2nd Edition

Includes Bonus Visual Studio .NET add-in

ASP.NET

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

G. Andrew Duthie & Matthew MacDonald

ASP.NET

IN A NUTSHELL

Other Microsoft .NET resources from O'Reilly

Related titles

Programming C#
C# in a Nutshell
Programming Visual Basic .NET
Programming ASP.NET
ASP.NET in a Nutshell
ADO.NET in a Nutshell
.NET Windows Forms in a Nutshell
.NET Framework Essentials
Mastering Visual Studio .NET

.NET Books Resource Center

dotnet.oreilly.com is a complete catalog of O'Reilly's books on .NET and related technologies, including sample chapters and code examples.

ONDotnet.com provides independent coverage of fundamental, interoperable and emerging Microsoft .NET programming and web services technologies.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

ASP.NET

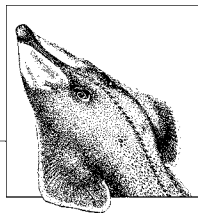
IN A NUTSHELL

Second Edition

G. Andrew Duthie and Matthew MacDonald

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

6



User Controls and Custom Server Controls

Reuse, a technique that is important to most developers, allows you to avoid constantly reinventing the wheel by using functionality that has already been built and tested. Reuse increases productivity, by reducing the total amount of code you need to write, and reliability, since by using tested code, you (presumably) already know the code works reliably.

ASP.NET provides a range of options for reuse. The first is the wide variety of built-in server controls that ship with ASP.NET. These server controls alone can eliminate hundreds, or even thousands, of lines of code that needed to be written to achieve the same effect in classic ASP. In addition, the .NET Framework Class Library (FCL) provides hundreds of classes to perform actions (such as sending SMTP email or making network calls) that in classic ASP would have required purchasing a third-party component or making calls into the Win32 API. Of course, the framework classes provide built-in functionality more than reuse. Fortunately, the framework also provides robust support for developing your own classes, user controls, and custom server controls, allowing you to reuse your own code as well.

Going hand-in-hand with reuse is the concept of *extensibility*, the ability to take the existing functionality provided by the .NET Framework and ASP.NET and extend it to perform actions that are more tailored to your particular applications and problem domains. ASP.NET provides a significant number of avenues for extensibility:

Custom server controls
: Allow you to create entirely new controls for use with ASP.NET or to derive from existing controls and extend or modify their functionality.

Components
: As in classic ASP, components are the primary means for extending an ASP.NET application by encapsulating the application's business logic into an easily reusable form. With the .NET Framework, it's easier than ever to

build components, and components are more interoperable across languages than in the COM world. .NET components can also communicate with COM components through an interoperability layer.

HttpHandlers and HttpModules
: HttpHandlers are components that are called to perform the processing of specific types of requests made to IIS. HttpModules are components that participate in the processing pipeline of all requests for a given ASP.NET application. These extensibility techniques are beyond the scope of this book, but you can get answers to questions on these topics in the HttpHandlers and HttpModules forum at *http://www.asp.net/forums*.

The rest of this chapter discusses employing ASP.NET user controls and custom server controls for reuse and employing custom server controls for extensibility. The chapter also explains how custom server controls can easily be shared across multiple applications, making reuse simpler than ever.

User Controls

The simplest form of reuse in classic ASP is the include file. By adding the following directive:

```
<!-- #include file = "filename.inc" -->
```

classic ASP developers can place the contents of the specified file inline with the page in which the directive appeared. Unfortunately, this reuse technique is a bit crude and sometimes makes applications harder to debug.

While ASP.NET still supports include files, a better way to provide the same kinds of reuse is through a new feature called *user controls*. User controls can consist of any of the following:

- HTML
- Server-side script
- Controls

all in a file with the *.ascx* file extension. When added to a Web Forms page, ASP.NET treats user controls as objects; these user controls can expose properties and methods like any other object. The rendered output of user controls can also be cached to improve application performance.

Example 6-1 shows a simple user control that provides navigational links to other examples in this chapter. The user control appears in each example to demonstrate how the use of a user control can provide a single point for modifying such frequently used elements as headers, footers, and navigation bars.

Example 6-1. Nav.ascx

```
<%@ Control Language="vb" %>
<table cellpadding="0" cellspacing="0">
   <tr>
      <td valign="top">
         <strong>Navigation Bar</strong><br/>
         <hr width='80%'>
```

Example 6-1. Nav.ascx (continued)

```
        <a href="NavBarClient.aspx"
           onmouseover="img1.src='node_rev.jpg';"
           onmouseout="img1.src='node.jpg';">
           <img border='0' align='absMiddle' alt='NavBar Client'
              src='node.jpg' id='img1' name='img1'></a>
        <a href="NavBarClient.aspx"
           onmouseover="img1.src='node_rev.jpg';"
           onmouseout="img1.src='node.jpg';">NavBar Client</a>
        <hr width='80%'>
        <a href="UCClient.aspx"
           onmouseover="img2.src='alt_node_rev.jpg';"
           onmouseout="img2.src='alt_node.jpg';">
           <img border='0' align='absMiddle' alt='User Control Client'
              src='alt_node.jpg' id='img2' name='img2'></a>
        <a href="UCClient.aspx"
           onmouseover="img2.src='alt_node_rev.jpg';"
           onmouseout="img2.src='alt_node.jpg';">User Control Client</a>
        <hr width='80%'>
        <a href="BlogClient.aspx"
           onmouseover="img3.src='node_rev.jpg';"
           onmouseout="img3.src='node.jpg';">
           <img border='0' align='absMiddle' alt='Blog Client'
              src='node.jpg' id='img3' name='img3'></a>
        <a href="BlogClient.aspx"
           onmouseover="img3.src='node_rev.jpg';"
           onmouseout="img3.src='node.jpg';">Blog Client</a>
        <hr width='80%'>
        <a href="BlogAdd.aspx"
           onmouseover="img3.src='alt_node_rev.jpg';"
           onmouseout="img3.src='alt_node.jpg';">
           <img border='0' align='absMiddle' alt='Add New Blog'
              src='alt_node.jpg' id='img3' name='img3'></a>
        <a href="BlogAdd.aspx"
           onmouseover="img3.src='node_rev.jpg';"
           onmouseout="img3.src='node.jpg';">Add New Blog</a>
        <hr width='80%'>
      </td>
   </tr>
</table>
```

With the exception of the `@ Control` directive, which is not strictly required, the code in Example 6-1 consists exclusively of HTML and client-side script (for performing a simple mouseover graphics switch). However, the user control could just as easily contain server controls and/or server-side script to perform more complicated tasks.

The `@ Control` directive performs essentially the same task as the `@ Page` directive, only for user controls. Chapter 3 lists the attributes of the `@ Page` and `@ Control` directives and the purpose of each.

The advantage of using a user control for this type of functionality is that it places all of our navigation logic in a single location. This placement makes it considerably easier to maintain the navigation links for a site. If you used ASP.NET's

built-in server controls instead of raw HTML in your navigation user control, you could manipulate those server controls programmatically from the page on which the control is used. For example, you could hide the link to the page that's currently displayed or highlight it in some fashion.

The disadvantage of a user control is that it is not reusable across multiple sites ("site," here, refers to an IIS virtual directory defined as an application). It's also not usually a good idea to tightly couple user interface elements and data, as this control does, because doing so tends to reduce the reusability of a control. Later in this chapter, you'll see how to improve this user control by turning it into a custom server control.

User controls are made available to a page through the use of either the `@ Register` directive, which prepares a user control on a page declaratively (i.e., using a tag-based syntax like server controls), or, programmatically, using the LoadControl method of the `TemplateControl` class (from which both the `Page` class and the `UserControl` class derive).

Example 6-2 shows a page that uses the `@ Register` directive and a declarative tag to create the user control shown in Example 6-1. The `@ Register` directive in Example 6-2 tells ASP.NET to look for any `<aspnetian:nav>` tags with the `runat="server"` attribute, and when it finds one, create an instance of the user control and place its output where the tag is located. This allows us to place our control very precisely.

Example 6-2. UCClient.aspx

```
<%@ Page Language="vb" %>
<%@ Register TagPrefix="aspnetian" TagName="nav" Src="Nav.ascx" %>
<html>
<head>
</head>
<body>
   <table border="1" width="100%" cellpadding="20" cellspacing="0">
      <tr>
         <td align="center" width="150">
            <img src="aspnetian.jpg"/>
         </td>
         <td align="center">
            <h1>User Control Client Page<h1>
         </td>
      </tr>
      <tr>
         <td width="150">
            <aspnetian:nav runat="server"/>
         </td>
         <td>
            This is where page content might be placed
            <br/><br/><br/><br/><br/><br/><br/><br/><br/>
         </td>
      </tr>
   </table>
</body>
</html>
```

You can instead create the control dynamically using the LoadControl method and add the control to either the Controls collection of the page, or, better yet, to the Controls collection of a PlaceHolder control. The latter allows you to control the location of the user control based on the location of the placeholder. You might choose to use this technique if you know where you want the control to reside on the page, but don't necessarily want the control loaded and displayed on every request. This technique is shown in Example 6-3.

Example 6-3. UCClient_Prog.aspx

```
<%@ Page Language="vb" %>
<html>
<head>
   <script runat="server">
      Sub Page_Init( )
         PH.Controls.Add(LoadControl("Nav.ascx"))
      End Sub
   </script>
</head>
<body>
   <table border="1" width="100%" cellpadding="20" cellspacing="0">
      <tr>
         <td align="center" width="150">
            <img src="aspnetian.jpg"/>
         </td>
         <td align="center">
            <h1>User Control Client Page<h1>
         </td>
      </tr>
      <tr>
         <td width="150">
            <asp:placeholder id="PH" runat="server"/>
         </td>
         <td>
            This is where page content might be placed
            <br/><br/><br/><br/><br/><br/><br/><br/><br/>
         </td>
      </tr>
   </table>
</body>
</html>
```

If you want to work with the control after loading it using LoadControl, you need to cast the control to the correct type using the *CType* function in Visual Basic .NET or by preceding the control with (*typename*) in C#. Note that this requires that the user control be defined in a class that inherits from UserControl, so this technique would not work with the user control in Example 6-1.

Custom Server Controls

For the reasons cited earlier in the chapter, user controls are not always the ideal choice for reuse. User controls tend to be very good for quickly reusing existing user interface elements and code, but custom server controls are much better for developing reusable building blocks for multiple web applications.

A custom server control is, in its essence, a class that derives from either the Control or `WebControl` class of the System.Web.UI namespace, or from one of the classes that derive from these controls. Custom server controls can be used in your ASP.NET Web Forms pages in very much the same way you use the built-in server controls that come with ASP.NET. There are two primary categories of custom server controls:

Rendered controls
: Rendered controls consist largely of custom rendering of the text, tags, and any other output you desire, which may be combined with the rendered output of any base class from which your control is derived. Rendered controls override the Render method of the control from which they derive. This method is called automatically by the page containing the control when it's time for the control output to be displayed.

Compositional controls
: Compositional controls get their name from the fact that they are composed of existing controls whose rendered output forms the UI of the custom control. Compositional controls create their constituent controls by overriding the CreateChildControls method of the control from which they derive. This method, like the Render method, is automatically called by ASP.NET at the appropriate time.

When designing a new custom server control, you need to consider some issues to decide which type of control to create:

- Does one existing control provide most, but not all, of the functionality you desire? A rendered control that derives from that control may be the right choice.
- Could the desired functionality be provided by a group of existing controls? A compositional control may be a great way to reuse those controls as a group.
- Do you want to do something that is completely beyond any existing control? You may want to derive your control from the `Control` class and override the Render method to create your custom output.

Note that by default, custom server controls expose all public members of the class from which they are derived. This exposure is important to consider when designing a control for use by other developers if you want to limit the customizations they can make. For instance, you might not want developers to change the font size of your control. In such a case, you should avoid deriving from a control that exposes that property.

Rendered Controls

Perhaps the best way to understand the process of creating a rendered custom server control is to see one. Example 6-4 shows a class written in Visual Basic .NET

that implements a custom navigation control with the same functionality as the *Nav.ascx* user control discussed earlier in this chapter. Unlike the user control, which has the linked pages and images hardcoded into the control itself, the custom control in Example 6-4 gets this information from an XML file.

Example 6-4. NavBar.vb

```
Imports Microsoft.VisualBasic
Imports System
Imports System.Data
Imports System.Drawing
Imports System.IO
Imports System.Text
Imports System.Web
Imports System.Web.UI
Imports System.Web.UI.WebControls

Namespace aspnetian

Public Class NavBar
   Inherits Panel

   Private NavDS As DataSet
   Private _showDividers As Boolean = True

   Public Property ShowDividers( ) As Boolean
      Get
         Return _showDividers
      End Get
      Set
         _showDividers = value
      End Set
   End Property

   Sub NavBar_Load(sender As Object, e As EventArgs) Handles MyBase.Load

      LoadData( )

   End Sub

   Protected Overrides Sub Render(Writer As HtmlTextWriter)

      Dim NavDR As DataRow
      Dim RowNum As Integer = 1
      Dim SB As StringBuilder

      MyBase.RenderBeginTag(Writer)
      MyBase.RenderContents(Writer)

      Writer.Write("<hr width='80%'>" & vbCrLf)

      For Each NavDR In NavDS.Tables(0).Rows

         SB = new StringBuilder( )
         SB.Append(vbTab)
```

Example 6-4. NavBar.vb (continued)

```
        SB.Append("<a href=""")
        SB.Append(NavDR("url"))
        SB.Append(""" onmouseover=""")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append(".src='")
        SB.Append(NavDR("moimageUrl"))
        SB.Append("';""")
        SB.Append(" onmouseout=""")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append(".src='")
        SB.Append(NavDR("imageUrl"))
        SB.Append("';""")
        SB.Append(" target='")
        SB.Append(NavDR("targetFrame"))
        SB.Append("'>")
        SB.Append(vbCrLf)
        SB.Append(vbTab)
        SB.Append(vbTab)
        SB.Append("<img border='0' align='absMiddle' alt='")
        SB.Append(NavDR("text"))
        SB.Append("' src='")
        SB.Append(NavDR("imageUrl"))
        SB.Append("' id='")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append("' name='")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append("'></a>")
        SB.Append(vbTab)
        SB.Append("<a href=""")
        SB.Append(NavDR("url"))
        SB.Append(""" onmouseover=""")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append(".src='")
        SB.Append(NavDR("moimageUrl"))
        SB.Append("';""")
        SB.Append(" onmouseout=""")
        SB.Append("img")
        SB.Append(RowNum.ToString( ))
        SB.Append(".src='")
        SB.Append(NavDR("imageUrl"))
        SB.Append("';""")
        SB.Append(" target='")
        SB.Append(NavDR("targetFrame"))
        SB.Append("'>")
        SB.Append(NavDR("text"))
        SB.Append("</a>")
        SB.Append(vbCrLf)
```

Example 6-4. NavBar.vb (continued)

```
            If _showDividers = True Then
               SB.Append("<hr width='80%'>")
            Else
               SB.Append("<br/><br/>")
            End If
            SB.Append(vbCrLf)
            Writer.Write(SB.ToString( ))

            RowNum += 1

         Next

         MyBase.RenderEndTag(Writer)

      End Sub

      Protected Sub LoadData( )

         NavDS = New DataSet( )

         Try
            NavDS.ReadXml(Page.Server.MapPath("NavBar.xml"))
         Catch fnfEx As FileNotFoundException
            CreateBlankFile( )
            Dim Html As String
            Html = "<br>No NavBar.xml file was found, so one was " & _
               "created for you. Follow the directions in the file " & _
               "to populate the required fields and, if desired, " & _
               "the optional fields."
            Me.Controls.Add(New LiteralControl(Html))
         End Try

      End Sub

      Public Sub CreateBlankFile( )
         'Code to create a blank XML file with the fields used by
         '   the control. This code is included as a part of the file
         '   NavBar.vb, included with the sample files for the book.
      End Sub

End Class

End Namespace
```

The real meat of the NavBar control begins with the class declaration, which uses the `Inherits` keyword to declare that the control derives from the Panel control. This gives the control the ability to show a background color, to be hidden or shown as a unit, and to display the contents of its begin and end tags as part of the control.

Next, a couple of local member variables are declared. The location of the declaration is important, since these members need to be accessible to any procedure in

the control. A property procedure is then added for the ShowDividers property, which will determine whether the control renders a horizontal line between each node of the control.

In the NavBar_Load method, which handles the Load event for the control (fired automatically by ASP.NET), the LoadData method is called to load the NavBar data from the XML file associated with the control.

Skipping over the Render method temporarily, the LoadData method creates a new instance of the ADO.NET `DataSet` class and calls its ReadXml method to read the data from the XML file. If no file exists, the LoadData method calls another method (CreateBlankFile) to create a blank XML file with the correct format for use by the developer consuming the control. This technique not only deals gracefully with an error condition; it provides an easier starting point for the developer using the control. Note that the CreateBlankFile method is declared as public, which means it can be called deliberately to create a blank file, if desired.

Last, but certainly not least, the overridden Render method, which is called automatically at runtime when the control is created, iterates through the first (and only) table in the dataset and uses an instance of the `StringBuilder` class to build the HTML output to render. Once the desired output has been built, the method uses the HtmlTextWriter passed to it by ASP.NET to write the output to the client browser. Note that prior to looping through the rows in the dataset, the render method calls the RenderBeginTag and RenderContents methods of the base Panel control. This renders the opening `<div>` tag that is the client-side representation of the Panel control, plus anything contained within the opening and closing tags of the NavBar control. Once all the rows have been iterated and their output sent to the browser, the RenderEndTag method is called to send the closing `</div>` tag to the browser.

This example uses a couple of helper classes that are fairly common in ASP.NET development. The first, the `StringBuilder` class, is a helper class that is used for constructing strings. Because strings are immutable in the .NET Framework (strings cannot be changed), each time you use string concatenation (i.e., use the VB `&` operator or the C# `+` operator), the original string is destroyed and a new string containing the result of the concatenation is created. This can get fairly expensive when you're doing a lot of concatenation, so the `StringBuilder` class provides a way of constructing strings without the expense of concatenation.

The `HtmlTextWriter` class, an instance of which is automatically created and passed to the Render method by the ASP.NET runtime, allows you to write text output to the client browser, and includes useful methods (such as WriteBeginTag, WriteEndTag, and WriteAttribute) and shared/static fields for correctly formatting HTML output.

You can compile the code in Example 6-4 by using the following single-line command (which can alternatively be placed in a batch file):

```
vbc /t:library /out:bin\NavBar.dll /r:System.dll,System.Data.dll,
System.Drawing.dll,System.Web.dll,System.Xml.dll NavBar.vb
```

The preceding command requires that you create a *bin* subdirectory under the directory from which the command is launched and that you register the path to the Visual Basic compiler in your PATH environment variable. If you have not registered this path, you will need to provide the full path to the Visual Basic .NET compiler (by default, this path is *%windir%\Microsoft.NET\Framework\%version%* where *%windir%* is the path to your Windows directory, and *%version%* is the version number of the framework version you have installed).

Example 6-5 shows the XML file used to populate the control, Example 6-6 shows the code necessary to use the NavBar control in a Web Forms page, and Figure 6-1 shows the output of this page.

Example 6-5. NavBar.xml

```
<navBar>
    <!-- node field describes a single node of the control -->
    <node>
        <!-- Required Fields -->
        <!-- url field should contain the absolute or relative
           URL to link to -->
        <url>NavBarClient.aspx</url>
        <!-- text field should contain the descriptive text for
           this node -->
        <text>NavBar Client</text>
        <!-- End Required Fields -->
        <!-- Optional Fields -->
        <!-- imageUrl field should contain the absolute or relative
           URL for an image to be displayed in front of the link -->
        <imageUrl>node.jpg</imageUrl>
        <!-- moimageUrl field should contain the absolute or
           relative URL for an image to be displayed in front of
           the link on mouseover -->
        <moImageUrl>node_rev.jpg</moImageUrl>
        <!-- targetFrame field should contain one of the following:
           _blank, _parent, _self, _top  -->
        <targetFrame>_self</targetFrame>
        <!-- End Optional Fields -->
    </node>
    <node>
        <url>UCClient.aspx</url>
        <text>User Control Client</text>
        <imageUrl>alt_node.jpg</imageUrl>
        <moImageUrl>alt_node_rev.jpg</moImageUrl>
        <targetFrame>_self</targetFrame>
    </node>
    <node>
        <url>BlogClient.aspx</url>
        <text>Blog Client</text>
        <imageUrl>node.jpg</imageUrl>
        <moImageUrl>node_rev.jpg</moImageUrl>
        <targetFrame>
        </targetFrame>
    </node>
    <node>
```

Example 6-5. NavBar.xml (continued)

```
        <url>BlogAdd.aspx</url>
        <text>Add New Blog</text>
        <imageUrl>alt_node.jpg</imageUrl>
        <moImageUrl>alt_node_rev.jpg</moImageUrl>
        <targetFrame>
        </targetFrame>
    </node>
</navBar>
```

Example 6-6. NavBarClient.aspx

```
<%@ Page Language="vb" %>
<%@ Register TagPrefix="aspnetian" Namespace="aspnetian"
   Assembly="NavBar" %>
<html>
<head>
   <script runat="server">
      Sub Page_Load( )
         'NB1.CreateBlankFile( )
      End Sub
   </script>
</head>
<body>
   <table border="1" width="100%" cellpadding="20" cellspacing="0">
      <tr>
         <td align="center" width="150">
            <img src="aspnetian.jpg"/>
         </td>
         <td align="center">
            <h1>NavBar Control Client Page<h1>
         </td>
      </tr>
      <tr>
         <td width="150">
            <form runat="server">
               <aspnetian:NavBar id="NB1"
                  showdividers="False" runat="server">
                  <strong>Navigation Bar</strong>
                  <br/>
               </aspnetian:NavBar>
            </form>
         </td>
         <td>
            This is where page content might be placed
            <br/><br/><br/><br/><br/><br/><br/><br/><br/>
         </td>
      </tr>
   </table>
</body>
</html>
```

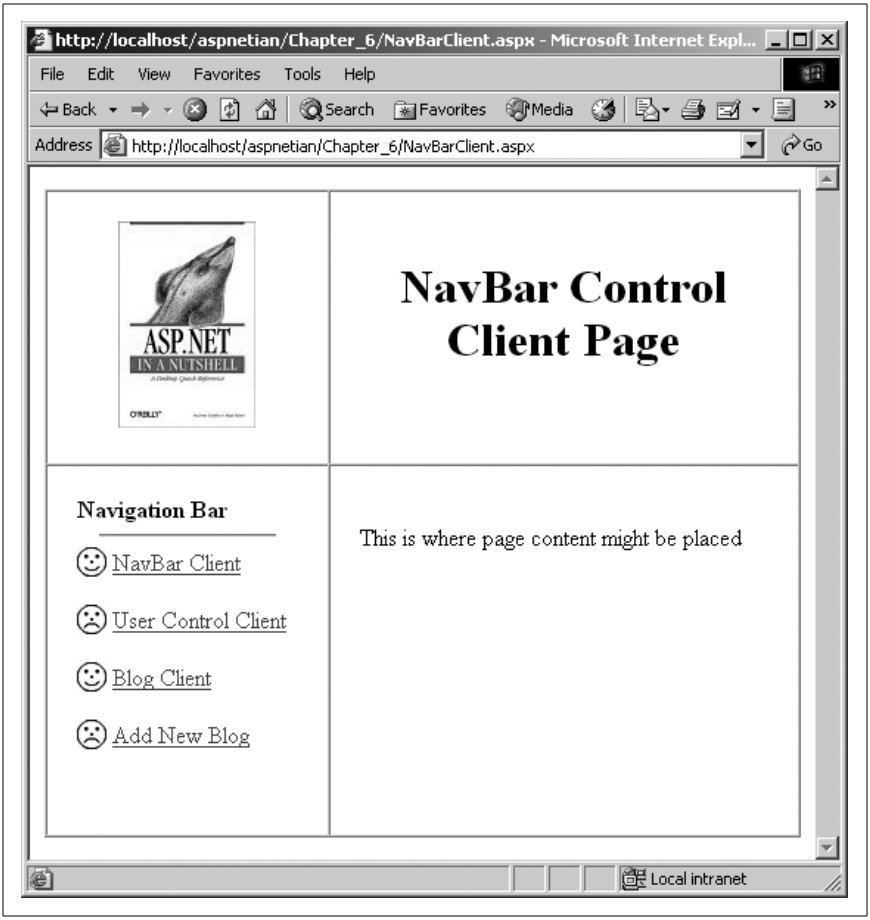


Figure 6-1. NavBarClient.aspx output

Compositional Controls

As mentioned earlier in the chapter, compositional controls render their output by combining appropriate controls within the CreateChildControls method, which is overridden in the custom control.

Example 6-7 shows the C# code for a compositional control that provides simple functionality for a *blog* (which is short for web log). The control has two modes: Add and Display. The mode is determined by the internal member_mode, which can be accessed by the public Mode property.

Like the NavBar control created in the previous example, the class definition for the Blog control specifies that the class derives from the Panel control (using C#'s : syntax), and also implements the INamingContainer interface. The INaming-Container interface contains no members, so there's nothing to actually implement. It's simply used to tell the ASP.NET runtime to provide a separate naming scope for controls contained within the custom control. This helps avoid

the possibility of naming conflicts at runtime, and also allows ASP.NET to properly manage the ViewState of child controls.

Also like the NavBar control, the Blog control uses an XML file to store the individual Blog entries. The example uses the same method of retrieving the data, namely creating a dataset and calling its ReadXml method, passing in the name of the XML file.

In addition to declaring the _mode member variable and the BlogDS dataset, the example declares two Textbox controls (which will be used when adding a new blog entry) and two more string member variables (_ addRedirect and _email).

The code in Example 6-7 then creates public property accessors for all three string variables. The Mode property determines whether the control displays existing blogs or displays fields for creating a new blog. The AddRedirect property takes the URL for a page to redirect to when a new blog is added. The Email property takes an email address to link to in each new blog field.

Next, the program overrides the OnInit method of the derived control to handle the Init event when it is called by the runtime. In this event handler, you call the LoadData method, which, like the same method in the NavBar control, loads the data from the XML file or, if no file exists, creates a blank file. It then calls the OnInit method of the base class to ensure that necessary initialization work is done.

Next is the overridden CreateChildControls method. Like the Render method, this method is called automatically by the ASP.NET runtime when the page is instantiated on the server. The timing of when CreateChildControls is called, however, is not predictable, since it may be called at different times during the lifecycle of the page, depending on how the control is coded, and other factors. Since the ASP.NET runtime will deliberately wait as long as possible to create the child controls, you may want to call the EnsureChildControls method (inherited from the Control class) to make sure that controls are created before you attempt to access them. A good example of this is when you expose a public property on your control that gets its value from a child control. If a client of your control attempts to access this property, and the child control has not yet been created, an exception will occur. To avoid this, you would add a call to EnsureChildControls to the property procedure:

```
Public Property MyTextValue( ) As String
   Get
      Me.EnsureChildControls( )
      Return CType(Controls(1), TextBox).Text
   End Get
   Set
      Me.EnsureChildControls( )
      CType(Controls(1), TextBox).Text = value.ToString( )
   End Set
End Property
```

Also unlike the Render method, you don't want to call the CreateChildControls method of the base class, or you'll create a loop in which this method calls itself recursively (and the ASP.NET process will hang). In the CreateChildControls method, you check the value of the _mode member variable and call either the

DisplayBlogs method or the NewBlog method, depending on the value of _mode. Note that this value is set by default to `display`, so if the property is not set, the control will be in display mode. Also note that the example uses the ToLower method of the `String` class to ensure that either uppercase or lowercase attribute values work properly.

The DisplayBlogs method iterates through the data returned in the dataset and instantiates controls to display this data. We use an `if` statement to determine whether more than one entry in a row has the same date. If so, we display only a single date header for the group of entries with the same date. We add an HtmlAnchor control to each entry to facilitate the readers' ability to bookmark the URL for a given entry. Then we write out the entry itself and add a contact email address and a link to the specific entry at the end of each entry.

Example 6-7. Blog.cs

```
using System;
using System.Data;
using System.Drawing;
using System.IO;
using System.Web;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;

namespace aspnetian
{

public class Blog:Panel, INamingContainer
{

   protected DataSet BlogDS;
   protected TextBox TitleTB;
   protected TextBox BlogText;

   private string _addRedirect;
   private string _email;
   private string _mode = "display";

   public string AddRedirect
   {
      get
      {
         return this._addRedirect;
      }
      set
      {
         this._addRedirect = value;
      }
   }

   public string Email
   {
```

Example 6-7. Blog.cs (continued)

```
        get
        {
            return this._email;
        }
        set
        {
            this._email = value;
        }
    }

    public string Mode
    {
        get
        {
            return this._mode;
        }
        set
        {
            this._mode = value;
        }
    }

    protected override void OnInit(EventArgs e)
    {
        LoadData( );
        base.OnInit(e);
    }

    protected override void CreateChildControls( )
    {
        this.Controls.Clear( );
        if (this._mode.ToLower( ) != "add")
        {
            DisplayBlogs( );
        }
        else
        {
            NewBlog( );
        }
    }

    protected void LoadData( )
    {
        BlogDS = new DataSet( );

        try
        {
            BlogDS.ReadXml(Page.Server.MapPath("Blog.xml"));
        }
        catch (FileNotFoundException fnfEx)
        {
            CreateBlankFile( );
```

Example 6-7. Blog.cs (continued)

```
        LoadData( );
    }
}

protected void DisplayBlogs( )
{
    DateTime BlogDate;
    DateTime CurrentDate = new DateTime( );

    DataRowCollection BlogRows = BlogDS.Tables[0].Rows;
    foreach (DataRow BlogDR in BlogRows)
    {
        string BDate = BlogDR["date"].ToString( );
        BlogDate = new DateTime(Convert.ToInt32(BDate.Substring(4, 4)),
            Convert.ToInt32(BDate.Substring(0, 2)),
            Convert.ToInt32(BDate.Substring(2, 2)));

        if (CurrentDate != BlogDate)
        {
            Label Date = new Label( );
            Date.Text = BlogDate.ToLongDateString( );
            Date.Font.Size = FontUnit.Large;
            Date.Font.Bold = true;
            this.Controls.Add(Date);
            this.Controls.Add(new LiteralControl("<br/><br/>"));
            CurrentDate = BlogDate;
        }

        HtmlAnchor Anchor = new HtmlAnchor( );
        Anchor.Name = "#" + BlogDR["anchorID"].ToString( );
        this.Controls.Add(Anchor);

        Label Title = new Label( );
        Title.Text = BlogDR["title"].ToString( );
        Title.Font.Size = FontUnit.Larger;
        Title.Font.Bold = true;
        this.Controls.Add(Title);

        this.Controls.Add(new LiteralControl("<p>"));
        LiteralControl BlogText = new LiteralControl("<div>" +
            BlogDR["text"].ToString( ) + "</div>");
        this.Controls.Add(BlogText);
        this.Controls.Add(new LiteralControl("</p>"));

        HyperLink Email = new HyperLink( );
        Email.NavigateUrl = "mailto:" + BlogDR["email"].ToString( );
        Email.Text = "E-mail me";
        this.Controls.Add(Email);

        this.Controls.Add(new LiteralControl(" | "));

        HyperLink AnchorLink = new HyperLink( );
```

Example 6-7. Blog.cs (continued)

```
            AnchorLink.NavigateUrl = Page.Request.Url.ToString( ) + "#" +
               BlogDR["anchorID"].ToString( );
            AnchorLink.Text = "Link";
            this.Controls.Add(AnchorLink);

            this.Controls.Add(new LiteralControl("<hr width='100%'/><br/>"));
        }
    }

    protected void NewBlog( )
    {
        Label Title = new Label( );
        Title.Text = "Create New Blog";
        Title.Font.Size = FontUnit.Larger;
        Title.Font.Bold = true;
        this.Controls.Add(Title);

        this.Controls.Add(new LiteralControl("<br/><br/>"));

        Label TitleLabel = new Label( );
        TitleLabel.Text = "Title: ";
        TitleLabel.Font.Bold = true;
        this.Controls.Add(TitleLabel);
        TitleTB = new TextBox( );
        this.Controls.Add(TitleTB);

        this.Controls.Add(new LiteralControl("<br/>"));

        Label BlogTextLabel = new Label( );
        BlogTextLabel.Text = "Text: ";
        BlogTextLabel.Font.Bold = true;
        this.Controls.Add(BlogTextLabel);
        BlogText = new TextBox( );
        BlogText.TextMode = TextBoxMode.MultiLine;
        BlogText.Rows = 10;
        BlogText.Columns = 40;
        this.Controls.Add(BlogText);

        this.Controls.Add(new LiteralControl("<br/>"));

        Button Submit = new Button( );
        Submit.Text = "Submit";
        Submit.Click += new EventHandler(this.Submit_Click);
        this.Controls.Add(Submit);
    }

    protected void Submit_Click(object sender, EventArgs e)
    {
        EnsureChildControls( );
        AddBlog( );
    }
```

Example 6-7. Blog.cs (continued)

```
    protected void AddBlog( )
    {
        DataRow NewBlogDR;
        NewBlogDR = BlogDS.Tables[0].NewRow( );
        NewBlogDR["date"] = FormatDate(DateTime.Today);
        NewBlogDR["title"] = TitleTB.Text;
        NewBlogDR["text"] = BlogText.Text;
        NewBlogDR["anchorID"] = Guid.NewGuid().ToString( );
        NewBlogDR["email"] = _email;
        BlogDS.Tables[0].Rows.InsertAt(NewBlogDR, 0);
        BlogDS.WriteXml(Page.Server.MapPath("Blog.xml"));
        Page.Response.Redirect(_addRedirect);
    }

    protected string FormatDate(DateTime dt)
    {
        string retString;

        retString = String.Format("{0:D2}", dt.Month);
        retString += String.Format("{0:D2}", dt.Day);
        retString += String.Format("{0:D2}", dt.Year);
        return retString;
    }

    protected void CreateBlankFile( )
    {
        // code to create new file...omitted to conserve space
    }

} // closing bracket for class declaration

} // closing bracket for namespace declaration
```

Displaying the blog entries is only half the battle. While it would certainly be possible to edit the XML file directly in order to add a new blog entry, it makes much more sense to make this a feature of the control. This is what the NewBlog method does. In the NewBlog method, we instantiate Label and TextBox controls for data entry and a Button control to submit the new blog entry. When the Button is clicked, the Submit_Click event handler method is called when the control is re-created on the server. The Submit_Click event handler, in turn, calls the AddBlog method to insert a new row into the BlogDS dataset and then writes the contents of the dataset back to the underlying XML file. Before using the control, of course, we'll need to compile it and place it in the application's *bin* directory. The following snippet can be used to compile the control:

```
csc /t:library /out:bin\blog.dll /r:system.dll,system.data.dll,
system.xml.dll,system.web.dll blog.cs
```

Example 6-8 shows the ASP.NET code necessary to instantiate the Blog control programmatically. Note the use of the PlaceHolder control to precisely locate the Blog control output. For this code to work correctly, the compiled assembly containing the Blog control must reside in the application's *bin* subdirectory.

Figure 6-2 shows the output of the control when used in the client page shown in Example 6-8.

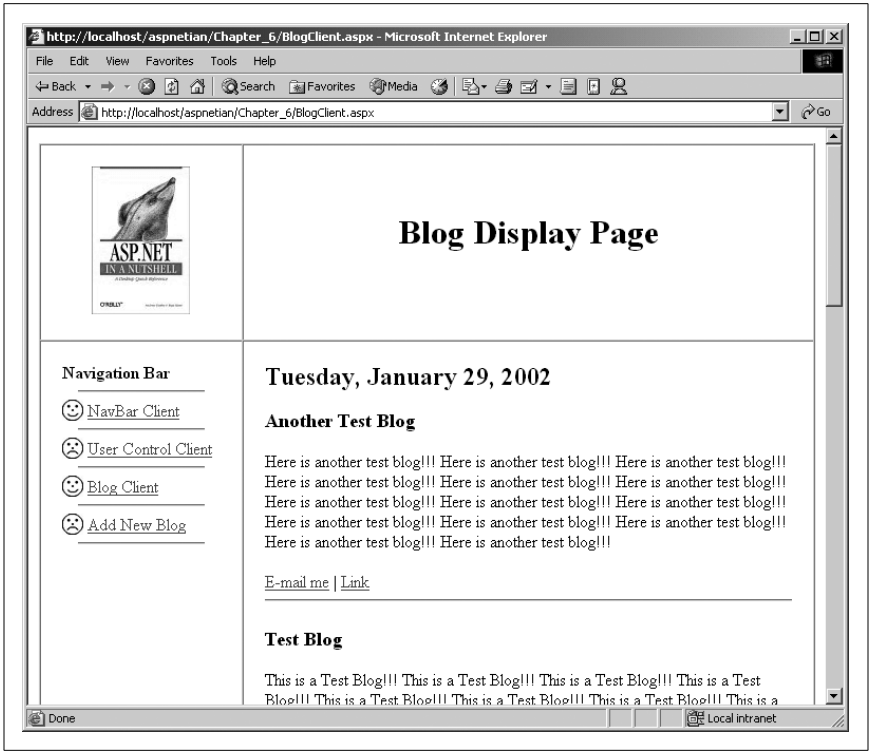


Figure 6-2. Output of BlogClient.aspx

Example 6-8. BlogClient.aspx

```
<%@ Page Language="vb" debug="true" %>
<%@ Register TagPrefix="aspnetian" Namespace="aspnetian"
   Assembly="NavBar" %>
<html>
<head>
   <script runat="server">
      Sub Page_Init( )
         Dim Blog1 As New Blog( )
         Blog1.SeparatorColor = System.Drawing.Color.Red
         PH.Controls.Add(Blog1)
      End Sub
   </script>
</head>
<body>
   <form runat="server">
   <table border="1" width="100%" cellpadding="20" cellspacing="0">
      <tr>
         <td align="center" width="150">
            <img src="aspnetian.jpg"/>
```

Example 6-8. BlogClient.aspx (continued)

```
        </td>
        <td align="center">
           <h1>Blog Display Page<h1>
        </td>
     </tr>
     <tr>
        <td width="150" valign="top">
           <aspnetian:NavBar id="NB1" runat="server">
              <strong>Navigation Bar</strong>
              <br/>
           </aspnetian:NavBar>
        </td>
        <td>
           <asp:placeholder id="PH" runat="server"/>
        </td>
     </tr>
  </table>
  </form>
</body>
</html>
```

Example 6-9 shows the code necessary to instantiate the control declaratively. The example uses the TagPrefix `aspnetian2` because both the NavBar control and the Blog control use the same namespace, but are compiled into separate assemblies (which means that using the same TagPrefix for both would result in an error).

Example 6-9. BlogAdd.aspx

```
<%@ Page Language="vb" debug="true" %>
<%@ Register TagPrefix="aspnetian" Namespace="aspnetian"
    Assembly="NavBar" %>
<%@ Register TagPrefix="aspnetian2" Namespace="aspnetian"
    Assembly="Blog" %>
<html>
<head>
   <script runat="server">
      Sub Page_Load( )
         'Uncomment the line below to explicitly create a blank
         '   XML file, then comment the line out again to run the control
         'NB1.CreateBlankFile( )
      End Sub
   </script>
</head>
<body>
   <form runat="server">
   <table border="1" width="100%" cellpadding="20" cellspacing="0">
      <tr>
         <td align="center" width="150">
            <img src="aspnetian.jpg"/>
         </td>
         <td align="center">
            <h1>Blog Add Page<h1>
         </td>
```

Example 6-9. BlogAdd.aspx (continued)

```
      </tr>
      <tr>
         <td width="150" valign="top">
            <aspnetian:NavBar id="NB1" runat="server">
               <strong>Navigation Bar</strong>
               <br/>
            </aspnetian:NavBar>
         </td>
         <td>
            <aspnetian2:Blog id="Blog1"
               mode="Add"
               addredirect="BlogClient.aspx"
               email="blogs@aspnetian.com"
               runat="server"/>
         </td>
      </tr>
   </table>
   </form>
</body>
</html>
```

As you can see, whether the control is used programmatically or declaratively, the amount of code necessary to provide simple blogging functionality is made trivial by the use of a custom server control. Note that you can also have the same page use the Blog control in either Display or Add mode, depending on the user's actions, as explained in the following section.

Adding Design-Time Support

While using the Blog control in a Web Forms page is fairly simple, it's still not 100% intuitive. For example, without documentation, there's no way for someone using the Blog control to know what the appropriate values for the Mode property are. Without explicitly telling developers using the control about the Add mode, it would be difficult for them to discover and use this mode on their own.

For developers using Visual Studio .NET (or another IDE that supports IntelliSense), you can solve this problem by adding design-time support to the control. This is done by using a combination of special metadata attributes added to the control and custom XSD schemas to support IntelliSense statement completion for Web Forms pages. IntelliSense support in code-behind modules is automatic and requires no additional coding.

Part of the challenge of providing design-time support for custom server controls is that different editors in the Visual Studio IDE require different techniques to support design-time functionality. Custom controls automatically support IntelliSense statement completion when working with code-behind modules in Visual Basic .NET or C#. Figure 6-3 shows this statement completion in action for the Blog control.

Unfortunately, when editing Web Forms pages, automatic support for statement completion does not extend to the Design or HTML views (nor does Visual Studio provide built-in support for viewing and editing properties in the Property

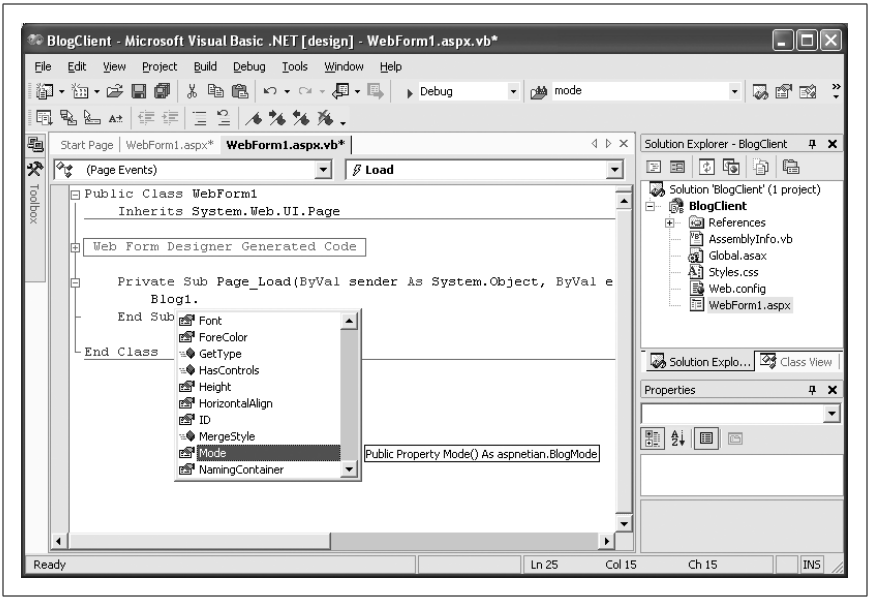


Figure 6-3. IntelliSense in code-behind

browser without additional work in your control). To complicate things further, one technique is necessary for supporting IntelliSense in the Property browser and Design view of the Web Forms editor, while another is necessary for supporting it in the HTML view of the Web Forms editor.

The technique required for supporting property browsing in Design view uses metadata attributes to inform Visual Studio .NET about how to handle the properties. Supporting statement completion and property browsing in HTML view requires creating a custom XSD schema that describes the types in your control. We'll discuss both techniques in the next sections.

Metadata attributes

Visual Studio .NET provides rich support for designing and modifying controls visually by using drag-and-drop techniques and tools, such as the Property browser, and related designers, such as the color picker. Support for these tools is provided by a series of metadata attributes that you can add to your control. These attributes tell the Visual Studio IDE whether to display any properties that your control exposes in the Properties browser, what type the properties are, and which designer should be used to set the properties' values.

To support editing of the AddRedirect property in the Property browser, we would add the following attributes before the Property procedure, as shown in the following code snippet:

```
[
Browsable(true),
Category("Behavior"),
Description("URL to which the page should redirect after
```

```
    successful submission of a new Blog entry."),
Editor(typeof(System.Web.UI.Design.UrlEditor), typeof(UITypeEditor))
]
public string AddRedirect
{ // property procedure code }
```

These attribute declarations allow the property to be displayed in the Property browser, set the desired category for the property (when properties are sorted by category), provide a description of the property, and tell Visual Studio .NET to use the UrlEditor designer to edit the property's value.

Additional Uses for Metadata

Metadata attributes aren't just for use by the Visual Studio .NET designer. In fact, metadata attributes are used throughout the .NET Framework to allow developers (both the framework developers, and those who use the framework) to add descriptive, configuration, and other types of information to assemblies, classes, and/or class members.

You can also create your own custom attributes in your applications, though the specifics of doing so is beyond the scope of this book.

The attribute syntax shown in this section is for C#. In C#, attributes take the form:

```
[AttributeName(AttributeParams)]
```

In Visual Basic .NET, attributes are declared with the following syntax:

```
<AttributeName(AttributeParams)>
```

Visual Basic .NET requires that the attribute declaration appear on the same line as the member it's modifying, so it's usually a good idea to follow the attribute with a VB line continuation character to improve readability:

```
<AttributeName(AttributeParams)> _
Public Membername( )
```

In both C# and VB, you can declare multiple attributes within a single set of [ ] or <> brackets by separating multiple attributes with commas.

In addition to setting attributes at the property level, you can set certain attributes at the class and assembly levels. For example, you can use the assembly-level attribute `TagPrefix` to specify the tag prefix to use for any controls contained in the assembly. Visual Studio .NET then inserts this tag prefix automatically when you add an instance of the control to a Web Forms page from the Visual Studio toolbox. The following code snippet shows the syntax for the `TagPrefix` attribute. This attribute should be placed within the class module that defines the control, but outside the class and namespace declarations.

```
[
assembly: TagPrefix("aspnetian", "aspnetian")
]
```

```
namespace aspnetian
{ // control classes, etc. }
```

To complete the integration of a control in the Visual Studio .NET environment, add the `ToolBoxData` attribute (which tells Visual Studio .NET your preferred tag name for controls inserted from the toolbox) to the class that implements the control:

```
[
ToolboxData("<{0}:Blog runat=server></{0}:Blog>")
]
public class Blog:Panel, INamingContainer
{ // control implementation }
```

Once compiled, the control will support automatic insertion of the `@ Register` directive, tag prefix, and tag name for the Blog control. To add the control to the Visual Studio .NET toolbox, follow these simple steps:

1. In Design view, select the Web Forms tab of the Visual Studio .NET toolbox.
2. Right-click anywhere in the tab and select Add/Remove Items....
3. With the .NET Framework Components tab selected, click Browse.
4. Browse to the location of the compiled control assembly, select it, and click Open.
5. Click OK.

Once the control has been added to the toolbox, you can add it to a Web Forms page by either double-clicking the control or dragging and dropping it from the toolbox onto the Web Forms page. In either case, Visual Studio .NET will automatically insert the correct `@ Register` directive, including setting the TagPrefix based on the assembly-level attribute, and will also create a set of tags for the control with the tag name specified in the `ToolBoxData` attribute.

Adding a control designer

As written, the Blog control will not have any visible interface in the Design view of the Web Forms editor. This can make it more difficult to select the control on the page, and also may make it more difficult to understand what the control will look like at runtime. To correct this problem, we can add support for a designer that will render HTML at design time that approximates the look of the Blog control at runtime. Note that you can also create designers that completely reproduce the runtime output of a control, but doing so is more involved and beyond the scope of this book.

All server control designers derive from the class `System.Web.UI.Design.ControlDesigner`, which exposes a number of methods you can override to provide design-time rendering for your control. Example 6-10 overrides the GetDesignTimeHtml method to return simple HTML. Note that the example shows the entire designer class for the Blog control, which you can add to the existing *Blog.cs* class file (making sure that the class declaration is within the `namespace` curly braces).

Example 6-10. BlogDesigner class

```
public class BlogDesigner:ControlDesigner
{
   public override string GetDesignTimeHtml( )
   {
      return "<h1>Blog</h1><hr/><hr/>";
   }
}
```

To tie this designer into the Blog class, we use the `Designer` attribute, as shown in the following snippet. Note that this code also adds a `Description` attribute that describes what the control does.

```
[
Description("Simple Blog control. Supports display of Web log / news
   items from an XML file."),
Designer(typeof(aspnetian.BlogDesigner)),
ToolboxData("<{0}:Blog runat=server></{0}:Blog>")
]
public class Blog:Panel, INamingContainer
{ // class implementation }
```

As you can see, the `BlogDesigner` class is extremely simple, but it adds a lot to the control's design-time appearance on a web page, as shown in Figure 6-4.

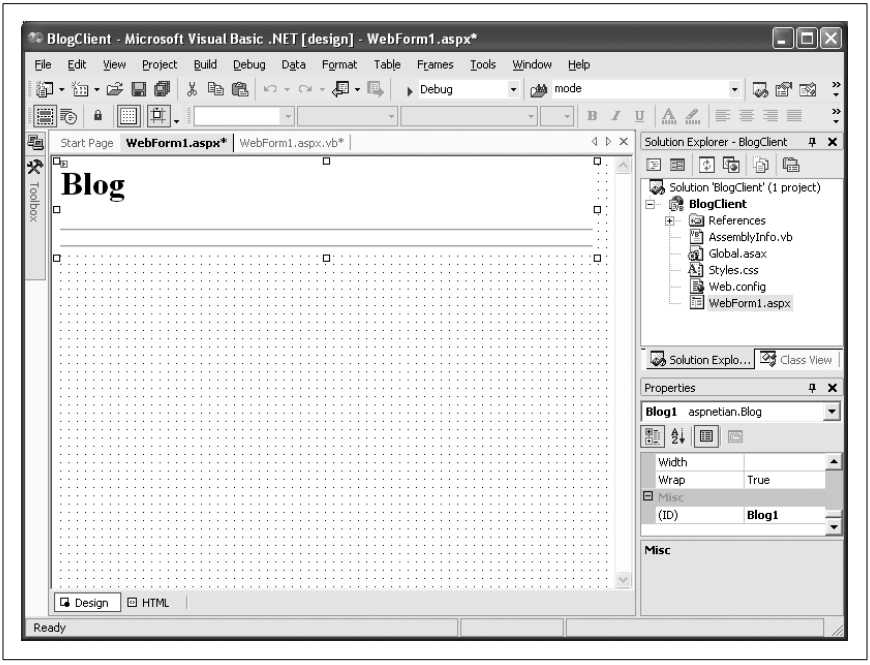


Figure 6-4. Adding design-time rendering

Example 6-11 shows the code for the Blog control, updated with attributes to enable design-time support for the control in Design view and the Property browser. Note that the example adds several using directives to import the namespaces needed to support the attributes and designer classes we've used. The example also adds an enumeration to be used for the value of the Mode property and a new property, SeparatorColor.

Example 6-11. Updated Blog.cs

```
using System;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Design;
using System.IO;
using System.Web;
using System.Web.UI;
using System.Web.UI.Design;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;

[
assembly: TagPrefix("aspnetian", "aspnetian")
]

namespace aspnetian
{

public enum BlogMode
{
   Add,
   Display
}

[
Description(@"Simple Blog control. Supports display of Web log / news
   items from an XML file."),
Designer(typeof(aspnetian.BlogDesigner)),
ToolboxData("<{0}:Blog runat=server></{0}:Blog>")
]
public class Blog:Panel, INamingContainer
{

   protected DataSet BlogDS;
   protected TextBox TitleTB;
   protected TextBox BlogText;

   private string _addRedirect;
   private string _email;
   private BlogMode _mode;
   private Color _separatorColor = Color.Black;

   [
```

Example 6-11. Updated Blog.cs (continued)

```
Browsable(true),
Category("Behavior"),
Description("URL to which the page should redirect after
   successful submission of a new Blog entry."),
Editor(typeof(System.Web.UI.Design.UrlEditor), typeof(UITypeEditor))
]
public string AddRedirect
{
   get
   {
      return this._addRedirect;
   }
   set
   {
      this._addRedirect = value;
   }
}

[
Browsable(true),
Category("Behavior"),
Description("Email address the control will use for listing in new
   Blog entries.")
]
public string Email
{
   get
   {
      return this._email;
   }
   set
   {
      this._email = value;
   }
}

[
Browsable(true),
Category("Behavior"),
Description("Controls whether existing Blogs are displayed, or
   fields for creating a new Blog entry.")
]
public BlogMode Mode
{
   get
   {
      return this._mode;
   }
   set
   {
      this._mode = value;
```

Example 6-11. Updated Blog.cs (continued)

```
    }
}

[
Browsable(true),
Category("Appearance"),
Description("Controls the color of the line that separates Blog
   entries when in display mode.")
]
public Color SeparatorColor
{
   get
   {
      return this._separatorColor;
   }
   set
   {
      this._separatorColor = value;
   }
}

protected override void OnInit(EventArgs e)
{
   LoadData( );
   base.OnInit(e);
}

protected override void CreateChildControls( )
{
   if (this._mode != BlogMode.Add)
   {
      DisplayBlogs( );
   }
   else
   {
      NewBlog( );
   }
}

protected void LoadData( )
{
   BlogDS = new DataSet( );

   try
   {
      BlogDS.ReadXml(Page.Server.MapPath("Blog.xml"));
   }
   catch (FileNotFoundException fnfEx)
   {
      CreateBlankFile( );
      LoadData( );
```

Example 6-11. Updated Blog.cs (continued)

```
    }
}

protected void DisplayBlogs( )
{
    DateTime BlogDate;
    DateTime CurrentDate = new DateTime( );

    DataRowCollection BlogRows = BlogDS.Tables[0].Rows;
    foreach (DataRow BlogDR in BlogRows)
    {
        string BDate = BlogDR["date"].ToString( );
        BlogDate = new DateTime(Convert.ToInt32(BDate.Substring(4, 4)),
            Convert.ToInt32(BDate.Substring(0, 2)),
            Convert.ToInt32(BDate.Substring(2, 2)));

        if (CurrentDate != BlogDate)
        {
            Label Date = new Label( );
            Date.Text = BlogDate.ToLongDateString( );
            Date.Font.Size = FontUnit.Large;
            Date.Font.Bold = true;
            this.Controls.Add(Date);
            this.Controls.Add(new LiteralControl("<br/><br/>"));
            CurrentDate = BlogDate;
        }

        HtmlAnchor Anchor = new HtmlAnchor( );
        Anchor.Name = "#" + BlogDR["anchorID"].ToString( );
        this.Controls.Add(Anchor);

        Label Title = new Label( );
        Title.Text = BlogDR["title"].ToString( );
        Title.Font.Size = FontUnit.Larger;
        Title.Font.Bold = true;
        this.Controls.Add(Title);

        this.Controls.Add(new LiteralControl("<p>"));
        LiteralControl BlogText = new LiteralControl("<div>" +
            BlogDR["text"].ToString( ) + "</div>");
        this.Controls.Add(BlogText);
        this.Controls.Add(new LiteralControl("</p>"));

        HyperLink Email = new HyperLink( );
        Email.NavigateUrl = "mailto:" + BlogDR["email"].ToString( );
        Email.Text = "E-mail me";
        this.Controls.Add(Email);

        this.Controls.Add(new LiteralControl(" | "));
        HyperLink AnchorLink = new HyperLink( );
        AnchorLink.NavigateUrl = Page.Request.Url.ToString( ) + "#" +
            BlogDR["anchorID"].ToString( );
```

Example 6-11. Updated Blog.cs (continued)

```
        AnchorLink.Text = "Link";
        this.Controls.Add(AnchorLink);

        this.Controls.Add(new LiteralControl("<hr color='" +
           _separatorColor.ToKnownColor( ) + "' width='100%'/><br/>"));
    }
}

protected void NewBlog( )
{
    Label Title = new Label( );
    Title.Text = "Create New Blog";
    Title.Font.Size = FontUnit.Larger;
    Title.Font.Bold = true;
    this.Controls.Add(Title);

    this.Controls.Add(new LiteralControl("<br/><br/>"));

    Label TitleLabel = new Label( );
    TitleLabel.Text = "Title: ";
    TitleLabel.Font.Bold = true;
    this.Controls.Add(TitleLabel);
    TitleTB = new TextBox( );
    this.Controls.Add(TitleTB);

    this.Controls.Add(new LiteralControl("<br/>"));

    Label BlogTextLabel = new Label( );
    BlogTextLabel.Text = "Text: ";
    BlogTextLabel.Font.Bold = true;
    this.Controls.Add(BlogTextLabel);
    BlogText = new TextBox( );
    BlogText.TextMode = TextBoxMode.MultiLine;
    BlogText.Rows = 10;
    BlogText.Columns = 40;
    this.Controls.Add(BlogText);

    this.Controls.Add(new LiteralControl("<br/>"));

    Button Submit = new Button( );
    Submit.Text = "Submit";
    Submit.Click += new EventHandler(this.Submit_Click);
    this.Controls.Add(Submit);
}

protected void Submit_Click(object sender, EventArgs e)
{
    EnsureChildControls( );
    AddBlog( );
}

protected void AddBlog( )
{
```

Example 6-11. Updated Blog.cs (continued)

```
        DataRow NewBlogDR;
        NewBlogDR = BlogDS.Tables[0].NewRow( );
        NewBlogDR["date"] = FormatDate(DateTime.Today);
        NewBlogDR["title"] = TitleTB.Text;
        NewBlogDR["text"] = BlogText.Text;
        NewBlogDR["anchorID"] = Guid.NewGuid().ToString( );
        NewBlogDR["email"] = _email;
        BlogDS.Tables[0].Rows.InsertAt(NewBlogDR, 0);
        BlogDS.WriteXml(Page.Server.MapPath("Blog.xml"));
        Page.Response.Redirect(_addRedirect);
    }

    protected string FormatDate(DateTime dt)
    {
        string retString;
        retString = String.Format("{0:D2}", dt.Month);
        retString += String.Format("{0:D2}", dt.Day);
        retString += String.Format("{0:D2}", dt.Year);
        return retString;
    }

    public void CreateBlankFile( )
    {
        // code to create new file...omitted to conserve space
    }

}

public class BlogDesigner:ControlDesigner
{
    public override string GetDesignTimeHtml( )
    {
        return "<h1>Blog</h1><hr/><hr/>";
    }
}

}
```


Custom schemas and Visual Studio annotations

As much as the metadata attributes described in the previous section help provide support for the Blog control at design time, they're missing one important piece: IntelliSense support for adding tags and attributes in the HTML view of the Web Forms editor. For developers who are more comfortable working in HTML than in WYSIWYG style, this oversight is significant.

Since the HTML view of the Web Forms editor uses XSD schemas to determine which elements and attributes to make available in a Web Forms page, to correct the oversight, we need to implement an XSD schema that describes the Blog control and the attributes that it supports. Optionally, we can add annotations to the schema that tell Visual Studio .NET about the various elements and how we'd like them to behave.

Example 6-12 contains the portion of the XSD schema specific to the Blog control. The actual schema file (contained in the sample code for the book, which may be obtained from the book's page at the O'Reilly web site: *http://www.oreilly.com/catalog/aspdotnetnut2*) also contains type definitions for the Panel control from which the Blog control is derived, as well as other necessary attribute and type definitions. These definitions were copied from the *asp.xsd* schema file created for the built-in ASP.NET Server Controls.

You should never modify the *asp.xsd* schema file directly, but should copy any necessary type or attribute definitions to your custom schema file. While this may seem redundant, if you edit *asp.xsd* directly and a later installation or service pack for the .NET Framework overwrites this file, your custom schema entries will be lost.

Example 6-12. Blog.xsd

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema targetNamespace="urn:http://www.aspnetian.com/schemas"
   elementFormDefault="qualified"
   xmlns="urn:http://www.aspnetian.com/schemas"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:vs="http://schemas.microsoft.com/Visual-Studio-Intellisense"
   vs:friendlyname="Blog Control Schema"
   vs:ishtmlschema="false"
   vs:iscasesensitive="false"
   vs:requireattributequotes="true" >
   <xsd:annotation>
      <xsd:documentation>
         Blog Control schema.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="Blog" type="BlogDef" />

   <!-- <aspnetian:Blog> -->
   <xsd:complexType name="BlogDef">
      <!-- <aspnetian:Blog>-specific attributes -->
      <xsd:attribute name="AddRedirect" type="xsd:string"
         vs:builder="url"/>
      <xsd:attribute name="Email" type="xsd:string"/>
      <xsd:attribute name="Mode" type="BlogMode"/>
      <xsd:attribute name="SeparatorColor" type="xsd:string"
         vs:builder="color"/>
      <!-- <asp:Panel>-specific attributes -->
      <xsd:attribute name="BackImageUrl" type="xsd:anyURI" />
      <xsd:attribute name="HorizontalAlign" type="HorizontalAlign" />
      <xsd:attribute name="Wrap" type="xsd:boolean" />
      <xsd:attribute name="Enabled" type="xsd:boolean" />
      <xsd:attribute name="BorderWidth" type="ui4" />
      <xsd:attribute name="BorderColor" type="xsd:string"
         vs:builder="color" />
      <xsd:attribute name="BorderStyle" type="BorderStyle" />
```

Example 6-12. Blog.xsd (continued)

```
      <xsd:attributeGroup ref="WebControlAttributes" />
   </xsd:complexType>

   <!-- DataTypes -->
   <xsd:simpleType name="BlogMode">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="Add" />
         <xsd:enumeration value="Display" />
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

In Example 6-12, note the `targetNamespace` and `xmlns` attributes on the root `schema` element, which define the XML namespace for the control's schema. The value of the `targetNamespace` and `xmlns` attributes will also be used as an attribute in your Web Forms page to "wire up" the schema. The `<xsd:element>` tag defines the root Blog element. The `<xsd:complexType>` tag defines the attributes for the Blog element, which includes the web control attributes referenced by the `<xsd:attributeGroup>` tag. Finally, the `<xsd:simpleType>` tag defines the enumeration for the BlogMode type used as one of the attributes for the Blog element.

Note that Example 6-12 uses the `vs:builder` annotation to tell Visual Studio .NET to use the Url builder for the `AddRedirect` attribute and the Color builder for the `SeparatorColor` attribute. The `vs:builder` annotation is one of many annotations available to modify schemas. The most commonly used are listed in Table 6-1.

Table 6-1. Common Visual Studio .NET annotations

Annotation	Purpose	Valid values
vs:absolutepositioning	Used at the root `<schema>` element to determine whether Visual Studio may insert style attributes for positioning.	true/false
vs:blockformatted	Indicates whether leading whitespace may be added to the element during automatic formatting.	true/false
vs:builder	Specifies the builder to be used for editing the related property's value.	color, style, or url
vs:deprecated	Allows a related property to be marked as "deprecated", which prevents it from showing up in the Properties browser and in statement completion.	true/false
vs:empty	Used at the element level to indicate that Visual Studio .NET should use single tag syntax for the related tag (no end tag).	true/false
vs:friendlyname	Used at the root level to provide a display name for the schema.	
vs:iscasesensitive	Used at the root level and specifies whether Visual Studio .NET will treat the related tags in a case-sensitive manner.	true/false
vs:ishtmlschema	Used at the root level and specifies whether the schema is an HTML document schema.	true/false
vs:nonbrowseable	Used at the attribute level and specifies that the attribute should not appear in statement completion.	true/false
vs:readonly	Used at the attribute level and specifies that the attribute may not be modified in the Properties window.	true/false
vs:requireattributequotes	Used at the root level and specifies that the attribute values must have quotes.	true/false

Once you've built your XSD schema, save it to the same location as the *asp.xsd* file (which defaults to *C:\ProgramFiles\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml*).

To allow Visual Studio .NET to read your custom schema, you'll need to add an `xmlns` attribute to the `<body>` tag of the page in which you wish to use the schema, as shown in the following snippet:

```
<body xmlns:aspnetian="urn:http://www.aspnetian.com/schemas">
```

Notice that this code uses the `aspnetian` prefix with the `xmlns` attribute to specify that the schema is for controls prefixed with the `aspnetian` tag prefix. This recall is set up by the `TagPrefix` attribute (described earlier in "Metadata attributes"). The value of the `xmlns` attribute should be the same as the `targetNamespace` attribute defined at the root of the schema.

Once you've wired up your schema via the `xmlns` attribute, you should be able to type an opening `<` character and the first few letters of the `aspnetian` namespace and have the Blog control appear as one of the options for statement completion, as shown in Figure 6-5.

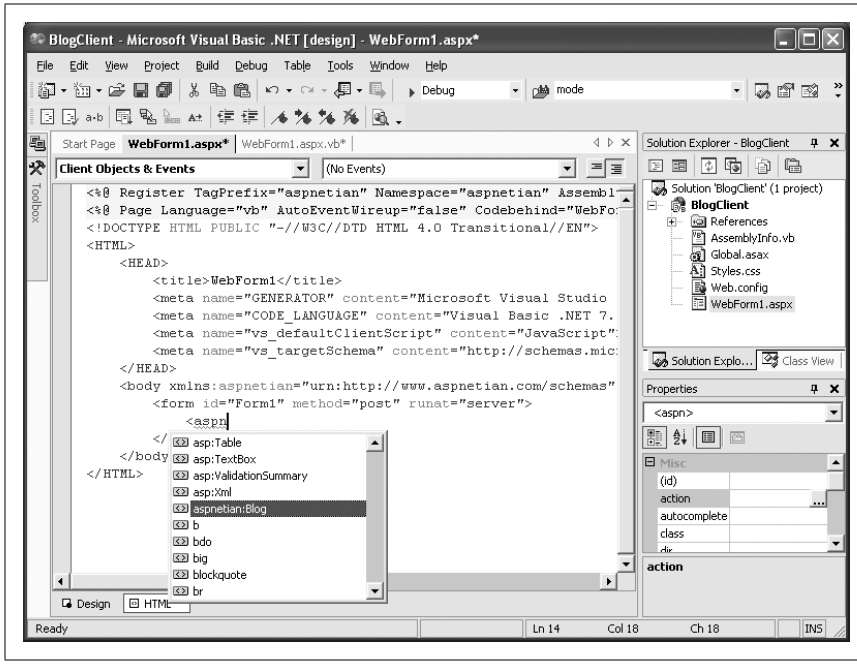


Figure 6-5. Statement completion in HTML view

Example 6-13 shows the code for a page that uses the Blog control from Visual Studio .NET, including the `xmlns` attribute added to the `<body>` element.

Example 6-13. BlogClient_VS.aspx

```
<%@ Register TagPrefix="aspnetian" Namespace="aspnetian"
   Assembly="Blog" %>
<%@ Page Language="vb" AutoEventWireup="True" Debug="True"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Blog Client</title>
      <meta content="Microsoft Visual Studio.NET 7.0" name="GENERATOR">
      <meta content="Visual Basic 7.0" name="CODE_LANGUAGE">
      <meta content="JavaScript" name="vs_defaultClientScript">
      <meta content="http://schemas.microsoft.com/intellisense/ie5"
        name="vs_targetSchema">
      <script runat="server">
         Sub Page_Load( )
            If Request.QueryString("mode") = "add" Then
               Blog1.Mode = BlogMode.Add
               Link1.Visible = False
               Link2.Visible = False
            Else
               Blog1.Mode = BlogMode.Display
               Link1.Visible = True
               Link2.Visible = True
            End If
         End Sub
      </script>
   </head>
   <body xmlns:aspnetian="urn:http://www.aspnetian.com/schemas">
      <form id="Form1" method="post" runat="server">
         <p><asp:hyperlink id="Link1" runat="server"
               navigateurl="WebForm1.aspx?mode=add">
               Add Blog
            </asp:hyperlink></p>
         <p><aspnetian:blog id="Blog1" addredirect="WebForm1.aspx"
               email="andrew@aspnetian.com" runat="server" >
            </aspnetian:blog></p>
         <p><asp:hyperlink id="Link2" runat="server"
               navigateurl="WebForm1.aspx?mode=add">
               Add Blog
            </asp:hyperlink></p>
      </form>
   </body>
</html>
```

Notice that Example 6-13 provides support for both displaying and adding blog entries from within the same page; this is done by omitting the Mode property in the tag that defines the control and setting the Mode programmatically (based on whether or not the page request was the result of the user clicking one of the "Add Blog" Hyperlink controls added to the page).

When the page is loaded for the first time, it will be in Display mode. Clicking one of the hyperlinks will request the page with the mode QueryString element set to `add`, which will cause the page to render in Add mode.

Adding Client Script

Sometimes you may want to use client-side script in your ASP.NET pages, either with controls or independent of them. In classic ASP, it was possible to write client script to the browser using Response.Write. However, this could get very messy—particularly if you needed to write the same set of code for use with more than one form element.

The ASP.NET `Page` class provides several methods for sending client script to the browser that make this process simpler and more reliable.

These methods include:

RegisterClientScriptBlock
: Renders a string containing the specified client script to the browser.

RegisterHiddenField
: Adds an `<input>` element whose type is set to `hidden`.

IsClientScriptBlockRegistered
: Allows you to test whether a given named script block has been already registered by another control to avoid redundancy.

You might use these methods to pop up a message box on the client with the number of Blogs that currently exist in the XML file. To accomplish this, add the following snippet to the DisplayBlogs method of the Blog control:

```
Page.RegisterClientScriptBlock("Blog", "<script>alert('There are now " +
   BlogRows.Count + " Blogs!');</script>");
```

Then, if any other controls need to use the same script, call IsClientScriptBlockRegistered, passing it the name of the script shown above, `Blog`, to determine whether to call RegisterClientScriptBlock again. In this way, a single client-side script block may be shared among multiple controls.

When using any of the methods discussed in this section, you should always check the built-in browser capabilities class to ensure that the client supports script (Request.Browser.JavaScript or Request.Browser.VBScript). Additionally, you should ensure that you call the method(s) either prior to or in the PreRender event handler, to ensure that the script is written to the client properly.

Sharing Controls Across Applications

The architecture of the .NET Framework makes using a custom server control or other assembly as simple as copying that assembly to the *bin* subdirectory of your application and adding the appropriate directives and tags to your page. However, there may be times when you would like multiple applications on the same machine to be able to use the same control, without having multiple local copies of the control's assembly floating around.

Fortunately, .NET addresses this need with the Global Assembly Cache (GAC), a repository of shared assemblies that are accessible to all .NET applications on a

given machine. Adding your own control assemblies to the GAC is a relatively straightforward process that requires four steps:

1. Use the *sn.exe* command-line utility to create a public key pair for use in signing your control:

   ```
   sn.exe -k Blog.snk
   ```

2. Add the `AssemblyKeyFileAttribute` to the file containing the control code, passing the path to the keyfile created in Step 1 as an argument. (This is an assembly-level attribute, so it should be placed outside of any namespace or class definitions.) When compiled, this attribute will result in a strongly named assembly that can be placed in the GAC:

   ```
   [assembly: AssemblyKeyFileAttribute("Blog.snk")]
   ```

3. Recompile the control.

4. Add the control to the GAC, either by dragging and dropping the assembly in Windows Explorer or by using the *gacutil.exe* utility, as follows:

   ```
   gacutil -i Blog.dll
   ```

Note that as with the *csc.exe* and *vbc.exe* command-line compilers, using the *sn.exe* and *gacutil.exe* utilities without a fully qualified path requires that you have the path to these utilities registered as part of your `PATH` environment variable. The *sn.exe* and *gacutil.exe* utilities are typically located in the *\FrameworkSDK\bin* directory, which is installed either under *ProgramFiles\Microsoft.NET* or *ProgramFiles\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin*, depending on whether you've installed just the .NET Framework SDK or Visual Studio .NET.

Once you've added the control assembly to the GAC, you can use it from any application on the machine. One caveat: to use custom controls that are installed in the GAC, you must supply the version, culture, and public key information for the assembly when adding the `@ Register` directive for the control, as shown in the following snippet (which should appear on a single line):

```
<%@ Register TagPrefix="aspnetian" Namespace="aspnetian" Assembly="Blog,
   Version=0.0.0.0, Culture=neutral, PublicKeyToken=6bd31f35fc9a113b" %>
```

If you've added your control to the Visual Studio .NET toolbox, when you use the control from the toolbox, the correct `@ Register` directive will be generated for you automatically.

Additional Resources

The following site provides additional information on the topics discussed in this chapter:

http://www.aspnextgen.com/
: The DotNetJunkies site, run by Microsoft MVP Award winners Donny Mack and Doug Seven, contains many ASP.NET tutorials, including some on building custom server controls and user controls.