pages the user requests from a server and that the server processes and returns as markup code—mostly HTML. How the requested resource is processed, and therefore how the markup is generated, is server-specific. In particular, when the resource happens to have an .aspx extension, IIS delegates any further processing to the ASP.NET runtime system.

The ASP.NET runtime transforms the source code of the requested .aspx page into the living instance of a .NET Framework class that inherits from a base class named Page. At the end of the day, a running ASP.NET page is an object, and so it is for some of its components—the server-side controls.

A large number of new ASP.NET features are just a direct or an indirect propagation of the .NET infrastructure. ASP.NET benefits from cross-language integration and exception handling, garbage collection and code access security, deployment and configuration, and an incredibly rich class library. All these features aren't the products of a self-contained engine, they are available to you because ASP.NET applications are a special breed of a .NET application.
A Model for Component Interaction

Any element in an ASP.NET page that is marked with the `runat` attribute can be given a unique ID, allowing you to access that element from your server-side code. Two factors make this approach possible:

- The component-based architecture of the .NET platform, and the fact that ASP.NET is a constituent part of that platform.
- The ASP.NET built-in mechanism for the application’s state management.

The component-based design of .NET makes component interaction easy and effective in all environments including ASP.NET applications. ASP.NET components access page features and interact by calling one another’s methods and setting properties.

The fact that all elements in the page are true components, and not simply parsable text, provides a flexible and powerful extensibility model. Creating new controls is as easy as deriving a new class; building a page inheritance hierarchy is as easy as specifying a parent class different from the base `Page` class.

The `runat` Attribute

The `runat` attribute is what determines whether a piece of markup text is to be emitted verbatim at render time or transformed into a stateful instance of a particular .NET class. In the latter case, the class makes itself responsible for emitting the related markup. In an ASP.NET page, all markup elements that have the `runat` attribute set to `server` are considered server-side controls. The control class exposes methods and properties that let you configure the state of the component. The control is responsible for emitting HTML code when the page is rendered to the browser. Let’s consider the following simple code that renders an anchor element in the client page:

```csharp
Response.Write("<A id=myAnchor href=www.asp.net>Click me</A>"
```

The anchor element is created programmatically and is not defined in the page layout. In classic ASP, code blocks and the `Response.Write` method are the only ways you have to create or configure controls dynamically. In some development environments, such as Microsoft Visual InterDev, design-time controls provided an object-based way to output dynamically generated HTML. Design-time controls, though, were just what the name indicates—that is, controls you can use at design-time to generate markup and script code. In ASP.NET, you have a kind of `run-time controls` to mark the contrast with design-time controls.
Working with Server-Side Controls

Within an ASP page, there's no way for you to code against the myAnchor element. It's just frozen, lifeless text, only good for sending to the browser. Once on a client, the myAnchor element comes to life and can accept script instructions. Suppose now that you need to set the href attribute of the anchor based on run-time conditions. In classic ASP, you could first obtain the value for the href attribute and then call Response.Write:

```csharp
strHref = "www.asp.net"
strHtml = "<A id=myAnchor 
strHtml = strHtml + "href=" + strHref
strHtml = strHtml +">Click me</A>
Response.Write(strHtml)
```

This code will work unchanged in an ASP.NET page but is certainly not the best you can do. By declaring the <A> tag with the runat attribute, you can give life to the anchor element on the server too:

```html
<A runat="server" id="myAnchor">Click me</A>
```

When the page is loaded, the ASP.NET runtime parses the source code and creates instances of all controls marked with the runat attribute. Throughout the page, the myAnchor ID identifies an instance of the server-side control mapped to the <A> tag. The following code can be used to set the href attribute programmatically when the page loads:

```csharp
<script runat="server" language="C#">
void Page_Load(object sender, EventArgs e)
{
    myAnchor.HRef = "http://www.asp.net";
}
</script>
```

The markup elements whose name matches an HTML element are mapped to the corresponding HTML server control. Note that not all feasible HTML tags have corresponding ASP.NET controls; for those that don't, a generic control is used. The list of tags and their associated controls is hard-coded in the ASP.NET runtime. Elements that belong to the <asp> namespace are mapped to Web server controls. Other markup elements are mapped to the assembly and class name declared by using an @Register directive.

Pagewide Tags

The runat attribute can also be used with pagewide tags such as <body>. These tags are represented through an instance of the HtmlGenericControl class. HtmlGenericControl is the .NET class used to represent an HTML server-side tag not directly represented by a .NET Framework class. The list of such tags also includes <span>, <font>, and <iframe>.
Part I Building an ASP.NET Page

In the following page, the background color is set programmatically when the page loads:

```csharp
<%@ Page Language="C#" %>
<script runat="server">
    private void Page_Load(object sender, EventArgs e)
    {
        TheBody.Style[HtmlTextWriterStyle.BackgroundColor] = "lightblue";
    }
</script>
<html>
<head><title>Background color</title></head>
<body id="TheBody" runat="server">
    <h3>The background color of this page has been set programmatically. Open View|Source menu to see the source code.</h3>
</body>
</html>
```

The resulting HTML code is as follows:

```html
<html>
<head>
    <title>Background color</title>
</head>
<body id="TheBody" style="background-color:lightblue;">
    <h3>The background color of this page has been set programmatically. Open View|Source menu to see the source code.</h3>
</body>
</html>
```

Likewise, you can set any of the attributes of the `<body>` tag, thus deciding programmatically, say, which style sheet or background image to use. You use the `HtmlGenericControl.Attributes` collection to create attributes on the tag:

```csharp
TheBody.Attributes["Background"] = "/proaspnet20/images/body.gif";
```

We'll discuss the programming interface of the `HtmlGenericControl` class in more detail in Chapter 4.

---

**Note**

Starting with ASP.NET 2.0, the contents of the `<head>` tag can be accessed programmatically as long as it is flagged with the `runat` attribute. The background class for the `<head>` tag is named `HtmlHead`. Through the facilities of the `HtmlHead` control, you can set the page title and link to external style sheets programmatically.

---

**Unknown Tags**

In case of unknown tags, namely tags that are neither predefined in the current schema nor user-defined, the ASP.NET runtime can behave in two different ways. If the tag doesn't
contain namespace information, ASP.NET treats it like a generic HTML control. The empty namespace, in fact, evaluates to the HTML namespace, thereby leading the ASP.NET runtime to believe the tag is really an HTML element. No exception is raised, and markup text is generated on the server. For example, let’s consider the following ASP.NET page:

```csharp
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e)
    {
        dinoe.Attributes["FavoriteFood"] = "T-bone steak";
    }
</script>
<html>
<head><title>Unknown tags</title></head>
<body>
<form runat="server">
    <Person id="dinoe" runat="server" />
    Click the <b>View|Source</b> menu item...
</form>
</body>
</html>
```

The `<Person>` tag is still processed as if it was a regular HTML tag, and the `FavoriteFood` attribute is added. Figure 1-4 shows what the HTML code for this page actually is. In the preceding sample, the type of the `dinoe` object is `HtmlGenericControl`.

![Figure 1-4 The HTML source code of the sample page](image)

ASP.NET also processes namespace-less custom tags, mapping them to the `HtmlGenericControl` class. If the tag does contain namespace information, it is acceptable as long as the namespace is `<asp>` or a namespace explicitly associated with the tag name using an `@Register` directive. If the namespace is unknown, a compile error occurs.
ASP.NET Server Controls

There are basically two families of ASP.NET server controls. They are HTML server controls and Web server controls. System.Web.UI.HtmlControls is the namespace of HTML server controls. System.Web.UI.WebControls groups all the Web server controls.

HTML Server Controls

HTML server controls are classes that represent a standard HTML tag supported by most browsers. The set of properties of an HTML server control matches a commonly used set of attributes of the corresponding tag. The control features properties such as InnerText, InnerHtml, Style, and Value plus collections such as Attributes. Instances of HTML server controls are automatically created by the ASP.NET runtime each time the corresponding HTML tag marked with runat="server" is found in the page source.

As mentioned, the available set of HTML server controls doesn't cover all possible HTML tags of any given version of the HTML schema. Only most commonly used tags found their way to the System.Web.UI.HtmlControls namespace. Tags such as <iframe>, <frameset>, <body>, and <h1> have been left out as well as less frequently used tags such as <fieldset>, <marquee>, and <pre>.

The lack of a specialized server control, however, doesn't limit your programming power when it comes to using and configuring those tags on the server. You only have to use a more generic programming interface—the HtmlGenericControl class, which we looked at briefly in this section.

Web Server Controls

Web server controls are controls with more features than HTML server controls. Web server controls include not only input controls such as buttons and text boxes, but also special-purpose controls such as a calendar, an ad rotator, a drop-down list, a tree view, and a data grid. Web server controls also include components that closely resemble some HTML server controls. Web server controls, though, are more abstract than the corresponding HTML server controls in that their object model doesn't necessarily reflect the HTML syntax. For example, let's compare the HTML server text control and the Web server TextBox control. The HTML server text control has the following markup:

```html
<input runat="server" id="FirstName" type="text" value="Dino" />
```

The Web server TextBox control has the following markup:

```html
<asp:textbox runat="server" id="FirstName" text="Dino" />
```

Both controls generate the same HTML markup code. However, the programming interface of the HTML server text control matches closely that of the HTML <input> tag, while
methods and properties of the Web server TextBox control are named in a more abstract way that coincides with how similar controls are named in the .NET Framework, for consistency. For example, to set the content of an HTML server text control you must use the Value property because Value is the corresponding HTML attribute name. If you work with the Web server TextBox control, you must resort to Text. With very few exceptions (that I’ll discuss in Chapter 3), using HTML server controls or Web server controls to represent HTML elements is only a matter of preference and ease of development and maintenance.

The ASP.NET Development Stack

At the highest level of abstraction, the development of an ASP.NET application passes through two phases—page authoring and run-time configuration. You build the pages that form the application, implement its user’s requirements, and then fine-tune the surrounding run-time environment to make it serve pages effectively and securely. As Figure 1-2 shows, the ASP.NET component model is the bedrock of all ASP.NET applications and their building blocks. With Figure 1-2 in mind, let’s examine the various logical layers to see what they contain and why they contain it.

The Presentation Layer

An ASP.NET page is made of controls, free text, and markup. When the source code is transformed into a living instance of a page class, the ASP.NET runtime makes no further distinction between verbatim text, markup, and server controls—everything is a control, including literal text and carriage-return characters. At run time, any ASP.NET page is a mere graph of controls.

Rich Controls

The programming richness of ASP.NET springs from the wide library of server controls that covers the basic tasks of HTML interaction—for example, collecting text through input tags—as well as more advanced functionalities such as grid-based data display. The native set of controls is large enough to let you fulfill virtually any set of requirements. In addition, the latest version of ASP.NET adds a few new rich controls to take developer’s productivity close to its upper limit.

In ASP.NET 3.5, you find controls to create list views, Web wizards, collapsible views of hierarchical data, advanced data reports, commonly used forms, declarative data binding, menus, and site navigation. You even find a tiny API to create portal-like pages. Availability of rich controls means reduction of development time and coding errors, more best practices implemented, and more advanced functionalities delivered to end users. We’ll specifically cover controls in Chapter 4, Chapter 6, and later on in Chapter 11.
Custom Controls

ASP.NET core controls provide you with a complete set of tools to build Web functionalities. The standard set of controls can be extended and enhanced by adding custom controls. The underlying ASP.NET component model greatly simplifies the task by applying the common principles and rules of object-oriented programming.

You can build new controls by enhancing an existing control or aggregating two or more controls together to form a new one. ASP.NET 3.5 comes with a small set of base classes to build brand new controls on. This set of classes includes base classes to simplify the development of new data-bound controls. For more in-depth coverage of control development, take a look at my book "Programming Microsoft ASP.NET 2.0 Applications—Advanced Topics", in particular chapters 11 through 15. Although the book is labeled with ASP.NET 2.0 and Visual Studio 2005, there's really nothing in it that doesn't work or is now considered wrong when working with ASP.NET 3.5.

Adaptive Rendering

Starting with version 2.0, ASP.NET ships a new control adapter architecture that allows any server control to create alternate renderings for a variety of browsers. Note, though, that the new adaptive rendering model doesn't apply to mobile controls. Mobile controls are a special family of Web controls designed to build applications for mobile devices. ASP.NET mobile controls still use the old adapter model, which was available since ASP.NET 1.1, for controls that inherit from MobileControl and are hosted on pages that inherit from MobilePage. In short, if you need to write a mobile application with ASP.NET 2.0, you should use the mobile controls, as you would have done with ASP.NET 1.1.

So what's the added value of the new adapter model? With this form of adaptive rendering, you can write control adapters to customize server controls for individual browsers. For example, you can write a control adapter to generate a different HTML markup for the Calendar control for a given desktop browser. In addition, by simply replacing the adapter component you can fix bugs that relate to rendering and, to the extent that it is possible, customize a solution to a given client. As an example, consider that Microsoft leveraged the adaptive rendering engine to deliver a CSS-friendly version of some native ASP.NET controls that use cascading style sheet (CSS) layout features instead of plain tables. For more information, check out http://www.asp.net/cssadapters.

The Page Framework

Any ASP.NET page works as an instance of a class that descends from the Page class. The Page class is the ending point of a pipeline of modules that process any HTTP request. The various system components that work on the original request build step by step all the
information needed to locate the page object to generate the markup. The page object model sports several features and capabilities that could be grouped in terms of events, scripting, personalization, styling, and prototyping.

Page Events

The life cycle of a page in the ASP.NET runtime is marked by a series of events. By wiring their code up to these events, developers can dynamically modify the page output and the state of constituent controls. In ASP.NET 1.x, a page fires events such as `Init`, `Load`, `PreRender`, and `Unload` that punctuate the key moments in the life of the page. In ASP.NET 2.0 and 3.5 a few new events have been added to allow developers to follow the request processing more closely and precisely. In particular, you find new events to signal the beginning and end of the initialization and loading phase. The page life cycle will be thoroughly examined in Chapter 3.

Page Scripting

The page scripting object model lets developers manage script code and hidden fields to be injected in client pages. This object model generates JavaScript code used to glue together the HTML elements generated by server controls, thus providing features otherwise impossible to program on the server. For example, in this way you can set the input focus to a particular control when the page displays in the client browser.

ASP.NET 2.0 pages can be architected to issue client calls to server methods without performing a full postback and subsequently refresh the whole displayed page. The ASP.NET 2.0 callback mechanism is supplanted by ASP.NET AJAX extensions in ASP.NET 3.5. With AJAX capabilities onboard, you can choose to just enhance your existing ASP.NET 2.0 applications with flicker-free page updates or build pages according to a new paradigm that leverages a set of script-callable Web services. I’ll return to AJAX enhancements in Chapter 19 and Chapter 20.

Cross-page posting is another feature that the community of ASP.NET developers loudly demanded. It allows the posting of content of a form to another page. Sounds like teaching old tricks to a new dog? Maybe. As mentioned earlier in this chapter, one defining characteristic of ASP.NET is that each page contains just one `<form>` tag, which continuously posts to itself. That’s the way ASP.NET has been designed, and it results in several advantages.

In previous versions of ASP.NET, cross-page posting could be implemented the same way as in classic ASP—that is, posting through an HTML pure `<form>` not marked with the `runat` attribute. This method works fine, but it leaves you far from the object-oriented and strongly typed world of ASP.NET. Cross-page posting as implemented in ASP.NET 2.0 and newer versions just fills the gap.
Part I Building an ASP.NET Page

Page Personalization

Starting with ASP.NET 2.0, you can store and retrieve user-specific information and preferences without the burden of having to write the infrastructural code. The application defines its own model of personalized data, and the ASP.NET runtime does the rest by parsing and compiling that model into a class. Each member of the personalized class data corresponds to a piece of information specific to the current user. Loading and saving personalized data is completely transparent to end users and doesn't even require the page author to know much about the internal plumbing. The user personalized information is available to the page author through a page property. Each page can consume previously saved information and save new information for further requests.

Page Styling

Much like Microsoft Windows XP themes, ASP.NET themes assign a set of styles and visual attributes to elements of the site that can be customized. These elements include control properties, page style sheets, images, and templates on the page. A theme is the union of all visual styles for all customizable elements in the pages—a sort of super-CSS file. A theme is identified by name and consists of CSS files, images, and control skins. A control skin is a text file that contains default control declarations in which visual properties are set for the control. With this feature enabled, if the developer adds, say, a Calendar control to a page, the control is rendered with the default appearance defined in the theme.

Themes are a great new feature because they allow you to change the look and feel of pages in a single shot and, perhaps more importantly, give all pages a consistent appearance.

Page Prototyping

Almost all Web sites today contain pages with a similar layout. For some sites, the layout is as simple as a header and footer; others sites might contain sophisticated navigational menus and widgets that wrap content. In ASP.NET 1.x, the recommended approach for developers was to wrap these UI blocks in user controls and reference them in each content page. As you can imagine, this model works pretty well when the site contains only a few pages; unfortunately, it becomes unmanageable if the site happens to contain hundreds of pages. An approach based on user controls presents several key issues for content-rich sites. For one thing, you have duplicate code in content pages to reference user controls. Next, application of new templates requires the developer to touch every page. Finally, HTML elements that span the content area are likely split between user controls.

Starting with ASP.NET 2.0, page prototyping is greatly enhanced thanks to master pages. Developers working on Web sites where many pages share some layout and functionality can now author any shared functionality in one master file, instead of adding the layout information to each page or separating the layout among several user controls. Based on the
shared master, developers can create any number of similar-looking content pages simply by referencing the master page through a new attribute. We’ll cover master pages in Chapter 6.

The HTTP Runtime Environment

The process by which a Web request becomes plain HTML text for the browser is not much different in any version of ASP.NET. The request is picked up by IIS, given an identity token, and passed to the ASP.NET ISAPI extension (\texttt{aspnet_isapi.dll})—the entry point for any ASP.NET-related processing. This is the general process, but a number of key details depend on the underlying version of IIS and the process model in use.

The process model is the sequence of operations needed to process a request. When the ASP.NET runtime runs on top of IIS 5.x, the process model is based on a separate worker process named \texttt{aspnet_wp.exe}. This Microsoft Win32 process receives control directly from IIS through the hosted ASP.NET ISAPI extension. The extension is passed any request for ASP.NET resources, and it hands them over to the worker process. The worker process loads the common language runtime (CLR) and starts the pipeline of managed objects that transform the original request from an HTTP payload into a full-featured page for the browser. The \texttt{aspnet_isapi} module and the worker process implement advanced features such as process recycling, page output caching, memory monitoring, and thread pooling. Each Web application runs in a distinct AppDomain within the worker process. By default, the worker process runs under a restricted, poorly privileged account named ASPNET.

\begin{quote}
\textbf{Note} In the CLR, an application domain (AppDomain) provides isolation, unloading, and security boundaries for executing managed code. An AppDomain is a kind of lightweight, CLR-specific process where multiple assemblies are loaded and secured to execute code. Multiple AppDomains can run in a single CPU process. There is not a one-to-one correlation between AppDomains and threads. Several threads can belong to a single AppDomain, and while a given thread is not confined to a single application domain, at any given time, a thread executes in a single AppDomain.
\end{quote}

When ASP.NET runs under IIS version 6.0 or later, the default process model is different and the \texttt{aspnet_wp.exe} process is not used. The worker process in use is the standard IIS worker process (\texttt{w3wp.exe}) implemented by IIS 6.0 and 7.0. It looks up the URL of the request and loads a specific ISAPI extension. For example, it loads \texttt{aspnet_isapi.dll} for ASP.NET-related requests. Under the IIS 6.0 process model, the \texttt{aspnet_isapi} module is responsible for loading the CLR and starting the HTTP pipeline.

Once in the ASP.NET HTTP pipeline, the request passes through various system and user-defined components that work on it until a valid page class is found and successfully instantiated. Developers can modify and adapt the run-time environment to some extent. This can
happen in a variety of ways: changing the list of installed HTTP modules, configuration files, state and personalization providers, and other application services.

System HTTP Modules
HTTP modules are the ASP.NET counterpart of ISAPI filters. An HTTP module is a .NET Framework class that implements a particular interface. All ASP.NET applications inherit a few system HTTP modules as defined in the `machine.config` file. Preinstalled modules provide features such as authentication, authorization, and session-related services. Generally speaking, an HTTP module can preprocess and postprocess a request, and it intercepts and handles system events as well as events raised by other modules.

The good news is that you can write and register your own HTTP modules and make them plug into the ASP.NET runtime pipeline, handle system events, and fire their own events. In addition, you can adapt on a per-application basis the list of default HTTP modules. You can add custom modules and remove those that you don’t need.

Application Configuration
The behavior of ASP.NET applications is subject to a variety of parameters; some are system-level settings, and some depend on the characteristics of the application. The common set of system parameters are defined in two files—`machine.config` and a machine level (global) `web.config`. All together, the two files contain machine-specific values for all supported settings. Machine settings are normally controlled by the system administrator, and applications should not be given write access to both files. These files are located outside the Web space of the application and, as such, cannot be reached even if an attacker succeeds in injecting malicious code into the system.

Note In ASP.NET 1.x, you have only one machine configuration file named machine.config. The original content of this file has been split between the two files I mentioned starting with ASP.NET 2.0.

Any application can override most of the default values stored in the machine configuration files by creating one or more application-specific `web.config` files. Almost all applications create a `web.config` file in their root folder. This file contains a subset of the machine settings and is written according to the same XML schema. The goal of the application-specific `web.config` is to override some of the default settings. Be aware, however, that not all settings that are defined at the machine level can be overridden in a child configuration file.

If the application contains child directories, you can define a `web.config` file for each directory. The scope of each configuration file is determined in a hierarchical, top-down manner. The settings that are valid for a page are determined by the sum of the changes that the various `web.config` files found along the way applied to the original machine configuration.
Any web.config file can extend, restrict, and override any type of settings defined at an upper level, including the machine level. If no configuration file exists in an application folder, the settings that are valid at the upper level are applied.

**Application Services**

Authentication, state management, and caching are all examples of essential services that the ASP.NET runtime environment supplies to running applications. Starting with ASP.NET 2.0, other services have been added to the list—including membership, role management, and personalization—as shown in Figure 1-5.

Most application services must persist and retrieve some data for internal purposes. While doing so, a service chooses a data model and a storage medium, and it gets to the data through a particular sequence of steps. Applications based on these services are constrained by the design to using those settings—which usually include a fixed data schema, a predefined storage medium, or a hard-coded behavior. What if you don’t like or don’t want these restrictions?
Run-time configuration, as achieved through machine.config and web.config files, adds some more flexibility to your code. However, run-time configuration does not provide a definitive solution that is flexible enough to allow full customization of the service that would make it extensible and smooth to implement. A more definitive solution is provided in version 2.0 of ASP.NET, which formalizes and integrates into the overall framework of classes a design pattern that was originally developed and used in several ASP.NET Starter Kits. Known as the provider model, this pattern defines a common API for a variety of operations—each known as the provider. At the same time, the provider's interface contains several hooks for developers to take complete control over the internal behavior of the API, data schema used, and storage medium.

Important The provider model is one of the most important and critical aspects of ASP.NET. A good understanding of it is crucial to conduct effective design and implementation of cutting-edge applications. The provider model is formalized starting with ASP.NET 2.0, but it is simply the implementation of a design pattern. As such, it is completely decoupled at its core from any platform and framework. So once you understand the basic idea, you can start using it in any application, even outside the boundaries of ASP.NET.

The ASP.NET Provider Model

There's a well-known design pattern behind the ASP.NET provider model—the strategy pattern. Defined, the strategy pattern indicates an expected behavior (say, sorting) that can be implemented through a variety of interchangeable algorithms (say, Quicksort or Mergesort). Each application then selects the algorithm that best fits while keeping the public, observable behavior and programming API intact.

The most notable feature of the strategy pattern is that it provides a way for an object, or an entire subsystem, to expose its internals so that a client can unplug the default implementation of a given feature and plug his own in. This is exactly what happens in ASP.NET for a number of services, including membership, roles, state management, personalization, and site navigation. The ASP.NET provider model is the ASP.NET implementation of the strategy pattern.

The Rationale Behind the Provider Model

The provider model is not an application feature that end users can see with their own eyes. In itself, it doesn't make an application show richer content, run faster, or be more responsive. The provider model is an infrastructural feature that improves an application's architecture by enabling developers and architects to operate under the hood of some system components.
At the same time, it enables developers to build new components that expose hooks for clients to plug in and customize behavior and settings. Implementing the strategy pattern doesn’t transform an application into an open-source project, allowing anybody to modify anything. It simply means that you have a simple, elegant, and effective pattern to make certain parts of your application customizable by clients. At the same time, the ASP.NET implementation of the pattern—the provider model—makes you capable of customizing certain parts of the ASP.NET runtime environment through special classes, named providers, from which you can derive your own.

**Exemplifying the Provider Model**

To see an example of the provider model and its major benefits, let’s look at Figure 1-6. The figure outlines the classic schema for authenticating a user. The blocks of the diagram follow closely the flow of operations in ASP.NET 1.1 and any Web applications on whatever platform and framework available.

![Classic Membership Scenario](image)

The user who attempts to connect to a protected page is shown a login page and invited to provide their credentials. Next, the name and password are passed on to a function, which is ultimately responsible for validating the user. ASP.NET can automatically check users against Windows accounts or, perhaps, a list of names in the *web.config* file. None of these approaches work well in a realistic Web application; in most cases, developers just end up writing a custom piece of code to validate credentials against a homemade data source. The schema and storage medium of the data source are fixed and determined by the developer. Likewise, the algorithm employed to validate credentials is constrained by the design.
Is there anything wrong with this solution? Not necessarily. It works just fine, puts you in control of everything, and can be adapted to work in other applications. The rub is that there’s no well-defined pattern that emerges from this solution. Sure, you can port it from one application to the next, but overall the solution relates to the adapter pattern mostly like cut-and-paste relates to object-oriented inheritance.

Let’s briefly consider another scenario—session state management. In ASP.NET 1.x, you can store the session state in a process separate from the running application—be it SQL Server or a Windows service (the ASP.NET state server). If you do so, though, you’re constrained to using the data schema that ASP.NET hard-codes for you. Furthermore, imagine you’re not a SQL Server customer. In this case, either you abandon the idea of storing session state to a database or you buy a set of licenses for SQL Server. Finally, there’s nothing you can do about the internal behavior of the ASP.NET session module. If you don’t like the way it, say, serializes data to the out-of-process storage, you can’t change it. Take it or leave it—there’s no intermediate choice.

Can you see the big picture? There are modules in ASP.NET that force you to take a fixed schema of data, a fixed storage medium, and a fixed internal behavior. The most that you can do is (sometimes) avoid using those modules and write your own from scratch, as we outlined in the membership example. However, rolling your own replacement is not necessarily a smart move. You end up with a proprietary and application-specific system that is not automatically portable from one application to another. In addition, if you hire new people, you have to train those people before they get accustomed to using your API. Finally, you have to put forth a lot of effort to build and test such a proprietary API and make it general enough to be reusable and extensible in a variety of contexts. (Otherwise, you get to reinvent the wheel time after time.)

In which way is the provider model a better solution? In the first place, it supplies a well-documented and common programming interface to perform common tasks. In addition, you gain the ability to completely control the internal business and data access logic of each API that falls under its umbrella.

In the end, in ASP.NET 1.1 you often have no other choice than writing your own API to roll certain functions the way you want. Starting with ASP.NET 2.0, the provider model offers a much better alternative. So much better that it’s practically a crime not to use it.

Figure 1-7 revisits Figure 1-6 in light of the provider model.
ASP.NET 2.0 makes available a bunch of static methods on a global class—Membership. (We'll cover the membership API in great detail in Chapter 17.) At the application level, you always invoke the same method to perform the same operation (for example, validating user credentials, creating new users, changing passwords). Below this common API, though, you can plug in your own provider to do the job just the way you want. Writing a new provider is as easy as deriving a new class from a known base and overriding a few well-known methods. The selection of the current provider for a given task takes place in the configuration file.

Benefits of the Provider Model

In the ASP.NET implementation, the strategy pattern brings you two major benefits: extensive customization of the application’s run-time environment, and code reusability. Several areas in ASP.NET are affected by the provider model. You can write providers to handle user membership and roles, persist session state, manage user profiles through personalization, and load site map information from a variety of sources. For example, by writing a provider you can change the schema of the data used to persist credentials and store this data in an Oracle or DB2 database. This level of customization of system components is unprecedented, and it opens up a new world of possibilities for application developers. At the same time, it gives you an excellent starting point for writing new providers and even extending the model to your own components.

If you look at ASP.NET 3.5 from the perspective of existing version 1.x applications, the provider model gains even more technical relevance because it is the key to code reuse and subsequent preservation of investments in programming and development time. As I pointed
out, a realistic membership system in ASP.NET 1.1 requires you to roll your own API as far as validation and user management are concerned. What should you do when the decision to upgrade to ASP.NET 2.0 or 3.5 is made? Should you drop all that code to embrace the new dazzling membership API of the newest ASP.NET platforms? Or would you be better sticking to the old-fashioned and proprietary API for membership?

The provider model delivers the answer (and a good answer, indeed) in its unique ability to switch the underlying algorithm while preserving the overall behavior. This ability alone wouldn’t be sufficient, though. You also need to adapt your existing code to make it pluggable in the new runtime environment. Another popular pattern helps out here—the adapter pattern. The declared intent of the adapter pattern is to convert a class A to an interface B that a client C understands. You wrap the existing code into a new provider class that can be seamlessly plugged into the existing ASP.NET 3.5 framework. You change the underlying implementation of the membership API, and you use your own schema and storage medium while keeping the top-level interface intact. And, more importantly, you get to fully reuse your code.

A Quick Look at the ASP.NET Implementation

The implementation of the ASP.NET provider model consists of three distinct elements—the provider class, configuration layer, and storage layer. The provider class is the component you plug into the existing framework to provide a desired functionality the way you want. The configuration layer supplies information used to identify and instantiate the actual provider. The storage layer is the physical medium where data is stored. Depending on the feature, it can be Active Directory, an Oracle or SQL Server table, an XML file, or whatever else.

The Provider Class

A provider class implements an interface known to its clients. In this way, the class provides clients with the functionality promised by that interface. Clients are not required to know anything about the implementation details of the interface. This code opacity allows for the magic of code driving other code it doesn’t even know about. In the ASP.NET provider model, the only variation to the original definition of the strategy pattern is that base classes are used instead of interfaces.

In ASP.NET, a provider class can’t just be any class that implements a given interface. Quite the reverse, actually. A provider class must inherit from a well-known base class. There is a base class for each supported type of provider. The base class defines the programming interface of the provider through a bunch of abstract methods.

All provider base classes derive from a common class named `ProviderBase`. This base class provides one overridable method—`Initialize`—through which the run-time environment passes any pertinent settings from configuration files. Figure 1-8 outlines the hierarchy of provider classes.
Interfaces vs. Base Classes

Raise your hand if you are a developer who has never been involved in hours and hours of debate on the subject of interfaces versus base classes. It's a discussion that rarely comes to an end and always leaves folks from different camps firmly holding to their respective positions. Should you use interfaces, or are base classes better? Which considerations is your answer based on? Consider the following fact, first.

Early builds of ASP.NET 2.0—the first version of ASP.NET to support the provider model—implemented the provider model literally with the definition of the strategy pattern, that is, through interfaces. In the Beta 1 timeframe, interfaces were replaced with base classes, and so it was with the released version. The ASP.NET team seemingly came to a conclusion on the issue, did it not?

An interface is a collection of logically related methods that contains only member definitions and no code. An interface type is a partial description of a type, which multiple classes can potentially support. In other words, a good interface is one that is implemented by a number of different types and encapsulates a useful, generalized piece of functionality that clients want to use. That's why many interfaces just end with the suffix "able," such as IDisposable, IComparable, and IFormattable. If an interface has only one useful implementing class, it is likely the offspring of a bad design choice. As a practical rule, new interfaces should be introduced sparingly and with due forethought.

A base class defines a common behavior and a common programming interface for a tree of child classes. Classes are more flexible than interfaces and support versioning.
If you add a new method to version 2.0 of a class, any existing derived classes continue to function unchanged, as long as the new method is not abstract. This is untrue for interfaces.

In light of these considerations, the emerging rule is that one should use base classes instead of interfaces whenever possible (which doesn’t read as, “always use base classes”). To me, base classes appear to be an excellent choice, as far as the provider model is concerned.

In more general terms, the debate between base classes and interfaces has no easy answer. In some application-specific cases, in fact, one could argue that using a base class for a behavior that can be described with an interface takes away the option of deriving that new class from a custom base class that could lead to an overall better design. ASP.NET uses base classes for its provider model, and that is the pattern you must follow.

The Configuration Layer

Each supported provider type is assigned a section in the configuration file, which is where the default provider for the feature is set and all available providers are listed. If the provider sports public properties, default values for these properties can be specified through attributes. The contents of the section are passed as an argument to the Initialize method of the ProviderBase class—the only method that all providers have in common. Within this method, each provider uses the passed information to initialize its own state. Here’s a snapshot of the configuration section for the membership provider.

```xml
<membership defaultProvider="AspNetSqlProvider">
  <providers>
    <add name="AspNetSqlProvider"
        connectionStringName="LocalSqlServer"
        enablePasswordRetrieval="false"
        enablePasswordReset="true"
        requiresQuestionAndAnswer="true"
        ...
        passwordFormat="Hashed" />
    ...
  </providers>
</membership>
```

Chapter 1. The ASP.NET Programming Model
Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.
The Storage Layer

All providers need to read and write information to a persistent storage medium. In many cases, two providers of the same type differ only by the storage they employ. Details of the storage medium are packed in the attributes of the provider in the `<providers>` section, as shown in the preceding code sample. For example, the preceding `AspNetSqlProvider` provider is the predefined membership provider that reads and writes to a SQL Server table. The connection string for the provider is specified through the `connectionStringName` attribute, which in turn refers to another centralized section of the configuration files that lists all available connection strings.

For the provider to work, any needed infrastructure (that is, database, tables, relationships) must exist. Setting up the working environment is a task typically accomplished at deployment time. ASP.NET makes it a breeze thanks to the Web site administration console, which is shown in Figure 1-9.

Available Types of Providers

The provider model is used to achieve several tasks, the most important of which are as follows:

- The implementation of a read/write mechanism to persist the user profile
- The creation of a user-defined repository of user credentials that supports most common operations, such as checking a user for existence, adding and deleting users, and changing passwords
- The creation of a user-defined repository for user roles
- The definition of the site map
- The introduction of newer types of data storage for the session state
Table 1-1 shows the list of the provider classes available in ASP.NET.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MembershipProvider</td>
<td>Base class for membership providers used to manage user account information.</td>
</tr>
<tr>
<td>PersonalizationProvider</td>
<td>Base class for managing personalization for Web Parts components.</td>
</tr>
<tr>
<td>ProfileProvider</td>
<td>Base class for personalization providers used to persist and retrieve a user's profile information.</td>
</tr>
<tr>
<td>ProtectedConfigurationProvider</td>
<td>Base class for encryption providers used to encrypt information in configuration files.</td>
</tr>
<tr>
<td>RoleProvider</td>
<td>Base class for role providers used to manage user role information.</td>
</tr>
<tr>
<td>SessionStateStoreProviderBase</td>
<td>Base class for session state store providers. These providers are used to save and retrieve session state information from persistent storage media.</td>
</tr>
<tr>
<td>SiteMapProvider</td>
<td>Base class for managing site map information.</td>
</tr>
<tr>
<td>WebEventProvider</td>
<td>Base class for health monitoring providers that process system events.</td>
</tr>
</tbody>
</table>

The classes listed in Table 1-1 define an abstract method for each aspect that's customizable in the feature they represent. For example, regarding membership management, the class MembershipProvider exposes methods such as ValidateUser, CreateUser, DeleteUser, ChangePassword, and so forth. Note that you'll never use MembershipProvider in your code just because it's an abstract class. Instead, you'll use a derived class such as SqlMembershipProvider or, perhaps, ActiveDirectoryMembershipProvider. The same holds true for other types of providers.

Finally, if you're going to write a custom membership provider that wraps your existing code, you create a class that inherits from MembershipProvider or similar classes if other provider-based features are involved.

The provider architecture is one of ASP.NET 2.0's most important new features and also one of the most delicate with regard to applications. To prevent developers from producing buggy providers, the ASP.NET team supplies a made-to-measure provider toolkit that details what you can and cannot do in a provider, plus lots of sample code to serve as a guide. Writing a custom provider can be tricky for at least a couple of reasons. First, ASP.NET providers must be thread-safe. Second, their initialization step can lead you straight into a deadly reentrancy. Be sure you download the ASP.NET provider toolkit from the ASP.NET Developer Center (msdn2.microsoft.com/en-us/asp.net/aa336558.aspx) before you leap into a new provider project.
Chapter 1. The ASP.NET Programming Model

Conclusion

As part of the .NET Framework, ASP.NET allows you to take full advantage of features of the common-language runtime (CLR), such as type safety, inheritance, language interoperability, and versioning. ASP.NET builds on the successes of a variety of other platforms, including classic ASP, JSP, and LAMP and promotes a programming model that, although built on top of the stateless HTTP protocol, appears to be stateful and event-driven to programmers. ASP.NET 3.5, in particular, extends previous versions of the Microsoft Web platform with key additions such as WCF, LINQ, and AJAX support.

In this chapter, we first analyzed the component model that backs up ASP.NET Web pages and then went through the development stack from top (presentation layer and rich controls) to bottom (infrastructure and providers). The provider model—in the end, an implementation of the strategy pattern—is a key element in the new ASP.NET architecture and a pillar of support for new applications. Extensively applied, it allows you to customize several low-level aspects of the application's run-time environment and reuse large portions of existing code. Fully understood, it gives you a way to build new components that are flexible and extensible beyond imagination and, as such, seamless to plug in to a variety of projects and easier to customize for clients.

Just the Facts

- In ASP.NET, you take full advantage of all CLR features such as type safety, inheritance, code access security, and language interoperability.
- At execution time, ASP.NET pages are represented by an instance of a class that descends from the Page class.
- The Page class is the ending point of a pipeline of modules that process any HTTP request.
- Only elements in an ASP.NET page marked with the runat attribute can be programmatically accessed when the page is executed on the server.
- Page elements devoid of the runat attribute are not processed on the server and are emitted verbatim.
- The runat attribute applies to virtually any possible tags you can use in an ASP.NET page, including custom and unknown tags.
- The process model is the sequence of operations needed to process a request. The selected process model determines which worker process takes care of running ASP.NET applications and under which account.
- ASP.NET applications run under a poorly privileged account.
The behavior of ASP.NET applications can be configured through a bunch of configuration files.

The ASP.NET provider model is an infrastructural feature that improves an application’s architecture by enabling developers and architects to operate under the hood of some system components.

The ASP.NET provider model brings you two major benefits: extensive customization of the application’s run-time environment and code reusability.