CHAPTER 8

# State Management

The most significant difference between programming for the Web and programming for the desktop is *state management*—how you store information over the lifetime of your application. This information can be as simple as a user's name or as complex as a stuffed-full shopping cart for an e-commerce store.

In a traditional desktop application, there's little need to think about state management. Memory is plentiful and always available, and you need to worry about only a single user. In a web application, it's a different story. Thousands of users can simultaneously run the same application on the same computer (the web server), each one communicating over a stateless HTTP connection. These conditions make it impossible to design a web application in the same way as a desktop application.

Understanding these state limitations is the key to creating efficient web applications. In this chapter, you'll see how you can use ASP.NET's state management features to store information carefully and consistently. You'll explore different storage options, including view state, session state, and custom cookies. You'll also consider how to transfer information from page to page by using cross-page posting and the query string.

## Understanding the Problem of State

In a traditional desktop application, users interact with a continuously running application. A portion of memory on the desktop computer is allocated to store the current set of working information.

In a web application, the story is quite a bit different. A professional ASP.NET site might look like a continuously running application, but that's really just a clever illusion. In a typical web request, the client connects to the web server and requests a page. When the page is delivered, the connection is severed, and the web server discards all the page objects from memory. By the time the user receives a page, the web page code has already stopped running, and there's no information left in the web server's memory.

This stateless design has one significant advantage. Because clients need to be connected for only a few seconds at most, a web server can handle a huge number of nearly simultaneous requests without a performance hit. However, if you want to retain information for a longer period of time so it can be used over multiple postbacks or on multiple pages, you need to take additional steps.

## Using View State

One of the most common ways to store information is in *view state*. View state uses a hidden field that ASP.NET automatically inserts in the final, rendered HTML of a web page. It's a perfect place to store information that's used for multiple postbacks in a single web page.

In Chapter 5, you learned how web controls use view state to keep track of certain details. For example, if you change the text of a label, the Label control automatically stores its new text in view state. That way, the text remains in place the next time the page is posted back. Web controls store most of their property values in view state, provided you haven't switched ViewState off (for example, by setting the EnableViewState property of the control or the page to false).

View state isn't limited to web controls. Your web page code can add bits of information directly to the view state of the containing page and retrieve it later after the page is posted back. The type of information you can store includes simple data types and your own custom objects.

## The ViewState Collection

The ViewState property of the page provides the current view-state information. This property provides an instance of the StateBag collection class. The StateBag is a dictionary collection, which means every item is stored in a separate "slot" using a unique string name, which is also called the *key name*.

For example, consider this code:

```
// The this keyword refers to the current Page object. It's optional.
this.ViewState["Counter"] = 1;
```

This places the value 1 (or rather, an integer that contains the value 1) into the ViewState collection and gives it the descriptive name Counter. If currently no item has the name Counter, a new item will be added automatically. If an item is already stored under the name Counter, it will be replaced.

When retrieving a value, you use the key name. You also need to cast the retrieved value to the appropriate data type by using the casting syntax you saw in Chapter 2 and Chapter 3. This extra step is required because the ViewState collection stores all items as basic objects, which allows it to handle many different data types.

Here's the code that retrieves the counter from view state and converts it to an integer:

```
int counter;
counter = (int)this.ViewState["Counter"];
```

---

■ **Note** ASP.NET provides many collections that use the same dictionary syntax. These include the collections you'll use for session and application state, as well as those used for caching and cookies. You'll see several of these collections in this chapter.

---

## A View-State Example

The following example is a simple counter program that records the number of times a button is clicked. Without any kind of state management, the counter will be locked perpetually at 1. With careful use of view state, the counter works as expected.

```
public partial class SimpleCounter : System.Web.UI.Page
{
    protected void cmdIncrement_Click(object sender, EventArgs e)
    {
        int counter;
        if (ViewState["Counter"] == null)
        {
            counter = 1;
        }
        else
        {
            counter = (int)ViewState["Counter"] + 1;
        }
```

```
        ViewState["Counter"] = counter;
        lblCount.Text = "Counter: " + counter.ToString();
    }
}
```

The code checks to make sure the item exists in view state before it attempts to retrieve it. Otherwise, you could easily run into problems such as the infamous *null reference exception* (which is described in Chapter 7).

Figure 8-1 shows the output for this page.

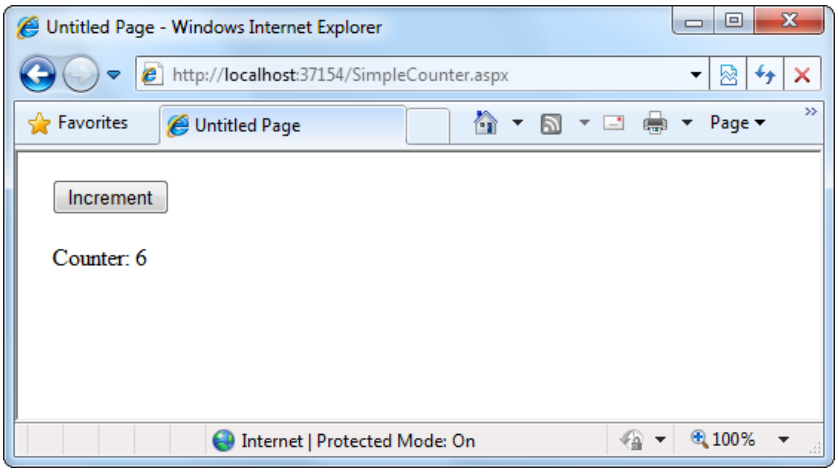


***Figure 8-1.*** *A simple view-state counter*

## Making View State Secure

You probably remember from Chapter 5 that view-state information is stored in a single jumbled string that looks like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
 value="dDw3NDg2NTI5MDg7Oz4=" />
```

As you add more information to view state, this value can become much longer. Because this value isn't formatted as clear text, many ASP.NET programmers assume that their view-state data is encrypted. It isn't. Instead, the view-state information is simply patched together in memory and converted to a *Base64 string* (which is a special type of string that's always acceptable in an HTML document because it doesn't include any extended characters). A clever hacker could reverse-engineer this string and examine your view-state data in a matter of seconds.

Fortunately, ASP.NET has the tools to make view state more secure. By default, it makes view state tamper-proof (see the following section, "Tamper-Proof View State") and gives you the additional option of making it secret (see the section after that, "Private View State").

### Tamper-Proof View State

ASP.NET uses a hash code to make sure your view-state information can't be altered without your knowledge. Technically, a *hash code* is a cryptographically strong checksum. The idea is that ASP.NET examines all the data

in view state, just before it renders the final page. It runs this data through a hashing algorithm (with the help of a secret key value). The hashing algorithm creates a short segment of data, which is the hash code. This code is then added at the end of the view-state data, in the final HTML that's sent to the browser.

The hash code becomes very useful when the page is posted back to the server. At this point, ASP.NET examines the view-state data and recalculates the hash code by using the same process it used to create the hash code in the first place. ASP.NET then checks whether the newly calculated hash code matches the original hash code, which is stored in the view state for the page. If a malicious user changes part of the view-state data, the new hash code won't match the original value. At this point, ASP.NET will reject the postback completely and show an error page. (You might think a really clever user could get around this by generating fake view-state information *and* a matching hash code. However, malicious users can't generate the right hash code, because they don't have the same cryptographic key as ASP.NET.)

Hash codes are enabled by default, so you don't need to take any extra steps to get this functionality. (In theory, you can disable it, but no one ever does—and you shouldn't.)

## Private View State

Even when you use hash codes, the view-state data will still be readable by the user. In many cases, this is completely acceptable—after all, the view state tracks information that's often provided directly through other controls. However, if your view state contains some information you want to keep secret, you can enable view-state *encryption*.

You can turn on encryption for an individual page by using the ViewStateEncryptionMode property of the Page directive:

```
<%@Page ViewStateEncryptionMode="Always" %>
```

Or you can set the same attribute in a configuration file to configure view-state encryption for all the pages in your website:

```
<configuration>
    ...
    <system.web>
      <pages viewStateEncryptionMode="Always" />
      ...
    </system.web>
</configuration>
```

Either way, this enforces encryption. You have three choices for your view-state encryption setting—always encrypt (Always), never encrypt (Never), or encrypt only if a control specifically requests it (Auto). The default is Auto, which means that the page won't encrypt its view state unless a control on that page specifically requests it. (Technically, a control makes this request by calling the Page.RegisterRequiresViewStateEncryption() method.) If no control calls this method to indicate that it has sensitive information, the view state is not encrypted, thereby saving the encryption overhead. On the other hand, a control doesn't have absolute power—if it calls Page.RegisterRequiresViewStateEncryption() and the encryption mode is Never, the view state won't be encrypted.

---

■ **Tip** Don't encrypt view-state data if you don't need to do so. The encryption will impose a performance penalty, because the web server needs to perform the encryption and decryption with each postback.

---

## Retaining Member Variables

You have probably noticed that any information you set in a member variable for an ASP.NET page is automatically abandoned when the page processing is finished and the page is sent to the client. Interestingly, you can work around this limitation by using view state.

The basic principle is to save all member variables to view state when the Page.PreRender event occurs, and retrieve them when the Page.Load event occurs. Remember, the Load event happens every time the page is created. In the case of a postback, the Load event occurs first, followed by any other control events.

The following example uses this technique with a single member variable (named Contents). The page provides a text box and two buttons. The user can choose to save a string of text and then restore it at a later time (see Figure 8-2). The Button.Click event handlers store and retrieve this text by using the Contents member variable. These event handlers don't need to save or restore this information by using view state, because the PreRender and Load event handlers perform these tasks when page processing starts and finishes.

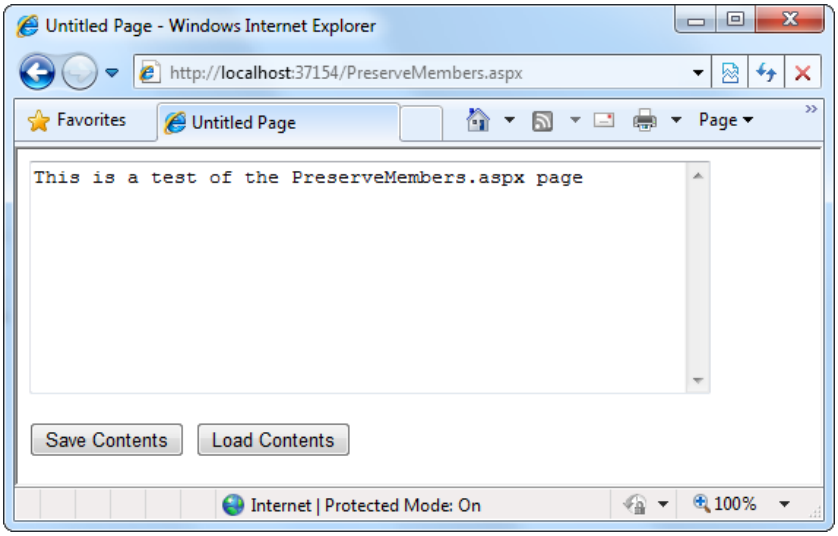


***Figure 8-2.*** *A page with state*

```
public partial class PreserveMembers : Page
{
    // A member variable that will be cleared with every postback.
    private string contents;

    protected void Page_Load(Object sender, EventArgs e)
    {
        if (this.IsPostBack)
        {
            // Restore variables.
            contents = (string)ViewState["contents"];
        }
    }
```

```
    protected void Page_PreRender(Object sender, EventArgs e)
    {
        // Persist variables.
        ViewState["contents"] = contents;
    }

    protected void cmdSave_Click(Object sender, EventArgs e)
    {
        // Transfer contents of text box to member variable.
        contents = txtValue.Text;
        txtValue.Text = "";
    }

    protected void cmdLoad_Click(Object sender, EventArgs e)
    {
        // Restore contents of member variable to text box.
        txtValue.Text = contents;
    }
}
```

The logic in the Load and PreRender event handlers allows the rest of your code to work more or less as it would in a desktop application. However, you must be careful not to store needless amounts of information when using this technique. If you store unnecessary information in view state, it will enlarge the size of the final page output and can thus slow down page transmission times. Another disadvantage with this approach is that it hides the low-level reality that every piece of data must be explicitly saved and restored. When you hide this reality, it's more likely that you'll forget to respect it and design for it.

If you decide to use this approach to save member variables in view state, use it *exclusively*. In other words, refrain from saving some view-state variables at the PreRender stage and others in control event handlers, because this is sure to confuse you and any other programmer who looks at your code.

---

■ **Tip** The previous code example reacts to the Page.PreRender event, which occurs just after page processing is complete and just before the page is rendered in HTML. This is an ideal place to store any leftover information that is required. You cannot store view-state information in an event handler for the Page.Unload event. Though your code will not cause an error, the information will not be stored in view state, because the final HTML page output is already rendered.

---

## Storing Custom Objects

You can store your own objects in view state just as easily as you store numeric and string types. However, to store an item in view state, ASP.NET must be able to convert it into a stream of bytes so that it can be added to the hidden input field in the page. This process is called *serialization*. If your objects aren't serializable (and by default they're not), you'll receive an error message when you attempt to place them in view state.

To make your objects serializable, you need to add a Serializable attribute before your class declaration. For example, here's an exceedingly simple Customer class:

```
[Serializable]
public class Customer
{
    private string firstName;
```

```
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    private string lastName;
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }

    public Customer(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Because the Customer class is marked as serializable, it can be stored in view state:

```
// Store a customer in view state.
Customer cust = new Customer("Marsala", "Simons");
ViewState["CurrentCustomer"] = cust;
```

Remember, when using custom objects, you'll need to cast your data when you retrieve it from view state.

```
// Retrieve a customer from view state.
Customer cust;
cust = (Customer)ViewState["CurrentCustomer"];
```

Once you understand this principle, you'll also be able to determine which .NET objects can be placed in view state. You simply need to find the class information in the Visual Studio Help. The easiest approach is to look up the class in the index. For example, to find out about the FileInfo class (which you'll learn about in Chapter 17), look for the index entry *FileInfo class*. In the class documentation, you'll see the declaration for that class, which looks something like this:

```
[Serializable]
[ComVisible(true)]
public sealed class FileInfo : FileSystemInfo
```

If the class declaration is preceded with the Serializable attribute (as it is here), instances of this class can be placed in view state. If the Serializable attribute isn't present, the class isn't serializable, and you won't be able to place instances of it in view state.

# Transferring Information Between Pages

One of the most significant limitations with view state is that it's tightly bound to a specific page. If the user navigates to another page, this information is lost. This problem has several solutions, and the best approach depends on your requirements.

In this section, you'll learn two basic techniques to transfer information between pages: cross-page posting and the query string.

## Cross-Page Posting

A *cross-page postback* is a technique that extends the postback mechanism you've already learned about so that one page can send the user to another page, complete with all the information for that page. This technique sounds conceptually straightforward, but it's a potential minefield. If you're not careful, it can lead you to create pages that are tightly coupled to others and difficult to enhance and debug.

The infrastructure that supports cross-page postbacks is a property named PostBackUrl, which is defined by the IButtonControl interface and turns up in button controls such as ImageButton, LinkButton, and Button. To use cross-posting, you simply set PostBackUrl to the name of another web form. When the user clicks the button, the page will be posted to that new URL with the values from all the input controls on the current page. (This posted-back information includes the hidden view-state field. As you'll see shortly, it allows ASP.NET to create an up-to-date instance of the source page in memory.)

Here's an example—a page named CrossPage1.aspx that defines a form with two text boxes and a button. When the button is clicked, it posts to a page named CrossPage2.aspx.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="CrossPage1.aspx.cs"
    Inherits="CrossPage1" %>
<html ="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>CrossPage1</title>
</head>
<body>
    <form id="form1" runat="server" >
      <div>
        First Name:
        <asp:TextBox ID="txtFirstName" runat="server"></asp:TextBox>
        <br />
        Last Name:
        <asp:TextBox ID="txtLastName" runat="server"></asp:TextBox>
        <br />
        <br />
        <asp:Button runat="server" ID="cmdPost"
          PostBackUrl="CrossPage2.aspx" Text="Cross-Page Postback" /><br />
      </div>
    </form>
</body>
</html>
```

The CrossPage1 page doesn't include any code. Figure 8-3 shows how it appears in the browser.

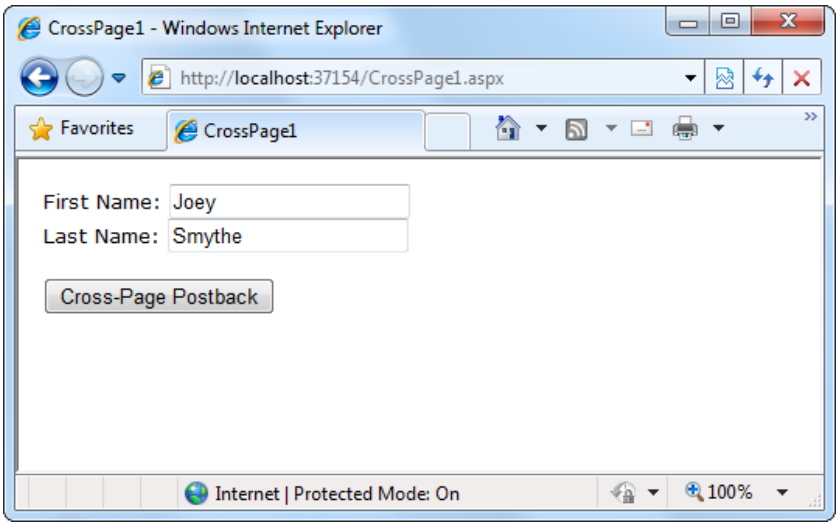


***Figure 8-3.*** *The starting point of a cross-page postback*

Now if you load this page and click the button, the page will be posted back to CrossPage2.aspx. At this point, the CrossPage2.aspx page can interact with CrossPage1.aspx by using the Page.PreviousPage property. Here's the code for the CrossPage2 page, which includes an event handler that grabs the title from the previous page and displays it:

```
public partial class CrossPage2 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (PreviousPage != null)
        {
            lblInfo.Text = "You came from a page titled " +
              PreviousPage.Title;
        }
    }
}
```

Note that this page checks for a null reference before attempting to access the PreviousPage object. If it's a null reference , no cross-page postback took place. This means CrossPage2.aspx was requested directly or CrossPage2.aspx posted back to itself. Either way, no PreviousPage object is available.

Figure 8-4 shows what you'll see when CrossPage1.aspx posts to CrossPage2.aspx.

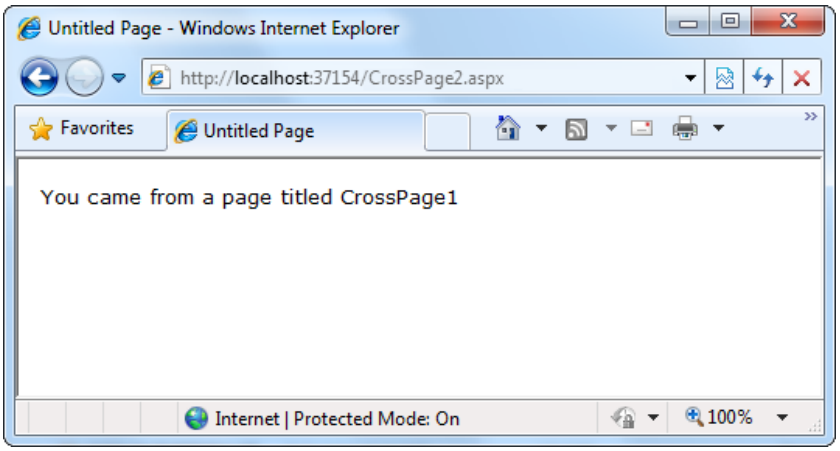


***Figure 8-4.*** *The target of a cross-page postback*

## Getting More Information from the Source Page

The previous example shows an interesting initial test, but it doesn't really allow you to transfer any useful information. After all, you're probably interested in retrieving specific details (such as the text in the text boxes of CrossPage1.aspx) from CrossPage2.aspx. The title alone isn't very interesting.

To get more-specific details, such as control values, you need to cast the PreviousPage reference to the appropriate page class (in this case, it's the CrossPage1 class). Here's an example that handles this situation properly, by checking first whether the PreviousPage object is an instance of the expected class:

```
public partial class CrossPage1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        CrossPage1 prevPage = PreviousPage as CrossPage1;
        if (prevPage != null)
        {
            // (Read some information from the previous page.)
        }
    }
}
```

---

■ **Note** In a projectless website, Visual Studio may flag this code as an error, indicating that it does not have the type information for the source-page class (in this example, that's CrossPage1). However, after you compile the website, the error will disappear.

---

You can also solve this problem in another way. Rather than casting the reference manually, you can add the PreviousPageType directive to the .aspx page that receives the cross-page postback (in this example, CrossPage2.aspx), right after the Page directive. The PreviousPageType directive indicates the expected type of the page initiating the cross-page postback. Here's an example:

```
<%@ PreviousPageType VirtualPath="~/CrossPage1.aspx" %>
```

Now the PreviousPage property will automatically use the CrossPage1 type. That allows you to skip the casting code and go straight to work using the previous page object, like this:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (PreviousPage != null)
    {
        // (Read some information from the previous page.)
    }
}
```

However, this approach is more fragile because it limits you to a single-page class. You don't have the flexibility to deal more than one page triggering a cross-page postback. For that reason, it's usually more flexible to use the casting approach.

After you've cast the previous page to the appropriate page type, you still won't be able to directly access the control objects it contains. That's because the controls on the web page are not publicly accessible to other classes. You can work around this by using properties.

For example, if you want to expose the values from two text boxes in the source page, you might add a property for each control variable. Here are two properties you could add to the CrossPage1 class to expose its TextBox controls:

```
public TextBox FirstNameTextBox
{
    get { return txtFirstName; }
}
public TextBox LastNameTextBox
{
    get { return txtLastName; }
}
```

However, this usually isn't the best approach. The problem is that it exposes too many details, giving the target page the freedom to read everything from the text in the text box to its fonts and colors. If you need to change the page later to use different input controls, it will be difficult to maintain these properties. Instead, you'll probably be forced to rewrite code in both pages.

A better choice is to define specific, limited methods or properties that extract just the information you need. For example, you might decide to add a FullName property that retrieves just the text from the two text boxes. Here's the full page code for CrossPage1.aspx with this property:

```
public partial class CrossPage1 : System.Web.UI.Page
{
    public string FullName
    {
        get { return txtFirstName.Text + " " + txtLastName.Text; }
    }
}
```

This way, the relationship between the two pages is clear, simple, and easy to maintain. You can probably change the controls in the source page (CrossPage1) without needing to change other parts of your application. For example, if you decide to use different controls for name entry in CrossPage1.aspx, you will be forced to revise the code for the FullName property. However, your changes would be confined to CrossPage1.aspx, and you wouldn't need to modify CrossPage2.aspx at all.

Here's how you can rewrite the code in CrossPage2.aspx to display the information from CrossPage1.aspx:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (PreviousPage != null)
    {
        lblInfo.Text = "You came from a page titled " +
            PreviousPage.Title + "<br />";

        CrossPage1 prevPage = PreviousPage as CrossPage1;
        if (prevPage != null)
        {
            lblInfo.Text += "You typed in this: " + prevPage.FullName;
        }
    }
}
```

Notice that the target page (CrossPage2.aspx) can access the Title property of the previous page (CrossPage1.aspx) without performing any casting. That's because the Title property is defined as part of the base System.Web.UI.Page class, and so every web page includes it. However, to get access to the more specialized FullName property, you need to cast the previous page to the right page class (CrossPage1) or use the PreviousPageType directive that was discussed earlier.

Figure 8-5 shows the new result.

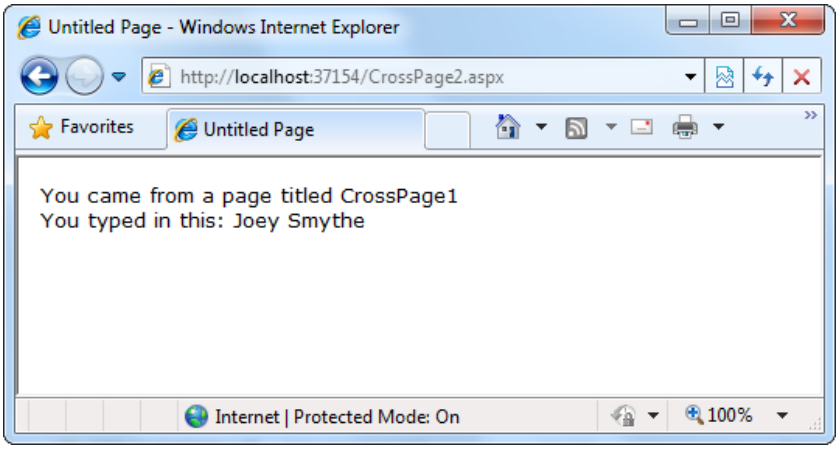


***Figure 8-5.*** *Retrieving specific information from the source page*

---

■ **Note** Cross-page postbacks are genuinely useful, but they can lead the way to more-complicated pages. If you allow multiple source pages to post to the same destination page, it's up to you to code the logic that figures out which page the user came from and then act accordingly. To avoid these headaches, it's easiest to perform cross-page postbacks between two specific pages only.

---

ASP.NET uses some interesting sleight of hand to make cross-page postbacks work. The first time the second page accesses Page.PreviousPage, ASP.NET needs to create the previous page object. To do this, it actually starts the page processing but interrupts it just before the PreRender stage, and it doesn't let the page render any HTML output.

However, this still has some interesting side effects. For example, all the page events of the previous page are fired, including Page.Load and Page.Init, and the Button.Click event also fires for the button that triggered the cross-page postback. ASP.NET fires these events because they might be needed to return the source page to the state it was last in, just before it triggered the cross-page postback.

## The Query String

Another common approach is to pass information by using a query string in the URL. This approach is commonly found in search engines. For example, if you perform a search on the Google website, you'll be redirected to a new URL that incorporates your search parameters. Here's an example:

```
http://www.google.ca/search?q=organic+gardening
```

The query string is the portion of the URL after the question mark. In this case, it defines a single variable named *q*, which contains the string *organic+gardening*.

The advantage of the query string is that it's lightweight and doesn't exert any kind of burden on the server. However, it also has several limitations:

- Information is limited to simple strings, which must contain URL-legal characters.
- Information is clearly visible to the user and to anyone else who cares to eavesdrop on the Internet.
- The enterprising user might decide to modify the query string and supply new values, which your program won't expect and can't protect against.
- Many browsers impose a limit on the length of a URL (usually from 1 KB to 2 KB). Therefore, you can't place a large amount of information in the query string and still be assured of compatibility with most browsers.

Adding information to the query string is still a useful technique. It's particularly well suited in database applications in which you present the user with a list of items that correspond to records in a database, such as products. The user can then select an item and be forwarded to another page with detailed information about the selected item. One easy way to implement this design is to have the first page send the item ID to the second page. The second page then looks that item up in the database and displays the detailed information. You'll notice this technique in e-commerce sites such as Amazon.com.

To store information in the query string, you first need to place it there. Unfortunately, you have no collection-based way to do this. Instead, you'll need to insert it into the URL yourself. Here's an example that uses this approach with the Response.Redirect() method:

```
// Go to newpage.aspx. Submit a single query string argument
// named recordID, and set to 10.
Response.Redirect("newpage.aspx?recordID=10");
```

You can send multiple parameters as long as they're separated with an ampersand (&):

```
// Go to newpage.aspx. Submit two query string arguments:
// recordID (10) and mode (full).
Response.Redirect("newpage.aspx?recordID=10&mode=full");
```

The receiving page has an easier time working with the query string. It can receive the values from the QueryString dictionary collection exposed by the built-in Request object:

```
string ID = Request.QueryString["recordID"];
```

Note that information is always retrieved as a string, which can then be converted to another simple data type. Values in the QueryString collection are indexed by the variable name. If you attempt to retrieve a value that isn't present in the query string, you'll get a null reference.

---

■ **Note** Unlike view state, information passed through the query string is clearly visible and unencrypted. Don't use the query string for information that needs to be hidden or made tamperproof.

---

## A Query String Example

The next program presents a list of entries. When the user chooses an item by clicking the appropriate item in the list, the user is forwarded to a new page. This page displays the received ID number. This provides a quick and simple query string test with two pages. In a sophisticated application, you would want to combine some of the data control features that are described later in Part 3 of this book.

The first page provides a list of items, a check box, and a submission button (see Figure 8-6).

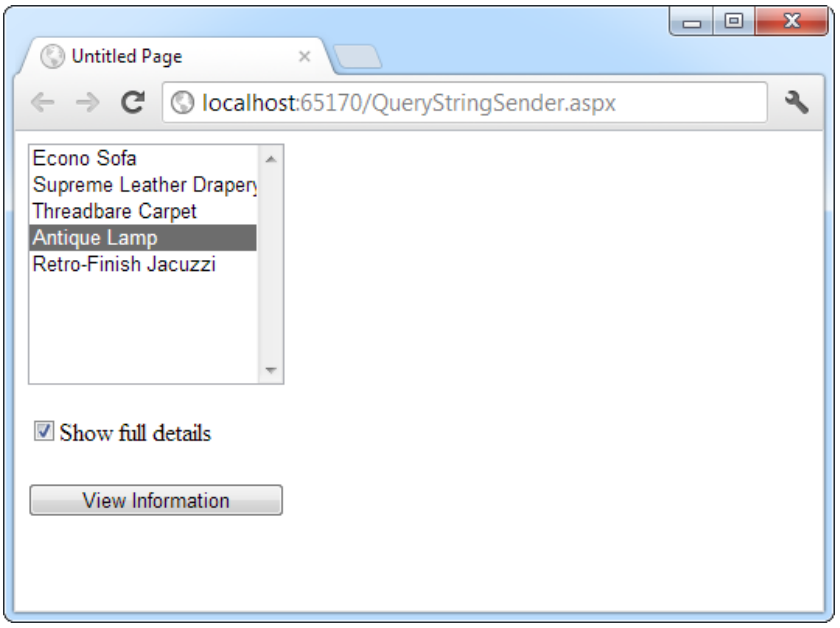


***Figure 8-6.*** *A query string sender*

Here's the code for the first page:

```
public partial class QueryStringSender : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        if (!this.IsPostBack)
```

```
        {
        // Add sample values.
            lstItems.Items.Add("Econo Sofa");
            lstItems.Items.Add("Supreme Leather Drapery");
            lstItems.Items.Add("Threadbare Carpet");
            lstItems.Items.Add("Antique Lamp");
            lstItems.Items.Add("Retro-Finish Jacuzzi");
        }
    }

    protected void cmdGo_Click(Object sender, EventArgs e)
    {
        if (lstItems.SelectedIndex == -1)
        {
            lblError.Text = "You must select an item.";
        }
        else
        {
            // Forward the user to the information page,
            // with the query string data.
            string url = "QueryStringRecipient.aspx?";
            url += "Item=" + lstItems.SelectedItem.Text + "&";
            url += "Mode=" + chkDetails.Checked.ToString();
            Response.Redirect(url);
        }
    }
}
```

Here's the code for the recipient page (shown in Figure 8-7):


***Figure 8-7.*** *A query string recipient*

```
public partial class QueryStringRecipient : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        lblInfo.Text = "Item: " + Request.QueryString["Item"];
        lblInfo.Text += "<br />Show Full Record: ";
        lblInfo.Text += Request.QueryString["Mode"];
    }
}
```

One interesting aspect of this example is that it places information in the query string that isn't valid—namely, the space that appears in the item name. When you run the application, you'll notice that ASP.NET encodes the string for you automatically, converting spaces to the valid %20 equivalent escape sequence. The recipient page reads the original values from the QueryString collection without any trouble. This automatic encoding isn't always sufficient. To deal with special characters, you should use the URL encoding technique described in the next section.

## URL Encoding

One potential problem with the query string is that some characters aren't allowed in a URL. In fact, the list of characters that are allowed in a URL is much shorter than the list of allowed characters in an HTML document. All characters must be alphanumeric or one of a small set of special characters (including `$-_.+!*'(),`). Some browsers tolerate certain additional special characters (Internet Explorer is notoriously lax), but many do not. Furthermore, some characters have special meaning. For example, the ampersand (&) is used to separate multiple query string parameters, the plus sign (+) is an alternate way to represent a space, and the number sign (#) is used to point to a specific bookmark in a web page. If you try to send query string values that include any of these characters, you'll lose some of your data. You can test this with the previous example by adding items with special characters in the list box.

To avoid potential problems, it's a good idea to perform *URL encoding* on text values before you place them in the query string. With URL encoding, special characters are replaced by escaped character sequences starting with the percent sign (%), followed by a two-digit hexadecimal representation. For example, the & character becomes %26. The only exception is the space character, which can be represented as the character sequence %20 or the + sign.

To perform URL encoding, you use the UrlEncode() and UrlDecode() methods of the HttpServerUtility class. As you learned in Chapter 5, an HttpServerUtility object is made available to your code in every web form through the Page.Server property. The following code uses the UrlEncode() method to rewrite the previous example, so it works with product names that contain special characters:

```
string url = "QueryStringRecipient.aspx?";
url += "Item=" + Server.UrlEncode(lstItems.SelectedItem.Text) + "&";
url += "Mode=" + chkDetails.Checked.ToString();
Response.Redirect(url);
```

Notice that it's important not to encode everything. In this example, you can't encode the & character that joins the two query string values, because it truly *is* a special character.

You can use the UrlDecode() method to return a URL-encoded string to its initial value. However, you don't need to take this step with the query string. That's because ASP.NET automatically decodes your values when you access them through the Request.QueryString collection. (Many people still make the mistake of decoding the query string values a second time. Usually, decoding already-decoded data won't cause a problem. The only exception occurs when you have a value that includes the + sign. In this case, using UrlDecode() will convert the + sign to a space, which isn't what you want.)

# Using Cookies

Cookies provide another way to store information for later use. *Cookies* are small files that are created in the web browser's memory (if they're temporary) or on the client's hard drive (if they're permanent). One advantage of cookies is that they work transparently, without the user being aware that information needs to be stored. They also can be easily used by any page in your application and even be retained between visits, which allows for truly long-term storage. They suffer from some of the same drawbacks that affect query strings—namely, they're limited to simple string information, and they're easily accessible and readable if the user finds and opens the corresponding file. These factors make them a poor choice for complex or private information or large amounts of data.

Some users disable cookies on their browsers, which will cause problems for web applications that require them. Also, users might manually delete the cookie files stored on their hard drives. But for the most part, cookies are widely adopted and used extensively on many websites.

Before you can use cookies, you should import the System.Net namespace so you can easily work with the appropriate types:

```
using System.Net;
```

Cookies are fairly easy to use. Both the Request and Response objects (which are provided through Page properties) provide a Cookies collection. The important trick to remember is that you retrieve cookies from the Request object, and you set cookies by using the Response object.

To set a cookie, just create a new HttpCookie object. You can then fill it with string information (using the familiar dictionary pattern) and attach it to the current web response:

```
// Create the cookie object.
HttpCookie cookie = new HttpCookie("Preferences");

// Set a value in it.
cookie["LanguagePref"] = "English";

// Add another value.
cookie["Country"] = "US";

// Add it to the current web response.
Response.Cookies.Add(cookie);
```

A cookie added in this way will persist until the user closes the browser and will be sent with every request. To create a longer-lived cookie, you can set an expiration date:

```
// This cookie lives for one year.
cookie.Expires = DateTime.Now.AddYears(1);
```

You retrieve cookies by cookie name, using the Request.Cookies collection. Here's how you retrieve the preceding cookie, which is named *Preferences*:

```
HttpCookie cookie = Request.Cookies["Preferences"];
```

This code won't cause an exception if the cookie doesn't exist. Instead, you'll simply get a null reference. Before you attempt to retrieve any data from the cookie, you should test the reference to make sure you actually *have* the cookie:

```
string language;
if (cookie != null)
{
    language = cookie["LanguagePref"];
}
```

The only way to remove a cookie is by replacing it with a cookie that has an expiration date that has already passed. This code demonstrates the technique:

```
HttpCookie cookie = new HttpCookie("Preferences");
cookie.Expires = DateTime.Now.AddDays(-1);
Response.Cookies.Add(cookie);
```

## A Cookie Example

The next example shows a typical use of cookies to store a customer name (Figure 8-8). To try this example, begin by running the page, entering a name, and clicking the Create Cookie button. Then close the browser, and request the page again. The second time, the page will find the cookie, read the name, and display a welcome message.

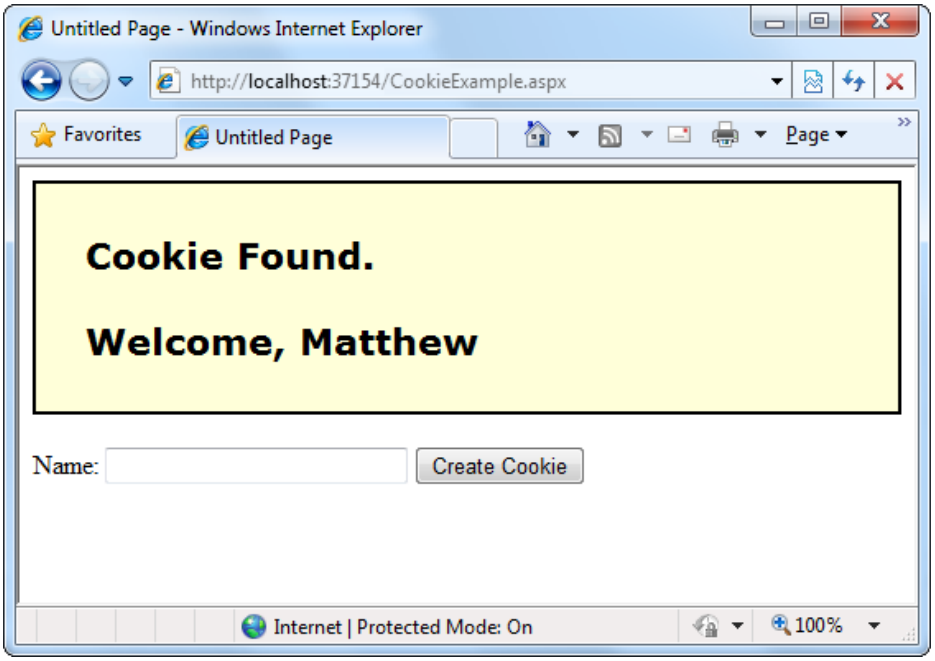


***Figure 8-8.*** *Displaying information from a custom cookie*

Here's the code for this page:

```
public partial class CookieExample : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        HttpCookie cookie = Request.Cookies["Preferences"];
        if (cookie == null)
        {
            lblWelcome.Text = "<b>Unknown Customer</b>";
        }
        else
        {
            lblWelcome.Text = "<b>Cookie Found.</b><br /><br />";
            lblWelcome.Text += "Welcome, " + cookie["Name"];
        }
    }

    protected void cmdStore_Click(Object sender, EventArgs e)
    {
        // Check for a cookie, and create a new one only if
        // one doesn't already exist.
        HttpCookie cookie = Request.Cookies["Preferences"];
        if (cookie == null)
        {
            cookie = new HttpCookie("Preferences");
        }

        cookie["Name"] = txtName.Text;
        cookie.Expires = DateTime.Now.AddYears(1);
        Response.Cookies.Add(cookie);

        lblWelcome.Text = "<b>Cookie Created.</b><br /><br />";
        lblWelcome.Text += "New Customer: " + cookie["Name"];
    }
}
```

---

■ **Note** You'll find that some other ASP.NET features use cookies. Two examples are session state (which allows you to temporarily store user-specific information in server memory) and forms security (which allows you to restrict portions of a website and force users to access it through a login page). Chapter 19 discusses forms security, and the next section of this chapter discusses session state.

---

# Managing Session State

There comes a point in the life of most applications when they begin to have more-sophisticated storage requirements. An application might need to store and access complex information such as custom data objects, which can't be easily persisted to a cookie or sent through a query string. Or the application might have stringent security requirements that prevent it from storing information about a client in view state or in a custom cookie. In these situations, you can use ASP.NET's built-in session-state facility.

Session-state management is one of ASP.NET's premiere features. It allows you to store any type of data in memory on the server. The information is protected, because it is never transmitted to the client, and it's uniquely bound to a specific session. Every client that accesses the application has a different session and a distinct collection of information. Session state is ideal for storing information such as the items in the current user's shopping basket when the user browses from one page to another.

## Session Tracking

ASP.NET tracks each session by using a unique 120-bit identifier. ASP.NET uses a proprietary algorithm to generate this value, thereby guaranteeing (statistically speaking) that the number is unique and it's random enough that a malicious user can't reverse-engineer or "guess" what session ID a given client will be using. This ID is the only piece of session-related information that is transmitted between the web server and the client.

When the client presents the session ID, ASP.NET looks up the corresponding session, retrieves the objects you stored previously, and places them into a special collection so they can be accessed in your code. This process takes place automatically.

For this system to work, the client must present the appropriate session ID with each request. You can accomplish this in two ways:

*Using cookies:* In this case, the session ID is transmitted in a special cookie (named ASP.NET_SessionId), which ASP.NET creates automatically when the session collection is used. This is the default.

*Using modified URLs:* In this case, the session ID is transmitted in a specially modified (or *munged*) URL. This allows you to create applications that use session state with clients that don't support cookies.

Ordinarily, ASP.NET uses cookies to track session state. You'll learn how to switch to the modified-URL system (called cookieless sessions) later in this chapter, when you tackle session-state configuration. But first, it's time to see for yourself how session state works in a website.

## Using Session State

You can interact with session state by using the System.Web.SessionState.HttpSessionState class, which is provided in an ASP.NET web page as the built-in Session object. The syntax for adding items to the collection and retrieving them is basically the same as for adding items to a page's view state.

For example, you might store a DataSet in session memory like this:

```
Session["InfoDataSet"] = dsInfo;
```

You can then retrieve it with an appropriate conversion operation:

```
dsInfo = (DataSet)Session["InfoDataSet"];
```

Of course, before you attempt to use the dsInfo object, you'll need to check that it actually exists—in other words, that it isn't a null reference. If the dsInfo is null, it's up to you to regenerate it. (For example, you might decide to query a database to get the latest data.)

---

■ **Note** Chapter 14 explores the DataSet.

---

Session state is global to your entire application for the current user. However, session state can be lost in several ways:

- If the user closes and restarts the browser.
- If the user accesses the same page through a different browser window, although the session will still exist if a web page is accessed through the original browser window. Browsers differ on how they handle this situation.
- If the session times out due to inactivity. More information about session timeout can be found in the configuration section.
- If your web page code ends the session by calling the Session.Abandon() method.

In the first two cases, the session actually remains in memory on the web server, because ASP.NET has no idea that the client has closed the browser or changed windows. The session will linger in memory, remaining inaccessible, until it eventually expires. (Ordinarily, that's after 20 minutes, but you'll learn how to configure it later in this chapter.)

Table 8-1 describes the key methods and properties of the HttpSessionState class.

***Table 8-1.*** *HttpSessionState Members*

| Member | Description |
|---|---|
| Count | Provides the number of items in the current session collection. |
| IsCookieless | Identifies whether the session is tracked with a cookie or modified URLs. |
| IsNewSession | Identifies whether the session was created only for the current request. If no information is in session state, ASP.NET won't bother to track the session or create a session cookie. Instead, the session will be re-created with every request. |
| Keys | Gets a collection of all the session keys that are currently being used to store items in the session-state collection. |
| Mode | Provides an enumerated value that explains how ASP.NET stores session-state information. This storage mode is determined based on the web.config settings discussed in the "Configuring Session State" section later in this chapter. |
| SessionID | Provides a string with the unique session identifier for the current client. |
| Timeout | Determines the number of minutes that will elapse before the current session is abandoned, provided that no more requests are received from the client. This value can be changed programmatically, letting you make the session collection longer when needed. |
| Abandon() | Cancels the current session immediately and releases all the memory it occupied. This is a useful technique in a logoff page to ensure that server memory is reclaimed as quickly as possible. |
| Clear() | Removes all the session items but doesn't change the current session identifier. |

## A Session-State Example

The next example uses session state to store several Furniture data objects. The data object combines a few related variables and uses a special constructor so it can be created and initialized in one easy line. Rather than use full property procedures, the class takes a shortcut and uses public member variables so that the code listing remains short and concise. (If you refer to the full code in the downloadable examples, you'll see that it uses property procedures.)

```
public class Furniture
{
    public string Name;
    public string Description;
    public decimal Cost;

    public Furniture(string name, string description,
      decimal cost)
    {
        Name = name;
        Description = description;
        Cost = cost;
    }
}
```

Three Furniture objects are created the first time the page is loaded, and they're stored in session state. The user can then choose from a list of furniture-piece names. When a selection is made, the corresponding object will be retrieved, and its information will be displayed, as shown in Figure 8-9.

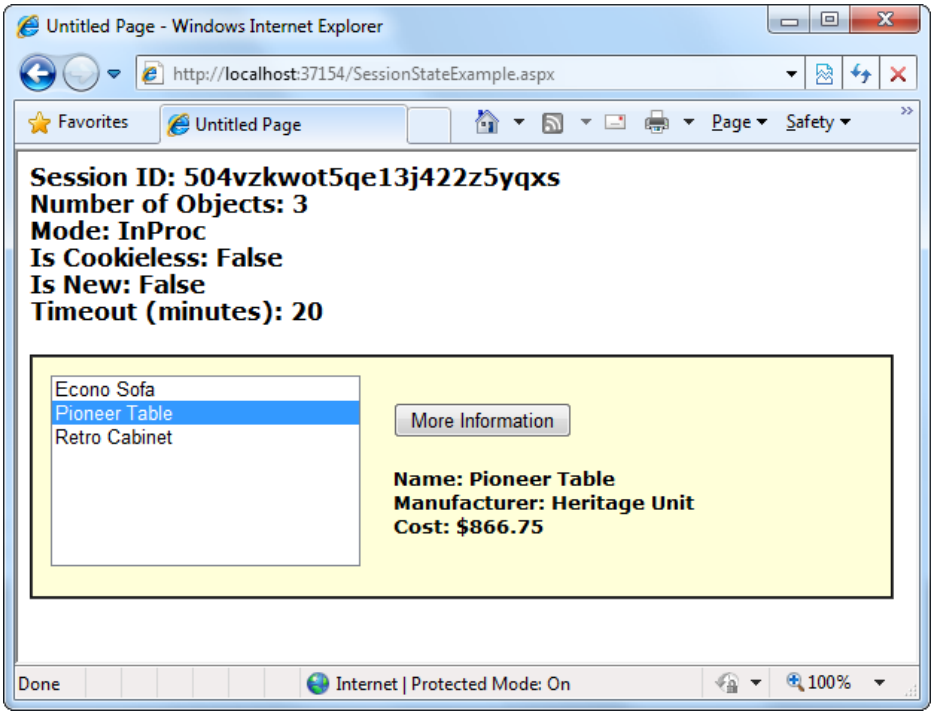


***Figure 8-9.*** *A session-state example with data objects*

```
public partial class SessionStateExample : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // Create Furniture objects.
            Furniture piece1 = new Furniture("Econo Sofa",
                                       "Acme Inc.", 74.99M);
            Furniture piece2 = new Furniture("Pioneer Table",
                                       "Heritage Unit", 866.75M);
            Furniture piece3 = new Furniture("Retro Cabinet",
                                       "Sixties Ltd.", 300.11M);

            // Add objects to session state.
            Session["Furniture1"] = piece1;
            Session["Furniture2"] = piece2;
            Session["Furniture3"] = piece3;

            // Add rows to list control.
            lstItems.Items.Add(piece1.Name);
            lstItems.Items.Add(piece2.Name);
            lstItems.Items.Add(piece3.Name);
        }

        // Display some basic information about the session.
        // This is useful for testing configuration settings.
        lblSession.Text = "Session ID: " + Session.SessionID;
        lblSession.Text += "<br />Number of Objects: ";
        lblSession.Text += Session.Count.ToString();
        lblSession.Text += "<br />Mode: " + Session.Mode.ToString();
        lblSession.Text += "<br />Is Cookieless: ";
        lblSession.Text += Session.IsCookieless.ToString();
        lblSession.Text += "<br />Is New: ";
        lblSession.Text += Session.IsNewSession.ToString();
        lblSession.Text += "<br />Timeout (minutes): ";
        lblSession.Text += Session.Timeout.ToString();
    }

    protected void cmdMoreInfo_Click(Object sender, EventArgs e)
    {
        if (lstItems.SelectedIndex == -1)
        {
            lblRecord.Text = "No item selected.";
        }
        else
        {
            // Construct the right key name based on the index.
            string key = "Furniture" +
                  (lstItems.SelectedIndex + 1).ToString();
```

```
            // Retrieve the Furniture object from session state.
            Furniture piece = (Furniture)Session[key];

            // Display the information for this object.
            lblRecord.Text = "Name: " + piece.Name;
            lblRecord.Text += "<br />Manufacturer: ";
            lblRecord.Text +=  piece.Description;
            lblRecord.Text += "<br />Cost: " + piece.Cost.ToString("c");
        }
    }
}
```

It's also a good practice to add a few session-friendly features in your application. For example, you could add a logout button to the page that automatically cancels a session by using the Session.Abandon() method. This way, the user will be encouraged to terminate the session rather than just close the browser window, and the server memory will be reclaimed faster. Otherwise, the memory won't be reclaimed until the session times out, which is a potential drag on performance.

# Configuring Session State

You configure session state through the web.config file for your current application (which is found in the same virtual directory as the .aspx web page files). The configuration file allows you to set advanced options such as the timeout and the session-state mode.

The following listing shows the most important options that you can set for the <sessionState> element. Keep in mind that you won't use all of these details at the same time. Some settings apply only to certain session-state *modes*, as you'll see shortly.

```
<configuration>
    ...
    <system.web>
        ...
        <sessionState
            cookieless="UseCookies"
            cookieName="ASP.NET_SessionId"
            regenerateExpiredSessionId="false"
            timeout="20"
            mode="InProc"
            stateConnectionString="tcpip=127.0.0.1:42424"
            stateNetworkTimeout="10"
            sqlConnectionString="data source=127.0.0.1;Integrated Security=SSPI"
            sqlCommandTimeout="30"
            allowCustomSqlDatabase="false"
            customProvider=""
            compressionEnabled="false"
        />
    </system.web>
</configuration>
```

The following sections describe the most important session-state settings. You'll learn how to change session timeouts, use cookieless sessions, and change the way session information is stored.

## Timeout

The timeout setting specifies the number of minutes that ASP.NET will wait, without receiving a request, before it abandons the session.

This setting represents one of the important compromises of session state. If sessions are kept too long, the memory usage of a popular web application will increase, and the performance will decline. Ideally, you will choose a timeframe that is short enough to allow the server to reclaim valuable memory after a client stops using the application but long enough to allow a client to pause and continue a session without losing it.

You can also programmatically change the session timeout in code. For example, if you know a session contains an unusually large amount of information, you may need to limit the amount of time the session can be stored. You would then warn the user and change the Timeout property. Here's a sample line of code that changes the timeout to 10 minutes:

```
Session.Timeout = 10;
```

## Cookieless

As you already learned, ASP.NET tracks sessions by using a cookie that it creates and maintains automatically (which shouldn't be confused with the custom cookies your web page may create and manipulate). The ASP.NET session cookie is a bit of behind-the-scenes plumbing, and most of the time you won't think twice about it. But in some environments, cookies aren't the best choice. For example, maybe they're restricted by super-strict security settings. In this case, you may choose to use cookieless sessions.

To allow (or enforce) cookieless sessions, you set the cookieless attribute to one of the values defined by the HttpCookieMode enumeration, as listed in Table 8-2.

***Table 8-2.*** *HttpCookieMode Values*

| Value | Description |
|---|---|
| UseCookies | Cookies are always used, even if the browser or device doesn't support cookies or they are disabled. This is the default. If the device does not support cookies, session information will be lost over subsequent requests, because each request will get a new ID. |
| UseUri | Cookies are never used, regardless of the capabilities of the browser or device. Instead, the session ID is stored in the URL. |
| UseDeviceProfile | ASP.NET chooses whether to use cookieless sessions by examining the BrowserCapabilities object. The drawback is that this object indicates what the device should support—it doesn't take into account that the user may have disabled cookies in a browser that supports them. |
| AutoDetect | ASP.NET attempts to determine whether the browser supports cookies by attempting to set and retrieve a cookie (a technique commonly used on the Web). This technique can correctly determine whether a browser supports cookies but has them disabled, in which case cookieless mode is used instead. |

Here's an example that forces cookieless mode:

```
<sessionState cookieless="UseUri" ... />
```

In cookieless mode, the session ID will automatically be inserted into the URL. When ASP.NET receives a request, it will remove the ID, retrieve the session collection, and forward the request to the appropriate directory. Figure 8-10 shows a munged URL.

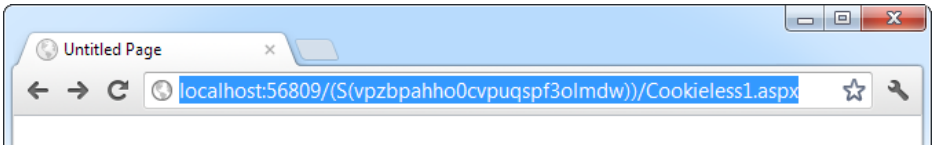


***Figure 8-10.*** *A munged URL with the session ID*

Because the session ID is inserted in the current URL, relative links also automatically gain the session ID. In other words, if the user is currently stationed on Page1.aspx and clicks a relative link to Page2.aspx, the relative link includes the current session ID as part of the URL. The same is true if you call Response.Redirect() with a relative URL, as shown here:

```
Response.Redirect("Page2.aspx");
```

Figure 8-11 shows a sample website (included with the online samples in the CookielessSessions directory) that tests cookieless sessions. It contains two pages and uses cookieless mode. The first page (Cookieless1.aspx) contains a HyperLink control and two buttons, all of which take you to a second page (Cookieless2.aspx). The trick is that these controls have different ways of performing their navigation. Only two of them work with cookieless sessions—the third loses the current session.

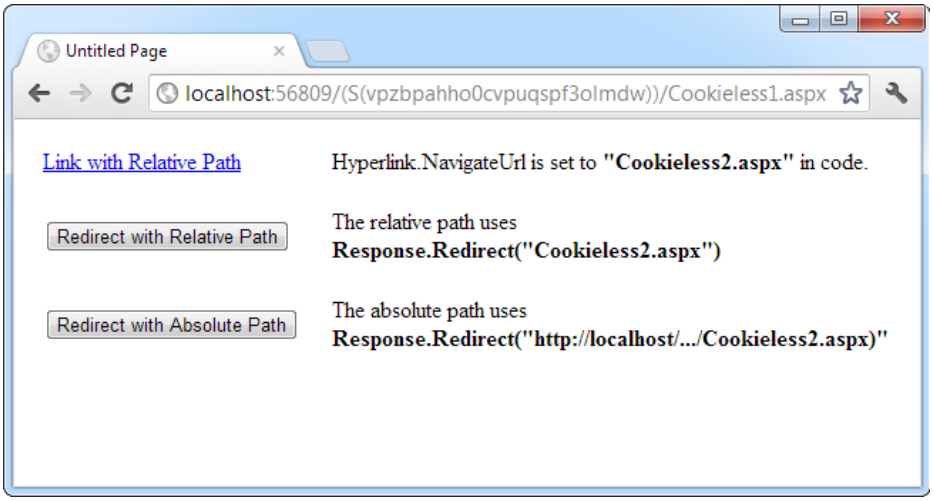


***Figure 8-11.*** *Three tests of cookieless sessions*

The HyperLink control navigates to the page specified in its NavigateUrl property, which is set to the relative path Cookieless2.aspx. If you click this link, the session ID is retained in the URL, and the new page can retrieve the session information. This demonstrates that cookieless sessions work with relative links.

The two buttons on this page use programmatic redirection by calling the Response.Redirect() method. The first button uses the relative path Cookieless2.aspx, much like the HyperLink control. This approach works with cookieless session state and preserves the munged URL with no extra steps required.

```
protected void cmdLink_Click(Object sender, EventArgs e)
{
    Response.Redirect("Cookieless2.aspx");
}
```

The only real limitation of cookieless state is that you cannot use absolute links (links that include the full URL, starting with `http://`). The second button uses an absolute link to demonstrate this problem. Because ASP.NET cannot insert the session ID into the URL, the session is lost.

```
protected void cmdLinkAbsolute_Click(Object sender, EventArgs e)
{
    Response.Redirect("http://localhost:56371/CookielessSessions/Cookieless2.aspx");
}
```

Now the target page checks for the session but can't find it. ASP.NET generates a new session ID (which it inserts in the URL), and the original information is lost. Figure 8-12 shows the result.


***Figure 8-12.*** *A lost session*

Writing the code to demonstrate this problem in a test environment is a bit tricky. The problem is that Visual Studio's integrated web server chooses a different port for your website every time you start it. As a result, you'll need to edit the code every time you open Visual Studio so that your URL uses the right port number (such as 56371 in the previous example).

There's another workaround. You can use some crafty code that gets the current URL from the page and just modifies the last part of it (changing the page name from Cookieless1.aspx to Cookieless2.aspx). Here's how:

```
// Create a new URL based on the current URL (but ending with
// the page Cookieless2.aspx instead of Cookieless1.aspx.
string url = "http://" + Request.Url.Authority +
  Request.Url.Segments[0] + Request.Url.Segments[1] +
  "Cookieless2.aspx";
Response.Redirect(url);
```

Of course, if you deploy your website to a real virtual directory that's hosted by IIS, you won't use a randomly chosen port number anymore, and you won't experience this quirk. Chapter 26 has more about virtual directories and website deployment.

### DEALING WITH EXPIRED SESSION IDS

By default, ASP.NET allows you to reuse a session identifier. For example, if you make a request and your query string contains an expired session, ASP.NET creates a new session and uses that session ID. The problem is that a session ID might inadvertently appear in a public place—such as in a results page in a search engine. This could lead to multiple users accessing the server with the same session identifier and then all joining the same session with the same shared data.

To avoid this potential security risk, you should include the optional regenerateExpiredSessionId attribute and set it to true whenever you use cookieless sessions. This way, a new session ID will be issued if a user connects with an expired session ID. The only drawback is that this process also forces the current page to lose all view state and form data, because ASP.NET performs a redirect to make sure the browser has a new session identifier.

## Mode

The remaining session-state settings allow you to configure ASP.NET to use different session-state services, depending on the mode that you choose. The next few sections describe the modes you can choose from.

■ **Note** Changing the mode is an advanced configuration task. To do it successfully, you need to understand the environment in which your web application will be deployed. For example, if you're deploying your application to a third-party web host, you need to know whether the host supports other modes before you try to use them. If you're deploying your application to a network server in your own organization, you need to team up with your friendly neighborhood network administrator.

### InProc

InProc is the default mode, and it makes the most sense for small websites. It instructs information to be stored in the same process as the ASP.NET worker threads, which provides the best performance but the least durability. If you restart your server, the state information will be lost. (In ASP.NET, application domains can be restarted for a variety of reasons, including configuration changes and updated pages, and when certain thresholds are met. If you find that you're losing sessions *before* the timeout limit, you may want to experiment with a more durable mode.)

InProc mode won't work if you're using a *web farm*, which is a load-balancing arrangement that uses multiple web servers to run your website. In this situation, different web servers might handle consecutive requests from the same user. If the web servers use InProc mode, each one will have its own private collection of session data. The end result is that users will unexpectedly lose their sessions when they travel to a new page or post back the current one.

■ **Note** When using the StateServer and SQLServer modes, the objects you store in session state must be serializable. Otherwise, ASP.NET will not be able to transmit the object to the state service or store it in the database. Earlier in this chapter, you learned how to create a serializable Customer class for storing in view state.

## Off

This setting disables session-state management for every page in the application. This can provide a slight performance improvement for websites that are not using session state.

## StateServer

With this setting, ASP.NET will use a separate Windows service for state management. This service runs on the same web server, but it's outside the main ASP.NET process, which gives it a basic level of protection if the ASP.NET process needs to be restarted. The cost is the increased time delay imposed when state information is transferred between two processes. If you frequently access and change state information, this can make for a fairly unwelcome slowdown.

When using the StateServer setting, you need to specify a value for the stateConnectionString setting. This string identifies the TCP/IP address of the computer that is running the StateServer service and its port number (which is defined by ASP.NET and doesn't usually need to be changed). This allows you to host the StateServer on another computer. If you don't change this setting, the local server will be used (set as address 127.0.0.1).

Of course, before your application can use the service, you need to start it. The easiest way to do this is to use the Microsoft Management Console (MMC). Here's how:

1. Select Start and type **Computer Management** into the search box.
2. When the Computer Management utility appears, click it.
3. In the Computer Management window, go to the Services and Applications ➤ Services node.
4. Find the service called ASP.NET State Service in the list, as shown in Figure 8-13.

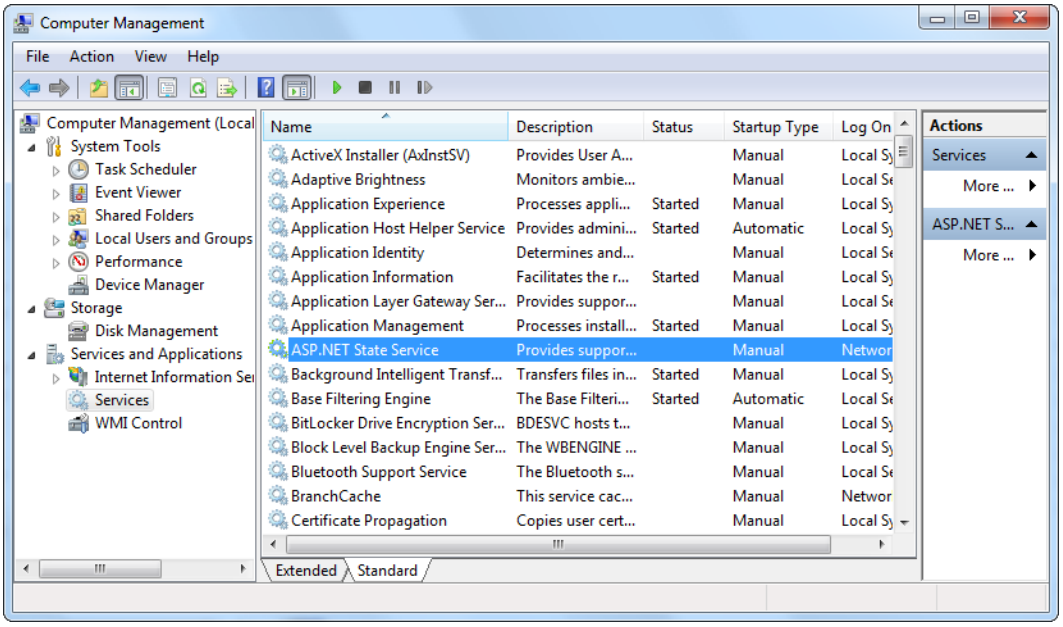


***Figure 8-13.*** *The ASP.NET state service*

5. When you find the service in the list, you can manually start and stop it by right-clicking it. Generally, you'll want to configure Windows to automatically start the service. Right-click it, select Properties, and modify the Startup Type, setting it to Automatic, as shown in Figure 8-14.

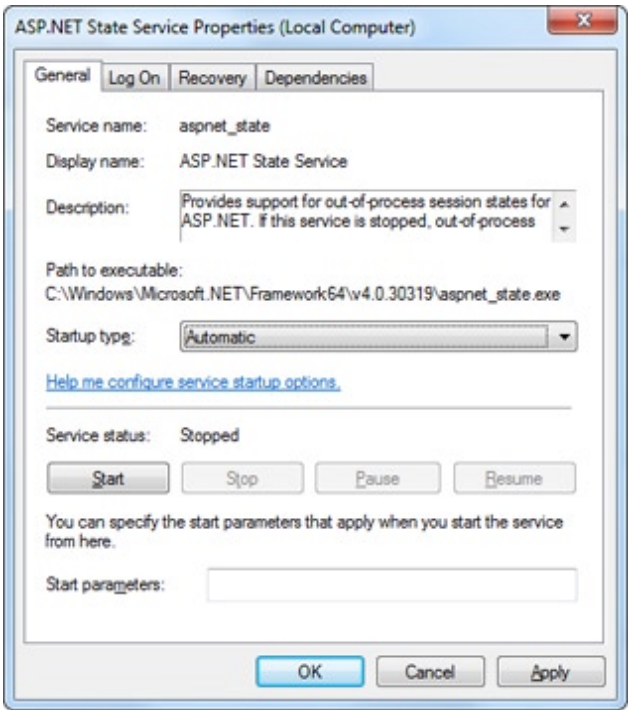


***Figure 8-14.*** *Changing the startup type*

---

■ **Note** When using StateServer mode, you can also set an optional stateNetworkTimeout attribute that specifies the maximum number of seconds to wait for the service to respond before canceling the request. The default value is 10 (seconds).

---

## SQLServer

This setting instructs ASP.NET to use an SQL Server database to store session information, as identified by the sqlConnectionString attribute. This is the most resilient state store but also the slowest by far. To use this method of state management, you'll need to have a server with SQL Server installed.

When setting the sqlConnectionString attribute, you follow the same sort of pattern you use with ADO.NET data access. Generally, you'll need to specify a data source (the server address) and a user ID and password, unless you're using SQL integrated security.

In addition, you need to install the special stored procedures and temporary session databases. These stored procedures take care of storing and retrieving the session information. ASP.NET includes a command-line tool that does the work for you automatically, called `aspnet_regsql.exe`. It's found in the c:\Windows\Microsoft.NET\Framework64\v4.0.30319 directory (because 4.0.30319 is the latest version of the CLR, and .NET 4.5 is actually a set of extensions *on top* of that core engine.) To run `aspnet_regsql.exe`, you need to open a Command Prompt window at this location. (One easy way to do that is to browse to the version folder in Windows Explorer, hold down Shift, and right-click the version folder. Then, choose "Open command menu window here" from the menu.) Once you're in the right folder, you can type in an `aspnet_regsql.exe` command.

You can use the aspnet_regsql.exe tool to perform several database-related tasks. As you travel through this book, you'll see how to use aspnet_regsql.exe with ASP.NET features such as membership (Chapter 20), profiles (Chapter 21), and caching (Chapter 23). To use aspnet_regsql.exe to create a session storage database, you supply the –ssadd parameter. In addition, you use the –S parameter to indicate the database server name, and the –E parameter to log in to the database via the currently logged-in Windows user account.

Here's a command that creates the session storage database on the current computer, using the default database name ASPState:

```
aspnet_regsql.exe -S localhost -E –ssadd
```

This command uses the alias *localhost*, which tells aspnet_regsql.exe to connect to the database server on the current computer.

---

■ **Note** The aspnet_regsql.exe command supports additional options that allow you to store session information in a database with a different name. You can find out about these options by referring to the Visual Studio help (look up aspnet_regsql in the index) or by surfing to http://msdn.microsoft.com/library/ms178586.aspx. This information also describes the extra steps you need to take to use the database-backed session storage with SQL Server Express.

---

After you've created your session-state database, you need to tell ASP.NET to use it by modifying the <sessionState> section of the web.config file. If you're using a database named ASPState to store your session information (which is the default), you don't need to supply the database name. Instead, you simply have to indicate the location of the server and the type of authentication that ASP.NET should use to connect to it, as shown here:

```
<sessionState mode="SQLServer"
 sqlConnectionString="data source=127.0.0.1;Integrated Security=SSPI"
 ... />
```

When using the SQLServer mode, you can also set an optional sqlCommandTimeout attribute that specifies the maximum number of seconds to wait for the database to respond before canceling the request. The default is 30 seconds.

### Custom

When using custom mode, you need to indicate which session-state store provider to use by supplying the customProvider attribute. The customProvider attribute indicates the name of the class. The class may be part of your web application (in which case the source code is placed in the App_Code subfolder), or it can be in an assembly that your web application is using (in which case the compiled assembly is placed in the Bin subfolder).

Creating a custom state provider is a low-level task that needs to be handled carefully to ensure security, stability, and scalability. Custom state providers are also beyond the scope of this book. However, other vendors may release custom state providers you want to use. For example, Oracle could provide a custom state provider that allows you to store state information in an Oracle database.

### Compression

When you set enableCompression to true, session data is compressed before it's passed out of process. The enableCompression setting has an effect only when you're using out-of-process session-state storage, because it's only in this situation that the data is serialized.

To compress and decompress session data, the web server needs to perform additional work. However, this isn't usually a problem, because compression is used in scenarios where web servers have plenty of CPU time to spare but are limited by other factors. Session-state compression makes sense in these two key scenarios:

> *When storing huge amounts of session-state data in memory:* Web server memory is a precious resource. Ideally, session state is used for relatively small chunks of information, while a database deals with the long-term storage of larger amounts of data. But if this isn't the case, and if the out-of-process state server is hogging huge amounts of memory, compression is a potential solution.
>
> *When storing session-state data on another computer:* In some large-scale web applications, session state is stored out of process (usually in SQL Server) and on a separate computer. As a result, ASP.NET needs to pass the session information back and forth over a network connection. Clearly, this design reduces performance from the speeds you'll see when session state is stored on the web server computer. However, it's still the best compromise for some heavily trafficked web applications with huge session-state storage needs.

The actual amount of compression varies greatly depending on the type of data. However, in testing, Microsoft saw clients achieve 30 percent to 60 percent size reductions, which is enough to improve performance in these specialized scenarios.

## Using Application State

*Application state* allows you to store global objects that can be accessed by any client. Application state is based on the System.Web.HttpApplicationState class, which is provided in all web pages through the built-in Application object.

Application state is similar to session state. It supports the same type of objects, retains information on the server, and uses the same dictionary-based syntax. A common example of using application state is a global counter that tracks the number of times an operation has been performed by all the web application's clients.

For example, you could create a global.asax event handler that tracks the number of sessions that have been created or the number of requests that have been received into the application. Or you can use similar logic in the Page.Load event handler to track the number of times a given page has been requested by various clients. Here's an example of the latter:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Retrieve the current counter value.
    int count = 0;
    if (Application["HitCounterForOrderPage"] != null)
    {
        count = (int)Application["HitCounterForOrderPage"];
    }

    // Increment the counter.
    count++;

    // Store the current counter value.
    Application["HitCounterForOrderPage"] = count;
    lblCounter.Text = count.ToString();
}
```

Once again, application-state items are stored as objects, so you need to cast them when you retrieve them from the collection. Items in application state never time out. They last until the application or server is restarted or the application domain refreshes itself (because of automatic process recycling settings or an update to one of the pages or components in the application).

Application state isn't often used, because it's generally inefficient. In the previous example, the counter would probably not keep an accurate count, particularly in times of heavy traffic. For example, if two clients requested the page at the same time, you could have a sequence of events like this:

1. User A retrieves the current count (432).
2. User B retrieves the current count (432).
3. User A sets the current count to 433.
4. User B sets the current count to 433.

In other words, one request isn't counted because two clients access the counter at the same time. To prevent this problem, you need to use the Lock() and Unlock() methods, which explicitly allow only one client to access the Application state collection at a time.

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Acquire exclusive access.
    Application.Lock();

    int count = 0;
    if (Application["HitCounterForOrderPage"] != null)
    {
        count = (int)Application["HitCounterForOrderPage"];
    }
    count++;
    Application["HitCounterForOrderPage"] = count;
```

```
    // Release exclusive access.
    Application.Unlock();

    lblCounter.Text = count.ToString();
}
```

Unfortunately, all other clients requesting the page will be stalled until the Application collection is released. This can drastically reduce performance. Generally, frequently modified values are poor candidates for application state. In fact, application state is rarely used in the .NET world because its two most common uses have been replaced by easier, more efficient methods:

- In the past, application state was used to store application-wide constants, such as a database connection string. As you saw in Chapter 5, this type of constant can be stored in the web.config file, which is generally more flexible because you can change it easily without needing to hunt through web page code or recompile your application.
- Application state can also be used to store frequently used information that is time-consuming to create, such as a full product catalog that requires a database lookup. However, using application state to store this kind of information raises all sorts of problems about how to check whether the data is valid and how to replace it when needed. It can also hamper performance if the product catalog is too large. Chapter 23 introduces a similar but much more sensible approach—storing frequently used information in the ASP.NET cache. Many uses of application state can be replaced more efficiently with caching.

---

■ **Tip** If you decide to use application state, you can initialize its contents when your application first starts. Just add the initialization code to the global.asax file in a method named Application_OnStart(), as described in Chapter 5.

---

# Comparing State Management Options

Each state management choice has a different lifetime, scope, performance overhead, and level of support. Table 8-3 and Table 8-4 show an at-a-glance comparison of your state management options.

***Table 8-3.*** *State Management Options Compared (Part 1)*

| | View State | Query String | Custom Cookies |
|---|---|---|---|
| Allowed Data Types | All serializable .NET data types. | A limited amount of string data. | String data. |
| Storage Location | A hidden field in the current web page. | The browser's URL string. | The client's computer (in memory or a small text file, depending on its lifetime settings). |
| Lifetime | Retained permanently for postbacks to a single page. | Lost when the user enters a new URL or closes the browser. However, this can be stored in a bookmark. | Set by the programmer. Can be used in multiple pages and can persist between visits. |
| Scope | Limited to the current page. | Limited to the target page. | The whole ASP.NET application. |
| Security | Tamperproof by default but easy to read. You can enforce encryption by using the ViewStateEncryptionMode property of the Page directive. | Clearly visible and easy for the user to modify. | Insecure, and can be modified by the user. |
| Performance Implications | Slow if a large amount of information is stored, but will not affect server performance. | None, because the amount of data is trivial. | None, because the amount of data is trivial. |
| Typical Use | Page-specific settings. | Sending a product ID from a catalog page to a details page. | Personalization preferences for a website. |

***Table 8-4.*** *State Management Options Compared (Part 2)*

| | Session State | Application State |
|---|---|---|
| Allowed Data Types | All .NET data types for the default in-process storage mode. All serializable .NET data types if you use an out-of-process storage mode. | All .NET data types. |
| Storage Location | Server memory, state service, or SQL Server, depending on the mode you choose. | Server memory. |
| Lifetime | Times out after a predefined period (usually 20 minutes, but can be altered globally or programmatically). | The lifetime of the application (typically, until the server is rebooted). |
| Scope | The whole ASP.NET application. | The whole ASP.NET application. Unlike other methods, application data is global to all users. |
| Security | Very secure, because data is never transmitted to the client. | Very secure, because data is never transmitted to the client. |
| Performance Implications | Slow when storing a large amount of information, especially if there are many users at once, because each user will have their own copy of session data. | Slow when storing a large amount of information, because this data will never time out and be removed. |
| Typical Use | Storing items in a shopping basket. | Storing any type of global data. |

■ **Note** ASP.NET has another, more specialized type of state management called *profiles*. Profiles allow you to store and retrieve user-specific information from a database. The only catch is that you need to authenticate the user in order to get the right information. You'll learn about profiles in Chapter 21.

# The Last Word

State management is the art of retaining information between requests. Usually, this information is user-specific (such as a list of items in a shopping cart, a username, or an access level), but sometimes it's global to the whole application (such as usage statistics that track site activity). Because ASP.NET uses a disconnected architecture, you need to explicitly store and retrieve state information with each request. The approach you choose to store this data affects the performance, scalability, and security of your application.

In this chapter, you toured a variety of storage options, including view state, cookies, and session state. You also learned to pass information through cross-page postbacks and the query string. As you develop your own web applications, you can consult Table 8-3 and Table 8-4 to help evaluate different types of state management and determine what's best for your needs.

PART 3



# Building Better Web Forms

CHAPTER 9



# Validation

This chapter presents some of the most useful controls that are included in ASP.NET: the *validation controls*. These controls take a potentially time-consuming and complicated task—verifying user input and reporting errors—and automate it. Each validation control, or *validator*, has its own built-in logic. Some check for missing data, others verify that numbers fall in a predefined range, and so on. In many cases, the validation controls allow you to verify user input without writing a line of code.

In this chapter, you'll learn how to use the validation controls in an ASP.NET web page and how to get the most out of them with sophisticated regular expressions, custom validation functions, and more. And as usual, you'll peer under the hood to see how ASP.NET implements these features.

## Understanding Validation

As any seasoned developer knows, the people using your website will occasionally make mistakes. What's particularly daunting is the range of possible mistakes that users can make. Here are some common examples:

- A user might ignore an important field and leave it blank.
- If you disallow blank values, a user might type in semi-random nonsense to circumvent your checks. This can create endless headaches on your end. For example, you might get stuck with an invalid e-mail address that causes problems for your automatic e-mailing program.
- A user might make an honest mistake, such as entering a typing error, entering a nonnumeric character in a number field, or submitting the wrong type of information. A user might even enter several pieces of information that are individually correct but when taken together are inconsistent (for example, entering a MasterCard number after choosing Visa as the payment type).
- A malicious user might try to exploit a weakness in your code by entering carefully structured wrong values. For example, an attacker might attempt to cause a specific error that will reveal sensitive information. A more dramatic example of this technique is the *SQL injection attack*, whereby user-supplied values change the operation of a dynamically constructed database command. (Of course, validation is no defense for poor coding. When you consider database programming in Chapter 14, you'll learn how to use parameterized commands, which avoid the danger of SQL injection attacks altogether.)

A web application is particularly susceptible to these problems, because it relies on basic HTML input controls that don't have all the features of their Windows counterparts. For example, a common technique in a Windows application is to handle the KeyPress event of a text box, check to see whether the current character is valid, and prevent it from appearing if it isn't. This technique makes it easy to create a text box that accepts only numeric input.

This strategy isn't as easy in a server-side web page. To perform validation on the web server, you need to post back the page, and it just isn't practical to post the page back to the server every time the user types a letter. To avoid this sort of problem, you need to perform all your validation at once when a page (which may contain multiple input controls) is submitted. You then need to create the appropriate user interface to report the mistakes. Some websites report only the first incorrect field, while others use a table, list, or window to describe them all. By the time you've perfected your validation strategy, you'll have spent a considerable amount of effort writing tedious code.

ASP.NET aims to save you this trouble and provide you with a reusable framework of validation controls that manages validation details by checking fields and reporting on errors automatically. These controls can even use client-side JavaScript to provide a more dynamic and responsive interface while still providing ordinary validation for older browsers (often referred to as *down-level* browsers).

## The Validation Controls

ASP.NET provides five validator controls, which are described in Table 9-1. Four are targeted at specific types of validation, while the fifth allows you to apply custom validation routines. You'll also see a ValidationSummary control in the Toolbox, which gives you another option for showing a list of validation error messages in one place. You'll learn about the ValidationSummary later in this chapter (see the "Other Display Options" section).

***Table 9-1.*** *Validator Controls*

| Control Class | Description |
|---|---|
| RequiredFieldValidator | Validation succeeds as long as the input control doesn't contain an empty string. |
| RangeValidator | Validation succeeds if the input control contains a value within a specific numeric, alphabetic, or date range. |
| CompareValidator | Validation succeeds if the input control contains a value that matches the value in another input control, or a fixed value that you specify. |
| RegularExpressionValidator | Validation succeeds if the value in an input control matches a specified regular expression. |
| CustomValidator | Validation is performed by a user-defined function. |

Each validation control can be bound to a single input control. In addition, you can apply more than one validation control to the same input control to provide multiple types of validation.

If you use the RangeValidator, CompareValidator, or RegularExpressionValidator, validation will automatically succeed if the input control is empty, because there is no value to validate. If this isn't the behavior you want, you should also add a RequiredFieldValidator and link it to the same input control. This ensures that two types of validation will be performed, effectively restricting blank values.

## Server-Side Validation

You can use the validator controls to verify a page automatically when the user submits it or manually in your code. The first approach is the most common.

When using automatic validation, the user receives a normal page and begins to fill in the input controls. When finished, the user clicks a button to submit the page. Every button has a CausesValidation property,

which can be set to true or false. What happens when the user clicks the button depends on the value of the CausesValidation property:

- If CausesValidation is false, ASP.NET will ignore the validation controls, the page will be posted back, and your event-handling code will run normally.
- If CausesValidation is true (the default), ASP.NET will automatically validate the page when the user clicks the button. It does this by performing the validation for each control on the page. If any control fails to validate, ASP.NET will return the page with some error information, depending on your settings. Your click event-handling code may or may not be executed—meaning you'll have to specifically check in the event handler whether the page is valid.

Based on this description, you'll realize that validation happens automatically when certain buttons are clicked. It doesn't happen when the page is posted back because of a change event (such as choosing a new value in an AutoPostBack list) or if the user clicks a button that has CausesValidation set to false. However, you can still validate one or more controls manually and then make a decision in your code based on the results. You'll learn about this process in more detail a little later (see the "Manual Validation" section).

---

■ **Note** Many other button-like controls that can be used to submit the page also provide the CausesValidation property. Examples include the LinkButton, ImageButton, and BulletedList. (Technically, the CausesValidation property is defined by the IButtonControl interface, which all button-like controls implement.)

---

## Client-Side Validation

In modern browsers (including Microsoft Internet Explorer, Mozilla Firefox, Apple Safari, and Google Chrome), ASP.NET automatically adds JavaScript code for client-side validation. In this case, when the user clicks a CausesValidation button, the same error messages will appear without the page needing to be submitted and returned from the server. This increases the responsiveness of your web page.

However, even if the page validates successfully on the client side, ASP.NET still revalidates it when it's received at the server. This is because it's easy for an experienced user to circumvent client-side validation. For example, a malicious user might delete the block of JavaScript validation code and continue working with the page. By performing the validation at both ends, ASP.NET makes sure your application can be as responsive as possible while also remaining secure.

## HTML5 Validation

HTML5, the most modern version of the HTML language, adds new client-side validation features that can help catch errors. The problem is that HTML5 validation is inconsistent—it works differently in different browsers, and many browsers offer only partial support. (For the complete details, you can refer to the compatibility table at `http://caniuse.com/form-validation`.)

HTML5 validation has essentially the same effect as JavaScript-based validation. When the user types in data that doesn't match the expected data type or validation rule—for example, if the user puts text in a numeric field—the browser detects the problem. It prevents the form from being submitted and displays an error message next to the offending field.

You can use HTML5 validation in an ASP.NET web form, but it probably isn't the best choice, for two reasons:

> *On its own, HTML5 validation isn't a complete solution.* If you use HTML5 validation, you'll need to add a JavaScript fallback for browsers that don't support HTML5 validation, and you'll need to perform similar checks on the server to catch data that's been tampered with. But if you use ASP.NET's validators, you'll get compatibility with all browsers and server-side checking for free.
>
> *HTML5 validation works on <input> elements, not web controls.* And because web controls are designed to work with all browsers, not just those with the latest HTML5 features, they don't include HTML5-specific properties. That means there's no clean and practical way to set the HTML5 *required* attribute (for required fields) and the *pattern* attribute (for regular expression validation).

ASP.NET does support one HTML5 validation feature: the enhanced *type* attribute, which allows you to create text boxes that are intended for specific types of data (such as e-mail addresses or numeric values). You can set the type attribute through the TextBox.TextMode property. Here's an example:

```
<asp:TextBox id="txtAge" runat="server" TextMode="Number" />
```

However, even this feature doesn't integrate well with ASP.NET. If you're creating a form that doesn't *need* validation, and doesn't use any ASP.NET validators, this approach is perfectly reasonable. And in some browsers, it will give the user additional editing conveniences, such as a numeric-only keypad on a tablet computer (for numeric data types), or a calendar-style date-picker control (for date data types). However, if you combine this approach with ASP.NET validators, you can end up with a mishmash of error messages. For example, see Figure 9-1, where ASP.NET validation gives the message *The email is missing the @ symbol*, and Google Chrome complains, *Please enter an email address*.

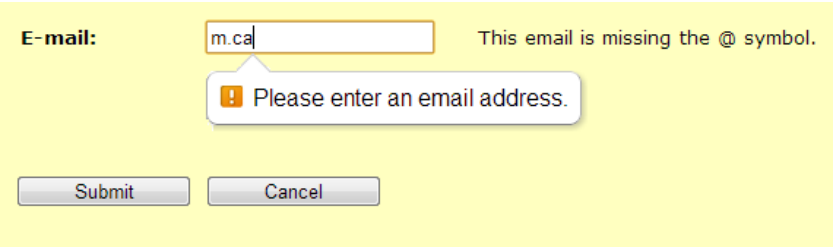


***Figure 9-1.*** *HTML5 validation and ASP.NET validation—an awkward match*

Here's the bottom line: For the vast majority of ASP.NET web forms, the best solution is to use ASP.NET validators. Sometimes a tried-and-true solution is better than living on the bleeding edge.

# Using the Validation Controls

The validation controls are found in the System.Web.UI.WebControls namespace and inherit from the BaseValidator class. This class defines the basic functionality for a validation control. Table 9-2 describes its key properties.

***Table 9-2.*** *Properties of the BaseValidator Class*

| Property | Description |
|---|---|
| ControlToValidate | Identifies the control that this validator will check. Each validator can verify the value in one input control. However, it's perfectly reasonable to "stack" validators—in other words, attach several validators to one input control to perform more than one type of error checking. |
| ErrorMessage and ForeColor | If validation fails, the validator control can display a text message (set by the ErrorMessage property). By changing the ForeColor, you can make this message stand out in angry red lettering. |
| Display | Allows you to configure whether this error message will be inserted into the page dynamically when it's needed (Dynamic) or whether an appropriate space will be reserved for the message (Static). Dynamic is useful when you're placing several validators next to each other. That way, the space will expand to fit the currently active error indicators, and you won't be left with any unseemly whitespace. Static is useful when the validator is in a table and you don't want the width of the cell to collapse when no message is displayed. Finally, you can also choose None to hide the error message altogether. |
| IsValid | After validation is performed, this returns true or false depending on whether it succeeded or failed. Generally, you'll check the state of the entire page by looking at its IsValid property instead to find out if all the validation controls succeeded. |
| Enabled | When set to false, automatic validation will not be performed for this control when the page is submitted. |
| EnableClientScript | If set to true, ASP.NET will add JavaScript and DHTML code to allow client-side validation on browsers that support it. |

When using a validation control, the only properties you need to implement are ControlToValidate and ErrorMessage. In addition, you may need to implement the properties that are used for your specific validator. Table 9-3 outlines these properties.

***Table 9-3.*** *Validator-Specific Properties*

| Validator Control | Added Members |
|---|---|
| RequiredFieldValidator | None required |
| RangeValidator | MaximumValue, MinimumValue, Type |
| CompareValidator | ControlToCompare, Operator, Type, ValueToCompare |
| RegularExpressionValidator | ValidationExpression |
| CustomValidator | ClientValidationFunction, ValidateEmptyText, ServerValidate event |

Later in this chapter (in the "A Validated Customer Form" section), you'll see a customer form example that demonstrates each type of validation.

## A Simple Validation Example

To understand how validation works, you can create a simple web page. This test uses a single Button web control, two TextBox controls, and a RangeValidator control that validates the first text box. If validation fails, the RangeValidator control displays an error message, so you should place this control immediately next to the TextBox it's validating. The second text box does not use any validation.

Figure 9-2 shows the appearance of the page after a failed validation attempt.

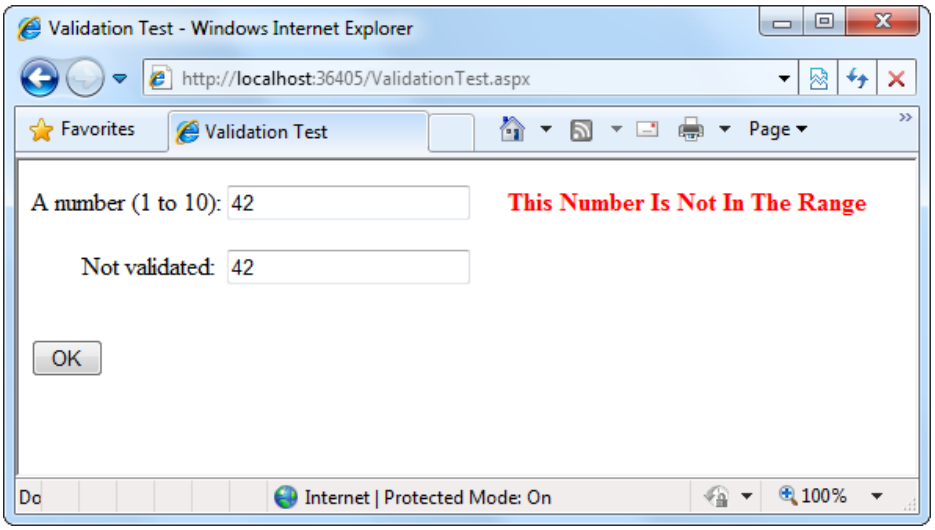


***Figure 9-2.*** *Failed validation*

In addition, place a Label control at the bottom of the form. This label will report when the page has been posted back and the event-handling code has executed. Disable its EnableViewState property to ensure that it will be cleared every time the page is posted back.

The markup for this page defines a RangeValidator control, sets the error message, identifies the control that will be validated, and requires an integer from 1 to 10. These properties are set in the .aspx file, but they could also be configured in the event handler for the Page.Load event. The Button automatically has its CauseValidation property set to true, because this is the default.

```
A number (1 to 10):
<asp:TextBox id="txtValidated" runat="server" />
<asp:RangeValidator id="RangeValidator" runat="server"
  ErrorMessage="This Number Is Not In The Range"
  ControlToValidate="txtValidated"
  MaximumValue="10" MinimumValue="1"
  ForeColor="Red" Font-Bold="true"
  Type="Integer" />
<br /><br />
```

```
Not validated:
<asp:TextBox id="txtNotValidated" runat="server" /><br /><br />
<asp:Button id="cmdOK" runat="server" Text="OK" OnClick="cmdOK_Click" />
<br /><br />
<asp:Label id="lblMessage" runat="server"
  EnableViewState="False" />
```

Finally, here is the code that responds to the button click:

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

If you're testing this web page in a modern browser, you'll notice an interesting trick. When you first open the page, the error message is hidden. But if you type an invalid number (remember, validation will succeed for an empty value) and press the Tab key to move to the second text box, an error message will appear automatically next to the offending control. This is because ASP.NET adds a special JavaScript function that detects when the focus changes. The actual implementation of this JavaScript code is somewhat complicated, but ASP.NET handles all the details for you automatically. As a result, if you try to click the OK button with an invalid value in txtValidated, your actions will be ignored, and the page won't be posted back.

Not all browsers will support client-side validation. To see what will happen on a down-level browser, set the RangeValidator.EnableClientScript property to false, and rerun the page. Now error messages won't appear dynamically as you change focus. However, when you click the OK button, the page will be returned from the server with the appropriate error message displayed next to the invalid control.

The potential problem in this scenario is that the click event-handling code will still execute, even though the page is invalid. To correct this problem and ensure that your page behaves the same on modern and older browsers, you must specifically abort the event code if validation hasn't been performed successfully.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    // Abort the event if the control isn't valid.
    if (!RangeValidator.IsValid) return;

    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

This code solves the current problem, but it isn't much help if the page contains multiple validation controls. Fortunately, every web form provides its own IsValid property. This property will be false if *any* validation control has failed. It will be true if all the validation controls completed successfully. If validation was not performed (for example, if the validation controls are disabled or if the button has CausesValidation set to false), you'll get an HttpException when you attempt to read the IsValid property.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    // Abort the event if any control on the page is invalid.
    if (!Page.IsValid) return;

    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Remember, client-side validation is just nice frosting on top of your application. Server-side validation will always be performed, ensuring that crafty users can't "spoof" pages.

## Other Display Options

In some cases, you might have already created a carefully designed form that combines multiple input fields. Perhaps you want to add validation to this page, but you can't reformat the layout to accommodate all the error messages for all the validation controls. In this case, you can save some work by using the ValidationSummary control.

To try this, set the Display property of the RangeValidator control to None. This ensures that the error message will never be displayed. However, validation will still be performed and the user will still be prevented from successfully clicking the OK button if some invalid information exists on the page.

Next, add the ValidationSummary in a suitable location (such as the bottom of the page):

```
<asp:ValidationSummary id="Errors" runat="server" ForeColor="Red" />
```

When you run the page, you won't see any dynamic messages as you enter invalid information and tab to a new field. However, when you click the OK button, the ValidationSummary will appear with a list of all error messages, as shown in Figure 9-3. In this case, it retrieves one error message (from the RangeValidator control). However, if you had a dozen validators, it would retrieve all their error messages and create a list.

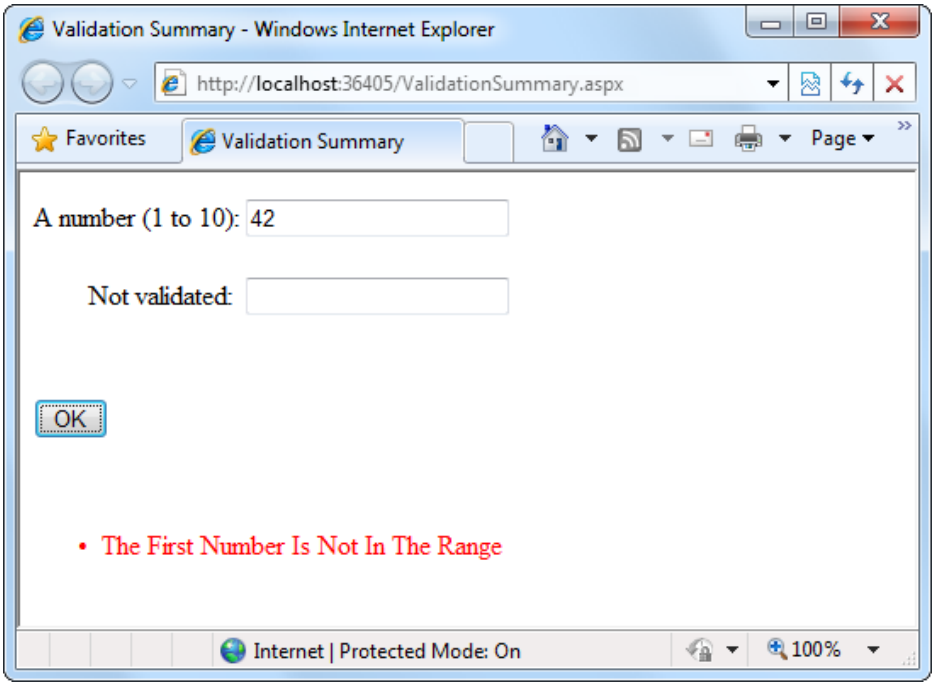


***Figure 9-3.*** *The validation summary*

When the ValidationSummary displays the list of errors, it automatically retrieves the value of the ErrorMessage property from each validator. In some cases, you'll want to display a full message in the summary and some sort of visual indicator next to the offending control. For example, many websites use an error icon or an asterisk to highlight text boxes with invalid input. You can use this technique with the help of the Text property of the validators. Ordinarily, Text is left empty, and the validator doesn't show any content in the web page. However, if you set both Text and ErrorMessage, the ErrorMessage value will be used for the summary while the

Text value is displayed in the validator. (Of course, you'll need to make sure you aren't also setting the Display property of your validator to None, which hides the validator—and its content—completely.)

Here's an example of a validator that includes a detailed error message (which will appear in the ValidationSummary) and an asterisk indicator (which will appear in the validator, next to the control that has the problem):

```
<asp:RangeValidator id="RangeValidator" runat="server"
  Text="*" ErrorMessage="The First Number Is Not In The Range"
  ControlToValidate="txtValidated"
  MaximumValue="10" MinimumValue="1" Type="Integer" />
```

If you have a lot of text, you may prefer to nest it inside the <asp:RangeValidator> element. For example, you can rewrite the markup shown earlier with this:

```
<asp:RangeValidator id="RangeValidator" runat="server"
  ControlToValidate="txtValidated"  MaximumValue="10" MinimumValue="1"
  ErrorMessage="The First Number Is Not In The Range"
  Type="Integer"><b>*** Error</b></asp:RangeValidator>
```

Here, ASP.NET automatically extracts the HTML inside the <asp:RangeValidator> element and uses it to set the RangeValidator.Text property.

You can even get a bit fancier by replacing the plain asterisk with a snippet of more-interesting HTML. Here's an example that uses the <img> tag to add a small error icon image when validation fails:

```
<asp:RangeValidator id="RangeValidator" runat="server">
  <img src="ErrorIcon.gif" alt="Error">
</asp:RangeValidator>
```

Figure 9-4 shows this validator in action.

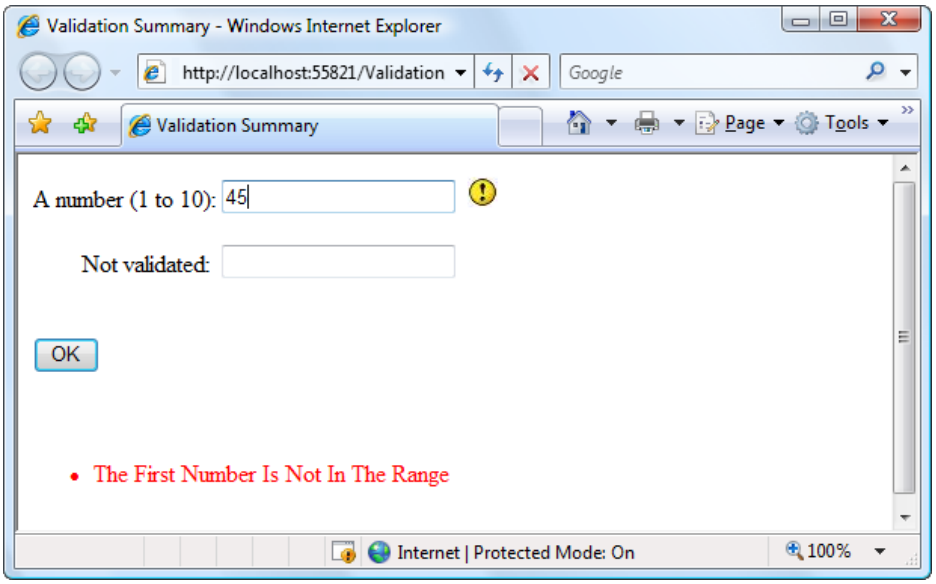


***Figure 9-4.*** *A validation summary and an error indicator*

The ValidationSummary control provides some useful properties you can use to fine-tune the error display. You can set the HeaderText property to display a special title at the top of the list (such as *Your page contains the following errors:*). You can also change the ForeColor and choose a DisplayMode. The possible modes are BulletList (the default), List, and SingleParagraph.

Finally, you can choose to have the validation summary displayed in a pop-up dialog box instead of on the page (see Figure 9-4). This approach has the advantage of leaving the user interface of the page untouched, but it also forces the user to dismiss the error messages by closing the window before being able to modify the input controls. If users will need to refer to these messages while they fix the page, the inline display is better.

To show the summary in a dialog box, set the ShowMessageBox property of the ValidationSummary to true. Keep in mind that unless you set the ShowSummary property to false, you'll see both the message box and the in-page summary (as in Figure 9-5).

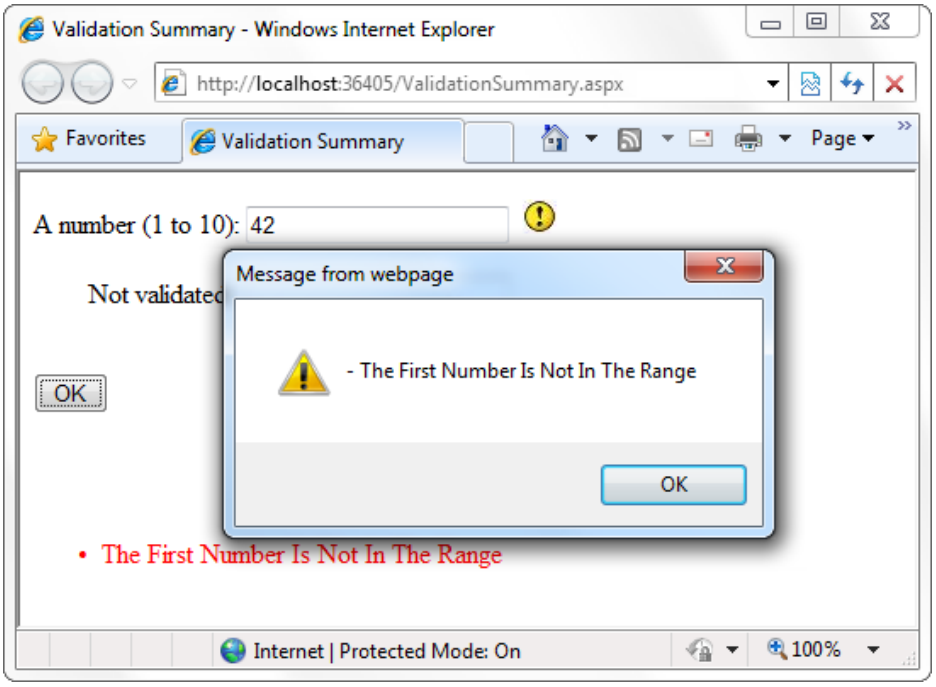


***Figure 9-5.*** *A message box summary*

## Manual Validation

Your final option is to disable validation and perform the work on your own, with the help of the validation controls. This allows you to take other information into consideration or create a specialized error message that involves other controls (such as images or buttons).

You can create manual validation in one of three ways:

- Use your own code to verify values. In this case, you won't use any of the ASP.NET validation controls.
- Disable the EnableClientScript property for each validation control. This allows an invalid page to be submitted, after which you can decide what to do with it depending on the problems that may exist.

- Add a button with CausesValidation set to false. When this button is clicked, manually validate the page by calling the Page.Validate() method. Then examine the IsValid property and decide what to do.

The next example uses the second approach. After the page is submitted, it examines all the validation controls on the page by looping through the Page.Validators collection. Every time it finds a control that hasn't validated successfully, it retrieves the invalid value from the input control and adds it to a string. At the end of this routine, it displays a message that describes which values were incorrect, as shown in Figure 9-6.

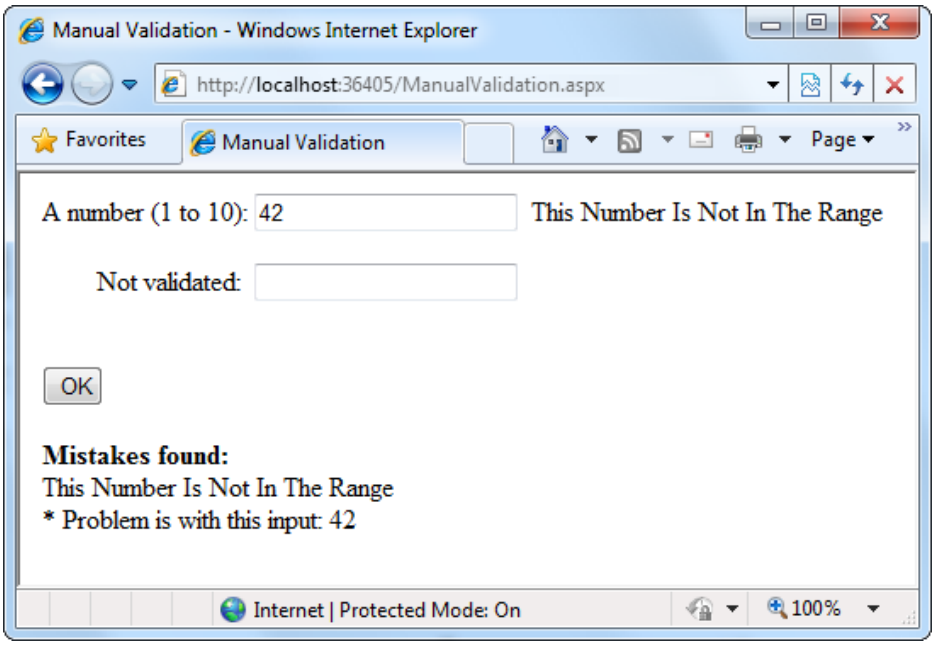


***Figure 9-6.*** *Manual validation*

This technique adds a feature that wouldn't be available with automatic validation, which uses the ErrorMessage property. In that case, it isn't possible to include the actual incorrect values in the message.

To try this example, set the EnableCientScript property of each validator to false. Then you can use the code in this event handler to check for invalid values.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    string errorMessage = "<b>Mistakes found:</b><br />";

    // Search through the validation controls.
    foreach (BaseValidator ctrl in this.Validators)
    {
        if (!ctrl.IsValid)
        {
            errorMessage += ctrl.ErrorMessage + "<br />";
```

```
            // Find the corresponding input control, and change the
            // generic Control variable into a TextBox variable.
            // This allows access to the Text property.
            TextBox ctrlInput =
              (TextBox)this.FindControl(ctrl.ControlToValidate);
            errorMessage += " * Problem is with this input: ";
            errorMessage += ctrlInput.Text + "<br />";
        }
    }
    lblMessage.Text = errorMessage;
}
```

This example uses an advanced technique: the Page.FindControl() method. It's required because the ControlToValidate property of each validator simply provides a string with the name of a control, not a reference to the actual control object. To find the control that matches this name (and retrieve its Text property), you need to use the FindControl() method. After retrieving the matching text box, the code can perform other tasks such as clearing the current value, tweaking a property, or even changing the text box color. Note that the FindControl() method returns a generic Control reference, because you might search any type of control. To access all the properties of your control, you need to cast it to the appropriate type (such as TextBox in this example).

---

■ **Tip** In this example, the code finds each validator and reads the ErrorMessage property. However, you can also set the ErrorMessage property at this time, which allows you to create customized error messages that incorporate information about the invalid values in their text.

---

## Validation with Regular Expressions

One of ASP.NET's most powerful validation controls is the RegularExpressionValidator, which validates text by determining whether it matches a specific pattern.

For example, e-mail addresses, phone numbers, and file names are all examples of text that has specific constraints. A phone number must be a set number of digits, an e-mail address must include exactly one @ character (with text on either side), and a file name can't include certain special characters such as \ and ?. One way to define patterns like these is with *regular expressions*.

Regular expressions have appeared in countless other languages and gained popularity as an extremely powerful way to work with strings. In fact, Visual Studio even allows programmers to perform a search-and-replace operation in their code using a regular expression (which may represent a new height of computer geekdom). Regular expressions can almost be considered an entire language of their own. How to master all the ways you can use regular expressions—including pattern matching, back references, and named groups—could occupy an entire book (and several books are dedicated to just that subject). Fortunately, you can understand the basics of regular expressions without nearly that much work.

### Using Literals and Metacharacters

All regular expressions consist of two kinds of characters: literals and metacharacters. *Literals* are not unlike the string literals you type in code. They represent a specific defined character. For example, if you search for the string literal `"a"`, you'll find the character *a* and nothing else.

*Metacharacters* provide the true secret to unlocking the full power of regular expressions. You're probably already familiar with two metacharacters from the DOS world (? and *). Consider the command-line expression shown here:

```
Del *.*
```

The expression *.* contains one literal (the period) and two metacharacters (the asterisks). This translates as "delete every file that starts with any number of characters and ends with an extension of any number of characters (or has no extension at all)." Because all files in DOS implicitly have extensions, this has the well-documented effect of deleting everything in the current directory.

Another DOS metacharacter is the question mark, which means "any single character." For example, the following statement deletes any file named *hello* that has an extension of exactly one character.

```
Del hello.?
```

The regular expression language provides many flexible metacharacters—far more than the DOS command line. For example, \s represents any whitespace character (such as a space or tab). \d represents any digit. Thus, the following expression would match any string that started with the numbers 333, followed by a single whitespace character and any three numbers. Valid matches would include 333 333 and 333 945 but not 334 333 or 3334 945.

```
333\s\d\d\d
```

One aspect that can make regular expressions less readable is that they use special metacharacters that are more than one character long. In the previous example, \s represents a single character, as does \d, even though they both occupy two characters in the expression.

You can use the plus (+) sign to represent a repeated character. For example, 5+7 means "one or more occurrences of the character 5, followed by a single 7." The number 57 would match, as would 555557. You can also use parentheses to group a subexpression. For example, (52)+7 would match any string that started with a sequence of 52. Matches would include 527, 52527, 5252527, and so on.

You can also delimit a range of characters by using square brackets. [a-f] would match any single character from *a* to *f* (lowercase only). The following expression would match any word that starts with a letter from *a* to *f*, contains one or more "word" characters (letters), and ends with *ing*—possible matches include *acting* and *developing*.

```
[a-f]\w+ing
```

The following is a more useful regular expression that can match any e-mail address by verifying that it contains the @ symbol. The dot is a metacharacter used to indicate any character except a newline. However, some invalid e-mail addresses would still be allowed, including those that contain spaces and those that don't include a dot (.). You'll see a better example a little later in the customer form example.

```
.+@.+
```

## Finding a Regular Expression

Clearly, picking the perfect regular expression may require some testing. In fact, numerous reference materials (on the Internet and in paper form) include useful regular expressions for validating common values such as postal codes. To experiment, you can use the simple RegularExpressionTest page included with the online samples, which is shown in Figure 9-7. It allows you to set a regular expression that will be used to validate a control. Then you can type in some sample values and see whether the regular expression validator succeeds or fails.

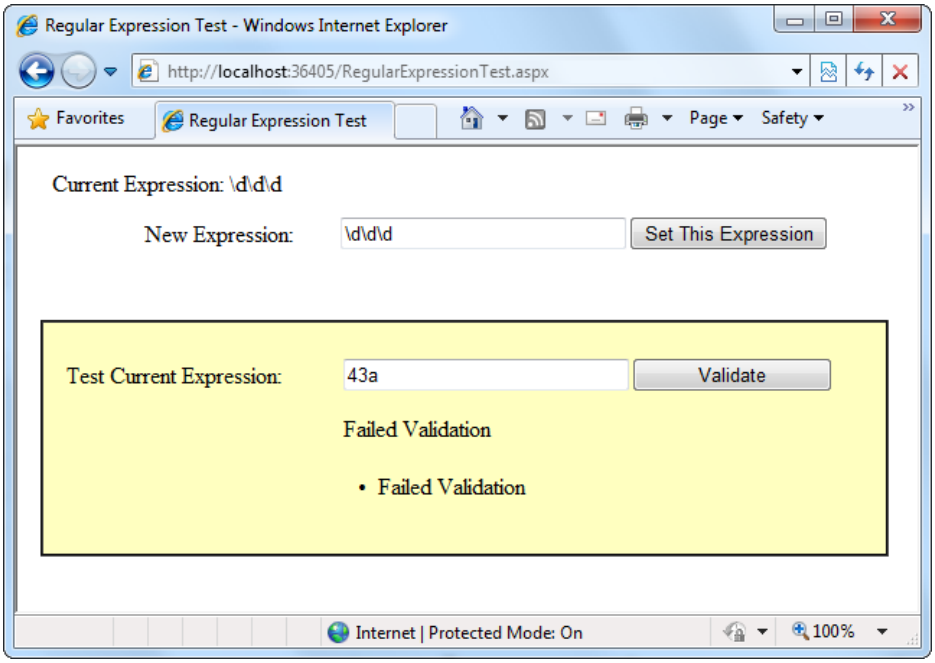


***Figure 9-7.*** *A regular expression test page*

The code is quite simple. The Set This Expression button assigns a new regular expression to the RegularExpressionValidator control (using whatever text you have typed). The Validate button simply triggers a postback, which causes ASP.NET to perform validation automatically. If an error message appears, validation has failed. Otherwise, it's successful.

```
public partial class RegularExpressionTest : System.Web.UI.Page
{
    protected void cmdSetExpression_Click(Object sender, EventArgs e)
    {
        TestValidator.ValidationExpression = txtExpression.Text;
        lblExpression.Text = "Current Expression: ";
        lblExpression.Text += txtExpression.Text;
    }
}
```

Table 9-4 shows some of the fundamental regular expression building blocks. If you need to match a literal character with the same name as a special character, you generally precede it with a \ character. For example, *hello* matches *hello* in a string, because the special asterisk (*) character is preceded by a slash (\).

***Table 9-4.*** *Regular Expression Characters*

| Character | Description |
|---|---|
| * | Zero or more occurrences of the previous character or subexpression. For example, 7*8 matches 7778 or just 8. |
| + | One or more occurrences of the previous character or subexpression. For example, 7+8 matches 7778 but not 8. |
| ( ) | Groups a subexpression that will be treated as a single element. For example, (78)+ matches 78 and 787878. |
| {m,n} | The previous character (or subexpression) can occur from *m* to *n* times. For example, A{1,3} matches A, AA, or AAA. |
| \| | Either of two matches. For example, 8\|6 matches 8 or 6. |
| [ ] | Matches one character in a range of valid characters. For example, [A-C] matches A, B, or C. |
| [^ ] | Matches a character that isn't in the given range. For example, [^A-B] matches any character except A and B. |
| . | Any character except a newline. For example, .here matches where and there. |
| \s | Any whitespace character (such as a tab or space). |
| \S | Any nonwhitespace character. |
| \d | Any digit character. |
| \D | Any character that isn't a digit. |
| \w | Any "word" character (letter, number, or underscore). |
| \W | Any character that isn't a "word" character (letter, number, or underscore). |

Table 9-5 shows a few common (and useful) regular expressions.

***Table 9-5.*** *Commonly Used Regular Expressions*

| Content | Regular Expression | Description |
|---|---|---|
| E-mail address* | \S+@\S+\.\S+ | Check for an at (@) sign and dot (.) and allow nonwhitespace characters only. |
| Password | \w+ | Any sequence of one or more word characters (letter, space, or underscore). |
| Specific-length password | \w{4,10} | A password that must be at least four characters long but no longer than ten characters. |
| Advanced password | [a-zA-Z]\w{3,9} | As with the specific-length password, this regular expression will allow four to ten total characters. The twist is that the first character must fall in the range of a–z or A–Z (that is to say, it must start with a nonaccented ordinary letter). |

(*continued*)

***Table 9-5.*** (*continued*)

| Content | Regular Expression | Description |
|---|---|---|
| Another advanced password | [a-zA-Z]\w*\d+\w* | This password starts with a letter character, followed by zero or more word characters, one or more digits, and then zero or more word characters. In short, it forces a password to contain one or more numbers somewhere inside it. You could use a similar pattern to require two numbers or any other special character. |
| Limited-length field | \S{4,10} | Like the password example, this allows four to ten characters, but it allows special characters (asterisks, ampersands, and so on). |
| US Social Security number | \d{3}-\d{2}-\d{4} | A sequence of three, two, and then four digits, with each group separated by a dash. You could use a similar pattern when requiring a phone number. |

** You have many ways to validate e-mail addresses with regular expressions of varying complexity. See www.4guysfromrolla.com/webtech/validateemail.shtml for a discussion of the subject and numerous examples.*

Some logic is much more difficult to model in a regular expression. An example is the Luhn algorithm, which verifies credit card numbers by first doubling every second digit, and then adding these doubled digits together, and finally dividing the sum by ten. The number is valid (although not necessarily connected to a real account) if there is no remainder after dividing the sum. To use the Luhn algorithm, you need a CustomValidator control that runs this logic on the supplied value. (You can find a detailed description of the Luhn algorithm at http://en.wikipedia.org/wiki/Luhn_formula.)

## A Validated Customer Form

To bring together these various topics, you'll now see a full-fledged web form that combines a variety of pieces of information that might be needed to add a user record (for example, an e-commerce site shopper or a content site subscriber). Figure 9-8 shows this form.

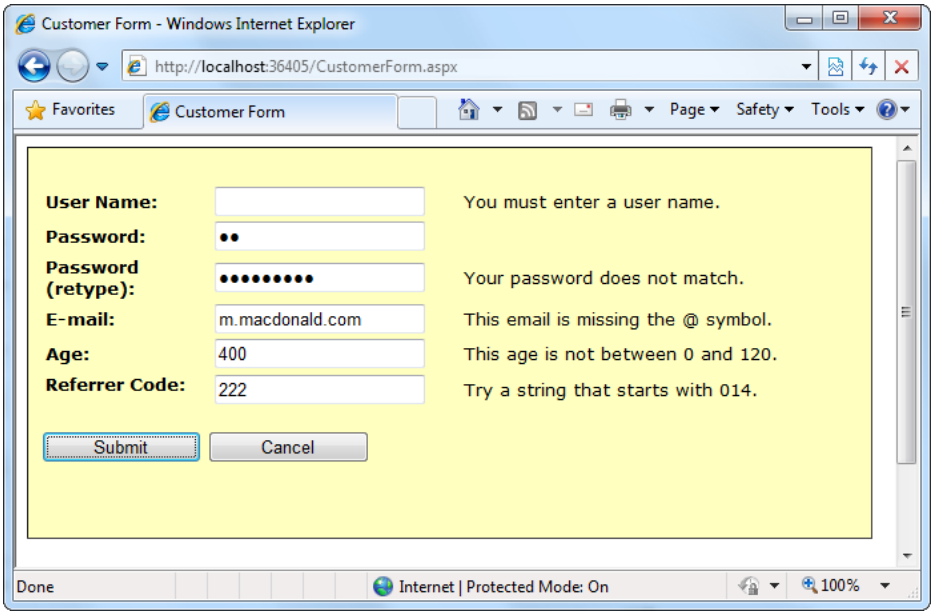


***Figure 9-8.*** *A sample customer form*

Several types of validation are taking place on the customer form:

- Three RequiredFieldValidator controls make sure the user enters a username, a password, and a password confirmation.
- A CompareValidator ensures that the two versions of the masked password match.
- A RegularExpressionValidator checks that the e-mail address contains an at (@) symbol.
- A RangeValidator ensures the age is a number from 0 to 120.
- A CustomValidator performs a special validation on the server of a "referrer code." This code verifies that the first three characters make up a number that is divisible by 7.

The tags for the validator controls are as follows:

```
<asp:RequiredFieldValidator id="vldUserName" runat="server"
     ErrorMessage="You must enter a user name."
     ControlToValidate="txtUserName" />

<asp:RequiredFieldValidator id="vldPassword" runat="server"
     ErrorMessage="You must enter a password."
     ControlToValidate="txtPassword" />

<asp:CompareValidator id="vldRetype" runat="server"
     ErrorMessage="Your password does not match."
     ControlToCompare="txtPassword" ControlToValidate="txtRetype" />

<asp:RequiredFieldValidator id="vldRetypeRequired" runat="server"
     ErrorMessage="You must confirm your password."
     ControlToValidate="txtRetype" />
```

```
<asp:RegularExpressionValidator id="vldEmail" runat="server"
     ErrorMessage="This email is missing the @ symbol."
     ValidationExpression=".+@.+" ControlToValidate="txtEmail" />

<asp:RangeValidator id="vldAge" runat="server"
     ErrorMessage="This age is not between 0 and 120." Type="Integer"
     MinimumValue="0" MaximumValue="120"
     ControlToValidate="txtAge" />

<asp:CustomValidator id="vldCode" runat="server"
     ErrorMessage="Try a string that starts with 014."
     ValidateEmptyText="False"
     OnServerValidate="vldCode_ServerValidate"
     ControlToValidate="txtCode" />
```

The form provides two validation buttons—one that requires validation and one that allows the user to cancel the task gracefully:

```
<asp:Button id="cmdSubmit" runat="server"
  OnClick="cmdSubmit_Click" Text="Submit"></asp:Button>
<asp:Button id="cmdCancel" runat="server"
  CausesValidation="False" OnClick="cmdCancel_Click" Text="Cancel">
</asp:Button>
```

Here's the event-handling code for the buttons:

```
protected void cmdSubmit_Click(Object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        lblMessage.Text = "This is a valid form.";
    }
}

protected void cmdCancel_Click(Object sender, EventArgs e)
{
    lblMessage.Text = "No attempt was made to validate this form.";
}
```

The only form-level code that is required for validation is the custom validation code. The validation takes place in the event handler for the CustomValidator.ServerValidate event. This method receives the value it needs to validate (e.Value) and sets the result of the validation to true or false (e.IsValid).

```
protected void vldCode_ServerValidate(Object source, ServerValidateEventArgs e)
{
    try
    {
        // Check whether the first three digits are divisible by seven.
        int val = Int32.Parse(e.Value.Substring(0, 3));
        if (val % 7 == 0)
        {
            e.IsValid = true;
        }
        else
```

```
        {
            e.IsValid = false;
        }
    }
    catch
    {
        // An error occurred in the conversion.
        // The value is not valid.
        e.IsValid = false;
    }
}
```

This example also introduces one new detail: error handling. This error-handling code ensures that potential problems are caught and dealt with appropriately. Without error handling, your code may fail, leaving the user with nothing more than a cryptic error page. The reason this example requires error-handling code is that it performs two steps that aren't guaranteed to succeed. First, the Int32.Parse() method attempts to convert the data in the text box to an integer. An error will occur during this step if the information in the text box is nonnumeric (for example, if the user entered the characters 4 G). Similarly, the String.Substring() method, which extracts the first three characters, will fail if fewer than three characters appear in the text box. To guard against these problems, you can specifically check these details before you attempt to use the Parse() and Substring() methods, or you can use error handling to respond to problems after they occur. (Another option is to use the TryParse() method, which returns a Boolean value that tells you whether the conversion succeeded. You saw TryParse() at work in Chapter 5.)

---

■ **Tip** In some cases, you might be able to replace custom validation with a particularly ingenious use of a regular expression. However, you can use custom validation to ensure that validation code is executed only at the server. That prevents users from seeing your regular expression template (in the rendered JavaScript code) and using it to determine how they can outwit your validation routine. A user who does not have a valid credit card number, for example, could create a false one more easily if that user knew the algorithm you use to test credit card numbers.

---

The CustomValidator has another quirk. You'll notice that your custom server-side validation isn't performed until the page is posted back. This means that if you enable the client script code (the default), dynamic messages will appear, informing the user when the other values are incorrect— but they will not indicate any problem with the referral code until the page is posted back to the server.

This isn't really a problem, but if it troubles you, you can use the CustomValidator.ClientValidationFunction property. Add a client-side JavaScript function to the .aspx portion of the web page. Remember, you can't use client-side ASP.NET code, because C# and VB aren't recognized by the client browser.

Your JavaScript function will accept two parameters (in true .NET style), which identify the source of the event and the additional validation parameters. In fact, the client-side event is modeled on the .NET ServerValidate event. Just as you did in the ServerValidate event handler, in the client validation function, you retrieve the value to validate from the Value property of the event argument object. You then set the IsValid property to indicate whether validation succeeds or fails.

The following is the client-side equivalent for the code in the ServerValidate event handler. The JavaScript code resembles C# superficially.

```
<script type="text/javascript">
function MyCustomValidation(objSource, objArgs)
{
    // Get value.
    var number = objArgs.Value;
```

```
    // Check value and return result.
    number = number.substr(0, 3);
    if (number % 7 == 0)
    {
        objArgs.IsValid = true;
    }
    else
    {
        objArgs.IsValid = false;
    }
}
</script>
```

You can place this block of JavaScript code in the <head> section of your page. (Or, you can put in a separate file and reference that file by using a <script> tag in the <head> section, just as you would with any other JavaScript library.)

After you've added the validation script function, you must set the ClientValidationFunction property of the CustomValidator control to the name of the function. You can edit the CustomValidator tag by hand or use the Properties window in Visual Studio.

```
<asp:CustomValidator id="vldCode" runat="server"
     ErrorMessage="Try a string that starts with 014."
     ControlToValidate="txtCode"
     OnServerValidate="vldCode_ServerValidate"
     ClientValidationFunction="MyCustomValidation" />
```

ASP.NET will now call this function on your behalf when it's required.

---

■ **Tip** Even when you use client-side validation, you must still include the ServerValidate event handler, both to provide server-side validation for clients that don't support the required JavaScript and DHTML features and to prevent clients from circumventing your validation by modifying the HTML page they receive.

---

By default, custom validation isn't performed on empty values. However, you can change this behavior by setting the CustomValidator.ValidateEmptyText property to true. This is a useful approach if you create a more detailed JavaScript function (for example, one that updates with additional information) and want it to run when the text is cleared.

## Validation Groups

In more-complex pages, you might have several distinct groups of controls, possibly in separate panels. In these situations, you may want to perform validation separately. For example, you might create a form that includes a box with login controls and a box underneath it with the controls for registering a new user. Each box includes its own submit button, and depending on which button is clicked, you want to perform the validation just for that section of the page.

This scenario is possible thanks to a feature called *validation groups*. To create a validation group, you need to put the input controls, the validators, and the CausesValidation button controls into the same logical group. You do this by setting the ValidationGroup property of every control with the same descriptive string (such as "LoginGroup" or "NewUserGroup"). Every control that provides a CausesValidation property also includes the ValidationGroup property.

For example, the following page defines two validation groups, named Group1 and Group2. The controls for each group are placed into separate Panel controls.

```
<form id="form1" runat="server">
  <asp:Panel ID="Panel1" runat="server">
    <asp:TextBox ID="TextBox1" ValidationGroup="Group1" runat="server" />
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
     ErrorMessage="*Required" ValidationGroup="Group1"
     runat="server" ControlToValidate="TextBox1" />
    <asp:Button ID="Button1" Text="Validate Group1"
     ValidationGroup="Group1" runat="server" />
  </asp:Panel>
  <br />
  <asp:Panel ID="Panel2" runat="server">
    <asp:TextBox ID="TextBox2" ValidationGroup="Group2"
     runat="server" />
    <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
     ErrorMessage="*Required" ValidationGroup="Group2"
     ControlToValidate="TextBox2" runat="server" />
    <asp:Button ID="Button2" Text="Validate Group2"
     ValidationGroup="Group2" runat="server" />
  </asp:Panel>
</form>
```

If you click the button in the topmost panel, only the first text box is validated. If you click the button in the second panel, only the second text box is validated (as shown in Figure 9-9).

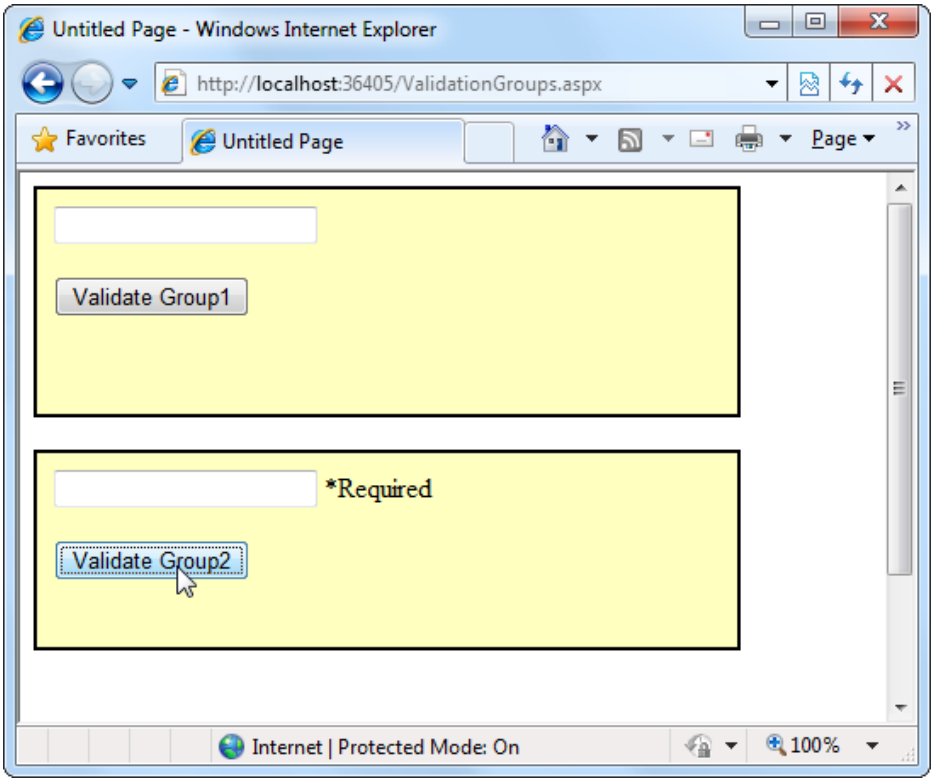


***Figure 9-9.*** *Grouping controls for validation*

What happens if you add a new button that doesn't specify any validation group? In this case, the button validates every control that isn't explicitly assigned to a named validation group. In the current example, no controls fit the requirement, so the page is posted back successfully and deemed to be valid.

If you want to make sure a control is always validated, regardless of the validation group of the button that's clicked, you'll need to create multiple validators for the control, one for each group (and one with no validation group).

# The Last Word

In this chapter, you learned how to use one of ASP.NET's most practical features: validation. You saw how ASP.NET combines server-side and client-side validation to ensure bulletproof security without sacrificing the usability of your web pages. You also looked at the types of validation provided by the various validation controls, and even brushed up on the powerful pattern-matching syntax used for regular expressions. Finally, you considered how to customize and extend the validation process to handle a few different scenarios.

CHAPTER 10

# Rich Controls

*Rich controls* are web controls that model complex user interface elements. Although no strict definition exists for what is and what isn't a rich control, the term commonly describes a web control that has an object model that's distinctly separate from the HTML it generates. A typical rich control can be programmed as a single object (and added to a web page with a single control tag) but renders itself using a complex sequence of HTML elements. Rich controls can also react to user actions (such as a mouse click on a specific region of the control) and raise more-meaningful events that your code can respond to on the web server. In other words, rich controls give you a way to create advanced user interfaces in your web pages without writing lines of convoluted HTML.

In this chapter, you'll take a look at several web controls that have no direct equivalent in the world of ordinary HTML. You'll start with the Calendar, which provides slick date-selection functionality. Next you'll consider the AdRotator, which gives you an easy way to insert a randomly selected image into a web page. Finally, you'll learn how to create sophisticated pages with multiple views by using two advanced controls: the MultiView and the Wizard. These controls allow you to pack a miniature application into a single page. Using them, you can handle a multistep task without redirecting the user from one page to another.

---

**Note** ASP.NET includes numerous rich controls that are discussed elsewhere in this book, including navigation controls, data-based list controls, and security controls. In this chapter, you'll focus on a few useful web controls that don't fit neatly into any of these categories. All of these controls appear in the Standard tab of the Visual Studio Toolbox.

---

## The Calendar

The *Calendar control* presents a miniature calendar that you can place in any web page. Like most rich controls, the Calendar can be programmed as a single object (and defined in a single simple tag), but it renders itself with dozens of lines of HTML output.

```
<asp:Calendar id="MyCalendar" runat="server" />
```

The Calendar control presents a single-month view, as shown in Figure 10-1. The user can navigate from month to month by using the navigational arrows, at which point the page is posted back and ASP.NET automatically provides a new page with the correct month values. You don't need to write any additional event-handling code to manage this process. When the user clicks a date, the date becomes highlighted in a gray box (by default). You can retrieve the selected day in your code as a DateTime object from the Calendar.SelectedDate property.

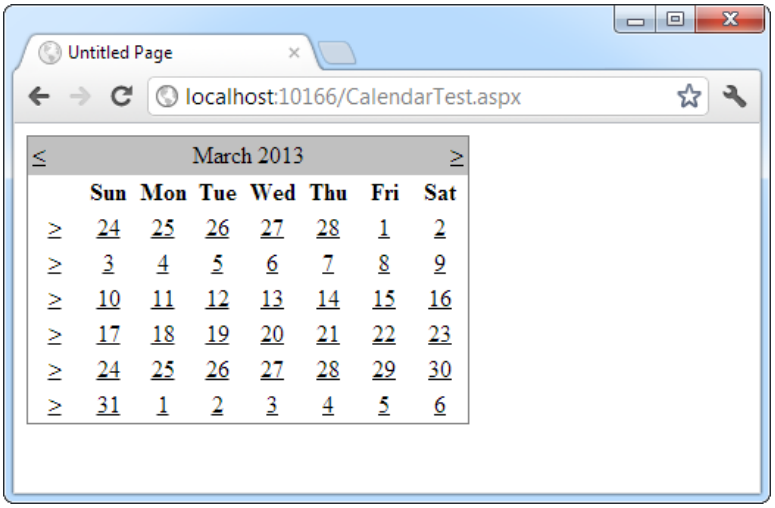


***Figure 10-1.*** *The default Calendar*

This basic set of features may provide everything you need in your application. Alternatively, you can configure different selection modes through the CalendarSelectionMode property. Depending on the value you choose, you can allow users to select days (Day), entire weeks (DayWeek), whole months (DayWeekMonth), or render the control as a static calendar that doesn't allow selection (None). The only fact you must remember is that if you allow month selection, the user can also select a single week or a day. Similarly, if you allow week selection, the user can also select a single day.

You may also want to set the Calendar.FirstDayOfWeek property to configure how a week is shown. (For example, set FirstDayOfWeek to the enumerated value Sunday, and weeks will be selected from Sunday to Saturday.)

When you allow multiple date selection, you need to examine the SelectedDates property, which provides a collection of all the selected dates. You can loop through this collection by using the foreach syntax. The following code demonstrates this technique:

```
lblDates.Text = "You selected these dates:<br />";
foreach (DateTime dt in MyCalendar.SelectedDates)
{
    lblDates.Text += dt.ToLongDateString() + "<br />";
}
```

Figure 10-2 shows the resulting page after this code has been executed.

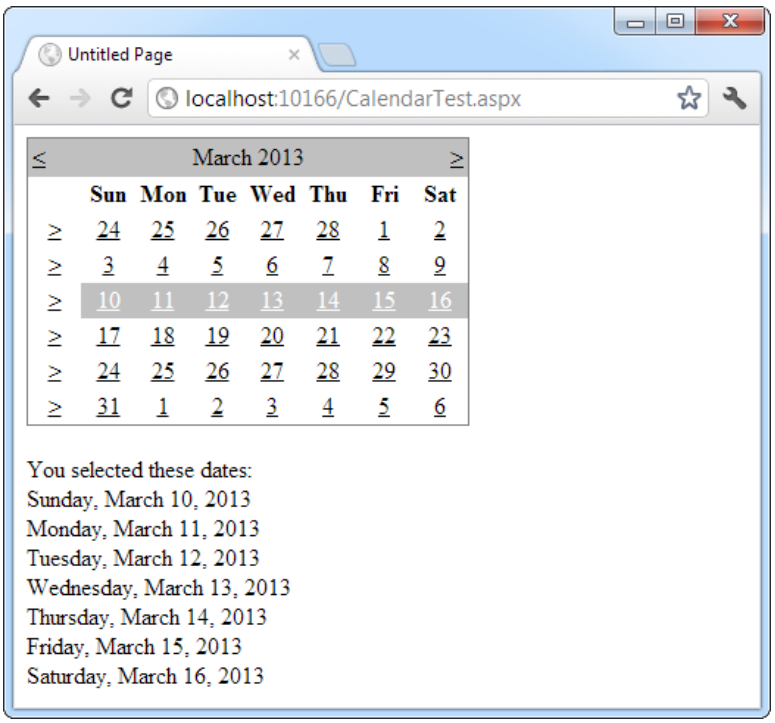


***Figure 10-2.*** *Selecting multiple dates*

## Formatting the Calendar

The Calendar control provides a whole host of formatting-related properties. You can set various parts of the calendar, such as the header, selector, and various day types, by using one of the style properties (for example, WeekendDayStyle). Each of these style properties references a full-featured TableItemStyle object that provides properties for coloring, border style, font, and alignment. Taken together, they allow you to modify almost any part of the calendar's appearance.

Table 10-1 lists the style properties that the Calendar control provides.

***Table 10-1.*** *Properties for Calendar Styles*

| Member | Description |
|---|---|
| DayHeaderStyle | The style for the section of the Calendar that displays the days of the week (as column headers). |
| DayStyle | The default style for the dates in the current month. |
| NextPrevStyle | The style for the navigation controls in the title section that move from month to month. |
| OtherMonthDayStyle | The style for the dates that aren't in the currently displayed month. These dates are used to "fill in" the calendar grid. For example, the first few cells in the topmost row may display the last few days from the previous month. |
| SelectedDayStyle | The style for the selected dates on the calendar. |
| SelectorStyle | The style for the week and month date selection controls. |
| TitleStyle | The style for the title section. |
| TodayDayStyle | The style for the date designated as today (represented by the TodaysDate property of the Calendar control). |
| WeekendDayStyle | The style for dates that fall on the weekend. |

You can adjust each style by using the Properties window. For a quick shortcut, you can set an entire related color scheme by using the Calendar's Auto Format feature. To do so, start by selecting the Calendar on the design surface of a web form. Then click the arrow icon that appears next to its top-right corner to show the Calendar's smart tag, and click the Auto Format link. You'll be presented with a list of predefined formats that set the style properties, as shown in Figure 10-3.

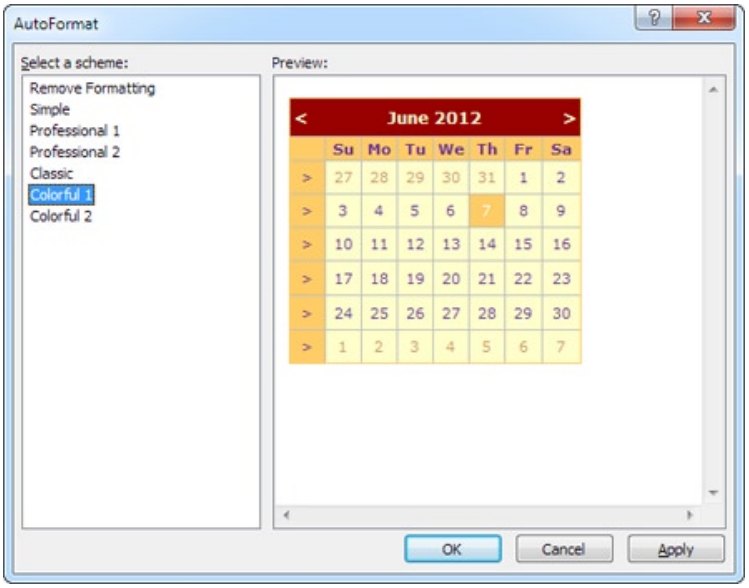


***Figure 10-3.*** *Calendar styles*

You can also use additional properties to hide some elements or configure the text they display. For example, properties that start with *Show* (such as ShowDayHeader, ShowTitle, and ShowGridLines) can be used to hide or show a specific visual element. Properties that end in *Text* (such as PrevMonthText, NextMonthText, and SelectWeekText) allow you to set the text that's shown in part of the calendar.

## Restricting Dates

In most situations where you need to use a calendar for selection, you don't want to allow the user to select any date in the calendar. For example, the user might be booking an appointment or choosing a delivery date—two services that are generally provided only on set days. The Calendar control makes it surprisingly easy to implement this logic. In fact, if you've worked with the date and time controls on the Windows platform, you'll quickly recognize that the ASP.NET versions are far superior.

The basic approach to restricting dates is to write an event handler for the Calendar.DayRender event. This event occurs when the Calendar control is about to create a month to display to the user. This event gives you the chance to examine the date that is being added to the current month (through the e.Day property) and decide whether it should be selectable or restricted.

The following code makes it impossible to select any weekend days or days in years later than 2013:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
    // Restrict dates after the year 2013 and those on the weekend.
    if (e.Day.IsWeekend || e.Day.Date.Year > 2013)
    {
        e.Day.IsSelectable = false;
    }
```

The e.Day object is an instance of the CalendarDay class, which provides various properties. Table 10-2 describes some of the most useful.

***Table 10-2.*** *CalendarDay Properties*

| Property | Description |
|---|---|
| Date | The DateTime object that represents this date. |
| IsWeekend | True if this date falls on a Saturday or Sunday. |
| IsToday | True if this value matches the Calendar.TodaysDate property, which is set to the current day by default. |
| IsOtherMonth | True if this date doesn't belong to the current month but is displayed to fill in the first or last row. For example, this might be the last day of the previous month or the next day of the following month. |
| IsSelectable | Allows you to configure whether the user can select this day. |

The DayRender event is extremely powerful. Besides allowing you to tailor what dates are selectable, it also allows you to configure the cell where the date is located through the e.Cell property. (The calendar is displayed using an HTML table.) For example, you could highlight an important date or even add information. Here's an example that highlights a single day—the fifth of May—by adding a new Label control in the table cell for that day:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
    // Check for May 5 in any year, and format it.
    if (e.Day.Date.Day == 5 && e.Day.Date.Month == 5)
    {
        e.Cell.BackColor = System.Drawing.Color.Yellow;

        // Add some static text to the cell.
        Label lbl = new Label();
        lbl.Text = "<br />My Birthday!";
        e.Cell.Controls.Add(lbl);
    }
}
```

Figure 10-4 shows the resulting calendar display.

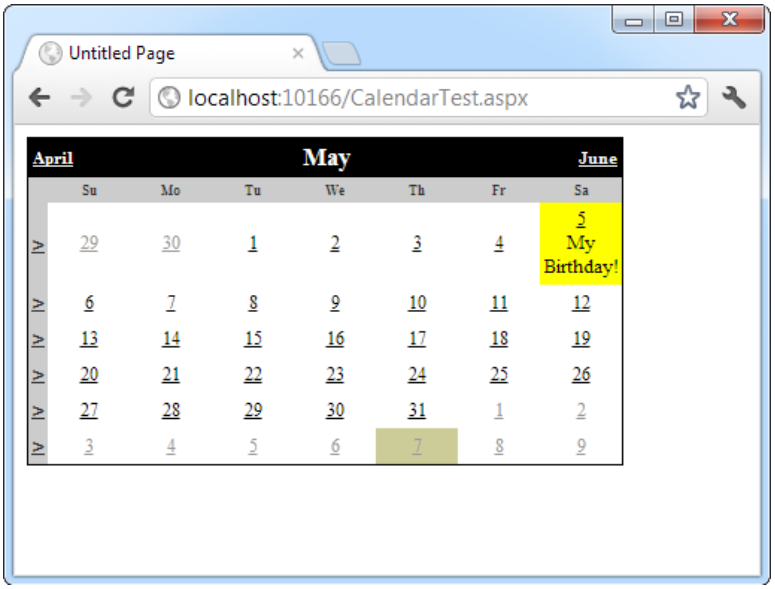


***Figure 10-4.*** *Highlighting a day*

The Calendar control provides two other useful events: SelectionChanged and VisibleMonthChanged. These occur immediately after the user selects a new day or browses to a new month (using the next month and previous month links). You can react to these events and update other portions of the web page to correspond to the current calendar month. For example, you could design a page that lets you schedule a meeting in two steps. First you choose the appropriate day. Then you choose one of the available times on that day.

The following code demonstrates this approach, using a different set of time values if a Monday is selected in the calendar than it does for other days:

```
protected void MyCalendar_SelectionChanged(Object source, EventArgs e)
{
    lstTimes.Items.Clear();

    switch (MyCalendar.SelectedDate.DayOfWeek)
    {
        case DayOfWeek.Monday:
            // Apply special Monday schedule.
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
        break;
        default:
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
            lstTimes.Items.Add("11:30");
            lstTimes.Items.Add("12:00");
            lstTimes.Items.Add("12:30");
            break;
    }
}
```

To try these features of the Calendar control, run the Appointment.aspx page from the online samples. This page provides a formatted Calendar control that restricts some dates, formats others specially, and updates a corresponding list control when the selection changes.

Table 10-3 gives you an at-a-glance look at almost all the members of the Calendar control class.

***Table 10-3.*** *Calendar Members*

| Member | Description |
|---|---|
| Caption and CaptionAlign | Gives you an easy way to add a title to the calendar. By default, the caption appears at the top of the title area, just above the month heading. However, you can control this to some extent with the CaptionAlign property. Use Left or Right to keep the caption at the top but move it to one side or the other, and use Bottom to place the caption under the calendar. |
| CellPadding | ASP.NET creates a date in a separate cell of an invisible table. CellPadding is the space, in pixels, between the border of each cell and its contents. |
| CellSpacing | The space, in pixels, between cells in the same table. |
| DayNameFormat | Determines how days are displayed in the calendar header. Valid values are Full (as in Sunday), FirstLetter (S), FirstTwoLetters (Su), and Short (Sun), which is the default. |
| FirstDayOfWeek | Determines which day is displayed in the first column of the calendar. The values are any day name from the FirstDayOfWeek enumeration (such as Sunday). By default, this is Sunday. |
| NextMonthText and PrevMonthText | Sets the text that the user clicks to move to the next or previous month. These navigation links appear at the top of the calendar and are the greater-than (>) and less-than (<) signs by default. This setting is applied only if NextPrevFormat is set to CustomText. |
| NextPrevFormat | Sets the text that the user clicks to move to the next or previous month. This can be FullMonth (for example, December), ShortMonth (Dec), or CustomText, in which case the NextMonthText and PrevMonthText properties are used. CustomText is the default. |
| SelectedDate and SelectedDates | Sets or gets the currently selected date as a DateTime object. You can specify this in the control tag in a format like this: `12:00:00 AM, 12/31/2010` (depending on your computer's regional settings). If you allow multiple date selection, the SelectedDates property will return a collection of DateTime objects, one for each selected date. You can use collection methods such as Add, Remove, and Clear to change the selection. |
| SelectionMode | Determines how many dates can be selected at once. The default is Day, which allows one date to be selected. Other options include DayWeek (a single date or an entire week) or DayWeekMonth (a single date, entire week, or entire month). You have no way to allow the user to select multiple noncontiguous dates. You also have no way to allow larger selections without also including smaller selections. (For example, if you allow full months to be selected, you must also allow week selection and individual day selection.) |
| SelectMonthText and SelectWeekText | The text shown for the link that allows the user to select an entire month or week. These properties don't apply if the SelectionMode is Day. |

(*continued*)

***Table 10-3.*** (*continued*)

| Member | Description |
|---|---|
| ShowDayHeader, ShowGridLines, ShowNextPrevMonth, and ShowTitle | These Boolean properties allow you to configure whether various parts of the calendar are shown, including the day titles, gridlines between every day, the previous/next month navigation links, and the title section. Note that hiding the title section also hides the next and previous month navigation controls. |
| TitleFormat | Configures how the month is displayed in the title area. Valid values include Month and MonthYear (the default). |
| TodaysDate | Sets which day should be recognized as the current date and formatted with the TodayDayStyle. This defaults to the current day on the web server. |
| VisibleDate | Gets or sets the date that specifies what month will be displayed in the calendar. This allows you to change the calendar display without modifying the current date selection. |
| DayRender event | Occurs once for each day that is created and added to the currently visible month before the page is rendered. This event gives you the opportunity to apply special formatting, add content, or restrict selection for an individual date cell. Keep in mind that days can appear in the calendar even when they don't fall in the current month, provided they fall close to the end of the previous month or close to the start of the following month. |
| SelectionChanged event | Occurs when the user selects a day, a week, or an entire month by clicking the date selector controls. |
| VisibleMonthChanged event | Occurs when the user clicks the next or previous month navigation controls to move to another month. |

# The AdRotator

The basic purpose of the *AdRotator* is to provide a graphic on a page that is chosen randomly from a group of possible images. In other words, every time the page is requested, an image is selected at random and displayed, which is the *rotation* indicated by the name AdRotator. One use of the AdRotator is to show banner-style advertisements on a page, but you can use it anytime you want to vary an image randomly.

Using ASP.NET, it wouldn't be too difficult to implement an AdRotator type of design on your own. You could react to the Page.Load event, generate a random number, and then use that number to choose from a list of predetermined image files. You could even store the list in the web.config file so that it can be easily modified separately as part of the application's configuration. Of course, if you wanted to enable several pages with a random image, you would either have to repeat the code or create your own custom control. The AdRotator provides these features for free.

## SHOULD YOU CHOOSE USE THE ADROTATOR FOR WEB ADVERTISING?

The name of the AdRotator control is slightly misleading. A better name might be RandomImage or RotatingImage. That's because there's absolutely no requirement for the AdRotator content to have anything to do with advertising.

In fact, these days the AdRotator is not the most common way to deal with website advertising, even in an ASP.NET application. You're more likely to use a block of pregenerated JavaScript that's supplied to you by an advertising service. For example, if you decide to make some extra money showing ads with Google AdSense (www.google.com/adsense), Google will provide you with a block of JavaScript code that fetches an appropriate ad (or a combination of ad links) for your page. The ads Google returns are random, but they are based on a variety of details, including the content it detects on your page and the amount of space you've allocated for advertising. Other advertising approaches are more sophisticated and use pop-up panels with Flash animation, among other tricks.

## The Advertisement File

The AdRotator stores its list of image files in an XML file. This file uses the format shown here:

```
<Advertisements>
  <Ad>
    <ImageUrl>prosetech.jpg</ImageUrl>
    <NavigateUrl>http://www.prosetech.com</NavigateUrl>
    <AlternateText>ProseTech Site</AlternateText>
    <Impressions>1</Impressions>
    <Keyword>Computer</Keyword>
  </Ad>
</Advertisements>
```

■ **Tip** An XML file is just a text file with specific tags. You can create an XML file by using nothing more than a text editor such as Notepad, but you can also use the Visual Studio text editor. Just choose Website ➤ Add New Item from the menu, and then choose XML File. It's up to you to fill in the right tags and content. You can place the advertisements file wherever you'd like—either in the main website folder or in a subfolder that you've created.

This example shows a single possible advertisement, which the AdRotator control picks at random from the list of advertisements. To add more advertisements, you would create multiple <Ad> elements and place them all inside the root <Advertisements> element:

```
<Advertisements>
  <Ad>
    <!-- First ad here. -->
  </Ad>

  <Ad>
    <!-- Second ad here. -->
  </Ad>
</Advertisements>
```

Each <Ad> element has a number of other important properties that configure the link, the image, and the frequency, as described in Table 10-4.

***Table 10-4.*** *Advertisement File Elements*

| Element | Description |
|---|---|
| ImageUrl | The image that will be displayed. This can be a relative link (a file in the current directory) or a fully qualified Internet URL. |
| NavigateUrl | The link that will be followed if the user clicks the banner. This can be a relative or fully qualified URL. |
| AlternateText | The text that will be displayed instead of the picture if it cannot be displayed. This text will also be used as a tooltip in some newer browsers. |
| Impressions | A number that sets how often an advertisement will appear. This number is relative to the numbers specified for other ads. For example, a banner with the value 10 will be shown twice as often (on average) as the banner with the value 5. |
| Keyword | A keyword that identifies a group of advertisements. You can use this for filtering. For example, you could create ten advertisements and give half of them the keyword Retail and the other half the keyword Computer. The web page can then choose to filter the possible advertisements to include only one of these groups. |

## The AdRotator Class

The actual AdRotator class provides a limited set of properties. You specify both the appropriate advertisement file in the AdvertisementFile property and the type of window that the link should follow (the Target window). The target can name a specific frame, or it can use one of the values defined in Table 10-5.

***Table 10-5.*** *Special Frame Targets*

| Target | Description |
|---|---|
| _blank | The link opens a new unframed window. |
| _parent | The link opens in the parent of the current frame. |
| _self | The link opens in the current frame. |
| _top | The link opens in the topmost frame of the current window (so the link appears in the full window). |

Optionally, you can set the KeywordFilter property so that the banner will be chosen from a specific keyword group. This is a fully configured AdRotator tag:

```
<asp:AdRotator ID="Ads" runat="server" AdvertisementFile="MainAds.xml"
     Target="_blank" KeywordFilter="Computer" />
```

Additionally, you can react to the AdRotator.AdCreated event. This occurs when the page is being created and an image is randomly chosen from the advertisements file. This event provides you with information about the image that you can use to customize the rest of your page. For example, you might display some related content or a link, as shown in Figure 10-5.

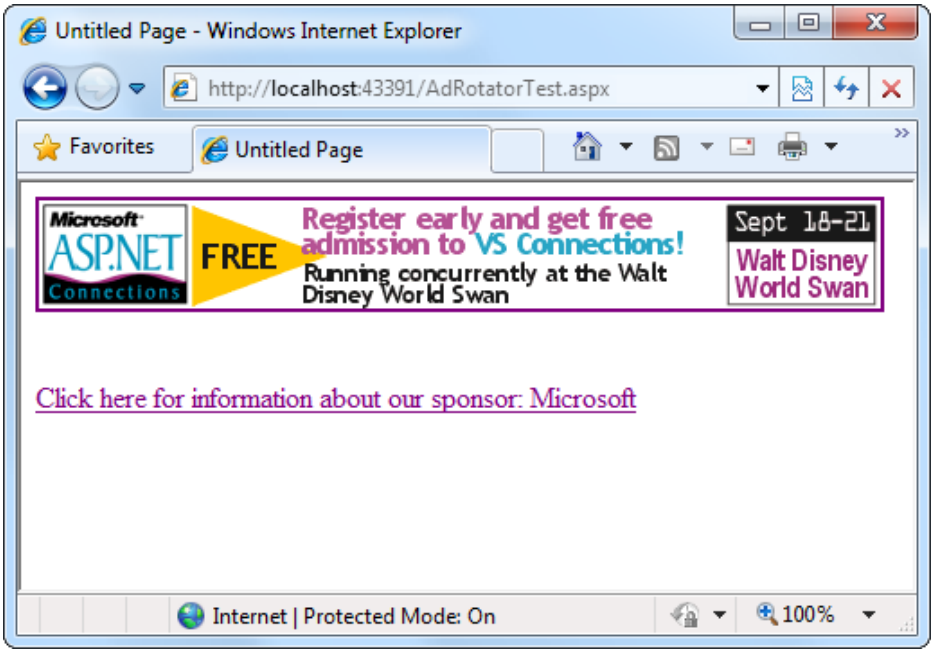


***Figure 10-5.*** *An AdRotator with synchronized content*

The event-handling code for this example simply configures a HyperLink control named lnkBanner based on the randomly selected advertisement:

```
protected void Ads_AdCreated(Object sender, AdCreatedEventArgs e)
{
    // Synchronize the Hyperlink control.
    lnkBanner.NavigateUrl = e.NavigateUrl;

    // Syncrhonize the text of the link.
    lnkBanner.Text = "Click here for information about our sponsor: ";
    lnkBanner.Text += e.AlternateText;
}
```

As you can see, rich controls such as the Calendar and AdRotator don't just add a sophisticated HTML output; they also include an event framework that allows you to take charge of the control's behavior and integrate it into your application.

# Pages with Multiple Views

In a typical website, you'll surf through many separate pages. For example, if you want to add an item to your shopping cart and take it to the checkout in an e-commerce site, you'll need to jump from one page to another. This design has its advantages—namely, it lets you carefully separate different tasks into different code files. It also presents some challenges; for example, you need to come up with a way to transfer information from one page to another (as you saw in Chapter 8).

However, in some cases it makes more sense to create a single page that can handle several different tasks. For example, you might want to provide several views of the same data (such as a grid-based view and a chart-based view) and allow the user to switch from one view to the other without leaving the page. Or, you might want to handle a small multistep task in one place (such as supplying user information for an account sign-up process). In these examples, you need a way to create dynamic pages that provide more than one possible view. Essentially, the page hides and shows different controls depending on which view you want to present.

The simplest way to understand this technique is to create a page with several Panel controls. Each panel can hold a group of ASP.NET controls. For example, imagine you're creating a simple three-step wizard. You'll start by adding three panels to your page, one for each step—say, panelStep1, panelStep2, and panelStep3. You can place the panels one after the other, because you'll show only one at a time. After you've added the panels, you can place the appropriate controls inside each panel. To start, the Visible property of each panel should be false, except for panelStep1, which appears the first time the user requests the page.

Here's an example that shows the way you can arrange your panels:

```
<asp:Panel ID="panelStep1" runat="server">...</asp:Panel>
<asp:Panel ID="panelStep2" Visible="False" runat="server">...</asp:Panel>
<asp:Panel ID="panelStep3" Visible="False" runat="server">...</asp:Panel>
```

---

■ **Note** When you set the Visible property of a control to false, the control won't appear in the page at runtime. Any controls inside an invisible panel are also hidden from sight, and they won't be present in the rendered HTML for the page. However, these controls will still appear in the Visual Studio design surface so that you can still select them and configure them.

---

Finally, you'll add one or more navigation buttons outside the panels. For example, the following code handles the click of a Next button, which is placed just after panelStep3 (so it always appears at the bottom of the page). The code checks which step the user is currently on, hides the current panel, and shows the following panel. This way, the user is moved to the next step.

```
protected void cmdNext_Click(object sender, EventArgs e)
{
    if (panelStep1.Visible)
    {
        // Move to step 2.
        panelStep1.Visible = false;
        panelStep2.Visible = true;
    }
    else if (panelStep2.Visible)
    {
        // Move to step 3.
        panelStep2.Visible = false;
        panelStep3.Visible = true;

        // Change text of button from Next to Finish.
        cmdNext.Text = "Finish";
    }
    else if (panelStep3.Visible)
    {
        // The wizard is finished.
        panelStep3.Visible = false;
```

```
        // Add code here to perform the appropriate task
        // with the information you've collected.
    }
}
```

This approach works relatively well. Even when the panels are hidden, you can still interact with all the controls on each panel and retrieve the information they contain. The problem is that you need to write all the code for controlling which panel is visible. If you make your wizard much more complex—for example, you want to add a button for returning to a previous step—it becomes more difficult to keep track of what's happening. At best, this approach clutters your page with the code for managing the panels. At worst, you'll make a minor mistake and end up with two panels showing at the same time.

Fortunately, ASP.NET gives you a more robust option. You can use two controls that are designed for the job—the MultiView and the Wizard. In the following sections, you'll see how you can use both of these controls with the GreetingCardMaker example developed in Chapter 6.

## The MultiView Control

The MultiView is the simpler of the two multiple-view controls. Essentially, the MultiView gives you a way to declare multiple views and show only one at a time. It has no default user interface—you get only whatever HTML and controls you add. The MultiView is equivalent to the custom panel approach explained earlier.

Creating a MultiView is suitably straightforward. You add the <asp:MultiView> tag to your .aspx page file and then add one <asp:View> tag inside it for each separate view:

```
<asp:MultiView ID="MultiView1" runat="server">
  <asp:View ID=”View1” runat=”server”>...</asp:View>
  <asp:View ID=”View2” runat=”server”>...</asp:View>
  <asp:View ID=”View3” runat=”server”>...</asp:View>
</asp:MultiView>
```

In Visual Studio, you create these tags by first dropping a MultiView control onto your form and then using the Toolbox to add as many View controls inside it as you want. This drag-and-drop process can be a bit tricky. When you add the first View control, you must make sure to drop it in the blank area inside the MultiView (not next to the MultiView, or on the MultiView's title bar). When you add more View controls, you must drop each one on one of the gray header bars of one of the existing views. The gray header has the View title (such as View1 or View2).

The View control plays the same role as the Panel control in the previous example, and the MultiView takes care of coordinating all the views so that only one is visible at a time.

Inside each view, you can add HTML or web controls. For example, consider the GreetingCardMaker example demonstrated in Chapter 6, which allows the user to create a greeting card by supplying some text and choosing colors, a font, and a background. As the GreetingCardMaker grows more complex, it requires more controls, and it becomes increasingly difficult to fit all those controls on the same page. One possible solution is to divide these controls into logical groups and place each group in a separate view.

### Creating Views

Here's the full markup for a MultiView that splits the greeting card controls into three views named View1, View2, and View3:

```
<asp:MultiView ID="MultiView1" runat="server" >

  <asp:View ID=”View1” runat=”server”>
    Choose a foreground (text) color:<br  />
```

```
    <asp:DropDownList ID="lstForeColor" runat="server" AutoPostBack="True"
     OnSelectedIndexChanged="ControlChanged" />
    <br  /><br  />
    Choose a background color:<br  />
    <asp:DropDownList ID="lstBackColor" runat="server" AutoPostBack="True"
     OnSelectedIndexChanged="ControlChanged" />
  </asp:View>

  <asp:View ID="View2" runat="server">
    Choose a border style:<br  />
    <asp:RadioButtonList ID="lstBorder" runat="server" AutoPostBack="True"
     OnSelectedIndexChanged="ControlChanged" RepeatColumns="2" />
    <br  />
    <asp:CheckBox ID="chkPicture" runat="server" AutoPostBack="True"
     OnCheckedChanged="ControlChanged" Text="Add the Default Picture"  />
  </asp:View>

  <asp:View ID="View3" runat="server">
    Choose a font name:<br  />
    <asp:DropDownList ID="lstFontName" runat="server" AutoPostBack="True"
     OnSelectedIndexChanged="ControlChanged" />
    <br  /><br  />
    Specify a font size:<br  />
    <asp:TextBox ID="txtFontSize" runat="server" AutoPostBack="True"
     OnTextChanged="ControlChanged" />
    <br  /><br  />
    Enter the greeting text below:<br  />
    <asp:TextBox ID="txtGreeting" runat="server" AutoPostBack="True"
     OnTextChanged="ControlChanged" TextMode="MultiLine" />
  </asp:View>

</asp:MultiView>
```

Visual Studio shows all your views at design time, one after the other (see Figure 10-6). You can edit these regions in the same way you design any other part of the page.

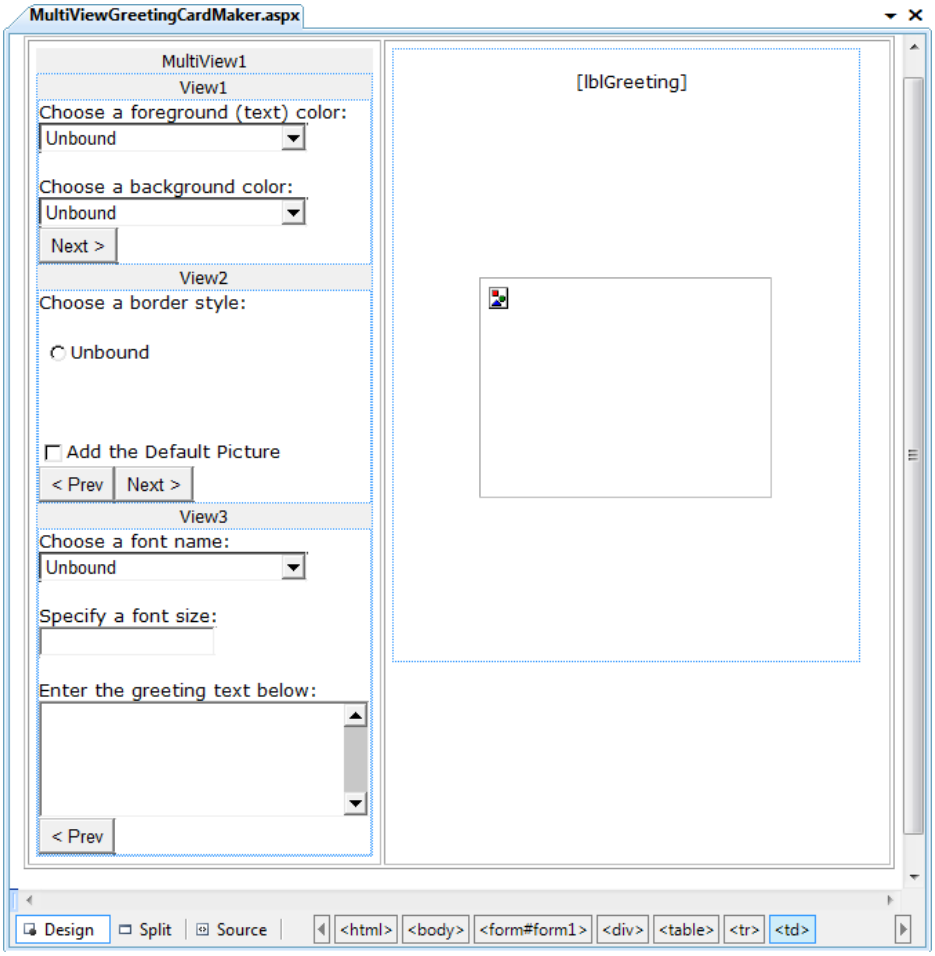


***Figure 10-6.*** *Designing multiple views*

## Showing a View

If you run this example, you won't see what you expect. The MultiView will appear empty on the page, and all the controls in all your views will be hidden.

The reason this happens is that the MultiView.ActiveViewIndex property is, by default, set to –1. The ActiveViewIndex property determines which view will be shown. If you set the ActiveViewIndex to 0, however, you'll see the first view. Similarly, you can set it to 1 to show the second view, and so on. You can set this property by using the Properties window or using code:

```
// Show the first view.
MultiView1.ActiveViewIndex = 0;
```

This example shows the first view (View1) and hides whatever view is currently being displayed, if any.

■ **Tip** To make more-readable code, you can create an enumeration that defines a name for each view. That way, you can set the ActiveViewIndex by using the descriptive name from the enumeration rather than an ordinary number. Refer to Chapter 3 for a refresher on enumerations.

You can also use the SetActiveView() method, which accepts any one of the view objects you've created, rather than the view name. By using the view objects, you can catch errors earlier. For example, if you misspell a view name, Visual Studio will spot the problem when you compile your code, rather than allowing your code to fail at runtime when it attempts to load the view.

```
MultiView1.SetActiveView(View1);
```

This gives you enough functionality that you can create previous and next navigation buttons. However, it's still up to you to write the code that checks which view is visible and changes the view. This code is a little simpler, because you don't need to worry about hiding views any longer, but it's still less than ideal.

Fortunately, the MultiView includes some built-in smarts that can save you a lot of trouble. Here's how it works: the MultiView recognizes button controls with specific command names. (Technically, a button control is any control that implements the IButtonControl interface, including the Button, ImageButton, and LinkButton.) If you add a button control to the view that uses one of these recognized command names, the button gets some automatic functionality. Using this technique, you can create navigation buttons without writing any code.

Table 10-6 lists all the recognized command names. Each command name also has a corresponding static field in the MultiView class, so you can easily get the right command name if you choose to set it programmatically.

***Table 10-6.*** *Recognized Command Names for the MultiView*

| Command Name | MultiView Field | Description |
|---|---|---|
| PrevView | PreviousViewCommandName | Moves to the previous view. |
| NextView | NextViewCommandName | Moves to the next view. |
| SwitchViewByID | SwitchViewByIDCommandName | Moves to the view with a specific ID (string name). The ID is taken from the CommandArgument property of the button control. |
| SwitchViewByIndex | SwitchViewByIndexCommandName | Moves to the view with a specific numeric index. The index is taken from the CommandArgument property of the button control. |

To try this, add this button to the first view:

```
<asp:Button ID="Button1" runat="server" CommandArgument="View2"
CommandName=”SwitchViewByID” Text=”Go to View2” />
```

When clicked, this button sets the MultiView to show the view specified by the CommandArgument (View2).

Rather than create buttons that take the user to a specific view, you might want a button that moves forward or backward one view. To do this, you use the PrevView and NextView command names. Here's an example that defines previous and next buttons in the second View:

```
<asp:Button ID="Button1" runat="server" Text="< Prev" CommandName="PrevView" />
<asp:Button ID=”Button2” runat=”server” Text=”Next >” CommandName=”NextView” />
```

After you add these buttons to your view, you can move from view to view easily. Figure 10-7 shows the previous example with the second view currently visible.

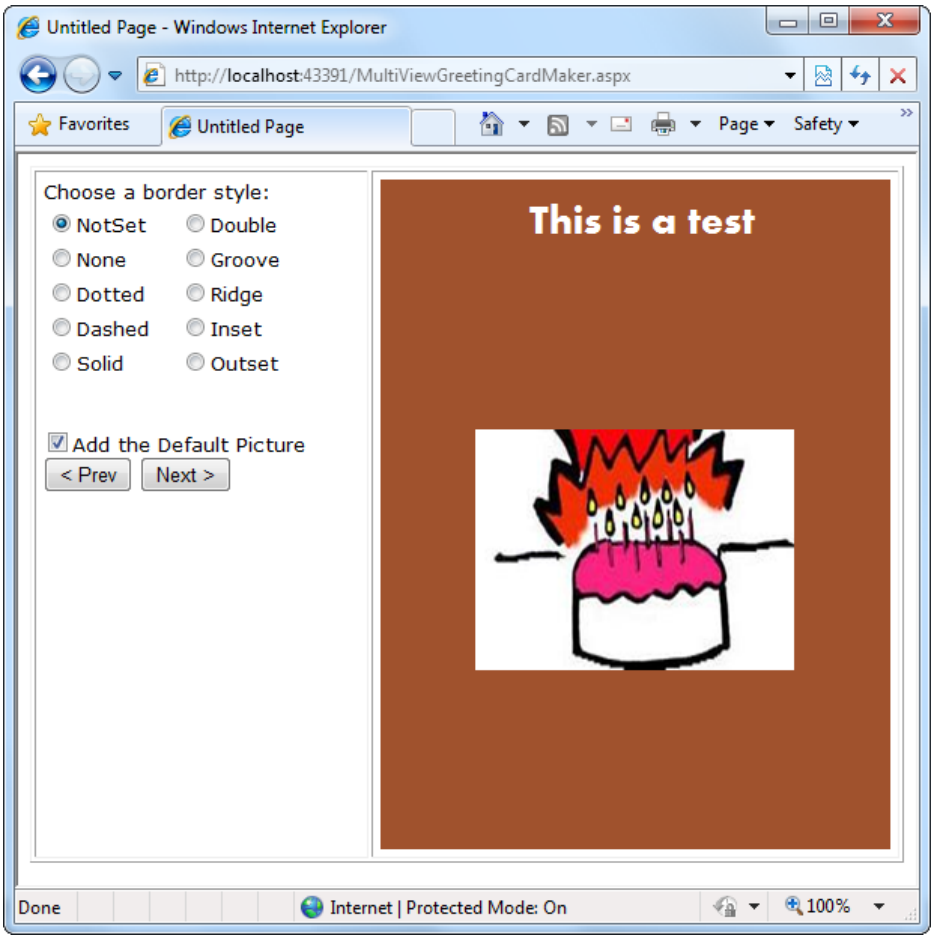


***Figure 10-7.*** *Moving from one view to another*

■ **Tip** Be careful how many views you cram into a single page. When you use the MultiView control, the entire control model—including the controls from every view—is created on every postback and persisted to view state. In most situations, this won't be a significant factor. However, it increases the overall page size, especially if you're tweaking controls programmatically (which increases the amount of information they need to store in view state).

## The Wizard Control

The Wizard control is a more glamorous version of the MultiView control. It also supports showing one of several views at a time, but it includes a fair bit of built-in yet customizable behavior, including navigation buttons, a sidebar with step links, styles, and templates.

Usually, wizards represent a single task, and the user moves linearly through them, moving from the current step to the one immediately following it (or the one immediately preceding it in the case of a correction). The ASP.NET Wizard control also supports nonlinear navigation, which means it allows you to decide to ignore a step based on the information the user supplies.

By default, the Wizard control supplies navigation buttons and a sidebar with links for each step on the left. You can hide the sidebar by setting the Wizard.DisplaySideBar property to false. Usually, you'll take this step if you want to enforce strict step-by-step navigation and prevent the user from jumping out of sequence. You supply the content for each step by using any HTML or ASP.NET controls. Figure 10-8 shows the region where you can add content to an out-of-the-box Wizard instance.

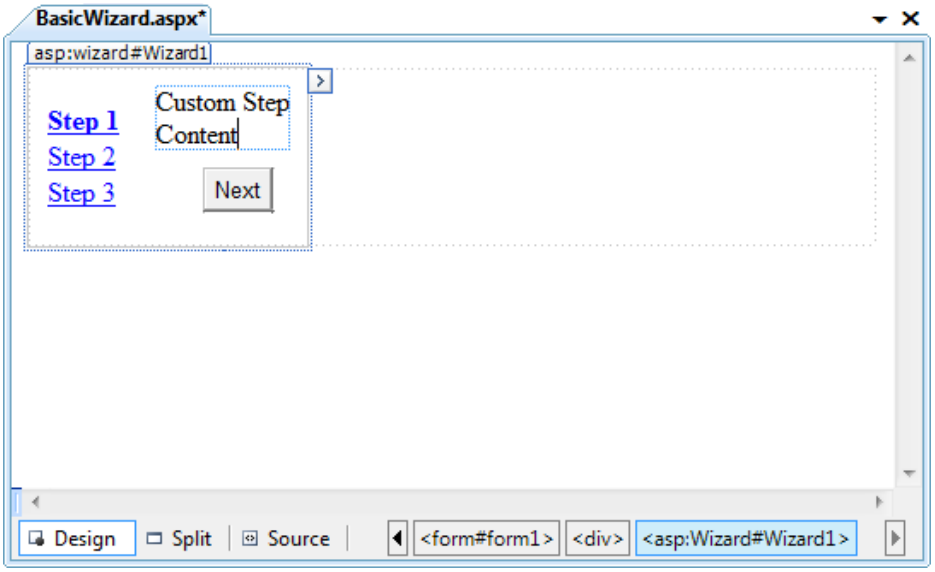


***Figure 10-8.*** *The region for step content*

## Defining Wizard Steps

To create a wizard in ASP.NET, you simply define the steps and their content by using <asp:WizardStep> tags. Here's the basic structure you'll use:

```
<asp:Wizard ID="Wizard1" runat="server" ... >
  <WizardSteps>

    <asp:WizardStep runat=”server” Title=”Step 1”>
      ...
    </asp:WizardStep>
```

```
    <asp:WizardStep runat=”server” Title=”Step 2”>
      ...
    </asp:WizardStep>

    ...
  <WizardSteps>
</asp:Wizard>
```

You can add as many WizardStep controls inside the Wizard as you want. Conceptually, the WizardStep plays the same role as the View in a MultiView (or the basic Panel in the first example that you considered). You place the content for each step inside the WizardStep control.

Before you start adding the content to your wizard, it's worth reviewing Table 10-7, which shows a few basic pieces of information that you can define for each step.

***Table 10-7.*** *WizardStep Properties*

| Property | Description |
|---|---|
| Title | The descriptive name of the step. This name is used for the text of the links in the sidebar. |
| StepType | The type of step, as a value from the WizardStepType enumeration. This value determines the type of navigation buttons that will be shown for this step. Choices include Start (shows a Next button), Step (shows Next and Previous buttons), Finish (shows Finish and Previous buttons), Complete (shows no buttons and hides the sidebar, if it's enabled), and Auto (the step type is inferred from the position in the collection). The default is Auto, which means the first step is Start, the last step is Finish, and all other steps are Step. |
| AllowReturn | Indicates whether the user can return to this step. If false, the user will not be able to return after passing this step. The sidebar link for this step will have no effect, and the Previous button of the following step will either skip this step or be hidden completely (depending on the AllowReturn value of the preceding steps). |

To see how this works, consider a wizard that again uses the GreetingCardMaker example. It guides the user through four steps. The first three steps allow the user to configure the greeting card, and the final step shows the generated card.

```
<asp:Wizard ID="Wizard1" runat="server" ActiveStepIndex="0"
 BackColor=”LemonChiffon” BorderStyle=”Groove” BorderWidth=”2px” CellPadding=”10”>

  <WizardSteps>
    <asp:WizardStep runat=”server” Title=”Step 1 - Colors”>
      Choose a foreground (text) color:<br />
      <asp:DropDownList ID=”lstForeColor” runat=”server” />
      <br />
      Choose a background color:<br />
      <asp:DropDownList ID=”lstBackColor” runat=”server” />
    </asp:WizardStep>

    <asp:WizardStep runat=”server” Title=”Step 2 - Background”>
      Choose a border style:<br />
```

```
      <asp:RadioButtonList ID="lstBorder" runat="server" RepeatColumns="2" />
      <br /><br />
      <asp:CheckBox ID="chkPicture" runat="server"
       Text="Add the Default Picture" />
    </asp:WizardStep>

    <asp:WizardStep runat="server" Title="Step 3 - Text">
      Choose a font name:<br />
      <asp:DropDownList ID="lstFontName" runat="server" />
      <br /><br />
      Specify a font size:<br />
      <asp:TextBox ID="txtFontSize" runat="server" />
      <br /><br />
      Enter the greeting text below:<br />
      <asp:TextBox ID="txtGreeting" runat="server"
        TextMode="MultiLine" />
    </asp:WizardStep>

    <asp:WizardStep runat="server" StepType="Complete" Title="Greeting Card">
      <asp:Panel ID="pnlCard" runat="server" HorizontalAlign="Center">
        <br />
        <asp:Label ID="lblGreeting" runat="server" />
        <asp:Image ID="imgDefault" runat="server" Visible="False" />
      </asp:Panel>
    </asp:WizardStep>
  </WizardSteps>

</asp:Wizard>
```

If you look carefully, you'll find a few differences from the original page and the MultiView-based example. First, the controls aren't set to automatically post back. That's because the greeting card isn't rendered until the final step, at the conclusion of the wizard. (You'll learn more about how to handle this event in the next section.) Another change is that no navigation buttons exist. That's because the wizard adds these details automatically based on the step type. For example, you'll get a Next button for the first two steps, a Previous button for steps 2 and 3, and a Finish button for step 3. The final step, which shows the complete card, doesn't provide any navigation links because the StepType is set to Complete. Figure 10-9 shows the wizard steps.

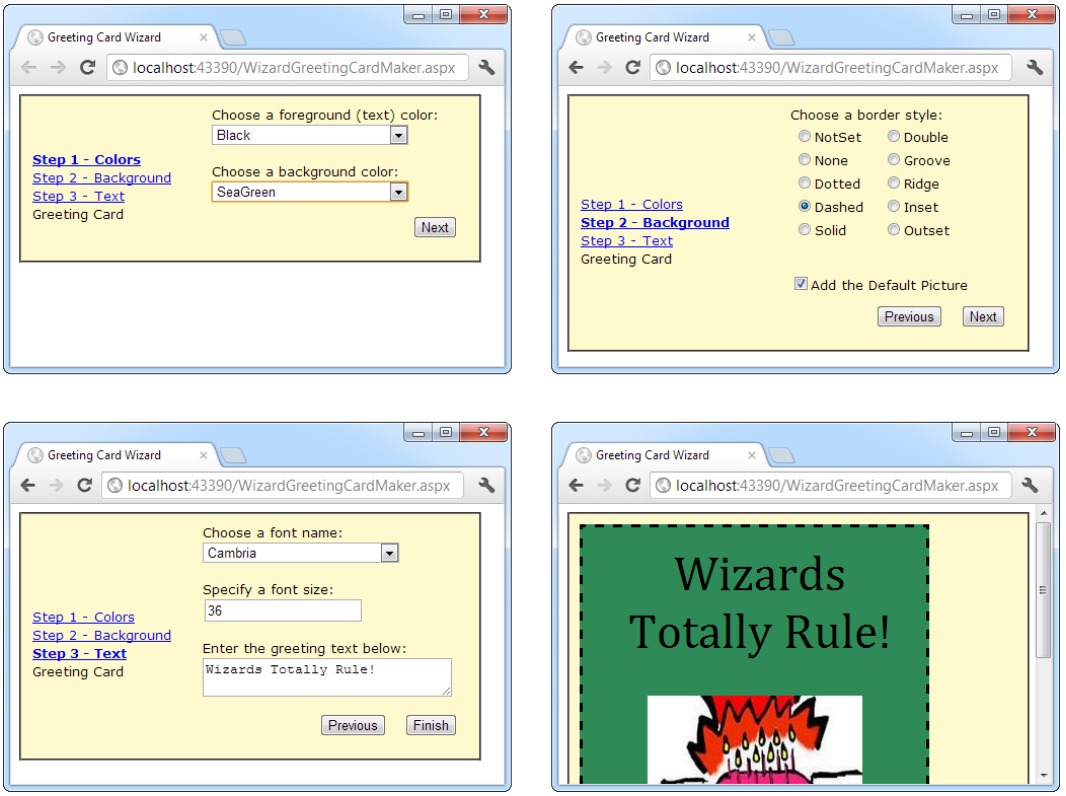


***Figure 10-9.*** *A wizard with four steps*

Unlike the MultiView control, you can see only one step at a time in Visual Studio. To choose which step you're currently designing, select it from the smart tag, as shown in Figure 10-10. But be warned—every time you do, Visual Studio changes the Wizard.ActiveStepIndex property to the step you choose. Make sure you set this back to 0 before you run your application so it starts at the first step.

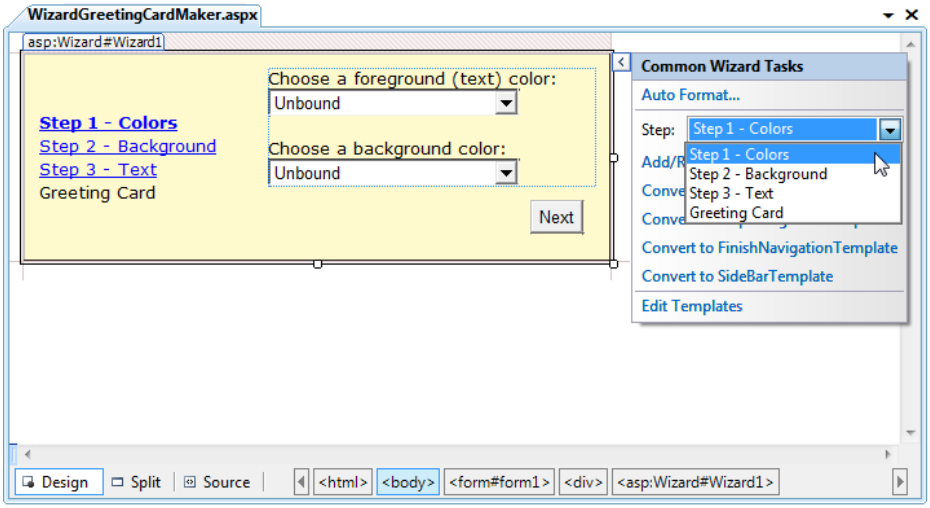


***Figure 10-10.*** *Designing a step*

■ **Note** Remember, when you add controls to separate steps on a wizard, the controls are all instantiated and persisted in view state, regardless of which step is currently shown. If you need to slim down a complex wizard, you'll need to split it into separate pages, use the Server.Transfer() method to move from one page to the next, and tolerate a less elegant programming model.

## Using Wizard Events

You can write the code that underpins your wizard by responding to several events (as listed in Table 10-8).

***Table 10-8.*** *Wizard Events*

| Event | Description |
|---|---|
| ActiveStepChanged | Occurs when the control switches to a new step (either because the user has clicked a navigation button or because your code has changed the ActiveStepIndex property). |
| CancelButtonClick | Occurs when the Cancel button is clicked. The Cancel button is not shown by default, but you can add it to every step by setting the Wizard.DisplayCancelButton property. Usually, a Cancel button exits the wizard. If you don't have any cleanup code to perform, just set the CancelDestinationPageUrl property, and the wizard will take care of the redirection automatically. |
| FinishButtonClick | Occurs when the Finish button is clicked. |
| NextButtonClick and PreviousButtonClick | Occurs when the Next or Previous button is clicked in any step. However, because there is more than one way to move from one step to the next, it's often easier to handle the ActiveStepChanged event. |
| SideBarButtonClick | Occurs when a button in the sidebar area is clicked. |

On the whole, two wizard programming models exist:

> *Commit-as-you-go:* This makes sense if each wizard step wraps an atomic operation that can't be reversed. For example, if you're processing an order that involves a credit card authorization followed by a final purchase, you probably don't want the user to step back and edit the credit card number after it's been authorized. To support the commit-as-you-go model, you set the AllowReturn property to false on some or all steps. You may also want to respond to the ActiveStepChanged event to commit changes for each step.
>
> *Commit-at-the-end:* This makes sense if each wizard step is collecting information for an operation that's performed only at the end. For example, if you're collecting user information and plan to generate a new account after you have all the information, you'll probably allow a user to make changes midway through the process. You execute your code for generating the new account when the wizard ends by reacting to the FinishButtonClick event.

To implement commit-at-the-end with the current example, just respond to the FinishButtonClick event. For example, to implement the greeting card wizard, you simply need to respond to this event and call UpdateCard(), the private method that refreshes the greeting card:

```
protected void Wizard1_FinishButtonClick(object sender,
  WizardNavigationEventArgs e)
{
    UpdateCard();
}
```

For the complete code for the UpdateCard() method, which generates the greeting card, refer to Chapter 6 (or check out the downloadable sample code).

If you decide to use the commit-as-you go model, you would respond to the ActiveStepChanged event and call UpdateCard() at that point to refresh the card every time the user moves from one step to another. This assumes the greeting card is always visible. (In other words, it's not contained in the final step of the wizard.) The commit-as-you-go model is similar to the previous example that used the MultiView.

## Formatting the Wizard

Without a doubt, the Wizard control's greatest strength is the way it lets you customize its appearance. This means if you want the basic model (a multistep process with navigation buttons and various events), you aren't locked into the default user interface.

Depending on how radically you want to change the wizard, you have several options. For less-dramatic modifications, you can set various top-level properties of the Wizard control. For example, you can control the colors, fonts, spacing, and border style, as you can with any ASP.NET control. You can also tweak the appearance of every button. For example, to change the Next button, you can use the following properties: StepNextButtonType (use a button, link, or clickable image), StepNextButtonText (customize the text for a button or link), StepNextButtonImageUrl (set the image for an image button), and StepNextButtonStyle (use a style from a style sheet). You can also add a header by using the HeaderText property.

More control is available through styles. You can use styles to apply formatting options to various portions of the Wizard control just as you can use styles to format parts of rich data controls such as the GridView. Table 10-9 lists all the styles you can use. As with other style-based controls, more-specific style settings (such as SideBarStyle) override more-general style settings (such as ControlStyle) when they conflict. Similarly, StartNextButtonStyle overrides NavigationButtonStyle on the first step.

***Table 10-9.*** *Wizard Styles*

| Style | Description |
|---|---|
| ControlStyle | Applies to all sections of the Wizard control. |
| HeaderStyle | Applies to the header section of the wizard, which is visible only if you set some text in the HeaderText property. |
| BorderStyle | Applies to the border around the Wizard control. You can use it in conjunction with the BorderColor and BorderWidth properties. |
| SideBarStyle | Applies to the sidebar area of the wizard. |
| SideBarButtonStyle | Applies to just the buttons in the sidebar. |
| StepStyle | Applies to the section of the control where you define the step content. |
| NavigationStyle | Applies to the bottom area of the control where the navigation buttons are displayed. |
| NavigationButtonStyle | Applies to just the navigation buttons in the navigation area. |
| StartNextButtonStyle | Applies to the Next navigation button on the first step (when StepType is Start). |
| StepNextButtonStyle | Applies to the Next navigation button on intermediate steps (when StepType is Step). |
| StepPreviousButtonStyle | Applies to the Previous navigation button on intermediate steps (when StepType is Step). |
| FinishPreviousButtonStyle | Applies to the Previous navigation button on the last step (when StepType is Finish). |
| FinishCompleteButtonStyle | Applies to the Complete navigation button on the last step (when StepType is Finish). |
| CancelButtonStyle | Applies to the Cancel button, if you have Wizard.DisplayCancelButton set to true. |

■ **Note** The Wizard control also supports templates, which give you a more radical approach to formatting. If you can't get the level of customization you want through properties and styles, you can use templates to completely define the appearance of each section of the Wizard control, including the headers and navigation links. Templates require data-binding expressions and are discussed in Chapter 15 and Chapter 16.

## Validating with the Wizard

The FinishButtonClick, NextButtonClick, PreviousButtonClick, and SideBarButtonClick events are cancellable. That means that you can use code like this to prevent the requested navigation action from taking place:

```
protected void Wizard1_NextButtonClick(object sender,
  WizardNavigationEventArgs e)
```

```
{
    // Perform some sort of check.
    if (e.NextStepIndex == 1 && txtName.Text == "")
    {
        // Cancel navigation and display a message elsewhere on the page.
        e.Cancel = true;
        lblInfo.Text =
          "You cannot move to the next step until you supply your name.";
    }
}
```

Here the code checks whether the user is trying to move to step 1 by using the NextStepIndex property. (Alternatively, you could examine the current step by using the CurrentStepIndex property.) If so, the code then checks a text box and cancels the navigation if it doesn't contain any text, keeping the user on the current step. Writing this sort of logic gets a little tricky, because you need to keep in mind that step-to-step navigation can be performed in several ways. To simplify your life, you can write one event handler that deals with the NextButtonClick, PreviousButtonClick, and SideBarButtonClick events, and performs the same check. You saw this technique in Chapter 6 with the GreetingCardMaker.

---

■ **Note** You can also use the ASP.NET validation controls in a Wizard without any problem. If the validation controls detect invalid data, they will prevent the user from clicking any of the sidebar links (to jump to another step), and they will prevent the user from continuing by clicking the Next button. However, by default the Previous button has its CausesValidation property set to false, which means the user *will* be allowed to step back to the previous step.

---

# The Last Word

This chapter showed you how the rich Calendar, AdRotator, MultiView, and Wizard controls can go far beyond the limitations of ordinary HTML elements. When you're working with these controls, you don't need to think about HTML at all. Instead, you can focus on the object model that's defined by the control.

Throughout this book, you'll consider some more examples of rich controls and learn how to use them to create rich web applications that are a world apart from HTML basics. Some of the most exciting rich controls that are still ahead include the navigation controls (Chapter 13), the data controls (Chapter 16), and the security controls (Chapter 20).

---

■ **Tip** You might also be interested in adding third-party controls to your websites. The Internet contains many hubs for control sharing. One such location is Microsoft's own www.asp.net, which provides a control gallery where developers can submit their own ASP.NET web controls. Some of these controls are free (at least in a limited version), and others require a purchase.

---