4 Introduction to ASP.NET Web Form Controls

Chapter 3 introduced Dreamweaver and many of its editing tools. It also showed you how to create a site. In this chapter we'll dive into ASP.NET Web Forms and the controls used in them. We'll start out by discussing what Web Forms are and how controls can turn them into dynamic pages. Then we'll create a Web Form and then add controls to see how they perform. Later in the chapter, we'll get into more advanced stuff, so that by the end of the chapter we'll be writing code to programmatically alter the contents of our Web Form.

Preparing Our Web Site

A Web Form is a potentially dynamic Web page. It doesn't have to contain dynamically created content, but it usually does. One use of a dynamic Web Form is a Web page that displays up-to-date items from a sales catalog. Web Forms are similar to HTML pages in that they contain HTML code and JavaScript. However, instead of ending with the .html extension, they end in the .aspx extension. More important, they serve as containers for most other parts of ASP.NET, such as controls. Controls are the building blocks of dynamic content, and that's what we'll be adding to the Web Form we create in this chapter. To process the logic that drives the dynamic content, we'll include blocks of code, another differentiating factor from plain HTML Web pages, which don't contain these code blocks.

Let's start fresh by creating a new site in Dreamweaver to feature our Web Forms. Since we learned in detail how to create a new site in Chapter 3, we'll use the Site Definition wizard to speed the process here.

To create a site:

1. From the Site drop-down menu at the top of the Dreamweaver interface, choose New Site (**Figure 4.1**).

   This opens the Site Definition wizard. Make sure it's in Basic mode (**Figure 4.2**).

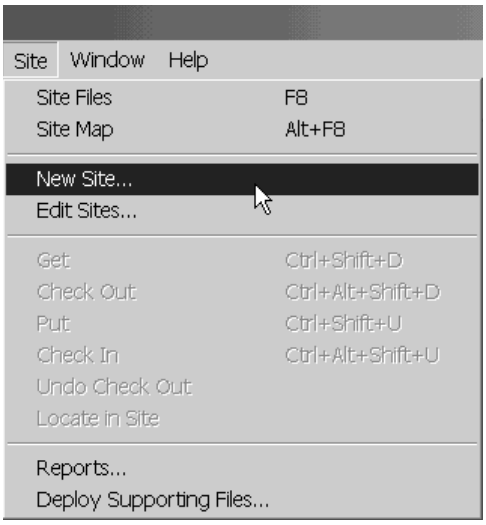

Figure 4.1 Site drop-down menu.

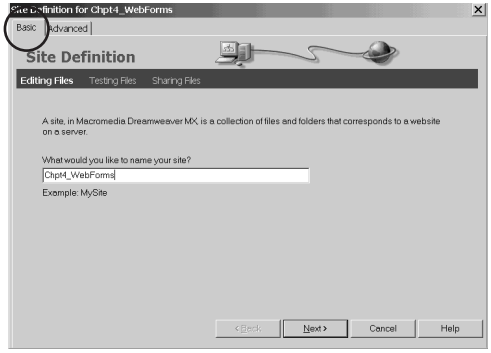

Figure 4.2 In Step 1 of the Site Definition wizard, you name your site.

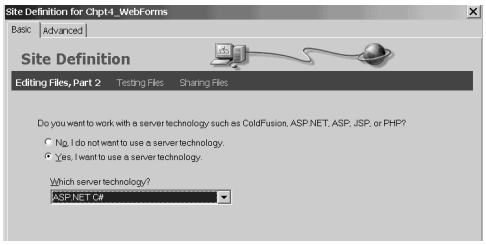

Figure 4.3 In Step 2 of the Site Definition wizard, you pick ASP.NET C# as your server technology.

2. Name the site "Chpt4_WebForms" and click Next.

   The wizard will present some new questions about your server technology preference (**Figure 4.3**).

3. Click the second radio button to indicate that you want to use a server technology. For this book, we'll use ASP.NET C#, so pick that as your server technology. Click Next to proceed to the next questions.

4. Accept the default answers for the rest of the screens the wizard presents.

   At the conclusion, the wizard will show a summary of the settings you've chosen (**Figure 4.4**).

5. Confirm your entries, then click Done.

   This will create the site according to the settings you've specified.

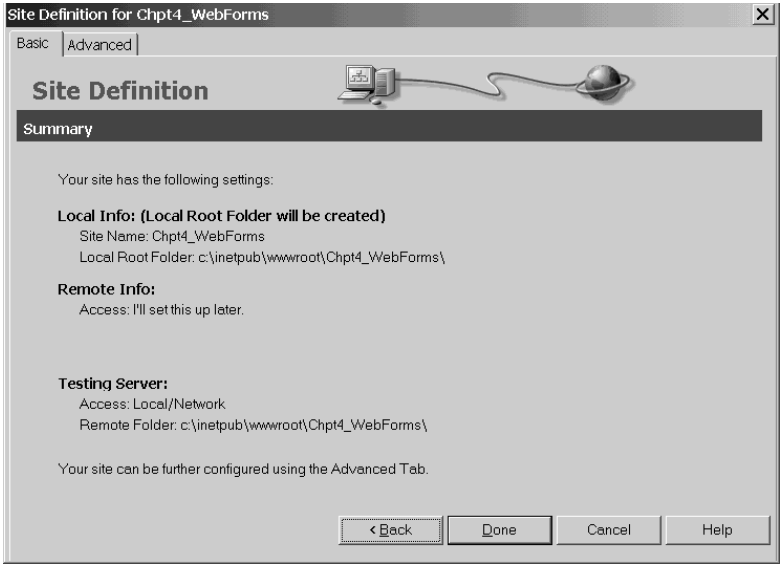

Figure 4.4 A summary of the settings you chose in the Site Definition wizard.

Building a Web Form

To start working with a Web Form and the controls it contains, we'll need to create one first.

To create a new Web Form:

1. Choose New from the File drop-down menu at the top of Dreamweaver.

   The New Document dialog box opens (**Figure 4.5**).

2. In the left pane of the dialog box's General tab, click Dynamic Page.

3. In the right pane, click ASP.NET C#.

4. Click the Create button to make the new .aspx file.

   For simplicity, we're not checking the box to make the new document XHTML compliant.

5. Save the file with its default name.

Now that we have a Web Form, let's start working with it. Switch your view to show Code and Design views simultaneously by clicking the Show Code and Design Views icon on the Document tool bar (**Figure 4.6**). This will help you see what's unique about a Web Form and its controls (**Figure 4.7**).

The only real difference we see between this page and an HTML page so far is in the code of the page. At the top is a command, commonly referred to as a Page Directive, that tells the Web server that the page is a C# Web Form:

```
<%@ Page Language="C#" ContentType=
"text/html" ResponseEncoding=
"iso-8859-1" %>
```

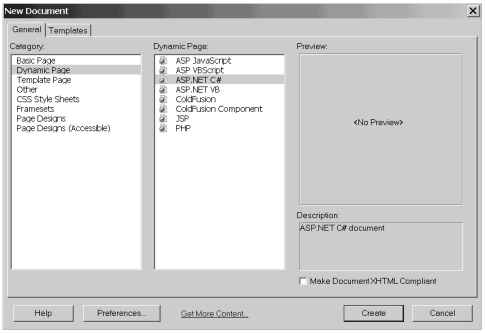

Figure 4.5 Create a new ASP.NET C# Web Form with the New Document dialog box.

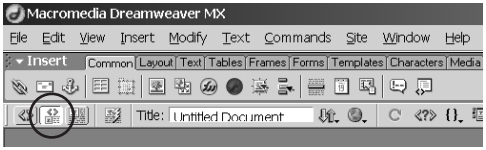

Figure 4.6 The Show Code and Design Views icon is located on the Common tab of the Insert bar.

Figure 4.7 A newly created Web Form in split view, with Code view on the top and Design view on the bottom.

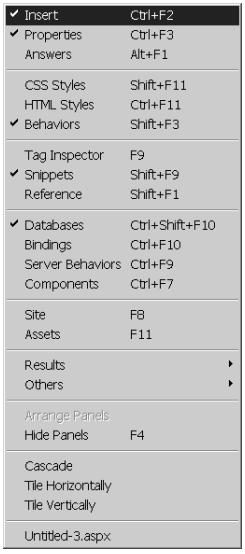

Figure 4.8 You know the Insert bar is open in Dreamweaver when you see a check next to its listing in the Window drop-down menu.

Figure 4.9 You click the Form icon on the Insert bar's Forms tab to insert a form element.

Figure 4.10 A Web Form with a form element in split Code/Design view.

The Page Directive also specifies that the resulting content should be HTML. Then it states that the resulting page should be encoded using `iso-8859-1`, which in English is `Western (Latin 1)`. This coding is the default; the only time you might need to change it is if you're writing pages for other languages.

To embed controls in the Web Form, we'll need to first define the form layout. Doing so involves inserting form tags, and then adding form elements, which is very similar to how you define the form layout for an HTML document—with one exception, as you'll see in Step 4 in the following instructions.

To add a form element:

1. Start by making sure the Insert bar is open.

   You can verify this by looking at the Window drop-down menu found at the top of Dreamweaver. There should be a check mark next to the word *Insert* (**Figure 4.8**).

   The Insert bar's default location is across the top of Dreamweaver, just below the menu bar where you found the Window drop-down menu.

2. Click on the Web Form in Design view, and then from the Insert bar, select the Forms tab and click on the Form icon (**Figure 4.9**).

   This automatically inserts a form element onto your page (**Figure 4.10**).

   You'll see the new HTML code for the form element in the Code view of the Web Form. Below that, in Design view, you'll see a red dotted line representing the boundaries of the form.

   Note: The Tag Editor starts if your cursor is in Code view when you click the Form icon to add a form to your Web page.

3. In the code of the Web Form, click just in front of the form element's name attribute.

continues on next page

4. Now go to the Insert bar and click on the ASP.NET tab (**Figure 4.11**). Then click the Runat Server icon.

   This inserts the command that tells the Web server that the form element should be available for server-side processing. It looks like this:

   ```
   <form runat="server" name="form1"
   method="post" action="">
   ```

This last step in the creation of a form element for an ASP.NET Web Form is critical. Controls need the `runat="server"` attribute setting to inform the Web server to process them as ASP.NET controls. We'll see exactly what that means in a moment. But for now, know that the most common error you'll likely make when building Web Forms is forgetting to add the `runat="server"` attribute setting to a control.

At long last we'll add a control to our Web Form. In the next exercise, we'll add an asp:label control that's handy for displaying text. It won't do anything dynamic right now, but it will make clear what distinguishes a control from a normal HTML tag.

To add a Label control:

1. Using the same Web Form we've been working with so far, click inside the red dotted area found in the Design view of the Web Form.
2. Click the asp:label icon found in the ASP.NET tab of the Insert bar.

   This will start the Tag Editor for a label control (**Figure 4.12**).
3. In the Tag Editor, set the label's ID to "lblMessage" and its Text to "Hello from ASP.NET." Click OK.

   The Tag Editor closes and writes the code for you (**Figure 4.13**).

Figure 4.11 The ASP.NET tab of the Insert bar displays the Runat Server icon.

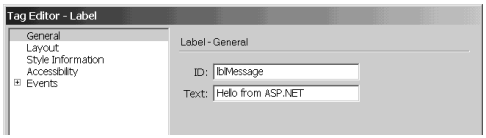

Figure 4.12 The Tag Editor for the lblMessage control.

Figure 4.13 The result of adding the lblMessage control with the Tag Editor.

4. Preview the Web Form in your default browser by saving the document and pressing the F12 key.

   Your browser should open and display the text "Hello from ASP.NET."

Let's review what we just completed. After adding an asp:label control to the Web Form, we just previewed the text "Hello from ASP.NET" in Design view. That makes perfect sense, since this control is called a Label. In the code, however, you'll notice a new tag that doesn't look quite like HTML but is very close in syntax.

ASP.NET controls are defined using XML (eXtensible Markup Language) coding standards, as is XHTML (or eXtensible HTML). We chose earlier not to force the document to be XHTML compliant, but that doesn't mean it can't contain XML. To understand what XML is, all you have to know is that it is text that must follow a strict format for describing information. Here, ASP.NET controls describe the programmable objects the Web server should create when processing the page.

Now, to see the new tag the asp:label control created, go back to the browser you previewed your page with in Step 4 and view the page's source code. You don't see the asp:label control anywhere, do you? In its place you'll find the following code:

```
<span id="lblMessage">Hello from
ASP.NET</span>
```

ASP.NET converted the asp:label control into the appropriate HTML—in this case the `<span>` tags—so that it could be rendered by a browser.

It's time to start looking into just what makes up a control. We'll also discuss why we'd want to use one instead of simply writing the resulting HTML in the first place.

Building Dynamic Pages Using Controls

There are lots of common uses for dynamic pages. One might be to have an online catalog for your products that dynamically checks the current price and quantity in your warehouse every time one of your customers visits the page. This is just one example; the list of possibilities is endless. You probably already have an idea of what you want to achieve since you're reading this book.

You may remember from Chapter 1 that historically, dynamic pages mixed the programming code with the HTML markup code, which has a number of disadvantages (see "A New Paradigm" in Chapter 1 for more information). ASP.NET uses controls like the asp:label control we presented earlier to allow programmable elements to be intermixed with the HTML markup code without mixing actual programming code with the HTML. Tools like Dreamweaver MX then simply have to be configured to know which HTML element each of these controls represents in order to present them graphically in Design view.

The `runat="server"` attribute setting in our controls tells the Web server that the control is a candidate for the programming code to manipulate. Then, once the Web server is done running all the programming code, it sends to the browser the HTML markup that the controls represent. Since the program might have manipulated the controls, the HTML sent to the browser could have different values than the controls had prior to execution.

To see this in action, let's take the Web Form we were just working on and make a dynamic page. The element that makes the page dynamic will involve the entry of a message into a textbox that will display in the label control when the user clicks on a button.

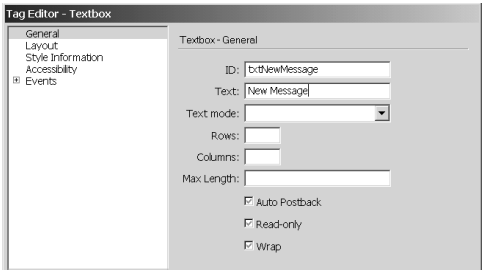

Figure 4.14 The Tag Editor for the txtNewMessage control.

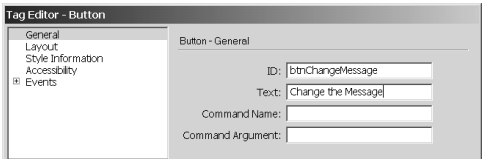

Figure 4.15 The Tag Editor for the btnChangeMessage control.

Figure 4.16 The code that results from adding three common controls: asp:label, asp:textbox, and asp:button.

To create a dynamic page:

1. In the Web Form, click beside the text "Hello from ASP.NET" in Design view.
2. From the Insert bar, click on the ASP.NET tab, and then click the asp:textbox icon .

   The Tag Editor for the asp:textbox control opens (**Figure 4.14**).
3. In the Tag Editor, set the ID to "txtNewMessage" and the Text to "New Message," and click OK.

   The Tag Editor closes and writes the code for you.
4. Click beside the new text box in Design view.
5. Then from the ASP.NET tab of the Insert bar, click the asp:button icon .

   The Tag Editor for the asp:button control opens (**Figure 4.15**).
6. This time in the Tag Editor, set the ID to "btnChangeMessage" and the Text to "Change the Message," and click OK.

   The resulting page will have three commonly used controls. The code for them will be in the Web Form's Code view, and the visual representation of them will be in Design view (**Figure 4.16**).
7. We still have some additional code to write in order to change the lblMessage control's text to what we enter into the txtNewMessage control. In Code view, click just before the `</head>` tag.

continues on next page

8. From the Insert bar's ASP.NET tab, click the Page_Load icon to automatically add the `Page_Load` function to the page.

   It'll look like this:

   ```
   <script runat="server">
   protected void Page_Load(Object Src,
   EventArgs E)
   {
     if (!IsPostBack) DataBind();
   }
   </script>
   ```

   You don't have to understand this right now—we're about to change it.

9. Change the contents of the `Page_Load` function to the following:

   ```
   if( IsPostBack )lblMessage.Text =
   txtNewMessage.Text;
   ```

   You can view the results in Code view (**Figure 4.17**).

10. Finally, save the document and press F12 to view your page in your default browser.

11. Try clicking the "Change the Message" button. You'll see the text displayed by the asp:label control change to read "New Message." To change the text again, you can change the value of the asp:textbox control and click the button.

Now, admittedly, this Web Form doesn't do a great deal, but this shows how easy it is to make a page dynamic.

Figure 4.17 The result of changing the `Page_Load` function.

Now, let's dive into the guts of the Web Form to learn what's really going on. As mentioned earlier, the Web server looks at the page to see if there are any elements on which it needs to run code. It also runs any code contained in the page. In our Web Form, for instance, we used a built-in function called `Page_Load`. Notice that this function resides in a script block that also contains the `runat="server"` attribute. This attribute tells the Web server to process the function before sending the page to the client. Script blocks without this attribute are assumed to be executed on the client and are ignored by the server.

The `Page_Load` function is automatically called by the Web server when the page is loaded for processing. The first line, `if( IsPostBack )`, checks if the page is in post-back mode. A Web Form is considered to be in post-back mode when it has already been sent to the client's computer and the client has then done something to cause the form to post back to the server. Therefore, the first time the visitor opens the Web Form, it's not a post back. The next section, "Handling Post Back," discusses this in more detail.

So the `Page_Load` function does nothing the first time the page is visited; then when the visitor clicks the button labeled Change the Message, the form is submitted back to the server. This time the Web server runs the same `Page_Load` function, but it's in post-back mode. Therefore, the Text property of the asp:label control is set to the value of the asp:textbox's Text property. Notice that now we called Text a property, not an attribute. The reason we call Text a property in the function and an attribute in the markup is that in the function we're dealing with a programmable object and in the markup we're dealing with the XML markup describing that object.

Handling Post Back

One of the biggest differences between ASP and ASP.NET is that in ASP.NET, a Web Form must post back to itself rather than post to a different page.

Historically, developers posted to a different page by setting the form's action attribute. Posting to a separate page used to be a good idea because it made for a cleaner separation of code from HTML. Now, because ASP.NET handles events in the same Web Form in which they're raised, the form must post back to the same page. Even if you set the action attribute of the form to a different page, the Web server finds the `runat="server"` attribute setting and overrides your action value.

The changes required to handle post back in the same Web Form instead of a different page are minimal. One of the nice things about posting to the same page is that it takes less code to process the data elements such as the Querystring or form fields. (See the sidebar "Form vs. Querystring Fields.") You saw that demonstrated in the previous exercise, "To create a dynamic page." There was no need to look in the Request.Form collection, as ASP developers would have had to do to access the form data being passed from the form. However, if you're stuck on using the Request.Form collection, you can still use it.

Form vs. Querystring Fields

Passing data back to the server is commonly done in one of two ways: via form field or Querystring.

We're all familiar with forms, because we've all had to enter information like our name into text boxes when purchasing something online. Those text boxes are placed in between opening and closing `<form>` tags in HTML, which makes them form fields. If you want to pass the data about these text boxes back to the server, you'd need to make sure you set the form's method attribute to its default value of `post`. Using `method="post"` means the form field values will be retrieved from the Response.Form collection.

The technique of posting fields in a Response.Form collection is sometimes called the "put" method. For example, say there was a text box named `txtFirstName` inside of a form, with the form's method attribute set to `post`. Then in the code of the page to which the form posted, the value entered into the `txtFirstName` text box would be available by referencing the `Response.Form["txtFirstName"]` field.

On the other hand, we could have set the form's method attribute to `get`. This attribute will still pass the data about the text boxes back to the server; however, in the code of the page to which the form posted, the value entered into the `txtFirstName` field will be available by referencing the Response.Querystring collection, rather than the Response.Form collection. It would look something like this: `Response.Querystring["txtFirstName"]`.

Posting fields in the Response.Querystring collection is usually referred to as the "get" method. The main difference between the two posting methods is that form fields can't be seen in a URL, while Querystring fields can. Most of the time you'll want to "put" fields, because then the field values won't be seen in the URL.This might be important, for example, if you need to pass sensitive information, such as social security numbers.

However, if you want to make a hyperlink pass information on to the page it links to without using a form at all, the "get" method of sending data to a page would be the perfect choice. For example, you might want to list summary information about products and provide links to a detail page for users to obtain more information about each product. Once a link is clicked, the detail page is passed data that indicates which product was selected via the Querystring. You won't need `<form>` tags around your data when passing it in this way.

The URL would look something like this:

```
http://localhost/detail.aspx?ProductID=002411027
```

In the above example, we have a Web Form called detail.aspx and the Querystring field, `ProductID`, is set to a value of `002411027`.

The code for the URL can just be a normal hyperlink, followed by a question mark ('?') and then the field name set to equal a value, just as we showed in the previous example:

```
<a href="http://localhost/detail.aspx?ProductID=002411027" > 002411027</a>
```

To use the Request.Form collection:

1. Once again, open up the same Web Form you've been working with.
2. Click beside the button in the Design view of the Web Form to set your insertion point.
3. On the ASP.NET tab, click the asp:label icon  .

   This opens its Tag Editor (**Figure 4.18**).
4. In the Tag Editor, set the ID to "lblOldWay" but leave the Text field blank. Click OK to close the Tag Editor and create the control.

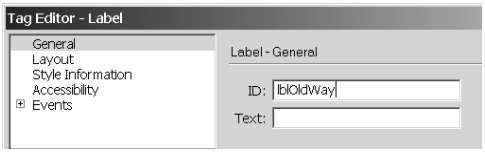

Figure 4.18 The Tag Editor for the lblOldWay control.

Figure 4.19 The result of adding the lblOldWay control and setting its Text to the Request.Form collection's txtNewMessage field.

5. Now add the following line of code to the `Page_Load` function, just above the code you entered in the last list, "To create a dynamic page."

   ```
   lblOldWay.Text = Request.Form
   ["txtNewMessage"];
   if ( IsPostBack )lblMessage.Text =
   txtNewMessage.Text;
   ```

   Note that the label has a visible placeholder in Design view (**Figure 4.19**).
6. Save the document and press the F12 key to view your changes.

   The lblOldWay element isn't shown, because it has no default value; but when you click the button, it changes to the text box's value, too.

✔ Tip

- If a Request.Form variable doesn't exist, it will be `null`, rather than an empty string, as it is the case in ASP.

If you looked at the source code of the page generated by the Web server while you were moving through the steps, you might have noticed a hidden form field called `__VIEWSTATE`. Don't bother to look for it now if you didn't see it before. It's really just a bunch of encoded stuff only the server understands. But what it's doing is keeping track of all your form field values so that they maintain state between post backs to the server. To remind you of what *maintaining state* means, remember that each time you visit a Web page, the server has no good idea who you are or what you were doing last. This *statelessness* is intrinsic to the Web and something programmers have always had to write code to get around. ASP.NET's `__VIEWSTATE` variable makes your life easier by not requiring you to write program code to maintain state anymore. It does it for you.

Moving Between Pages

So how do you help your visitors navigate to other pages on your site if you're always posting back to the same Web Form? The answer is the Response.Redirect command. First, handle the post back in your Web Form. Then give the Response.Redirect command the URL of the next page you want the visitor to go to, like this:

```
Response.Redirect("NextPage.aspx");
```

To increase performance there's a second, optional parameter you can add to the command. It determines whether the server should halt processing the current page and transfer immediately or whether it should finish the page first. The Boolean value of `true` halts processing and transfers immediately. That would look like this:

```
Response.Redirect("NextPage.aspx", true);
```

Transferring to a new page is when we really have to start worrying about maintaining state. The reason is that we won't have that handy `__VIEWSTATE` hidden form element doing the work for us. It's not available when transferring between pages using the Response.Redirect command.

In the past, when programmers posted to a different page, hidden form fields were often used to keep track of things such as the current visitor's ID number. Other options for maintaining state are common as well. For example, it's still possible to post a form that lacks the `runat="server"` attribute to your next page and gather those form values in the same way we described in the previous list, "To use the Request.Form collection." However, a better way to maintain state in ASP.NET is to use session variables.

Session variables are a great way to store data that only concerns an individual user for the duration of time the user is interacting with the Web site. Session variables are commonly used in E-commerce sites for the "shopping cart" feature, which stores items the visitor wants to buy. Don't let what you may have heard in the past about session variables put you off of ASP.NET's version; they're flexible and they let you easily overcome or minimize most of the problems associated with session variables, such as accessibility and performance. We'll start by using them in a simple way.

In the next list, we'll illustrate how to create and reference a session variable. Two Web Forms will be created: the first we'll use to create a session variable to store a text value and the second we'll use to retrieve and display the session variable's contents.

To use a session variable:

1. Create a Web Form as outlined earlier in this chapter in "To create a new Web Form," and name it "SetSession.aspx." Make sure the page shows both Code and Design views by clicking the Show Code and Design Views icon  .
2. Add a form element to it. Make sure to set the `runat="server"` attribute as we did in "To add a form element."
3. Now click inside the form's red dotted line to set your insertion point.
4. On the ASP.NET tab, click the asp:button icon  .

   Its Tag Editor opens (**Figure 4.20**).
5. In the Tag Editor, set the ID to "btnSetSessionVariable" and Text to "Set Session Variable."

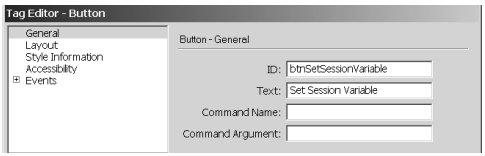

Figure 4.20 The Tag Editor for the btnSetSessionVariable control.

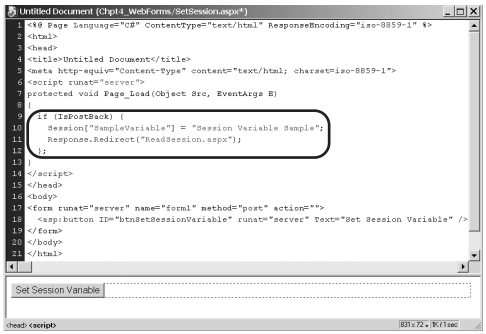

Figure 4.21 The code setting the SampleVariable session variable and redirecting to the ReadSession Web Form.

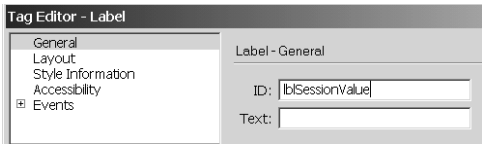

Figure 4.22 The Tag Editor for the lblSessionValue control.

6. In the page's code, add the Page_Load function by clicking the Page_Load icon like we did in "To create a dynamic page." This time, replace the Page_Load function's contents with code to set a session variable called "SampleVariable" to the value "Session Variable Sample," as follows:

```
protected void Page_Load(Object Src,
EventArgs E)
{
  Session["SampleVariable"] =
  "Session Variable Sample";
}
```

7. On the line that follows the code we just added, add the command to redirect to a page called ReadSession.aspx (**Figure 4.21**).

```
Response.Redirect("ReadSession.aspx");
```

8. Now create the Web Form you'll be redirecting to. Do it the same way you did in Step 1, but name it "ReadSession.aspx."

9. In the ReadSession Web Form you just created, add an asp:label, setting its ID to "lblSessionValue" (**Figure 4.22**). This time a form element is not necessary.

10. Add the `Page_Load` function with code that sets lblSessionValue's Text property to the session variable.

```
protected void Page_Load(Object Src,
EventArgs E)
{
  lblSessionValue.Text =
  Session["SampleVariable"].ToString();
}
```

continues on next page

Make sure to change the session variable to a string using the `ToString()` method, since that's what Text properties expect (**Figure 4.23**).

11. Finally, save the files and preview the SetSession Web Form in your default browser.

12. To test, click the button in the SetSession Web Form.

This will cause the page to post back to the server, set the session variable, and redirect to the ReadSession Web Form.

You learned from "To use the Request.Form collection" earlier in this chapter that if the lblSessionValue control hadn't been set in the ReadSession page's `Page_Load` function, the page would have been blank.

These instructions present a generic scenario for using a session variable. Session variables can be used to store almost anything the programmer needs. However, Web sites usually store things like the identifier for the logged in user, the number of the products the user is looking for more details about, and so on. Typically, it's best to try to store just a small number of temporary but reusable things that require little memory. This is because session variables are stored in the server's memory by default. You easily can change this default, however, by specifying that the server store the session variables in a database.

Then, also by default, an in-memory cookie on the visitor's computer tells the server which session variables belong to which visitors. You can also change how this session identifier is stored: Instead of the server placing a cookie on the visitor's computer, you can have the server automatically tack it onto the URL of every page the user visits (for both hyperlinks and redirect commands) using a Querystring variable.

Figure 4.23 The result of adding the lblSessionValue control and setting it's Text property to the SampleVariable Session variable.

Working with Lists

The last type of control we'll work with in this chapter is the list control. Most anyone who's ever filled out their address on a Web page has likely encountered either a list of countries or a list of state in the United States. Because those lists are rather long, we'll make our own short list.

There are actually many types of lists. The two examples above are usually shown as drop-down lists, but we can have check box lists, radio button lists, or just plain list boxes. Since drop-down lists are the most common, we'll use that type in the following list. We encourage you to try some of the other types as well.

The following instructions create an asp:dropdownlist control and add list items to it in two different ways.

To make a drop-down list:

1. Create a new Web Form, and then add a form element, making sure to add the `runat="server"` attribute setting.

   If you need to refresh your memory, see "To create a new Web Form" and "To add a form element" earlier in this chapter.

2. Click inside the form element in Design view to set your insertion point.

3. Then from the Insert bar's ASP.NET tab, click the asp:dropdownlist icon  .

   This opens the Tag Editor (**Figure 4.24**).

4. In the Tag Editor, set the ID to "ddlSampleList" and click OK.

   Never mind the faded check in the box next to Auto Postback. The default is to not post back automatically when the user's dropdownlist control selection changes, which is what we want right now.

5. Between the `<asp:dropdownlist>` and `</asp:dropdownlist>` tags, type the following code:

```
<asp:dropdownlist ID="ddlSampleList"
runat="server">
  <asp:listitem>
    List Item 1
  </asp:listitem>
</asp:dropdownlist>
```

6. Once again, add the `Page_Load` function and change its contents to the following:

```
protected void Page_Load(Object Src,
EventArgs E)
{
  ddlSampleList.Items.Add(
  "List Item 2" );
}
```

7. Review your Web Form and try to guess what the results will be (**Figure 4.25**).

8. Save the page and view it by pressing the F12 key to see if your guess was correct.

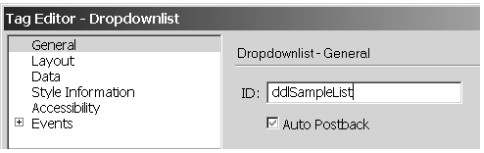

Figure 4.24 The Tag Editor for the ddlSampleList control.

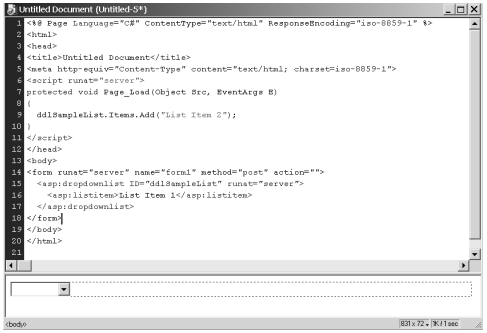

Figure 4.25 Adding list items to the ddlSampleList control through two different methods: code and markup.

Inheritance

.NET is built using object-oriented design, and the .NET languages are object oriented (OO). OO programming is a huge subject in itself, but one key concept in it is called *inheritance*. In OO languages you can create what is called a Class. A *Class* defines an Object—specifying what data is part of the Object, as well as what functions you can run on it. ListControl is one such Class.

Inheritance is when you create a second Class, say a DropDownList, and then specify in its definition that it should inherit all the functions and properties from another Class, such as the ListControl. You could then customize some of those functions to better fit your new Class. If several Classes inherit from the same Class, they will all inherit those same functions.

For a deeper discussion of this subject, refer to Chapter 11.

You see from the resulting page that both ways of adding items to the drop-down list work well. Step 5 was not a dynamic approach because you just typed the item directly within the drop-down list. That's actually a good way to do it for lists that are short and won't change, such as a list of gender choices.

Step 6 uses a more programmatic approach to adding a list item by working with the list control's Items collection. All list type controls have an Items collection so they all have the Add function. In fact they have a slew of common functions because they all inherit from the ListControl class (see the "Inheritance" sidebar).

There is a completely different way to dynamically add list items, however. It's called *data binding*, and it's handy for building dynamic pages. Data binding is the process of associating data from a database table or an array, for example, to a control at runtime. Data binding isn't limited to lists—many of the standard controls can also be bound to data. The stepped instructions that follow use two different syntaxes for achieving data binding; you'll see both regularly. You just have to know that the two styles exist and that sometimes one will be more useful than the other.

To bind data to a control:

1. Using the same Web Form you used in the previous exercise, "To make a drop-down list," remove both the programming code and the markup code that added list items to ddlSampleList.
2. Click beside the ddlSampleList control in Design view to set your insertion point.
3. Then on the ASP.NET tab, click the asp:textbox icon .
   The Tag Editor opens (**Figure 4.26**).
4. In the Tag Editor, set the ID to "txtDataString" and Text to "str." Click OK.
5. Back in the code of the Web Form, highlight the value of `"str"` assigned to txtDataString. Don't include its surrounding quotes in the highlighting.
6. Then on the ASP.NET tab, click the Bound Data icon . This will change the value of the Text attribute to <%# str %>.
   Note: You'll run into situations where the bound data syntax will need to use quotation marks. Because of the outer double quotation marks of the Text attribute, you would need to use the single quotation marks inside.
7. Now up above the `Page_Load` function, create a String variable called `str` like the following:

```
<script runat="server">
String str = "Sample String";

protected void Page_Load(Object Src,
EventArgs E)
```

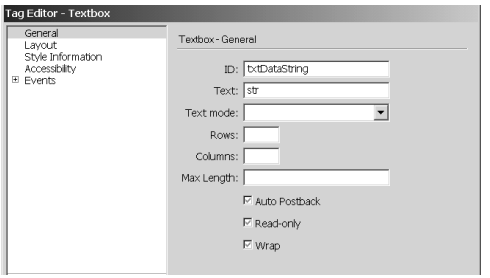

Figure 4.26 The Tag Editor for the txtDataString control.

8. Inside the `Page_Load` function, create a String array called `dataList` and set it to be ddlSampleList's data source. Also set the Page object to execute the `BindData` command (**Figure 4.27**).

```
protected void Page_Load(Object Src,
EventArgs E)
{
  String[] dataList = new
  String[2]
      {"List Item 1", "List Item
      2"};
  ddlSampleList.DataSource =
      dataList;
  Page.BindData();
}
```

9. Save the Web Form, and press F12 to view it in your default browser.

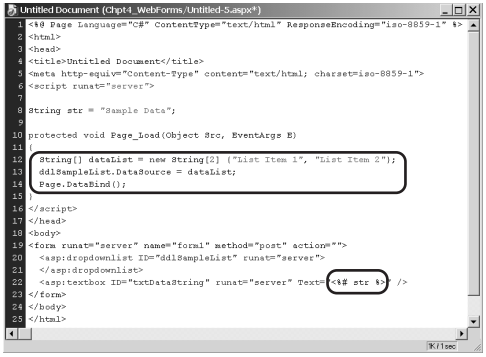

Figure 4.27 The two techniques for binding data to controls.

First we set the text box's Text attribute to a dynamic value by using the `<%# %>` bound data syntax. At first glance this may seem to be the same as the old ASP way of inserting dynamic data into HTML. However, the syntax is where the similarity stops. The biggest difference is that the data doesn't get bound until you call the `BindData` function. We did so in the `Page_Load` function by calling the Page object's `BindData` function. We could have called the `BindData` function for ddlSampleList and txtDataString separately; however, the call cascades down to all contained objects, so calling it on the Page object requires less code.

The second style of binding data we used was to set the `DataSource` attribute of ddlSampleList to the array of strings. This style of code is less clear in describing exactly what will happen, but the style better separates the programming code from the markup code, which can be beneficial.

We'll thoroughly explore the subject of binding data in Chapter 8, but next we'll take a closer look at the programmability of Web Forms in Chapter 5, "Effectively Using Web Form Controls."