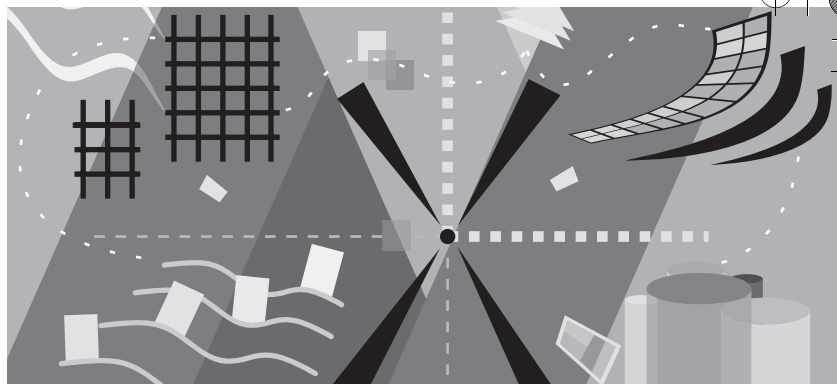


3



ASP.NET

*Interdependence absolute, foreseen, ordained, decreed,
To work, Ye'll note, at any tilt an' every rate o' speed.
Fra skylight-lift to furnace-bars, backed, bolted, braced an' stayed,
An' singin' like the Mornin' Stars for joy that they are made;*
—Rudyard Kipling, writing on interoperation
and scalability, “McAndrew’s Hymn,” 1894.

Problem Background

The Web was first used to deliver static pages of text and pictures. Programming a server to do this was relatively easy—just accept the URL identifying the file, fetch the file that it names from the server’s disk, and write that file back to the client, as shown in Figure 3-1. You can do a lot with just this simple architecture. For example, my local art cinema has a small Web site that I can browse to see what’s playing tonight (for example, *Happy, Texas*, in which two escaped convicts are mistaken for beauty pageant organizers in a small Texas town), learn about coming attractions, and follow links to trailers and reviews. They’re using the Web like a paper brochure, except with richer content, faster delivery, and lower marginal cost—in a word, lower friction. (OK, two words, I was off by one.)¹

The Web was initially used for viewing static pages, which was relatively easy to program.

1. One of the smartest, albeit geekiest, guys I know once said that there’s only one bug in all of computing, and it’s being “off by one.” “Can’t you be off by 25,000?” I asked him. “You’ve got to be off by one first,” he replied.

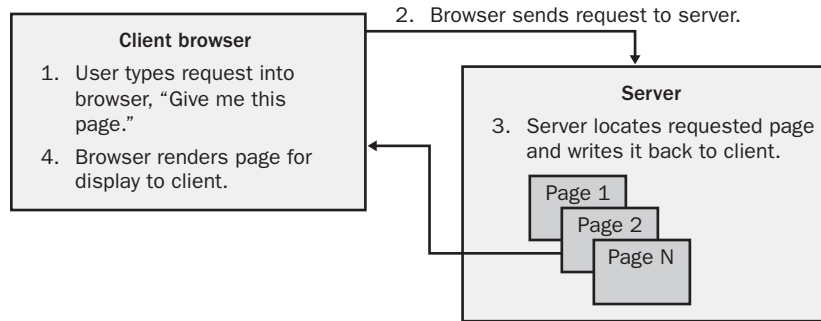


Figure 3-1 Server delivering static Web pages.

Web programmers today need to dynamically generate HTML pages in response to input received from the user, which poses new problems.

This approach worked well in the prehistoric days when all data on the Web was static (rendering the author's content verbatim, with no user input or programming logic) and public (available to anyone who knew or could find the magic address). But customers soon started asking, "If I can see what movie's playing tonight, why can't I see my current bank balance?" which the static page approach can't handle. A bank can't create a new page every day for every possible view of every account—there are far too many of them. Instead, the user needs to provide input such as the account number, and the bank's computer needs to create on demand an HTML page showing the user's account balance, as shown in Figure 3-2. This data is neither static nor public, which raises thorny new design problems

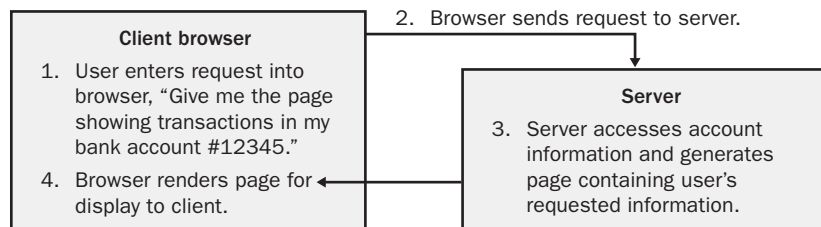


Figure 3-2 Server dynamically generating Web pages based on client input.

Our Web server application requires program logic that generates pages.

A Web server application that dynamically generates pages for a client needs several things. First, it needs a way of associating some sort of program logic with a page request. When a user requests a page, the server doesn't simply fetch the page from the disk; the page doesn't exist before the request. Instead, the server executes program logic that generates the page. In the bank example, we probably need to do a database lookup in the bank's central ledger to find out the customer's current balance and recent transactions.

We'd like to be able to write this logic quickly and easily, using languages and tools with which we are already familiar. And we'd like it to run as quickly and efficiently as possible in production.

Second, our Web server needs a way to get input from the user into the server-side program logic and output from that program logic back to the user. The user's browser submits to the server an HTML form, whose input controls specify the data that he'd like to see on his page—for example, his account number and the range of dates for which he wants to see transactions. Parsing the interesting data values from this HTML is tedious and highly repetitive, so we'd like a prefabricated way of doing it that's quick and easy for us to program. Think how much easier the text box (edit control for you C++ geeks) makes reading character input in a Windows user interface compared to assembling character strings yourself from individual keystrokes. We'd also like a similar level of prefabrication in assembling the HTML page for output to the user. Raw HTML is difficult and tedious to write. Think how much easier the label control makes output in a Windows user interface compared to writing all the GDI calls needed to set font, color, text, and so on. We'd like something similar for the HTML output that a browser requires.

Third, since at least some of our data is now private, our Web server needs to ensure that we know who a user is and that we only allow the user to see and do the things that he's allowed to. You'd like to see your bank account, but you really don't want your disgruntled former spouse looking at it, or, far worse, moving money out of it. This type of code is notoriously difficult and expensive to write—and proving to your wary customers that you've made the code bulletproof, so that they have the confidence to use it, is equally difficult and expensive. We need a prefabricated infrastructure that provides these security services to us.

Finally, our Web server needs a mechanism for managing user sessions. A user doesn't think of her interaction with a Web site in terms of individual page requests. Instead, she thinks of it in terms of a conversation, a "session," that takes place with multiple pages on your site over some reasonable amount of time. She expects the Web site to be able to remember things that she's told it a few minutes previously. For example, a user expects to be able to place items in an e-commerce site's shopping cart and have the cart remember these items until check out. Individual page requests don't inherently do this; we have to write the code ourselves to make it happen. Again, it's an integral part of most Web applications, so we'd like a prefabricated implementation of it that's easy to use. Ideally, it would work correctly in a multi-server environment and survive crashes.

In short, our Web server needs a run-time environment that provides prefabricated solutions to the programming problems common to all Web

Our Web server needs a convenient way of receiving input from and writing output to the user's browser.

Our Web server needs security services to keep unauthorized users from seeing or doing things that they shouldn't.

Our Web server needs a way to manage user sessions.

We need an entire Web server programming and run-time environment.

servers. We'd like it to be easy to program as well as to administer and deploy. We'd like it to scale to at least a few servers, ideally more. And we don't want to pay a lot for it. Not asking for much, are we?

Solution Architecture

The original ASP was a Web server run-time environment that was easy to use for simple tasks.

Microsoft released a relatively simple Web run-time environment called Active Server Pages (ASP) as part of Internet Information Server (IIS), itself part of the Windows NT 4 Option Pack, in the fall of 1997. IIS served up Web pages requested by the user. ASP allowed programmers to write program logic that dynamically constructed the pages that IIS supplied by mixing static HTML and scripting code, as shown in Listing 3-1. When a client requested an ASP page, IIS would locate the page and activate the ASP processor. The ASP processor would read the page and copy its HTML elements verbatim to an output page. In this case, the *style* attribute sets the text color to blue. It would also interpret the scripting elements, located within the delimiters `<% %>`. This code would perform program logic that rendered HTML strings as its output, which the ASP processor would copy to the location on the output page where the scripting element appeared. The resulting page, assembled from the static HTML elements and HTML dynamically generated by the script, would be written back to the client, as shown in Figure 3-3. ASP was relatively easy to use for simple tasks, which is the hallmark of a good first release.

```
<html style="color:#0000FF;">  
  The time is: <% =time %> on <% =date %>  
</html>
```

Listing 3-1 Intermingling of code and HTML in ASP.

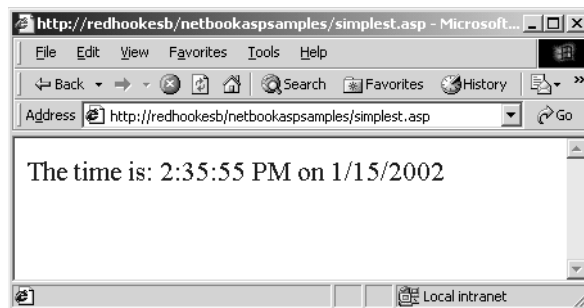


Figure 3-3 The Web page produced by ASP after processing the HTML/code mixture in Listing 3-1.

As the Web spread and user demands increased, Web programmers required more sophistication from their Web run-time environment in two key areas: making it easier to program and making it run better. ASP.NET is a big improvement in both these areas. ASP.NET looks somewhat like the original ASP, and most code will port between the versions unchanged or very close to it. But internally ASP.NET has been completely redone to take advantage of the .NET Framework. You'll find that it runs better, crashes less, and is easier to program. These are good things to have. I discuss all of these features in more depth later in this chapter.

ASP.NET is a complete rewrite of the original ASP, keeping the best concepts.

ASP's mingling of HTML output elements and scripting code may look logical, but it's the devil to program and maintain in any but the simplest cases. Because code and data could and did appear anywhere on the page, intelligent development environments such as Visual Basic couldn't make sense of it, so no one ever wrote a development environment that could handle ASP well. This meant that ASP code was harder to write than other types of code, such as a Visual Basic user interface application using forms. I know it drove me bats.

ASP.NET separates the HTML output from the program logic using a feature called *code-behind*. Instead of mixing HTML and code, you write the code in a separate file to which your ASP page contains a reference. You'd be astounded how much easier your code is to understand when you remove the distraction of HTML output syntax. It's like getting your three year old to shut up while you're talking taxes on the phone with your accountant. Because of this separation, Microsoft was able to enhance the Visual Studio .NET programming and debugging environment so that you can use it while developing your Web applications.

ASP.NET disentangles your code from the HTML.

Input and output in original ASP could be tricky, as the HTML environment was imperfectly abstracted. By this I mean that the programmer often had to spend time grappling with some fairly grotty HTML language constructs instead of thinking about her program logic, which isn't the best use of resources. For example, parsing data out of HTML forms required much more work than doing the same thing on a desktop app. Producing output required the programmer to assemble HTML output streams, which again isn't where you want your programmers to be spending their time. In Listing 3-1, our programmer had to know the proper HTML syntax for turning the text color blue.

ASP.NET supports *Web Forms*, which is a Web page architecture that makes programming a Web page very similar to programming a desktop application form. You add controls to the page and write handlers for their events, just as you do when writing a desktop app in Visual Basic. The ease of use that

84 Introducing Microsoft .NET, Third Edition

ASP.NET contains prefabricated controls that do for HTML pages what Windows controls did for Windows applications.

ASP.NET contains much good prefabricated support for securing your applications.

ASP.NET contains new session state management features that work on larger-scale Web farms.

ASP.NET contains many features, making it run better and making it easier to administer.

made Visual Basic so popular is now available for constructing Web applications. Just before the first edition of this book went to press, I taught a class on the beta 1 release of ASP.NET to a seasoned bunch of ASP developers, and this is the feature that made them literally stand up and cheer.

As Visual Basic depended on Windows controls and third-party ActiveX controls, so does Web Forms depend on a new type of control, named *ASP.NET Server Controls*. For convenience, I refer to these as Web controls in this chapter. These are pieces of prefabricated logic dealing with input and output that a designer places on ASP.NET pages, just as he did with a Windows application form. The controls abstract away the details of dealing with HTML, just as Windows controls did for the Windows GDI. For example, you no longer have to remember the HTML syntax for setting the foreground and background color for a line of text. Instead, you'll use a label control and write code that accesses the control's properties, just as you did in any other programming language. Think of how easy controls make it for you to program a simple desktop app in Visual Basic. Web controls do the same thing for ASP.NET pages.

Original ASP included very little support for security programming. Security in ASP.NET is much easier to write. If you're running in a Windows-only environment, you can authenticate a user (verify that he really is who he says he is) automatically using Windows built-in authentication. For the majority of installations that are not Windows-only, ASP.NET contains prefabricated support for implementing your own authentication scheme. And it also contains prefabricated support for Microsoft's Passport worldwide authentication initiative, if you decide to go that route.

Original ASP supported session state management with an easy-to-use API. Its main drawbacks were that the state management couldn't expand to more than one machine and that it couldn't survive a process restart. ASP.NET expands this support by adding features that allow it to do both automatically. You can set ASP.NET to automatically save and later restore session state either to a designated server machine or to Microsoft SQL Server.

ASP.NET contains many useful new run-time features as well. Original ASP executed rather slowly because the scripting code on its pages was interpreted at run time. ASP.NET automatically compiles pages, either when a page is placed on the server or when it's first requested, which makes it run much faster. Because it uses the .NET Framework's just-in-time compilation, you don't have to stop the server to replace a component or a page. It also

supports recycling of processes. Instead of keeping a process running indefinitely, ASP.NET can be set to automatically shut down and restart its server process after a configurable time interval or number of hits. As garbage collection solves the problem of memory leaks within managed .NET code, this process recycling solves many of the problems caused by leaking unmanaged external resources. This solves most problems of memory leaks in user code. ASP.NET also has the potential of being easier to administer because all settings are stored in human-readable files in the ASP directory hierarchy itself.

Simplest Example: Writing a Simple ASP.NET Page

Let's look at the simplest ASP.NET example that I could think of. Shown in Figure 3-4, it displays the current time on the server, either with or without the seconds digits, in the user's choice of text color. You can download the code from this book's Web site, <http://www.introducingmicrosoft.net>, and follow along with the discussion.

An ASP.NET sample program begins here.

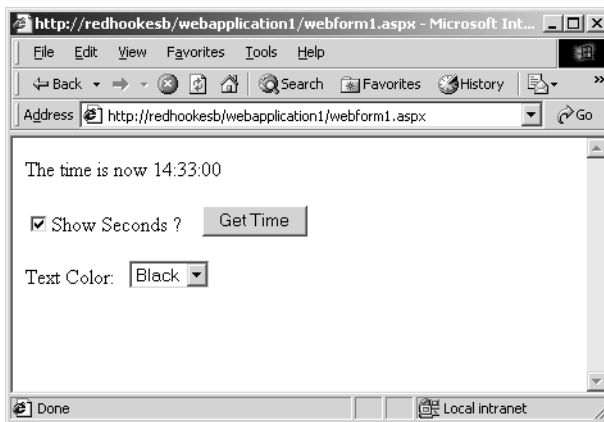


Figure 3-4 Simplest example of ASP.NET.

I wrote this sample using Visual Studio .NET. While I've tried to write my examples in a tool-agnostic manner, Visual Studio .NET contains so much built-in support for ASP.NET, especially debugging, that not using it would have been like using Notepad to write a Visual Basic Windows desktop app. You can do it if you really, REALLY want to and probably get it to

You put Web controls on your .ASPX pages, just as you do on a Visual Basic form.

work eventually. But it will be much faster and less painful if you take the right approach from the beginning.

The key to understanding this example is to think of your Web page as a Visual Basic form. I use that analogy because I think that my readers are most familiar with it, but you could write this same code in any other Visual Studio .NET language—C#, C++, or J#—if you want to. (I've provided C# version of all the code in the sample files.) I started by generating an ASP.NET Web Application project from Visual Studio .NET, as shown in Figure 3-5. Doing this creates, among other things, an ASP.NET page called `WebForm1.aspx`. I then used the Toolbox to drag Web controls onto the form, shown in Figure 3-6, adding a check box, a button, a drop-down list, and a couple of labels. I then set the properties of these controls using the Properties window in the lower right corner of the Visual Studio .NET window, specifying the text of each control, the font size of the labels, and the items in my drop-down list collection. I set the *AutoPostBack* property of the drop-down list control to *True*, meaning that it automatically posts the form back to the server when the user makes a new selection in the drop-down list. Setting user interface properties with a slick, familiar editor like this is so much easier than writing HTML strings in Notepad (which is what you had to do in ASP) that it isn't funny.

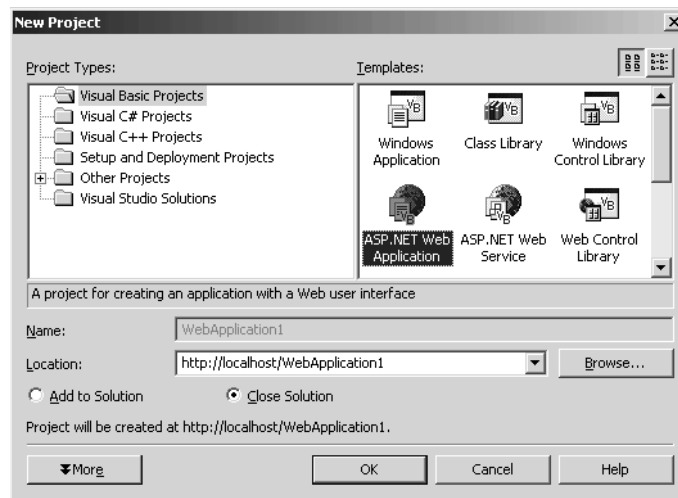


Figure 3-5 Choosing an ASP.NET Web Application in Visual Studio .NET.

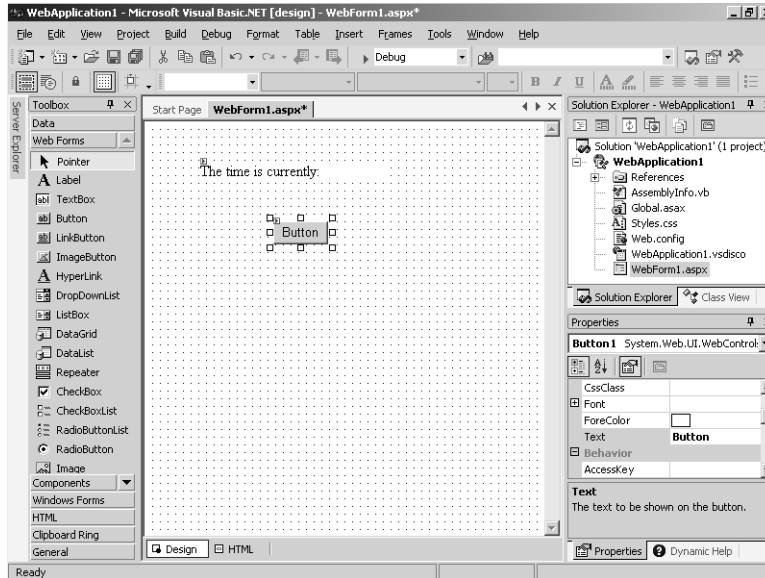


Figure 3-6 Using the Toolbox to drag Web controls onto the form.

Now that I've laid out my controls, I need to write the code that ties them together. When I generated the project, Visual Studio .NET also generated the class that represents the code behind my ASP.NET page. You view it by right-clicking on the page and selecting View Code from the shortcut menu, just as you do today with forms in Visual Basic 6.0. Web controls fire events to their forms, again just as in Visual Basic, so I need to write event handlers for them. I can add an event handler by choosing the control from the upper-left drop-down list and choosing the appropriate event from the upper-right drop-down list. I show excerpts from my Visual Basic class in Listing 3-2, and all of it is included with this book's downloadable sample code. In this simple example, when the button reports a *Click* event, I change the label's text property to hold the current time. I also added a handler for my drop-down list box's *SelectedIndexChanged* event. When the user makes a color selection from the list box, I set the label's color to the value selected by the user. When I build the project, Visual Studio .NET automatically publishes it to my machine's Web root directory.

You write event handlers for your controls' events, just as you do for a Visual Basic form.

```

Public Class WebForm1
    Inherits System.Web.UI.Page

    ' User clicked the button. Fetch the time and display it in the
    ' label control.

    Public Sub Button1_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
        If CheckBox1.Checked = True Then
            Label1.Text = "The time is now " + now.ToLongTimeString
        Else
            Label1.Text = "The time is now " + now.ToShortTimeString
        End If
    End Sub

    ' User has selected a different color from the list box.
    ' Change the label's color to reflect the user's choice.

    Protected Sub DropDownList1_SelectedIndexChanged(ByVal sender _
        As System.Object, _
        ByVal e As System.EventArgs) _
        Handles DropDownList1.SelectedIndexChanged
        Label1().ForeColor = _
            Color.FromName(DropDownList1().SelectedItem.Text)
    End Sub

End Class

```

Listing 3-2 Excerpts from the code behind the WebForm1.aspx page.

The event handler code that you write runs on the server.

The event handler code that you write runs on the server, not the user's browser. That's why Web Forms controls are often known as "server controls." When the user clicks the Get Time button or makes a selection from the drop-down list, the page is posted back to the server. ASP.NET catches the incoming page, detects which control event triggered the postback, and invokes the handler code for that event. This causes the page to be regenerated, with whatever changes the event handler code might have made to it, and written back to the client. You can read more about this process in the next section of this chapter. If you'd like even more detail, you can download the chapter on Web Forms controls from this book's Web site, *<http://www.introducingmicrosoft.net>*.

Now that I've finished writing my code, I'd like to see it in action. The easiest way to do this is to right-click on the page in Solution Explorer and select Build And Browse from the shortcut menu. This opens a browser window within the Visual Studio .NET environment. When this browser (or any other client) requests the .ASPX page, IIS loads it into the ASP.NET execution engine, which then parses the page. The first time the engine encounters the page it compiles its code "just-in-time." Subsequent requests for the same page will load and run the compiled code. The engine then executes the class's compiled code, creating the Web controls that I earlier placed on the form. The controls run only on the server side, rendering HTML that reflects their current state, which is incorporated in the page that gets sent to the client. The engine also executes the event handlers' code and renders the output HTML created by the code's interaction with the controls. The final HTML is written back to the client, which produces the page shown previously in Figure 3-4.

That's all I had to do to write an .ASPX page using Visual Studio .NET and Web controls. It's much easier than original ASP. It looks like and feels like writing Visual Basic code, which we're already familiar and comfortable with.

An execution engine executes the class associated with the ASP.NET page and renders HTML from its controls and code.

Tips from the Trenches

My clients report that the key to most user interface design problems is locating a third-party Web Forms control that provides the prefabricated behavior that they are looking for. Often they'll change their user interface requirements to conform to the controls they can buy off the shelf, rather than spend weeks writing code that would provide slightly different behavior.

More on Web Controls

Web controls spring from the same philosophy that led to the creation of Windows user interface controls 15 years ago. That architecture made Bill Gates the world's richest man (even after the recent tech stock slide, according to Forbes.com). That's my definition of a successful architecture; don't know about yours. Microsoft stock must be holding more of its value.²

2. As opposed to Sun, whose share price today will just barely buy you a good cup of coffee, as long as you don't want too many extras in it.

90 Introducing Microsoft .NET, Third Edition

Controls exist to encapsulate reusable program logic dealing with user interfaces.

Web controls are richer, more numerous, and easier to program than standard HTML controls.

Web controls exist for many different functions.

Input and output operations are highly repetitive. Prefabricating them—rolling them up in a capsule for any programmer to use—is a great idea. You kids today don't know how lucky you are not to have to worry about writing your own button controls, for example, and painting the pixels differently to make the button look as if it's been clicked. This not only saves an enormous amount of expensive programming time, but also makes every program's user interface more consistent and thus easier for users to understand. Remember (I'm dating myself here) Windows 3.0, which didn't contain a standard File Open dialog box? Every application programmer had to roll her own. It cost a lot, and every application's implementation looked and worked a little (or a lot) differently. Windows 3.1 provided a File Open dialog box as part of the operating system, which made the lives of both programmers and users much easier. A Web control, in the ASP.NET sense of the word, is any type of user-interface-related programming logic that is capable of a) exposing a standard programming model to a development environment such as Visual Studio .NET, and b) rendering its appearance into HTML for display in a standard browser—more or less as a Windows control renders its appearance into Windows GDI function calls.

“But HTML already *has* controls,” you say. “Buttons and checkboxes and links. I use them all the time. Why do I have to learn a new set?” HTML does support a few controls as part of the language, but these have severe limitations. First, they're not very numerous or very rich. A text box and a drop-down list box are about as far as they go. Compare their number to the controls advertised in *Visual Studio* magazine (formerly *Visual Basic Programmers Journal*) or their functionality to Visual Basic's data-bound grid control. We'd like to be able to package a lot more functionality than the few and wimpy existing HTML elements provide. Second, they are hard to write code for. The communication channel between an HTML control and its run-time environment isn't very rich. The Web Forms environment that Web controls inhabit contains an event-driven programming model similar to Visual Basic. It's much easier to program, there's much better support in development environments, and it abstracts away many of the differences from one browser to another. Web controls do more and are easier to program. Sounds decisive to me.

ASP.NET comes with the set of basic Web controls listed in Table 3-1. In addition, third-party vendors have written many controls for sale. Microsoft's official ASP.NET Web site, <http://www.asp.net>, lists several hundred in its control gallery, and a Google search turns up dozens more. You can also write your own, which isn't very hard. I discuss writing your own in a chapter you can download for free from <http://www.introducingmicrosoft.net>.

Table 3-1 Web Forms Server Controls by Function

Function	Control	Description
Text display (read only)	Label	Displays text that users can't directly edit.
Text edit	TextBox	Displays text entered at design time that can be edited by users at run time or changed programmatically. Note: Although other controls allow users to edit text (for example, DropDownList), their primary purpose is not usually text editing.
Selection from a list	DropDownList	Allows users to either select from a list or enter text.
	ListBox	Displays a list of choices. Optionally, the list can allow multiple selections.
Graphics display	Image	Displays an image.
	AdRotator	Displays a sequence (predefined or random) of images.
Value setting	CheckBox	Displays a box that users can click to turn an option on or off.
	RadioButton	Displays a single button that can be selected or not.
Date setting	Calendar	Displays a calendar to allow users to select a date.
Commands	Button	Used to perform a task.
	LinkButton	Like a Button control but has the appearance of a hyperlink.
	ImageButton	Like a Button control but incorporates an image instead of text.
Navigation controls	HyperLink	Creates Web navigation links.
Table controls	Table	Creates a table.
	TableCell	Creates an individual cell within a table row.
	TableRow	Creates an individual row within a table.
Grouping other controls	CheckBoxList	Creates a collection of check boxes.
	Panel	Creates a borderless division on the form that serves as a container for other controls.
	RadioButtonList	Creates a group of radio buttons. Inside the group, only one button can be selected.
List controls	Repeater	Displays information from a data set using a set of HTML elements and controls that you specify, repeating the elements once for each record in the data set.
	DataList	Like the Repeater control but with more formatting and layout options, including the ability to display information in a table. The DataList control also allows you to specify editing behavior.

Table 3-1 Web Forms Server Controls by Function

Function	Control	Description
Place holding	DataGrid	Displays information, usually data-bound, in tabular form with columns. Provides mechanisms to allow editing and sorting.
	Placeholder	Enables you to place an empty container control in the page and then dynamically add child elements to it at run time.
	Literal	Renders static text into a Web page without adding any HTML elements.
	XML	Reads XML and writes it into a Web Forms page at the location of the control.

The execution engine creates and uses the controls on the server side. The controls render their own HTML for the client.

When I placed controls on my form in the preceding example, Visual Studio .NET generated the statements in the .ASPX page, shown in Listing 3-3. Every statement that starts with `<asp: >` is a directive to the ASP.NET parser to create a control of the specified type when it generates the class file for the page. For example, in response to the statement `<asp:label>`, the parser creates a label control in the class. When the page is executed, the ASP.NET execution engine executes the event handler code on the page that interacts with the controls (fetching the time and setting the text and background color, as shown previously in Listing 3-2). Finally, the engine tells each control to render itself into HTML in accordance with its current properties, just as a Windows control renders itself into GDI calls in accordance with its current properties. The engine then writes the resulting HTML back to the client's browser. Figure 3-7 shows this process schematically. Excerpts from the actual HTML sent to the client are shown in Listing 3-4.

```
<%@ Page Language="vb" AutoEventWireup="false"
Codebehind="WebForm1.aspx.vb" Inherits="SimplestASPVb.WebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>
    <title>WebForm1</title>
    <meta name="GENERATOR" content="Microsoft Visual Studio.NET 7.0">
    <meta name="CODE_LANGUAGE" content="Visual Basic 7.0">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
          content="http://schemas.microsoft.com/intellisense/ie5"
    </HEAD>
    <body MS_POSITIONING="GridLayout">
      <form id="Form1" method="post" runat="server">
```

Listing 3-3 Excerpts from .ASPX page created by Visual Studio .NET, showing control statements.

```

    <asp:Label id="Label1"
    style="Z-INDEX: 101; LEFT: 32px; POSITION: absolute; TOP: 24px"
    runat="server">Time will be shown here</asp:Label>
    <asp:DropDownList id="DropDownList1"
    style="Z-INDEX: 102; LEFT: 136px; POSITION: absolute; TOP: 120px"
    runat="server" AutoPostBack="True">
        <asp:ListItem Value="Black">Black</asp:ListItem>
        <asp:ListItem Value="Red">Red</asp:ListItem>
        <asp:ListItem Value="Green">Green</asp:ListItem>
        <asp:ListItem Value="Blue">Blue</asp:ListItem>
    </asp:DropDownList>
    <asp:Button id="Button1"
    style="Z-INDEX: 103; LEFT: 184px; POSITION: absolute; TOP: 72px"
    runat="server" Text="Get Time"></asp:Button>
    <asp:CheckBox id="CheckBox1"
    style="Z-INDEX: 104; LEFT: 32px; POSITION: absolute; TOP: 72px"
    runat="server" Text="Show seconds?"></asp:CheckBox>
    <asp:Label id="Label2"
    style="Z-INDEX: 105; LEFT: 32px; POSITION: absolute; TOP: 120px"
    runat="server">Text color:</asp:Label>
</form>
</body>
</HTML>

```

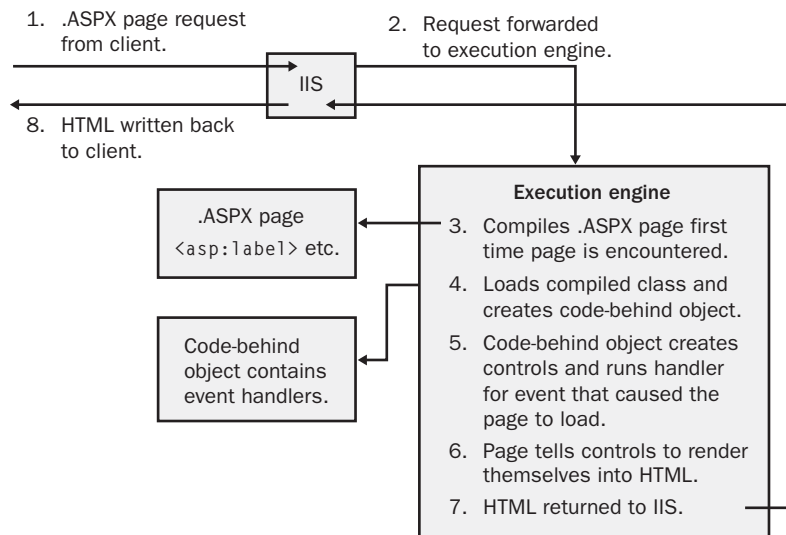


Figure 3-7 .ASPX page execution engine sequence.

```

<span id="Label1" style="color:Black;">
    (time will be displayed here)
</span>

<span>
<input type="checkbox" id="CheckBox1" name="CheckBox1" />
<label for="CheckBox1">Show Seconds ?</label>
</span>

<input type="submit" name="Button1" value="Get Time" id="Button1" />

<p>Text Color:

<select name="DropDownList1" id="DropDownList1"
    onchange="javascript: __doPostBack('DropDownList1','')">
    <option selected value="Black">Black</option>
    <option value="Red">Red</option>
    <option value="Green">Green</option>
    <option value="Blue">Blue</option>
</select>

```

Listing 3-4 Excerpts from the HTML generated by the controls.

Input controls can maintain their contents and selection from one round-trip to another. Display controls can also maintain their state from one round-trip to another.

Another great feature of the input controls in the Web controls package (list box, text box, check box, radio button, and so on) is that they can automatically remember the state in which the user left them when the page they are on makes a round-trip to the server. Microsoft calls this feature *postback data*. Observe the behavior of the check box control in the simplest example that I showed in the previous section. The check box correctly maintains its checked or unchecked state when the page is posted to the server for event handling and returned to the client browser afterward. I didn't have to write any code to get this behavior, as I would have had to in classic HTML. Input Web controls automatically remember their state. During the postback operation, the data actually resides in a dedicated field in the HTTP header of the page. This is an automatic feature that you can't turn off.

The display controls in the Web controls package, such as label, data list, data grid, and repeater, support their own version of property retention, called *view state*. Even though the user doesn't set them to specific values, they still remember the state in which the program left them in the previous round trip. The .ASPX page environment automatically places a hidden input control called `__VIEWSTATE` on each of its pages. Web controls automatically serialize their state into this hidden control when the page is being destroyed

and then retrieve their state when the page is next created, which is what you want most of the time. If you don't want this, you easily can turn it off by setting the control's *MaintainState* property to *False*.

Why does Microsoft use two separate mechanisms for maintaining control state, one for input controls and the other for display controls? Probably to make it easier for programmers to comply with the Principle of Least Astonishment, which states simply that astonishing a human user is not a good thing, therefore you should do it as seldom as possible. When a user types something into a field, she expects to see what she typed stay there until she does something that changes it. If an input control didn't automatically maintain its state, a developer would have to write code to make it do so or astonish the heck out of his users. Since you have to do this all the time, Microsoft built it into the input control behavior and didn't provide an off switch. On the other hand, a display control often doesn't display what a user typed in. Instead, it usually shows the results of an internal computing operation. Sometimes you might want a display control to remember its state between operations, and sometimes you might not. Hence a different mechanism, this one with an off switch.

Web controls can also discover the specific browser on which the user will be viewing the page so that they can render their HTML to take advantage of different browser features if they want to. A good example of this is the range validator control, shown in Figure 3-8. Ensuring that a user-entered number falls between a certain maximum and minimum value is such a common task that Microsoft provides a Web control for it. You place the control on a form and set properties that tell it which text box to validate and what the maximum and minimum values are. When the control renders its HTML, it checks the version of the browser it's running on. If it's a newer browser that supports DHTML (specifically MSDOM 4.0 or later, and EcmaScript version 1.2 or later), the control renders HTML that includes a client-side script that will perform the numeric validation in the user's browser before submitting the form to the server, thereby avoiding a network round-trip if a control contains invalid data. If all the data passes validation on the client, the form is submitted to the server. Somewhat counterintuitively, the validation operation is then repeated on the server, even though it passed on the client. This guarantees that the validation is performed before your server-side code runs, even if the control somehow messed up the client-side script on a flaky browser. If the control detects an older browser that doesn't support DHTML, it automatically generates HTML code that performs a round-trip and validates on the server.

Web controls can detect the capabilities of the browser they are running on and render different code to take advantage of them.

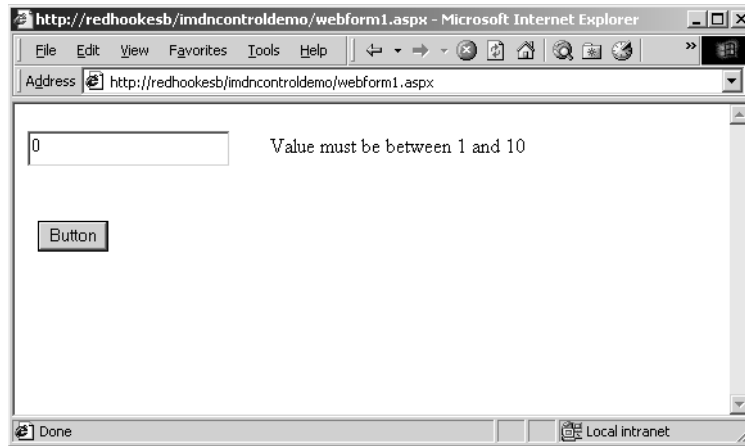


Figure 3-8 The range validator control.

You can write your own Web controls without too much trouble.

Writing your own Web controls isn't that hard because the .NET Framework contains prefabricated base classes from which you can inherit most of your control's necessary infrastructure (sort of like MFC on steroids). I show several examples in a chapter about Web controls that's available for free download at <http://www.introducingmicrosoft.net>.

Tips from the Trenches

The process of creating controls, loading their state, running their code, and rendering their state obviously takes some amount of time. ASP.NET supports a number of different options for caching page output pages to reduce this time and improve throughput. You can cache an entire page if you want, but you can also cache the output of individual controls. You do this by adding the *Vary-ByControl* attribute to the `<%@ OutputCache %>` directive in the page's .aspx file.

Managing and Configuring Web Application Projects: The Web.config File

A powerful run-time environment like ASP.NET requires excellent configuration management.

A powerful run-time environment like ASP.NET provides many prefabricated services. As you can imagine, an individual application has many, many options for its configuration. The configuration mechanism of an environment

is as important as the underlying software that it controls. Cool features that you can't configure, or can't figure out how to configure, are about as useful as unsecured Enron bonds.

Previous run-time environments have used a central store for their configuration information. For example, classic COM used the system registry and COM+ added the COM+ catalog. ASP.NET takes a different, decentralized approach. Each application stores its ASP.NET configuration management information in a file named `web.config`, which is stored in the application's own directory. Each file contains configuration information expressed in a particular XML vocabulary. Excerpts from our sample program's `web.config` file are shown in Listing 3-5. This excerpt shows the compilation and custom error handling settings of this particular application. I will discuss other settings later in this chapter.

ASP.NET keeps its configuration information in individual files, each having the name `web.config`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>

    <!-- DYNAMIC DEBUG COMPILATION
    Set compilation debug="true" to insert debugging symbols
    (.pdb information) into the compiled page. Because this
    creates a larger file that executes more slowly, you should
    set this value to true only when debugging and to false at
    all other times. For more information, refer to the
    documentation about debugging ASP.NET files.
    -->
    <compilation defaultLanguage="vb" debug="true" />

    <!-- CUSTOM ERROR MESSAGES
    Set customErrors mode="On" or "RemoteOnly" to enable
    custom error messages, "Off" to disable. Add <error>
    tags for each of the errors you want to handle.
    -->
    <customErrors mode="RemoteOnly" />

  </system.web>

</configuration>
```

Listing 3-5 Excerpts from the sample application's `web.config` file.

`Web.config` files can also reside in various subdirectories of an application. Each `web.config` file governs the operation of pages within its directory and all lower subdirectories unless overridden. Entries made in a lower-level

A setting made in a `web.config` file overrides the settings made in directories above it.

web.config file override those made in levels above them. Some users find this counterintuitive; some don't. The master file specifying the defaults for the entire ASP.NET system has the name machine.config and lives in the directory [system, e.g. WINNT]\Microsoft.NET\Framework\[version]\Config. Entries made in web.config files in the root directories of an individual Web application override entries made in the master file. Entries made in web.config files in subdirectories of an application override those made in the root. This relationship is shown in Figure 3-9.

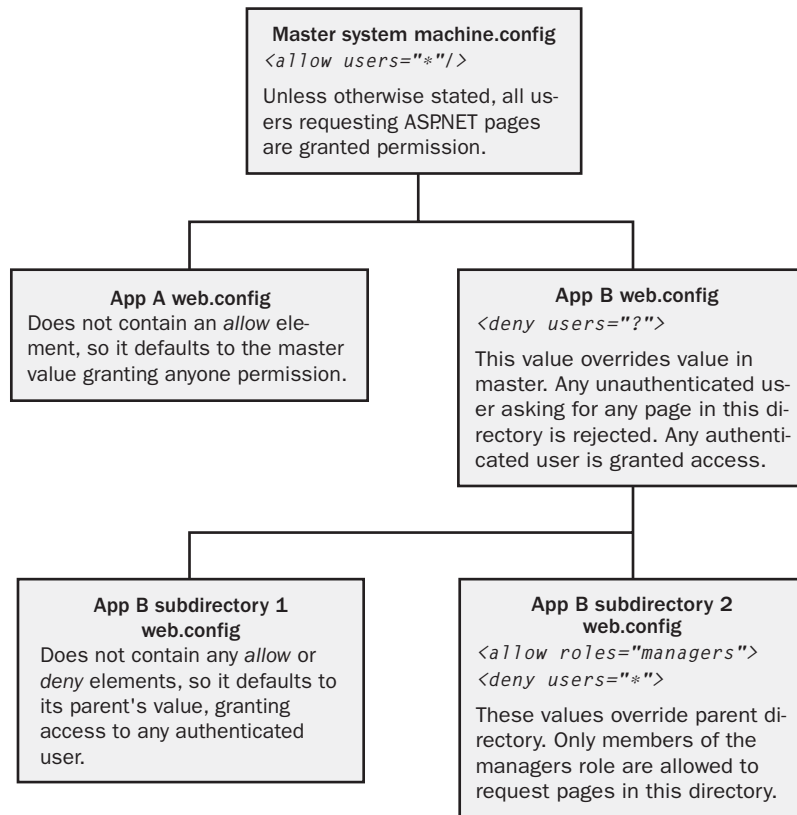


Figure 3-9 The hierarchical nature of web.config files.

Not all sections of the web.config file are configurable on all levels of subdirectory. The <sessionstate> section, for example, may not appear below an application's root directory. You'll have to examine the details of individual sections to find out the granularity of each.

Microsoft still has not provided a decent tool for maintaining the web.config file, even in version 1.1. You can edit the raw file using any XML editor, but it still falls far below the ease of using the .NET Framework configuration utility described in Chapter 2, or any other administrative tool in the Windows 2000 environment.

Third-party tools exist to make administration easier.

Tips from the Trenches

While I find this lapse inexplicable, third-party products solve the problem quite well. My favorite is the Web.Config Editor from Hunter-Stone (<http://www.hunterstone.com>), shown in Figure 3-10. You'll recover the \$44 they charge for it in the first hour of head-banging it helps you avoid.

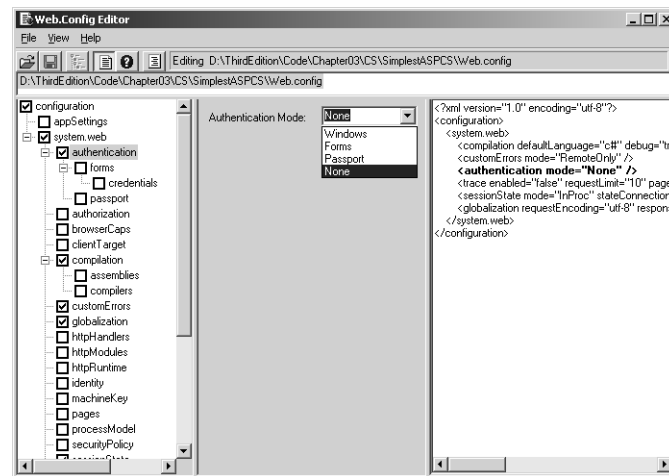


Figure 3-10 The Web.Config Editor from Hunter-Stone.

ASP.NET State Management

Web page requests are by default independent of each other. If I request page B from a server, the contents of page B don't know whether I have or haven't viewed page A a few minutes ago. This arrangement works well for simple read-only requests, such as the movie theater example I discussed in the open-

Internet page requests from a single user didn't originally know about each other. Sometimes this is OK.

100 Introducing Microsoft .NET, Third Edition

ing paragraph of this chapter. It's easy to write a server that does this because you don't have to store any data from one page request to another.

Then again, as Web interactions get richer, sometimes it isn't.

As Web interactions get more sophisticated, however, this design no longer meets your users' needs. What I did on page A *does* affect the content that page B should show. For example, I book a lot of air travel directly on airline Web sites because they offer extra frequent flyer miles. It's not acceptable to me to search for an outgoing flight, write it down on paper, search for a return flight, write it down as well, and then manually type both of these flight numbers into yet another page for purchasing my ticket. I want the airline's site to automatically remember my outgoing flight selection while I choose a return flight and then remember both of these while I buy the ticket.

A Web programmer often must maintain separate data for many users simultaneously.

Remembering data from one form to another was never a problem for a desktop application, which targeted a single user. The programmer could simply store input data in the program's internal memory without worrying about which user it belonged to, since only one person used the program at a time. But keeping track of a user's actions over multiple pages is much more difficult in a Web application, used by (you hope) many different people simultaneously. A Web programmer needs to keep my data separate from all the other users' so that their flights don't get mixed with mine (unless they're headed to Hawaii in February, in which case, hooray!). We call this *managing session state*, and the Web programmer needs some efficient, easy-to-program way of doing it.

ASP.NET provides a set of data tied to a specific user. Called a *Session*, programmers can use this object to store data for a specific user.

Original ASP provided a simple mechanism for managing session state, which ASP.NET supports and extends. Every time a new user accesses an .ASPX page, ASP.NET creates an internal object called a *Session*. This object is an indexed collection of data living on the server and tied to a particular active user. You can access items in the collection by means of a numerical index or a string name that you specify. The *Session* object can hold any number of items for each user, but since they all use server memory, I advise you to limit the amount of data you store in session state to the minimum necessary.

The *Session* object automatically remembers (placing a unique ID in a browser cookie or appending it to the URL) which user the data refers to, so the programmer doesn't have to write code to do that. For example, when I submit a form selecting a flight, the .ASPX page programmer can store that flight's information in the *Session* object. When I request the page to buy my ticket, the programmer reads the flight from my *Session* object. The .ASPX run time automatically knows who the user is (me), so the programmer's code automatically fetches my flights and not someone else's, as shown in Figure 3-11.

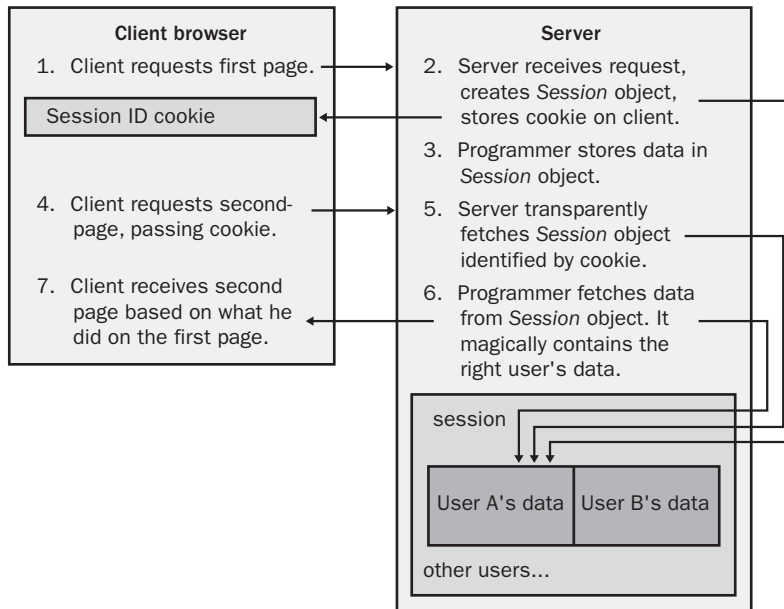


Figure 3-11 Managing session state.

This feature is easy for programmers to use, and hence quite popular. Each .ASPX page contains a property named *Session* that provides access to the session object for the current user. You place items in this object using an integer index or a string key, and retrieve them the same way. If you attempt to fetch an item that doesn't exist in the session object, you'll get a null object reference in return. A sample of the code that does this is shown in Listing 3-6. You will also find this sample on this book's Web site. The pages will look like Figure 3-12.

ASP.NET provides a simple API for setting and getting session state variables.

```
' Code that stores data in session state.

Protected Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button1.Click

    ' Store current text string in session state.

    Session("DemoString") = TextBox1().Text

    ' Redirect user to page for viewing state.

    Response().Redirect("WebForm2.aspx")
```

Listing 3-6 Code for session state management.

102 Introducing Microsoft .NET, Third Edition

```

End Sub

' Fetch designated string from session state object
' Display to user in label

Private Sub Page_Load(ByVal sender As System.Object, _
                      ByVal e As System.EventArgs) Handles MyBase.Load

    Dim str As String
    str = Session("DemoString")

    If (Not str Is Nothing) Then
        Label1.Text = str
    Else
        Label1.Text = ("(the item ""DemoString"" does not exist " & _
                       "in the session collection)")
    End If

End Sub

```

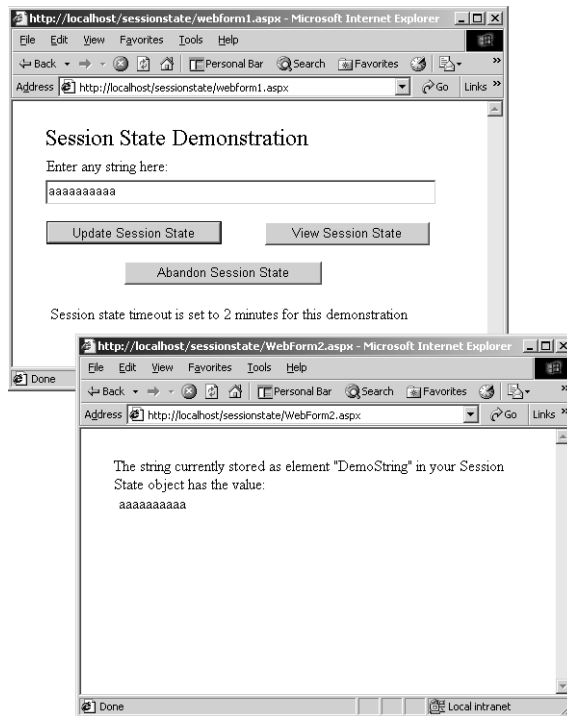


Figure 3-12 Session state management sample application.

Obviously, if a server were to maintain a permanent session for every user that ever viewed even one page on it, you'd run out of memory very quickly. The session mechanism is designed to maintain state only for active users—those who are currently performing tasks that require the server to maintain state for them. ASP.NET automatically deletes a user's *Session* object, dumping its contents, after it has been idle for a configurable timeout interval. This interval, in minutes, is set in the `<sessionstate>` section of the web.config file, shown in Listing 3-7. The default is 20 minutes. You can also dump the session yourself by calling the method *Session.Abandon*.

ASP.NET automatically deletes sessions after a configurable timeout interval.

```
<sessionState
  mode="inproc"
  stateConnectionString="tcpip=127.0.0.1:42424"
  sqlConnectionString="data source=127.0.0.1;user id=sa;password="
  cookieless="false"
  timeout="20"
/>
```

Listing 3-7 Session state management entries in the web.config file.

While the session mechanism in original ASP was easy to use, it had a number of drawbacks that hampered its expansion to large-scale systems. First, it stored session state in the worker processes that actually ran page scripts and called code living in programmer-written custom objects. A badly behaved object could and often did crash this process, thereby killing the session state of every user it was currently serving, not just the one whose call caused the crash. ASP.NET fixes this problem by providing the ability to store session state in a separate process, one that runs as a system service (see Figure 3-13), so badly behaved user code can't kill it. This means that worker processes can come and go without losing their session state. It slows down the access to the session state somewhat, as applications now need to cross process boundaries to get to their session state, but most developers figure that's worth it for reliability. To turn this feature on, set the *mode* attribute in the web.config file to *stateserver* and ensure that the state server process is running on the server machine.

Session state can be stored in a separate process for robustness.

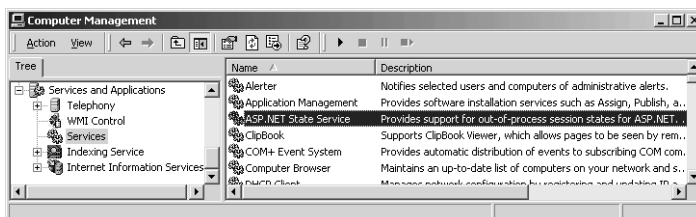


Figure 3-13 ASP.NET session state process running as a system service.

104 Introducing Microsoft .NET, Third Edition

Session state can be easily stored on a different machine.

You can also store session state in a SQL Server database.

ASP.NET also provides another state collection for data at an application level.

Original ASP always stored session state on the server machine on which it was created. This architecture didn't scale well to a Web farm in which every request is potentially handled by a different server. ASP.NET allows each application to specify the machine on which it wants its session state to be stored. You do this by setting the *stateConnectionString* attribute in the web.config file. In this way, any machine that handles a subsequent call can access the session state stored by a previous call handled by a different machine. Obviously, you now incur the overhead of another network round-trip, so maybe you'd rather set up your load balancer to route subsequent requests to the same machine. You can swap out that machine by simply changing its name in the configuration files of all the clients.

You can also store session state in SQL Server for better management of large collections. You do this by setting the *mode* attribute to *sqlserver* and providing a SQL connection string in the *sqlConnectionString* attribute. The current setup scripts create a temporary database for holding session state. This provides fast access because the data isn't written to the slow iron disk, but it also means that the session state won't survive a crash. If you want that durability and don't mind paying the performance penalty for it, you can change the database attributes yourself to use a permanent table.

ASP.NET also helps you manage application state, where an application represents a top-level IIS virtual directory and all of its subdirectories. Application-level state is stuff that changes from time to time, so you don't want to hard code it, but it applies to all users of the application and isn't tied to an individual user. An example of application state might be the current unadvertised special offer that every page would display. Each application contains one state object named *Application*, similar in use to the *Session* object, except that all accesses of the *Application* object operate on the same data regardless of the user on the other end.

Tips from the Trenches

My customers report that they can get a significant performance boost in heavily loaded systems by turning off session state for pages that don't need it, by adding the attribute *EnableSessionState="false"* to the *@page* directive in the raw .ASPX file.

Security in ASP.NET

Security is vital to any type of distributed programming, and discussions of it are often highly charged with emotion. I will attempt to outline the problems that arise in this area and discuss how ASP.NET provides your Web application with prefabricated functionality that gives you the level of security you need without too much programming effort.

The security requirements of a Web application are somewhat like those of a city hall. You have large numbers of people coming and going anonymously, accessing public areas such as the tourism office that dispenses maps. Because these areas aren't sensitive, you don't want to waste time and annoy users by subjecting them to strip searches, or else you won't get many customers and the businesses that sponsor the map giveaways will be angry. But other areas of the same city hall, such as the floor containing the mayor's office, are sensitive. You don't want to allow anyone in who can't prove their identity (authentication) and that they have business on the floor (authorization). And you might want to run them through a metal detector to make sure they can't hurt anybody.

Security is vital to any distributed system.

Different areas of an application require different levels of security.

Authentication

The first problem in security is authentication—Who are you, and how do I know you really are that person? Authenticating a user usually takes the form of examining some sort of credential presented by the user, sometimes agreed upon directly between the two parties, such as a PIN or password; sometimes issued by a third party that both trust, such as a driver's license or passport. If the credentials are satisfactory to the server, the server knows who the user is and can use its internal logic to determine what actions she is allowed to perform. A user who is not authenticated is called an anonymous user. That doesn't necessarily mean that she can't have access to anything on the Web site. It means that she can have access only to the features that the designers have chosen to make available to anonymous users—perhaps checking an airline schedule but not redeeming a frequent flyer award.

Authentication has historically been the most difficult problem in security design. Most application designers don't want to deal with it because it's so important but so difficult to get right. You need a full-time team of very smart geeks who do nothing but eat, drink, and sleep security because that's what the bad guys have who are trying to crack your Web site and weasel

Securely and definitively identifying a user is called authentication.

An authentication system is difficult and expensive to build.

free first-class upgrades (or worse). For example, you can't just send a password over the network, even if it's encrypted. If the network packet containing an encrypted password doesn't somehow change unpredictably every time you submit it, a bad guy could record it and play it back. That's how I broke into the foreign exchange system of a certain bank whose name you would recognize—fortunately with the bank's permission—under a consulting contract to test their security. They were very proud of their password encryption algorithm, which I never even tried to crack, but it took me only 20 minutes with a packet sniffer to record a user ID/password packet and play it back to enter their server. Too bad I was charging them by the hour. Next time I'll vacation for a billable week before declaring success. If you'd like more information about the design problems of writing secure applications, I recommend the book *Writing Secure Code, Second Edition*, by Michael Howard and David LeBlanc (Microsoft Press, 2003). I never even imagined half the ways to crack a system that these guys discuss.

Fortunately, several different types of authentication come built into ASP.NET

Because of the difficulty and importance of authentication, it's one of the first features that gets built into a (hopefully) secure operating system. ASP.NET supports three different mechanisms for authentication (four if you count "none"). They are shown in Table 3-2.

Table 3-2 ASP.NET authentication modes

Name	Description
None	No ASP.NET authentication services are used.
Windows	Standard Windows authentication is used from IIS.
Forms	ASP.NET requires all page request headers to contain a cookie issued by the server. Users attempting to access protected pages without a cookie are redirected to a login page that verifies credentials and issues a cookie.
Passport	Same idea as Forms, except that user ID info is held and cookies are issued by Microsoft's external Passport authentication service.

You need to think carefully about which of your site's resources require authentication for access and which don't.

You need to think carefully about exactly where in your Web site authentication should take place. As with a city hall, you need to balance security versus ease of use. For example, a financial Web site will require authentication before viewing accounts or moving money, but the site probably wants to make its marketing literature and fund prospectuses available to the anonymous public. It is important to design your site so that areas that require security have it, but you don't want this security to hamper your unsecure

operations. I have a hard time thinking of anything more detrimental to sales than making a user set up and use an authenticated account before allowing him to see a sales pitch, although some sites do just that.

Note Unlike DCOM, which used its own wire format to support packet privacy, none of the authentication schemes available to ASP.NET provide for encryption of the data transmitted from client to server. This problem doesn't come from ASP.NET itself, but from the common Web transport protocol HTTP. If your Web site provides data that you don't want on the front page of *USA Today*, you need to use the Secure Sockets Layer (SSL) transport or provide some other mechanism for encrypting. Typically, you will do this only for the most sensitive operations because it is slower than unsecure transports. For example, an airline Web site will probably use encrypted transport for the credit card purchase page but not for the flight search page.

Windows Authentication

ASP.NET supports what it calls *Windows-based authentication*, which basically means delegating the authentication process to IIS, the basic Web server infrastructure on which ASP.NET sits. IIS can be configured to pop up a dialog box on the user's browser and accept a user ID and password. These credentials must match a Windows user account on the domain to which the IIS host belongs. Alternatively, if the client is running Microsoft Internet Explorer 4 or higher on a Windows system and not connecting through a proxy, IIS can be configured to use the NTLM or Kerberos authentication systems built into Windows to automatically negotiate a user ID and password based on the user's current logged-in session.

Windows authentication works quite well for a Windows-only intranet over which you have full administrative control. For certain classes of installation—say, a large corporation—it's fantastic. Just turn it on and go. But it's much less useful on the wide-open Internet, where your server wants to be able to talk to any type of system, (say, a palmtop) using any type of access (say, not Internet Explorer), and where you don't want to set up a Windows login account for every user.

Windows authentication works well on a Windows-only intranet.

Forms-based authentication starts with a user ID and password.

The server supplies the browser with a cookie, an admission ticket identifying the authenticated user.

Forms-Based, or Cookie, Authentication

Most designers of real-life Web applications will choose *forms-based authentication*, otherwise known as *cookie authentication*. The user initially sets up an account with a user ID and password. Some Web sites, such as most airline sites, allow you to do this over the Web through a page open to anonymous access. Other Web sites, such as most financial services companies, require a signed paper delivered via snail mail.

When the user first requests a page from a secure Web site, he is directed to a form that asks for his ID and password. The Web server matches these against the values it has on file and allows access if they match. The server then provides the browser with a cookie that represents its successful login. Think of this cookie as the hand stamp you get at a bar when they check your ID and verify that you really are over 21 years old. It contains the user's identification in an encrypted form. The browser will automatically send this cookie in the request header section of every subsequent page request so that the server knows which authenticated user it comes from. This relay keeps you from having to enter a user ID and password on every form submittal or page request. Figure 3-14 illustrates this process.

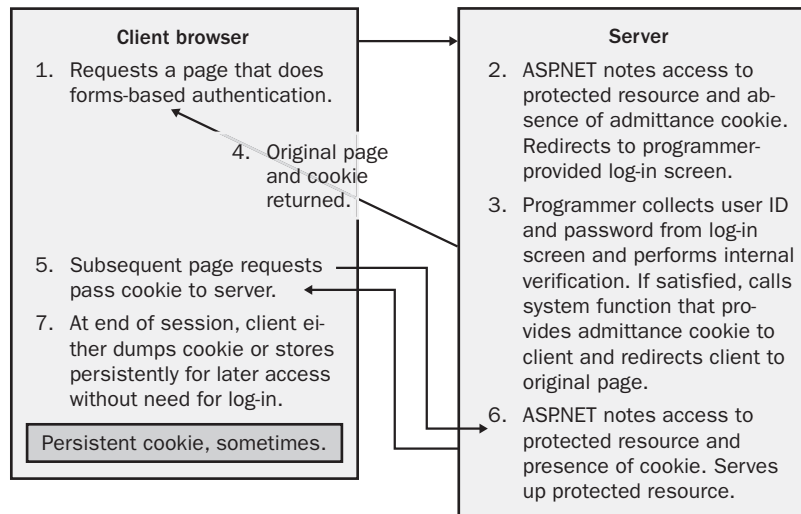


Figure 3-14 Forms-based authentication.

When it supplies the cookie, the server specifies whether the cookie is to be thrown away when the browser session ends or should be saved on the user's hard disk so that he won't have to log in next time. Financial Web sites often require the former in order to be ultraprojective against unauthorized use that could cost thousands of dollars. The latter, however, is obviously much more convenient for users. Most Web sites whose data isn't very sensitive, on-line periodicals such as the *Wall Street Journal* for example, often provide the user with the capability to choose it. A sample persistent cookie is shown in Figure 3-15.

Cookies may be temporary or persistent.

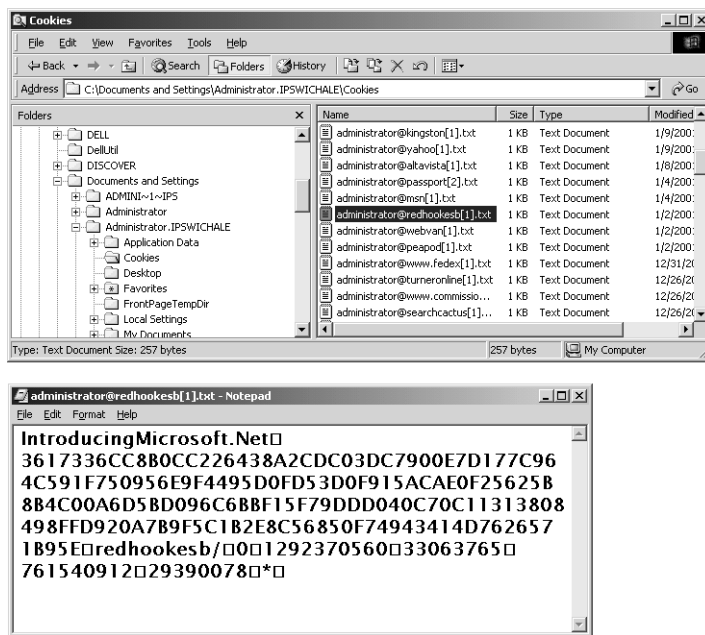


Figure 3-15 Persistent cookie created by sample application.

ASP.NET contains good support for forms-based authentication. A sample application is available on this book's Web site and shown in Figure 3-16. You tell ASP.NET to use forms-based authentication by making an entry in the web.config file as shown in Listing 3-8. The *authorization* element tells ASP.NET to deny access to any unauthenticated users.

ASP.NET contains good prefabricated support for forms-based authentication.

110 Introducing Microsoft .NET, Third Edition

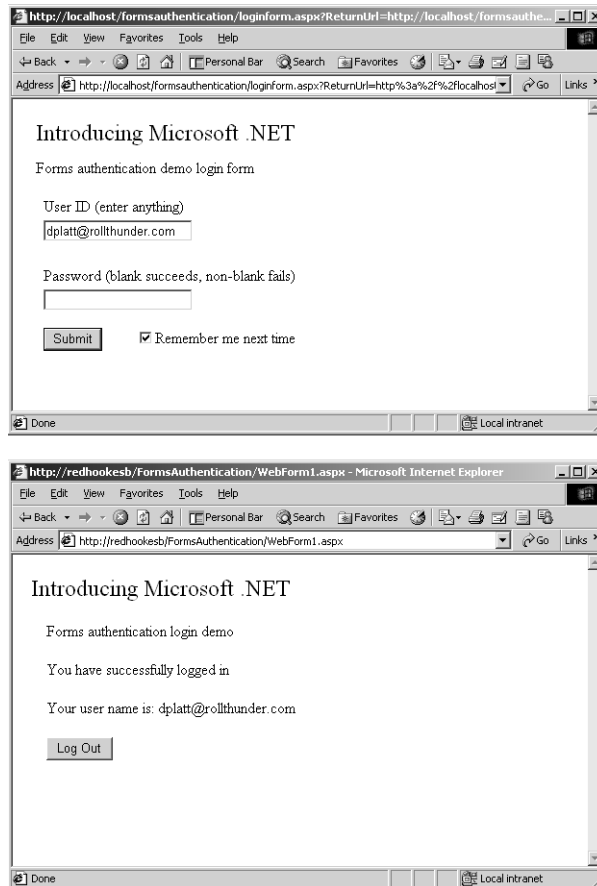


Figure 3-16 Sample application for forms-based authentication.

```
<authentication mode = "Forms" >
  <forms name="IntroducingMicrosft.NET" loginUrl="loginform.aspx"/>
</authentication>

<authorization>
  <deny users="?"/>
</authorization>
```

Listing 3-8 Web.config file for forms-based authentication.

This means that ASP.NET will require that any request for a page in that directory or its subdirectories must contain a cookie from the browser saying that you've authenticated the user and found him worthy. If the cookie is not present, as it won't be prior to your first login, ASP.NET directs the user to the page you have specified in the *loginUrl* attribute. You use this page to pro-

vide controls for the user to enter his ID and password. The page's handler contains whatever logic you need for checking that the user really is who he says he is. The sample code simply accepts a blank user password and rejects any that are nonblank. If the server accepts the incoming user, a simple function call (to *System.Web.Security.FormsAuthentication.RedirectFromLoginPage*) brings the user back to the page that he was trying to access when the authentication scheme shunted him to the login screen. If you'd like to send him somewhere else instead, you can use the function *Response.Redirect*. The code is shown in Listing 3-9.

```
Public Sub Button1_Click(ByVal sender As System.Object, _  
                        ByVal e As System.EventArgs)  
  
    ' Check for legal password (in this case, blank)  
  
    If TextBox2.Text = "" Then  
        System.Web.Security.FormsAuthentication.RedirectFromLoginPage( _  
            TextBox1.Text, CheckBox1.Checked)  
  
        ' Reject a non-blank password. Don't need to make any system calls,  
        ' just refrain from making the one that would grant access.  
  
    Else  
        Label5.Text = "Invalid Credentials: Please try again"  
    End If  
  
End Sub
```

Listing 3-9 Code for forms-based authentication form.

You can also set up ASP.NET to automatically perform forms-based authentication using user IDs and passwords stored in XML configuration files. This approach would probably work well for a small installation. Unfortunately, a demonstration is beyond the scope of this book, at least in this edition.

Tips from the Trenches

Forms authentication is the approach my customers use the most, by far. They report that they often want to change the default behavior described here; for example, redirecting the user to a different form after successfully authenticating him. You'll find methods for supporting these kinds of activities on the utility object *System.Web.Security.FormsAuthentication*.

As Web sites proliferate, so do the user IDs and passwords that a user must remember. This is a growing menace to the security of all data.

Passport Authentication

The third authentication alternative is Passport-based authentication. The forms-based scheme described in the previous section sounds fine, and it's a lot easier to write today than it used to be as the result of ASP.NET's prefabricated support. But it suffers from the fatal problem of unchecked proliferation. Every Web site with anything the least bit sensitive needs some sort of login security. For example, I use five different airline Web sites for booking all the travel I have to do, and because they take credit cards, every one of them requires authentication. It is a colossal pain in the ass to keep track of all the user names and passwords I use on these different sites. On one of them my name is "dplatt"; on another one that ID was taken when I signed up so I'm "daveplatt". I've tried using my e-mail address, which no one else ought to be using and which I can usually remember. This occasionally works, but many sites won't accept the @ sign and others won't accept a string that long. For passwords, some use 4-character PINs for identification, others use a password of at least six (or sometimes eight) characters, and one insists that the password contain both letters and numbers. Many sites consider their data to be so sensitive (my frequent flyer record? get a life) that they won't allow a persistent cookie to remember my login credentials. The only way I can keep track of all my user names and passwords is to write them down and keep them near my computer, where any thief would look for them first. That's one way Nobel physicist Richard Feynman broke into classified safes and left goofy notes for fun while working on the atomic bomb project at Los Alamos during World War II. (See his memoirs, *Surely You're Joking, Mr. Feynman*, for the other two ways. I guess Los Alamos has had this problem for a while.) As a client told me a decade ago, the greatest threat to security isn't the packet sniffer, it's the Post-It® note.

Microsoft Passport is a one-step secure login service.

Microsoft .NET Passport (<http://www.passport.com>) is an attempt to provide a universal one-step secure login procedure. It's very much like the forms-based authentication mechanism described in the preceding section, except that Microsoft's Passport Web servers handle the storing and verifying of user IDs and passwords. A user can go to the Passport Web site and create a passport, essentially a user ID and password stored on Microsoft's central server farm. When the user requests a page from a site that uses Passport authentication, that site looks for a Passport cookie in the request. If it doesn't find one, it redirects the request to Passport's own site, where the user enters her ID and password and receives a Passport cookie. Subsequent browser requests to any Passport-compliant site will use that cookie until the user logs out. The process is illustrated in Figure 3-17. A user can thus use the same user ID and password pair at any participating site. This scheme could greatly

simplify the proliferation problem and make many users' lives easier, with a net gain in security.

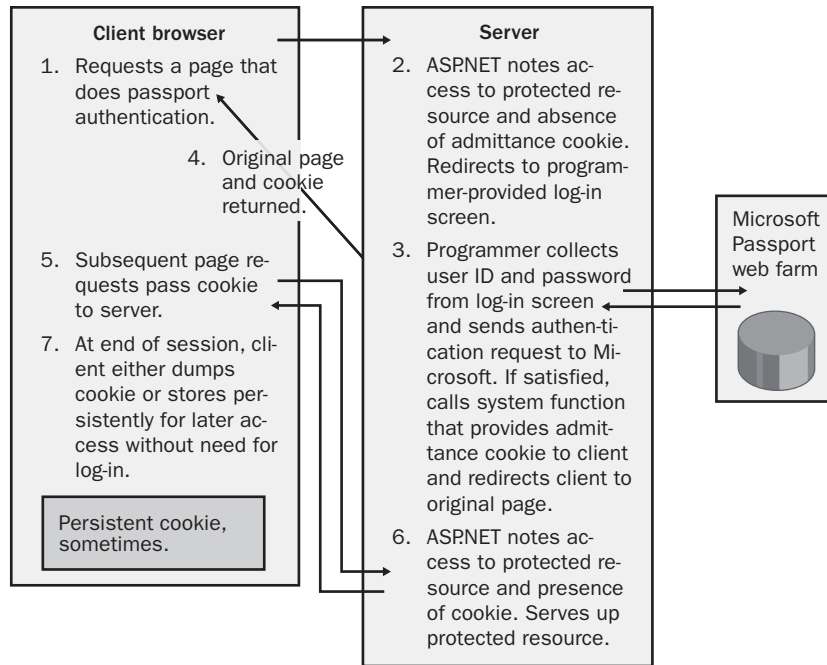


Figure 3-17 Passport authentication.

You would think that such a useful idea would be wildly popular, but Passport hasn't quite caught on yet outside of Microsoft. According to Passport's own Web site, only 90 sites support Passport authentication at the time of this writing (February 1, 2003), of which 21 belong to Microsoft and 12 represent various flavors of eBay.

Clearly, Passport hasn't crossed over to the mainstream yet. Part of the problem might be the cost, currently \$10,000 per year. Businesses might also be worried about outsourcing such an important piece of their online software presence. Since it's not under their control, how can they be sure it won't crash, or even slow below acceptable limits? How many would fear a repetition of January 23, 2001, when a boneheaded Microsoft technician made an incorrect DNS entry that blocked access to most Microsoft sites, including Passport, for almost a day? (I wonder if that technician had to use Notepad to edit a raw XML file, in which case the problem might have been avoided with a dedicated configuration tool.) And what about a falling out later? How could a Web company tell Microsoft to go to hell if Microsoft holds its login data?

Passport has been slow to catch on outside of Microsoft.

114 Introducing Microsoft .NET, Third Edition

Public distrust of Microsoft hinders widespread acceptance of Passport.

The basic problem hindering the advance of Passport is customers' willingness (or lack thereof) to allow Microsoft to hold their authentication data. I discussed Passport while teaching a class on .NET in a European country in September 2001. (Before.) I asked if any of the students' companies had considered using Passport for basic authentication. Students representing three separate banks said that they had looked at it for their customer Web sites, but decided not to use it. The student with the best English explained, the others nodding agreement, that "As banks, the basic commodity we sell is trust. Would you trust your brother to hold your money? No, [he must know my brother] but you trust us [he must NOT know my bank, whom I trust only because I owe them far more money in loans than they owe me in deposits]. We did research and focus groups, and found that an uncomfortable [he wouldn't quantify it] percentage of our customers would feel that we had betrayed their trust if we let Microsoft hold their authentication data instead of keeping it ourselves. They worry about it getting hacked, and they worry about it being used without their permission. I'm not necessarily saying that they're correct to worry about this, and I'm not necessarily saying that we're any better on our own, but like it or not, that IS how our customers told us they feel. So if we don't want them to think that we've done to them the worst thing that a bank can do, betray their trust, we can't use it, and that's the end of that."

Bill Gates has declared "Trustworthy Computing" to be Microsoft's top priority.

Microsoft seems finally to be getting the message. Bill Gates sent a company-wide e-mail on January 15, 2002, announcing that what he called "Trustworthy Computing" was Microsoft's highest priority. Trustworthy Computing, he wrote, was composed of availability, security, and privacy. "... ensuring .NET is a platform for Trustworthy Computing is more important than any other part of our work. If we don't do this, people simply won't be willing—or able—to take advantage of all the other great work we do." I've said for years that Microsoft needed to change priorities; that spending one single developer day, nay, one single developer minute, on a paper clip's blinking eyes before plugging every possible security hole and fixing every possible crash is to worship false gods. I'm ecstatic (OK, I'm a geek) that they finally seem to be doing it. How far and how quickly Trustworthy Computing (TC?) will go, and how much and how quickly the market will believe it, we'll just have to wait and see. Because of its late-breaking nature, I'll discuss this more in my chapter on .NET My Services, available on this book's Web site. But I see a major turn in the right direction, and I'm glad of it. Now tell those airlines, OK?

ASP.NET includes prefabricated support for Passport authentication.

Passport is an interesting idea, and the market will determine its success or failure, as it should. I would not choose it as my sole authentication

mechanism today, but I'd give serious thought to offering it as an option side by side with non-Passport alternatives. It will make the lives of some unknown (until you do it) percentage of your users easier, which should bring you more money, especially if your competitors don't do it. You'll find that the prefabricated support in ASP.NET makes it quite easy. You set the authentication mode to Passport in the web.config file. You will have to download the Passport SDK, sign a licensing agreement, and then write some simple code that delegates the authentication process to Passport.

Authorization

Once the authentication process is complete, we know who the user is or know that the user is anonymous. We now need to decide whether the user should be allowed to see the page he is requesting. This is called *authorization*. You can perform authorization administratively by restricting access to pages in your ASP.NET application, or you can perform it programmatically at run time.

ASP.NET contains good support for controlling access to .ASPX pages. You can administratively limit access to pages by making entries in your application's web.config files, as shown in Listing 3-10.

Once we've authenticated the user, we need to check whether the user is authorized to do what he's trying to do.

You can administratively specify which users are allowed to view various pages by making entries in the web.config files for specific pages or directories.

```
<!-- Authorization

    This allows a user named Simba, and any users belonging to the
    role Doctors, to access pages in this directory (and any
    subdirectories that don't override this setting in their own
    web.config files).
-->

<authorization>
  <allow users="Simba" roles="Doctors" />
  <deny users="*" />
</authorization>
```

Listing 3-10 Authorization entries in a web.config file.

The `<authorization>` section contains `<allow>` and `<deny>` elements that respectively allow and deny³ the specified users and members of the specified roles (groups of users) access to pages in the directories to which the web.config files apply. These elements can also accept a *verb* attribute (not shown) so that you can allow some users to view the page with a get

3. Well, duh!

116 Introducing Microsoft .NET, Third Edition

ASP.NET applies the administrative access rules in the order in which they appear.

operation but forbid them to post anything back to the server. Although web.config files apply to an entire directory (and its subdirectories unless overridden lower down), you can administratively restrict access to a single file by using the *<location>* element (not shown).

When a user requests a page, ASP.NET applies the rules specified in the web.config file in the order in which they appear until it finds a rule that tells it whether to grant or deny the requesting user access to the requested page. If ASP.NET can't tell from the entries made in that file, it will look in the web.config file in that directory's parent and so on until it reaches the master ASP.NET machine.config file discussed previously. That file, by default, grants access to all requests. So if you don't want people to see a certain file, you'll have to explicitly put in a *<deny>* element. In the example shown in Listing 3-9, ASP.NET will first apply the *<allow>* element. If the user is named "Simba" or is a member of a role named "Doctors," access is granted and the checking process ends. If neither of these tests is true, ASP.NET applies the next rule, which says to deny access to everybody (the * character). ASP.NET will fail the request, and the checking process will end. In this way, only Doctors and Simba can view the pages of this subdirectory. Note that when using Windows authentication on a domain, you must prepend the domain name onto the role or user name in order for ASP.NET to recognize it, as shown in Listing 3-11.

```
<!-- Authorization

    This does the same thing as the previous example, except
    user ID and role names are prefixed with the domain name so as to
    work correctly with Windows authentication on a domain.
-->

<authorization>
  <allow users="REDHOOKESB0\Simba"/>
  <allow roles="REDHOOKESB0\Doctors"/>
  <deny users="*" />
</authorization>
```

Listing 3-11 Web.config file showing authorization entries for domain users using Windows authentication.

The "?" character allows you to set permissions for unauthenticated users.

Since it is likely that our Web site will be used by many anonymous users—those who have not been authenticated—we need a way of specifying what they are allowed to do. The question mark character (?) denotes anonymous users. You use it exactly like any other name or the * character. You can see an example in Listing 3-8, the web.config file used for forms-

based authentication. It says that unauthenticated users aren't allowed to view pages in that directory, thereby forcing them to the login form to become authenticated.

ASP.NET allows you to authorize either individual users or entire roles. Most real-life installations perform little or no authorization on an individual user basis; it's almost always done on a group basis. For example, a hospital prescription application might allow physician's assistants to prescribe Tylenol, but only licensed doctors to prescribe controlled narcotics. An administrator will set permissions for each role, and add individual users to or remove them from role membership as required.

Most real-life authorization works with roles, which are groups of users.

When you use Windows authentication, ASP.NET automatically recognizes a role as any standard Windows user group set up by the administrator. Every user assigned to that group is a member of the role. You don't have to program anything, recognition just automatically happens. When you use forms or Passport authentication, however, the definition of a role becomes much more difficult. ASP.NET doesn't know how your user data is stored or what constitutes role membership in your application. You therefore have to write your own code and plug it into the ASP.NET environment to tell it which role(s) a user belongs to when it asks.

ASP.NET automatically determines role membership when you use Windows authentication.

I've written a sample program that does this, which you can download from the book's Web site. The application file `Global.asax` contains handlers for application-level events. I've added code to the handler for the *AuthenticateRequest* event. Somewhat confusingly named, this event gets called at the beginning of each page request when ASP.NET has already discovered the user's identity and authenticated it by whatever means you have told it to use. This is your code's chance to make any changes, and where you have to place the code that tells ASP.NET the roles to which a user belongs. The code is shown in Listing 3-12. We check to see if the user has been authenticated by ASP.NET. If so, we create a new object of type *GenericPrincipal*, which contains the user's ID that we've been given as well as the roles to which we have decided the user belongs, and set it into ASP.NET.

How do we know at this juncture what roles this user belongs to? I could certainly do my own lookup from within the event handler, based on the user ID I've been given. But this could involve an expensive database lookup and possibly a network round-trip happening every time a page request comes in, so we'd like a faster way. The sample program accomplishes this by placing the role strings in the cookie itself when it's first issued, at login time. The code is shown in Listing 3-13. The function *MyOwnGetRole* looks up user names and comes back with a role string. I've hardwired users named Seuss and Kevorkian to members of the role Doctors,

and users named Ratched and Houlihan to members of the role Nurses. In order to store this information in the cookie, I have to manually create my admission ticket, placing the role string into an element called *UserData* that exists for this purpose. The sample code demonstrates the degree of control that I have over the admission ticket cookie, setting its persistence and expiration time. As I described previously, my *AuthenticateRequest* event code reads the user data from the cookie, thereby providing me the list of roles to which my user belongs. Since every forms authentication program that wants to use role-based security needs to write this code, I wish Microsoft had provided a prefabricated implementation of it. Maybe in the next version.

```
Sub Application_AuthenticateRequest(ByVal sender As Object, _
                                   ByVal e As EventArgs)

    ' If ASP.NET has figured out who the user is

    If (Not HttpContext.Current.User Is Nothing) Then

        ' If the user has been authenticated by means of forms,
        ' and thus contains the admission ticket in a cookie

        If (TypeOf (HttpContext.Current.User.Identity) _
            Is System.Web.Security.FormsIdentity) Then

            ' Fetch the object giving the identity of the user

            Dim id As System.Web.Security.FormsIdentity
            id = HttpContext.Current.User.Identity

            ' Fetch the authentication ticket from this identity object

            Dim ticket As System.Web.Security.FormsAuthenticationTicket
            ticket = id.Ticket

            ' Create a new object identifying the user. Pass the role
            ' strings that we placed in the UserData section of the
            ' ticket back in the logon form where we issued it.

            Dim NewPrincipal As New _
                System.Security.Principal.GenericPrincipal( _
                    id, New String() {ticket.UserData})

            ' Place the new principal object, carrying the roles, into
            ' the context to identify the user in subsequent processing
            ' of this request.
```

Listing 3-12 Global.asax.vb code to set role membership on a user's page request.


```

        HttpContext.Current.User = NewPrincipal

    End If
End If
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button1.Click

    If (TextBox2.Text = "") Then

        ' Create new ticket

        Dim ticket As New System.Web.Security.FormsAuthenticationTicket( _
            1, TextBox1.Text, Now, Now.AddYears(10), True, _
            MyOwnGetRole(TextBox1.Text))

        ' Get the encrypted representation suitable for placing
        ' in an HTTP cookie

        Dim cookieStr As String = _
            System.Web.Security.FormsAuthentication.Encrypt(ticket)

        ' Place ticket in cookie

        Dim cookie As New _
            HttpCookie( _
                System.Web.Security.FormsAuthentication.FormsCookieName, _
                cookieStr)
        Response.Cookies.Add(cookie)

        ' redirect to the URL user asked for

        Dim returnUrl As String = Request("ReturnURL")
        Response.Redirect(returnUrl, False)

    Else
        Label4.Text = "Invalid credentials, try again"
    End If
End Sub

```

Listing 3-13 Code to place role membership in authentication cookie for later use.

Administratively restricting access on a per-page basis is fine, but sometimes your application requires finer granularity. For example, several classes of user might need access to a prescription form page. Suppose that members

of the role Doctors can prescribe morphine, but members of the role Nurses can only prescribe aspirin. You could provide two separate pages, administratively restricting access to the morphine page to doctors, but this would require constant page redesign as drugs change their prescription status. It would be much easier to check in the middle tier. Your page code can find out who a user is by accessing the member variable *Page.User*. This object contains a method called *IsInRole*, which you can call to see if the currently authenticated user is a member of the specified role. The page is shown in Figure 3-18, and the code in Listing 3-14. If you'd like finer granularity, you can access the object *Page.User.Identity*, which contains the user's name.

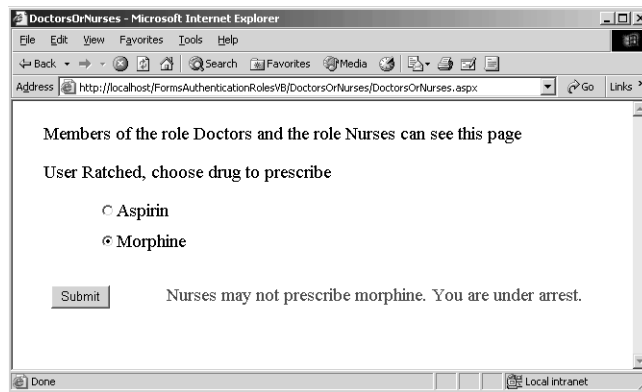


Figure 3-18 Sample application for forms role authentication.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
    Handles Button1.Click

    ' User wants to prescribe aspirin. Our business logic allows this for
    ' the roles Doctors and Nurses. Since these are the only two roles
    ' allowed to see this page, the prescription is automatically approved

    If (RadioButton1.Checked) Then

        Label13.Text = "Doctors and Nurses can both prescribe " & _
            "aspirin. Prescription approved."

    Else

        ' User clicked button to prescribe morphine. Doctors are allowed
        ' to do this, but Nurses are not. Check the role of the user.
        ' Approve prescription if user is a Doctor, fail if it's a Nurse.

        If User.IsInRole("Doctors") Then
```

Listing 3-14 Code to discover the role a user belongs to.

```
        Label3.Text = "Because you are a member of the role " & _  
                    "'Doctors', your prescription is approved"  
    Else  
        Label3.ForeColor = Color.Red  
        Label3.Text = "Nurses may not prescribe morphine. " & _  
                    "You are under arrest."  
    End If  
  
End If  
End Sub
```

Identity

Your Web pages usually represent the middle tier of a three-tier system, mediating access to a database behind it (the data tier). The middle tier performs business logic, such as determining whether the user attempting to prescribe morphine is a licensed doctor. If this business logic is successful, the middle tier will communicate with the data tier to subtract morphine from the pharmacy inventory and to add another item to the patient's billing statement. If not, the middle tier will probably generate a fascinating game to keep the user occupied while it calls the police.

The back-end data tier almost certainly has its own security mechanisms. All commercial databases allow administrators to set access permissions of various granularity, specifying which users are allowed access to various databases and tables and which are not. Even the NTFS file system allows an administrator to specify which users or groups of users are allowed to perform which operations on which files and directories. The operation of these back ends depends critically on what the data tier considers to be the identity of the user making the request for services.

One security mechanism, and probably the one that you'll wind up using, is called the *trusted user* model. In this model, the server process runs with a particular identity, say, PharmacyApp, known as the trusted user. The data tier is configured to allow this user to do anything in the database that it might need to in order to get its work done. For example, the trusted user identity used for the pharmacy application will have permission to make entries in the pharmacy inventory tables and the patient billing tables but probably not the patient census tables or the employee payroll records. The data tier trusts the middle tier server to have done all the authorization checking that needs to happen—for example, to have checked that the doctor's license is current—so the data tier doesn't perform a second authorization. You trust your spouse to go through your wallet, so you don't bother asking

Your Web site is usually the middle tier of a three-tier system, mediating access to a data tier.

The security identity of the Web server process is important.

The data tier usually trusts the middle tier to perform authorization.

what the money is for. This relationship is shown in Figure 3-19. You specify the identity of your server process via the `<identity>` element in your application's web.config file, as shown in Listing 3-15.

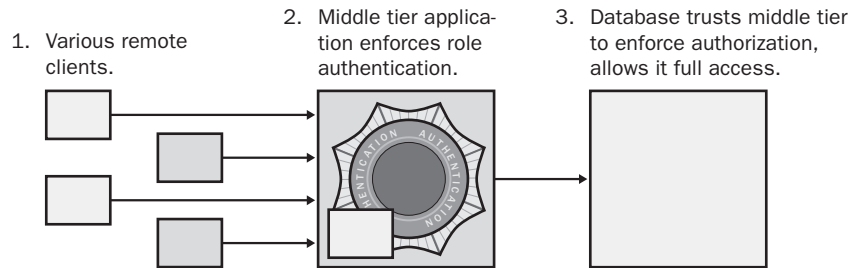


Figure 3-19 Trusted user model of authorization.

```
<identity impersonate="false"
  userName="MyDomain\MyUsername
  password="MyPassword" />
```

Listing 3-15 Web.config file specifying server process identity.

Sometimes the trusted user model doesn't fit.

In this case, the Web server can impersonate the client, if you're using Windows authentication.

Not every database installation is comfortable with this approach. Many databases were designed and database administrators trained before three-tier programming became popular. Particularly with legacy systems, it often happens that the database layer contains security checks or audit trails that depend on the actual database entry being made under the network identity of the base client.

In this case, if you're using Windows authentication, you can fall back on the older *impersonation-delegation* model, as used by original ASP. In this model, the .ASPX page code takes on the identity of ("impersonates") the authenticated user (or uses a special identity designated for anonymous users). The page code then attempts to access the data-tier resource, and the resource itself, say, SQL Server, performs the authorization, checking to see whether the user is allowed access. This case is shown in Figure 3-20.

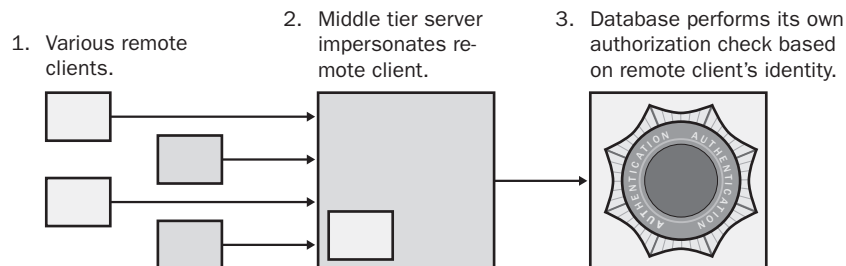


Figure 3-20 Impersonation/delegation model of authorization.

While this approach may sound attractive, it is expensive in terms of performance. It requires two authentications instead of one: the first when ASP.NET authenticates the user prior to impersonating him, and the second when the database authenticates the server object's current impersonating identity. Furthermore, when authorization rejects a user, it does so rather late in the process. You've already gone down three tiers instead of two and made two network hops instead of one. It's better that you notice that you forgot your wallet when you arrive at the supermarket instead of after you've shopped and waited in the checkout line. It's better still to notice it when you first get in your car.

The impersonation-delegation model is expensive in terms of performance.

The Web server impersonates a client in one of two ways. Original ASP automatically impersonated the client every time a request was made. ASP.NET does not do this by default, but you can turn on this functionality in the web.config file by setting the *impersonation* attribute shown in Listing 3-16 to True. Alternatively, you can impersonate the client programmatically by calling the function *System.Security.Principal.WindowsIdentity.GetCurrent().Impersonate*.

A Web server can impersonate a client programmatically, if using Windows authentication.

```
<identity>
  <impersonation enable="true" />
</identity>
```

Listing 3-16 Web.config setting for automatic impersonation.

Process Management

One of the main features that we want in a Web server is unattended 24/7 operation. This level of service is a problem developers never really had to address in writing desktop applications. For example, memory leaks in a Solitaire game that runs for 5 minutes at a time (OK, 2 hours) probably won't waste enough memory to hurt anything. But if I kept the Solitaire game running for a year, redealing instead of closing it, any memory leaks would eventually exhaust the process's address space and cause a crash.

Robustness under load is a very important feature of a Web server.

That sort of robustness is very hard to develop, partly because it takes a long time to test. The only way to find out whether something runs for two weeks under load is to run it under load for two weeks. And when the crash does happen, it's often caused by a banana peel dropped days earlier, which is essentially impossible to find at the time of the crash.

It's also extremely difficult to develop. That figures.

I once (14 years ago) worked on an application that ran in a major bank. It was a DOS-based system (remember DOS?) that would usually run all day, but it just couldn't run for a week without crashing and we couldn't wring it out so that it could. I had the bright idea of putting a watchdog timer card into the machine that our software would periodically reset while it ran. If the

Most software needs periodic restarting.

124 Introducing Microsoft .NET, Third Edition

ASP.NET supports process recycling to continue working robustly in the face of imperfect user code.

You configure process recycling in the `machine.config` file.

timer ever actually expired without being reset, it would automatically reboot the system, sort of a “dead-geek” switch. The bank balked at installing such a god-awful kludge (we probably shouldn’t have told them; we probably should have just done it and smiled) and instead agreed to have the administrator reboot the system every night.

Original ASP kept a user process running indefinitely. Any bugs or memory leaks in any of the user code would accumulate and eventually cause crashes. ASP.NET recognizes that user code probably isn’t going to be perfect. It therefore allows an administrator to configure the server to periodically shut down and restart worker processes.

You configure process recycling using the `<processModel>` element of the machine-level `machine.config` file, as shown in Listing 3-17. I wish Microsoft had allowed process model configuration on a per-application basis, but they haven’t at the time of this writing. You can tell ASP.NET to shut down your worker process and launch a new one after a specified amount of time (*timeout* attribute), a specified number of page requests (*requestLimit* attribute), or if the percentage of system memory it consumes grows too large (*memoryLimit* attribute). While not removing the need to make your code as robust as possible, this will allow you to run with a few memory leaks without rebooting the server every day or two. You can see that the process model contains other configurable capabilities as well. IIS 6.0, which is scheduled to ship with Windows Server 2003, contains a mode called worker process isolation. When this mode is active, ASP.NET’s process model settings are ignored in favor of IIS’s own internal settings that provide the same sorts of functionality.

```
<processModel
  enable="true"
  timeout="infinite"
  idleTimeout="infinite"
  shutdownTimeout="0:00:05"
  requestLimit="infinite"
  requestQueueLimit="5000"
  memoryLimit="80"
  webGarden="false"
  cpuMask="0xffffffff"
  userName=""
  password=""
  logLevel="errors"
  clientConnectedCheck="0:00:05"
/>
```

Listing 3-17 Machine.config file showing process recycling settings.