4
State Management

WHERE DO YOU STORE per-client state in a Web application? This question is at the root of many heated debates over how to best design Web applications. The disconnected nature of HTTP means that there is no "natural" way to keep state on behalf of individual clients, but that certainly hasn't stopped developers from finding ways of doing it. Today there are many choices for keeping client-specific state in an ASP.NET Web application, including Session state, View state, cookies, the HttpContext.Items collection, and any number of custom solutions. The best choice depends on many things, including the scope (Do you need the state to last for an entire user session or just between two pages?), the size (Are you worried about passing too much data in the response and would prefer to keep it on the server?), and the deployment environment (Is this application deployed on a Web farm so that server state must be somehow shared?), just to name a few.

ASP.NET 2.0 does not offer a penultimate solution for storing client state, but it does introduce three new features that should be considered any time you are looking for a place to store state on behalf of individual users. The first feature, **cross-page posting**, is actually the resurrection of a common technique used in classic ASP and other Web development environments for propagating state between two pages. This technique was not available in ASP.NET 1.1 because of the way POST requests were parsed and processed by individual pages, but has now been reincorporated into

ASP.NET in such a way that it works in conjunction with server-side controls and other ASP.NET features. The second feature is a trio of new server-side controls that implement the common technique of showing and hiding portions of a page as the user interacts with it. The Wizard control gives developers a simple way to construct a multistep user interface on a single page, and the MultiView and View controls provide a slightly lower-level (and more flexible) way of hiding and displaying panes.

The last feature, Profile, is by far the most intriguing. **Profile** provides a prebuilt implementation that will store per-client state across requests and even sessions of your application in a persistent back-end data store. It ties into the Membership provider of ASP.NET 2.0 for identifying authenticated clients, and generates its own identifier for working with anonymous users as well, storing each client's data in a preconfigured database table. This feature provides a flexible and extensible way of storing client data and should prove quite useful in almost any ASP.NET application.

Cross-Page Posting

This version of ASP.NET reintroduces the ability to perform cross-page posts. Once a common practice in classic ASP applications, ASP.NET 1.x made it nearly impossible to use this technique for state propagation because of server-side forms and view state. This section covers the fundamentals of cross-page posting in general, and then looks at the support added in ASP.NET 2.0.

Fundamentals

One common mechanism for sending state from one page to another in Web applications is to use a form with input elements whose action attribute is set to the URL or the target page. The values of the source page's input elements are passed as name-value pairs to the target page in the body of the POST request (or in the query string if the form's method attribute is set to GET), at which point the target page has access to the values. Listings 4-1 and 4-2 show a pair of sample pages that request a user's name, age, and marital status, and display a customized message on the target page.

LISTING 4-1: sourceform.aspx—sample form using a cross-page post

```
<!-- sourceform.aspx -->
<%@ Page language="C#" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Source Form</title>
</head>
<body>
    <form action="target.aspx" method="post">
        Enter your name:
        <input name="_nameTextBox" type="text" id="_nameTextBox" />
        <br />
        Enter your age:
        <input name="_ageTextBox" type="text" id="_ageTextBox" /><br />
        <input id="_marriedCheckBox" type="checkbox"
               name="_marriedCheckBox" />
        <label for="_marriedCheckBox">Married?</label><br />
        <input type="submit" name="_nextPageButton" value="Next page" />
    </form>
</body>
</html>
```

LISTING 4-2: target.aspx—sample target page for a cross-page post

```
<!-- target.aspx -->
<%@ Page language="C#" %>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Target Page</title>
</head>
<body>
  <h3>
  Hello there
  <%= Request.Form["_nameTextBox"] %>, you are
  <%= Request.Form["_ageTextBox"] %> years old and are
  <%= (Request.Form["_marriedCheckBox"] == "on") ? "" : "not " %>
  married!
  </h3>
</body>
</html>
```

This example works fine in both ASP.NET 1.1 and 2.0, and with a few simple modifications would even work in classic ASP. This technique is rarely used in ASP.NET, however, because the form on the source page cannot be marked with *runat="server"*; thus, many of the advantages of

ASP.NET, including server-side controls, cannot be used. ASP.NET builds much of its server-side control infrastructure on the assumption that pages with forms will generate POST requests back to the same page. In fact, if you try and change the action attribute of a form that is also marked with runat="server", it will have no effect, as ASP.NET will replace the attribute when it renders the page with the page's URL itself. As a result, most ASP.NET sites resort to alternative techniques for propagating state between pages (like Session state or using Server.Transfer while caching data in the Context.Items collection).

In the 2.0 release of ASP.NET, cross-page posting is now supported again, even if you are using server-side controls and all of the other ASP.NET features. The usage model is a bit different from the one shown in Listings 4-1 and 4-2, but in the end it achieves the desired goal of issuing a POST request from one page to another, and allowing the secondary page to harvest the contents from the POST body and process them as it desires. To initiate a cross-page post, you use the new PostBackUrl attribute defined by the IButtonControl interface, which is implemented by the Button, LinkButton, and ImageButton controls. When the PostBackUrl property is set to a different page, the OnClick handler of the button is set to call a JavaScript function that changes the default action of the form to the target page's URL. Listing 4-3 shows a sample form that uses cross-page posting to pass name, age, and marital status data entered by the user to a target page.

Listing 4-3: SourcePage1.aspx—using cross-page posting support in ASP.NET 2.0

```
<!-- SourcePage1.aspx -->
<%@ Page Language="C#" CodeFile="SourcePage1.aspx.cs"
         Inherits="SourcePage1" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Source page 1</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            Enter your name:
            <asp:TextBox ID="_nameTextBox" runat="server" /><br />
            Enter your age:
            <asp:TextBox ID="_ageTextBox" runat="server" /><br />
```

```
            <asp:CheckBox ID="_marriedCheckBox" runat="server"
                          Text="Married?" /><br />
            <asp:Button ID="_nextPageButton" runat="server"
                     Text="Next page" PostBackUrl="~/TargetPage.aspx" />
        </div>
    </form>
</body>
</html>
```

Once you have set up the source page to post to the target page, the next step is to build the target page to use the values passed by the source page. Because ASP.NET uses POST data to manage the state of its server-side controls, it would not have been sufficient to expect the target page to pull name/value pairs from the POST body, since many of those values (like __VIEWSTATE) need to be parsed by the server-side controls that wrote the values there in the first place. Therefore, ASP.NET will actually create a fresh instance of the source page class and ask it to parse the POST body on behalf of the target page. This page instance is then made available to the target page via the PreviousPage property, which is now defined in the Page class. Listings 4-4 and 4-5 show one example of how you could use this property in a target page to retrieve the values of the controls from the previous page: by calling FindControl on the Form control, you can retrieve individual controls whose state has been initialized with values from the post's body.

LISTING 4-4: TargetPage.aspx—target page of a cross-page post

```
<!-- TargetPage.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Tar-
getPage.aspx.cs"
         Inherits="TargetPage" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Target Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label runat="server" ID="_messageLabel" />
        </div>
    </form>
</body>
</html>
```

Listing 4-5: TargetPage.aspx.cs—target page of a cross-page post codebehind

```
// TargetPage.aspx.cs
public partial class TargetPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (PreviousPage != null)
        {
            TextBox nameTextBox =
                 (TextBox)PreviousPage.Form.FindControl("_nameTextBox");
            TextBox ageTextBox =
                 (TextBox)PreviousPage.Form.FindControl("_ageTextBox");
            CheckBox marriedCheckBox =
            (CheckBox)PreviousPage.Form.FindControl("_marriedCheckBox");

            _messageLabel.Text = string.Format(
   "<h3>Hello there {0}, you are {1} years old and {2} married!</h3>",
            nameTextBox.Text, ageTextBox.Text,
            marriedCheckBox.Checked ? "" : "not");
        }
    }
}
```

The technique shown in Listing 4-5 for retrieving values from the previous page is somewhat fragile, as it relies on the identifiers of controls on the previous page as well as their hierarchical placement, which could easily be changed. A better approach is to expose any data from the previous page to the target page by writing public property accessors in the codebehind, as shown in Listing 4-6.

Listing 4-6: SourcePage1.aspx.cs—exposing public properties to the target page

```
// File: SourcePage1.aspx.cs
public partial class SourcePage1 : Page
{
    public string Name
    {
      get { return _nameTextBox.Text; }
    }

    public int Age
    {
      get { return int.Parse(_ageTextBox.Text); }
    }
```

```
    public bool Married
    {
      get { return _marriedCheckBox.Checked; }
    }
}
```

Once the public properties are defined, the target page can cast the PreviousPage property to the specific type of the previous page and retrieve the values using the exposed properties, as shown in Listing 4-7.

LISTING 4-7: TargetPage.aspx.cs—target page using properties to retrieve source page values

```
// TargetPage.aspx.cs
public partial class TargetPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        SourcePage1 sp = PreviousPage as SourcePage1;
        if (sp != null)
        {
            _messageLabel.Text = string.Format(
    "<h3>Hello there {0}, you are {1} years old and {2} married!</h3>",
            sp.Name, sp.Age, sp.Married ? "" : "not");
        }
    }
}
```

Because this last scenario is likely to be the most common use of cross-page posting—that is, a specific source page exposes properties to be consumed by a specific target page—there is a directive called PreviousPageType that will automatically cast the previous page to the correct type for you. When you specify a page in the VirtualPath property of this directive, the PreviousPage property that is generated for that page will be strongly typed to the previous page type, meaning that you no longer have to perform the cast yourself, as shown in Listings 4-8 and 4-9.

LISTING 4-8: TargetPage.aspx with strongly typed previous page

```
<!-- TargetPage.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Tar-
getPage.aspx.cs"
         Inherits="TargetPage" %>
<%@ PreviousPageType VirtualPath="~/SourcePage1.aspx" %>
...
```

Listing 4-9: TargetPage.aspx.cs—using strongly typed PreviousPage accessor

```
// TargetPage.aspx.cs
public partial class TargetPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (PreviousPage != null)
        {
            _messageLabel.Text = string.Format(
    "<h3>Hello there {0}, you are {1} years old and {2} married!</h3>",
            PreviousPage.Name, PreviousPage.Age,
            PreviousPage.Married ? "" : "not");
        }
    }
}
```

Implementation

When you set the PostBackUrl property of a button to a different page, it does two things. First, it sets the client-side OnClick handler for that button to point to a JavaScript method called WebForm_DoPostBackWithOptions, which will programmatically set the form's action to the target page. Second, it causes the page to render an additional hidden field, __PREVIOUSPAGE, which contains the path of the source page in an encrypted string along with an accompanying message authentication code for validating the string. Setting the action dynamically like this enables you to have multiple buttons on a page that all potentially post to different pages and keeps the architecture flexible. Storing the path of the previous page in a hidden field means that no matter where you send the POST request, the target page will be able to determine where the request came from, and will know which class to instantiate to parse the body of the message.

Once the POST request is issued to the target page, the path of the previous page is read and decrypted from the __PREVIOUSPAGE hidden field and cached. As you have seen, the PreviousPage property on the target page gives access to the previous page and its data, but for efficiency, this property allocates the previous page class on demand. If you never actually access the PreviousPage property, it will never create the class and ask it to parse the body of the request.

The first time you do access the PreviousPage property in the target page, ASP.NET allocates a new instance of the previous page type, as

determined by the cached path to the previous page extracted from the __PREVIOUSPAGE hidden field. Once it is created, it then executes the page much like it would if the request had been issued to it. The page is not executed in its entirety, however, since it only needs to restore the state from the POST body, so it runs through its life cycle up to and including the LoadComplete event. The Response and Trace objects of the previous page instance are also set to null during this execution since there should be no output associated with the process.

It is important to keep in mind that the preceding page will be created and asked to run through LoadComplete. If you have any code that generates side effects, you should make an effort to exclude that code from running when the page is executed during a cross-page postback. You can check to see whether you are being executed for real or for the purpose of evaluating the POST body of a cross-page post by checking the IsCrossPagePostBack property. For example, suppose that the source page wrote to a database in its Load event handler for logging purposes. You would not want this code to execute during a cross-page postback evaluation since the request was not really made to that page. Listing 4-10 shows how you might exclude your logging code from being evaluated during a cross-page postback.

LISTING 4-10: Checking for IsCrossPagePostBack before running code with side effects

```
public partial class SourcePage1 : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsCrossPagePostBack)
        {
            WriteDataToLogFile();
        }
    }
}
```

Caveats

While this new support for cross-page posting is a welcome addition to ASP.NET, it does have some potential drawbacks you should be aware of before you elect to use it. The first thing to keep in mind is that the entire contents of the source page is going to be posted to the target page. This includes the entire view state field and all input elements on the page. If

you are using cross-page posting to send the value of a pair of TextBox controls to a target page, but you have a GridView with view state enabled on the source page, you're going to incur the cost of posting the entire contents of the GridView in addition to the TextBox controls just to send over a pair of strings. If you can't reduce the size of the request on the source page to an acceptable amount, you may want to consider using an alternative technique (like query strings) to propagate the values.

Validation is another potential trouble area with cross-page posting. If you are using validation controls in the client page to validate user input prior to the cross-page post, you should be aware that server-side validation will not take place until you access the PreviousPage property on the target page. Client-side validation will still happen as usual before the page issues the POST, but if you are relying on server-side validation at all, you must take care to check the IsValid property of the previous page before accessing the data exposed by the PreviousPage property.

A common scenario where this may occur is with custom validation controls. If you have set up a custom validation control with a server-side handler for the ServerValidate event, that method will not be called until you access the PreviousPage after the cross-page posting has occurred. Then there is the question of what to do if the previous page contains invalid data, since you can no longer just let the page render back to the client with error messages in place (because the client has already navigated away from the source page). The best option is probably just to place an indicator message that the data is invalid and provide a link back to the previous page to enter the data again. Listings 4-11 and 4-12 show a sample of a source page with a custom validation control and a button set up to use cross-page posting, along with a target page. Note that the code in the target page explicitly checks the validity of the previous page's data before using it and the error handling added if something is wrong.

Listing 4-11: Source page with custom validator

```
<!-- SourcePageWithValidation.aspx -->
<%@ Page Language="C#" %>

<script runat="server">
    public int Prime
    {
        get { return int.Parse(_primeNumberTextBox.Text); }
    }
```

```
    private bool IsPrime(int num)
    {
        // implementation omitted
    }

    protected void _primeValidator_ServerValidate(object source,
                            ServerValidateEventArgs args)
    {
        args.IsValid = IsPrime(Prime);
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Source page with validation</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            Enter your favorite prime number:
            <asp:TextBox ID="_primeNumberTextBox" runat="server" />
            <asp:CustomValidator ID="_primeValidator" runat="server"
                 ErrorMessage="Please enter a prime number"
               OnServerValidate="_primeValidator_ServerValidate">
                                        **</asp:CustomValidator><br />
            <asp:Button ID="_nextPageButton" runat="server"
                        Text="Next page"
                        PostBackUrl="~/TargetPageWithValidation.aspx"
                         /><br />
            <br />
            <asp:ValidationSummary ID="_validationSummary"
                                   runat="server" />
        </div>
    </form>
</body>
</html>
```

LISTING 4-12: Target page checking for validation

```
<!-- TargetPageWithValidation.aspx -->
<%@ Page Language="C#" %>
<%@ PreviousPageType VirtualPath="~/SourcePageWithValidation.aspx" %>

<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        if (PreviousPage != null && PreviousPage.IsValid)
        {
          _messageLabel.Text = "Thanks for choosing the prime number " +
                        PreviousPage.Prime.ToString();
        }
```

continues

```
        else
        {
            _messageLabel.Text = "Error in entering data";
            _messageLabel.ForeColor = Color.Red;
            _previousPageLink.Visible = true;
        }
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Target Page With validation</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label runat="server" ID="_messageLabel" /><br />
            <asp:HyperLink runat="server" ID="_previousPageLink"
                           NavigateUrl="~/SourcePageWithValidation.aspx"
                           visible="false">
                  Return to data entry page...</asp:HyperLink>
        </div>
    </form>
</body>
</html>
```

Finally, it is important to be aware that the entire cross-page posting mechanism relies on JavaScript to work properly, so if the client either doesn't support or has disabled JavaScript, your source pages will simply post back to themselves as the action on the form will not be changed on the client in response to the button press.

Multi-Source Cross-Page Posting

Cross-page posting can also be used to create a single target page that can be posted to by multiple source pages. Such a scenario may be useful if you have a site that provides several different ways of collecting information from the user but one centralized page for processing it.

If we try and extend our earlier example by introducing a second source page, also with the ability to collect the name, age, and marital status of the client, we run into a problem because each page is a distinct type with its own VirtualPath, and the target page will somehow have to distinguish between a post from source page 1 and one from source page 2. One way to

solve this problem is to implement a common interface in each source page's base class; this way, the target page assumes only that the posting page implements a particular interface and is not necessarily of one specific type or another. For example, we could write the IPersonInfo interface to model our cross-page POST data, as shown in Listing 4-13.

LISTING 4-13: IPersonInfo interface definition

```
public interface IPersonInfo
{
  string Name { get; }
  int Age { get; }
  bool Married { get; }
}
```

In each of the source pages, we then implement the IPersonInfo on the codebehind base class, and our target page can now safely cast the PreviousPage to the IPersonInfo type and extract the data regardless of which page was the source page, as shown in Listing 4-14.

LISTING 4-14: Generic target page using interface for previous page

```
IPersonInfo pi = PreviousPage as IPersonInfo;
if (pi != null)
{
  _messageLabel.Text = string.Format("<h3>Hello there {0}, you are {1}
years old and {2} married!</h3>",
                    pi.Name, pi.Age, pi.Married ? "" : "not");
}
```

It would be even better if we could use the PreviousPageType directive to strongly type the PreviousPage property to the IPersonInfo interface. In fact, there is a way to associate a type with a previous page instead of using the virtual path, which is to specify the TypeName attribute instead of the VirtualPath attribute in the PreviousPageType directive. Unfortunately, the TypeName attribute of the PreviousPageType directive requires that the specified type inherit from System.Web.UI.Page. You can introduce a workaround to get the strong typing by defining an abstract base class that implements the interface (or just defines abstract methods directly) and inherits from Page, as shown in Listing 4-15.

Listing 4-15: Abstract base class inheriting from Page for strong typing with PreviousPageType

```
public abstract class PersonInfoPage : Page, IPersonInfo
{
  public abstract string Name { get; }
  public abstract int Age { get; }
  public abstract bool Married { get; }
}
```

This technique then requires that each of the source pages you author change their base class from Page to this new PersonInfoPage base, and then implement the abstract properties to return the appropriate data. Listing 4-16 shows an example of a codebehind class for a source page using this new base class.

Listing 4-16: Codebehind class for a sample source page inheriting from PersonInfoPage

```
public partial class SourcePage1 : PersonInfoPage
{
  public override string Name
  {
    get { return _nameTextBox.Text; }
  }
  public override int Age
  {
    get { return int.Parse(_ageTextBox.Text); }
  }
  public override bool Married
  {
    get { return _marriedCheckBox.Checked; }
  }
}
```

Once all source pages are derived from our PersonInfoPage and the three abstract properties are implemented, our target page can be rewritten with a strongly typed PreviousPageType directive, which saves the trouble of casting, as shown in Listing 4-17.

Listing 4-17: Strongly typed target page using TypeName

```
<%@ PreviousPageType TypeName="PersonInfoPage" %>

<script runat="server">
protected void Page_Load(object sender, EventArgs e)
{
  if (PreviousPage != null)
```

```
  {
    _messageLabel.Text = string.Format(
 "<h3>Hello there {0}, you are {1} years old and {2} married!</h3>",
               PreviousPage.Name, PreviousPage.Age,
               PreviousPage.Married ? "" : "not");
  }
}
</script>
<!-- ... -->
```

The effort required to get the strong typing to work for multiple source pages hardly seems worth it in the end. You already have to check to see whether the PreviousPage property is null or not, and casting it to the interface using the *as* operator in C# is about the same amount of work as checking for null. However, both ways are valid approaches, and it is up to you to decide how much effort you want to put into making your previous pages strongly typed.

Wizard and MultiView Controls

This section covers a new collection of controls in ASP.NET 2.0 that simplify the process of collecting data from the user by using a sequence of steps that are all on a single page. The controls include the new Wizard control as well as the View and MultiView controls.

Same Page State Management

Another alternative to storing per-client state across requests is to have the user post back to the same page instead of navigating from one page to another. You can achieve the same sequential set of steps for data collection that you can using multiple pages with this technique by toggling the display of various panels, showing only one of several panels at any given time based on the user's progress. Instead of placing input controls on separate pages, you place them all on the same page, but separate them with Panel (or Placeholder) controls as shown in Figure 4-1. When the user selects the Next button in one panel, the handler for that button sets the visibility of the current panel to false and of the next panel to true.

This technique works well because all of the state for all the controls is kept on a single page, and even when the controls in a particular panel are

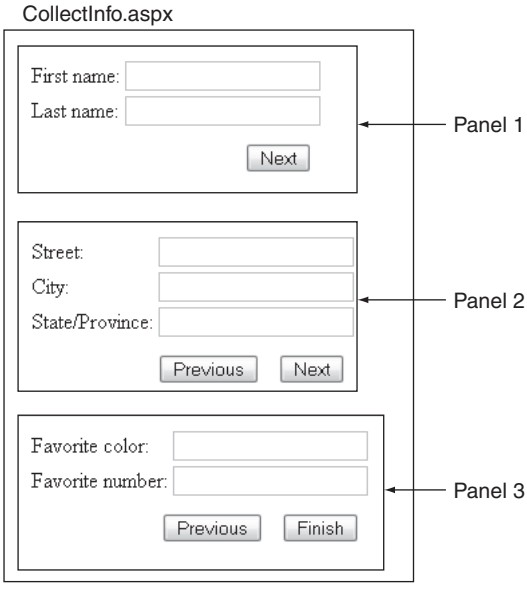

FIGURE 4-1: Multipanel page

not displayed, their state is maintained in view state, so programmatically it is just like working with one giant form. It is also quite efficient, since the contents of invisible panels are not even sent to the client browser; just the state of the controls is sent through view state.

Wizard Control

In the 2.0 release of ASP.NET this technique has been standardized in the form of the Wizard control. Instead of laying out the Panel controls yourself and adding the logic to flip the visibility of each panel in response to button presses, you can use the Wizard control to manage the details for you and focus on laying out the controls for each step. The control itself consists of a collection of WizardSteps which act as containers for any controls you want to add. Listing 4-18 shows a sample Wizard control populated with the input elements described in Figure 4-1 (also included is an adjacent Label control to display the data on completion).

LISTING 4-18: Sample Wizard control with three steps

```
<asp:Wizard ID="_infoWizard" runat="server" ActiveStepIndex="0"
            OnFinishButtonClick="_infoWizard_FinishButtonClick"
            DisplaySideBar="False">
    <WizardSteps>
      <asp:WizardStep ID="_step1" runat="server" Title="Name">
      <table>
        <tr>
          <td>First name:</td>
          <td><asp:TextBox ID="_firstNameTextBox" runat="server" /></td>
        </tr>
        <tr>
          <td>Last name:</td>
          <td><asp:TextBox ID="_lastNameTextBox" runat="server" /></td>
        </tr>
      </table>
      </asp:WizardStep>
      <asp:WizardStep ID="_step2" runat="server" Title="Address">
        <table>
          <tr>
            <td>Street:</td>
            <td><asp:TextBox ID="_streetTextBox" runat="server" /></td>
          </tr>
          <tr>
            <td>City:</td>
            <td><asp:TextBox ID="_cityTextBox" runat="server" /></td>
          </tr>
          <tr>
            <td>State/Province:</td>
            <td><asp:TextBox ID="_stateTextBox" runat="server" /></td>
          </tr>
        </table>
      </asp:WizardStep>
      <asp:WizardStep ID="_step3" runat="server" Title="Preferences">
        <table>
          <tr>
            <td>Favorite color:</td>
            <td><asp:TextBox ID="_colorTextBox" runat="server" /></td>
          </tr>
          <tr>
            <td>Favorite number:</td>
            <td><asp:TextBox ID="_numberTextBox" runat="server" /></td>
          </tr>
        </table>
      </asp:WizardStep>
    </WizardSteps>
  </asp:Wizard>
  <asp:Label ID="_summaryLabel" runat="server" />
```

Like most controls in ASP.NET, both the appearance and behavior of the Wizard control are extremely customizable. In the previous example, the control's SideBar portion, which generates a set of navigation hyperlinks on the left side to let the user jump between steps in the wizard without using the Next/Previous buttons, was not displayed. Figure 4-2 shows two different renderings of the Wizard control: the first is exactly how the Wizard control shown in Listing 4-18 would appear, and the second shows the same control with a different formatting applied and with the ShowSideBar attribute set to true. This control also supports templates so that you can completely customize the look and feel of it as desired.

The advantage of working with the Wizard control like this is that it manages all of the details of the sequential interaction with the user, and you can treat all of the input elements in each of the separate steps as if they were all part of a single page. For example, in our OnFinishButtonClick handler for the Wizard control we can easily use all of the data the user has entered. Listing 4-19 shows an example of printing a message back to the user in the form of a label and then hiding the Wizard control as an indicator that the input sequence is complete.

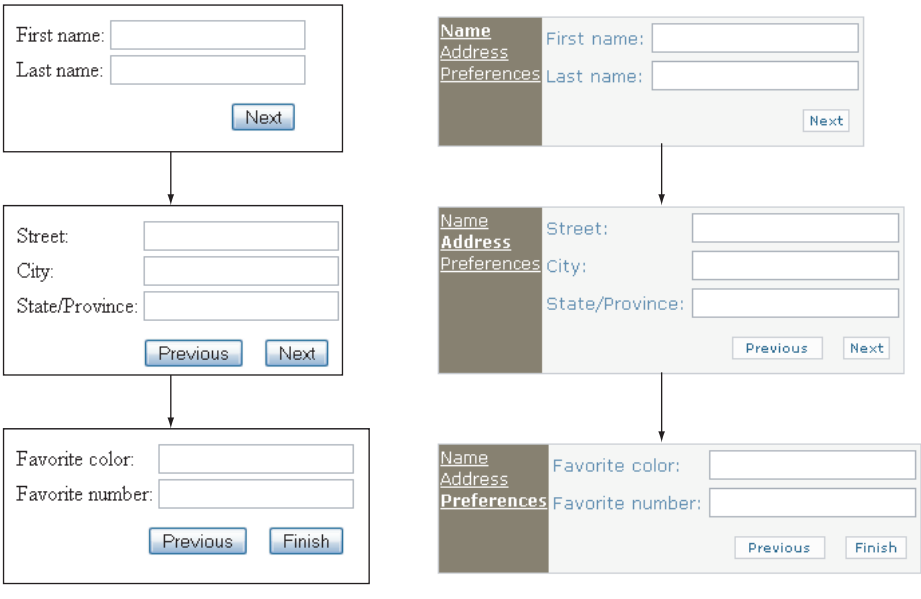

FIGURE 4-2: Wizard control, unadorned, and with SideBar and formatting

LISTING 4-19: Handler for the Wizard's Finish button click event

```
protected void _infoWizard_FinishButtonClick(object sender,
                                        WizardNavigationEventArgs e)
{
  _summaryLabel.Text = string.Format(
            "<h2>Thank you for submitting your information!</h2>" +
            "Name: {0} {1}<br /><br/>Address: {2}<br/>" +
            "{3}, {4}<br /><br />Prefs: {5} {6}<br />",
                        _firstNameTextBox.Text,
                        _lastNameTextBox.Text, _streetTextBox.Text,
                        _cityTextBox.Text, _stateTextBox.Text,
                        _colorTextBox.Text, _numberTextBox.Text);
  _infoWizard.Visible = false;
}
```

MultiView and View Controls

If you want the ability to toggle among multiple panels on a page but find the Wizard control too constraining, you might instead consider using the MultiView control. A MultiView control consists of several child View controls, and it maintains an active index indicating which of those child views should be visible. In fact, the WizardStep control used by the Wizard control inherits from the View class used by the MultiView, so the similarity is not a coincidence. Unlike the Wizard control, the MultiView renders nothing but the contents of the active view—there are no buttons, links, or titles of any kind. This means that it is up to you to determine how the user switches between the various views. Listings 4-20 and 4-21 show a sample MultiView with three embedded views to collect data from the user. This example uses three link buttons to let the user toggle among the three views by setting the ActiveViewIndex of the MultiView depending on which button was selected.

LISTING 4-20: MultiView with LinkButtons

```
        <asp:LinkButton ID="_view1LinkButton" runat="server"
                OnClick="_view1LinkButton_Click">
                View 1</asp:LinkButton>  
        <asp:LinkButton ID="_view2LinkButton" runat="server"
                OnClick="_view2LinkButton_Click">
                View 2</asp:LinkButton>  
        <asp:LinkButton ID="_view3LinkButton" runat="server"
                OnClick="_view3LinkButton_Click">
                View 3</asp:LinkButton><br />
```

continues

```
<asp:MultiView ID="_infoMultiView" runat="server"
               ActiveViewIndex="0">
    <asp:View ID="_view1" runat="server">
     <table>
        <tr>
          <td>First name:</td>
          <td><asp:TextBox ID="_firstNameTextBox"
                           runat="server" /></td>
        </tr>
        <tr>
          <td>Last name:</td>
          <td><asp:TextBox ID="_lastNameTextBox"
                           runat="server" /></td>
        </tr>
      </table>
    </asp:View>
    <asp:View ID="_view2" runat="server">
        <table>
          <tr>
            <td>Street:</td>
            <td><asp:TextBox ID="_streetTextBox"
                             runat="server" /></td>
          </tr>
          <tr>
            <td>City:</td>
            <td><asp:TextBox ID="_cityTextBox"
                             runat="server" /></td>
          </tr>
          <tr>
            <td>State/Province:</td>
            <td><asp:TextBox ID="_stateTextBox"
                             runat="server" /></td>
          </tr>
        </table>
    </asp:View>
    <asp:View ID="_view3" runat="server">
        <table>
          <tr>
            <td>Favorite color:</td>
            <td><asp:TextBox ID="_colorTextBox"
                             runat="server" /></td>
          </tr>
          <tr>
            <td>Favorite number:</td>
            <td><asp:TextBox ID="_numberTextBox"
                             runat="server" /></td>
          </tr>
        </table>
    </asp:View>
</asp:MultiView>
```

LISTING 4-21: LinkButton handlers for MultiView switching

```
protected void _view1LinkButton_Click(object sender, EventArgs e)
{
   _infoMultiView.ActiveViewIndex = 0;
}
protected void _view2LinkButton_Click(object sender, EventArgs e)
{
    _infoMultiView.ActiveViewIndex = 1;
}
protected void _view3LinkButton_Click(object sender, EventArgs e)
{
    _infoMultiView.ActiveViewIndex = 2;
}
```

Profile

Profile provides a simple way of defining database-backed user profile information. With just a few configuration file entries, you can quickly build a site that stores user preferences (or any other data, for that matter) into a database, all with a simple type-safe interface for the developer. In many ways, Profile looks and feels much like Session state, but unlike Session state, Profile is persistent across sessions and is also tied into the Membership provider, so authenticated clients have data stored associated with their real identities instead of some arbitrary identifier. Anonymous clients will have an identifier generated for them, stored as a persistent cookie so that subsequent access from the same machine will retain their preferences as well. In addition, Profile is retrieved on demand and written only when modified, so unlike out-of-process Session state storage, you only incur a trip to the database when you actually use Profile, not implicitly with each request.

Fundamentals

The first step in using Profile is to declare the properties you would like to store on behalf of each user in your web.config file under the <profile> element. Your first decision is whether you want to allow anonymous clients to store profile data or only authenticated clients. If you elect to support anonymous clients, you must enable anonymous identification by adding the anonymousIdentification element in your web.config file with its

enabled attribute set to true. This will cause ASP.NET to generate a unique identifier (a GUID) to associate with each anonymous user via a persistent cookie. If the user is authenticated, the membership identifier for that user will be used directly and no additional cookie will be created. You also have control over whether individual properties are stored on behalf of anonymous users through the allowAnonymous attribute of the add element. Listing 4-22 shows a sample web.config file with anonymous identification enabled, and three property declarations, one each for the user's favorite color, favorite number, and favorite HTTP status code. Note that all properties in this example allow anonymous access.

Listing 4-22: Defining three Profile properties in web.config

```
<configuration>
  <system.web>
     <anonymousIdentification enabled="true" />

     <profile enabled="true">
        <properties>
          <add name="FavoriteColor" defaultValue="blue"
                    type="System.String"
                    allowAnonymous="true" />

          <add name="FavoriteNumber" defaultValue="42"
                    type="System.Int32"
                    allowAnonymous="true" />

          <add name="FavoriteHttpStatusCode"
                    type="System.Net.HttpStatusCode"
                    allowAnonymous="true" serializeAs="String"
                    defaultValue="OK" />
        </properties>
    </profile>
  </system.web>
</configuration>
```

When ASP.NET compiles your site, it creates a new class that derives from ProfileBase with type-safe accessors to the properties you declared. These accessors use the Profile provider to save and retrieve these properties to and from whatever database the provider is configured to interact with. Listing 4-23 shows what the generated class would look like for the three profile properties defined in Listing 4-22.

LISTING 4-23: Generated ProfileCommon Class

```
public class ProfileCommon : ProfileBase {
    public virtual HttpStatusCode FavoriteHttpStatusCode {
        get {
            return ((HttpStatusCode)(this.GetPropertyValue(
                        "FavoriteHttpStatusCode")));
        }
        set {
            this.SetPropertyValue("FavoriteHttpStatusCode",
                         value);
        }
    }

    public virtual int FavoriteNumber {
        get {
            return ((int)(this.GetPropertyValue(
                                "FavoriteNumber")));
        }
        set {
            this.SetPropertyValue("FavoriteNumber", value);
        }
    }

    public virtual string FavoriteColor {
        get {
            return ((string)(this.GetPropertyValue(
                                    "FavoriteColor")));
        }
        set {
            this.SetPropertyValue("FavoriteColor", value);
        }
    }

    public virtual ProfileCommon GetProfile(string username)
    {
        return ((ProfileCommon)(ProfileBase.Create(
                            username)));
    }
}
```

The second thing that happens is ASP.NET adds a property declaration to each generated Page class in your site named *Profile*, which is a type-safe accessor to the current Profile class (which is part of the HttpContext), as shown in Listing 4-24.

Listing 4-24: Type-safe property added to Page-derived class for profile access

```
public partial class Default_aspx : Page
{
   protected ProfileCommon Profile {
      get {
          return ((ProfileCommon)(this.Context.Profile));
      }
   }
  //...
}
```

This lets you interact with your profile properties in a very convenient way. For example, Listing 4-25 shows a snippet of code that sets the profile properties based on fields in a form.

Listing 4-25: Setting profile properties

```
void enterButton_Click(object sender, EventArgs e)
{
  Profile.FavoriteColor = colorTextBox.Text;
  Profile.FavoriteNumber  = int.Parse(numberTextBox.Text);
  Profile.FavoriteHttpStatusCode = (HttpStatusCode)
                  Enum.Parse(typeof(HttpStatusCode),
                          statusCodeTextBox.Text);
}
```

If you look in the database used by the Profile provider (by default a local SQL Server 2005 Express database in your application's App_Data directory), you will see a table called aspnet_Profile with 5 columns:

```
UserId
PropertyNames
PropertyValuesString
PropertyValuesBinary
LastUpdatedDate
```

In the example shown in Listings 4-22, 4-23, and 4-25 these columns were populated with the following values:

```
405A7333-2C8D-4E63-AB56-BA54398D47DF
FavoriteColor:S:0:3:FavoriteNumber:S:3:2:FavoriteHttpStatusCode:S:5:16:
red42MovedPermanently
2006-1-1 09:00:00.000
```

So you can see that by default the Profile provider uses a string serialization with property names and string lengths carefully stored as well on

a per-user basis. In our example the user was anonymous, so a GUID was generated and used to index the property values in the aspnet_Profile table. The UserId column is actually a foreign key reference to the UserId column of the aspnet_Users table, where the membership system keeps user information (anonymous user information is also stored in this table).

Migrating Anonymous Profile Data

If your application supports both anonymous and authenticated clients, you may find that clients are frustrated when they store data as an anonymous user only to find it disappear when they log in and become authenticated. You can take steps to migrate their anonymous data to their authenticated identity by using the MigrateAnonymous event of the ProfileModule. This event, which you would typically add as a handler in global.asax, is triggered when an anonymous client with profile information transitions to an authenticated user. Listing 4-26 shows a sample global.asax file with a handler for this event transferring all profile state to the new profile data store for the newly authenticated client.

LISTING 4-26: Sample global.asax file migrating anonymous profile data

```
<%@ Application Language="C#" %>

<script runat="server">
  void Profile_MigrateAnonymous(object sender,
                                ProfileMigrateEventArgs e)
  {
    ProfileCommon prof = Profile.GetProfile(e.AnonymousID);
    Profile.FavoriteColor = prof.FavoriteColor;
    Profile.FavoriteNumber = prof.FavoriteNumber;
    Profile.FavoriteHttpStatusCode = prof.FavoriteHttpStatusCode;
  }
</script>
```

Note that the anonymous identifier previously associated with the client is available through the ProfileMigrateEventArgs parameter, and the actual profile for that identity is accessible using the static GetProfile method of the Profile class. In most cases it would be wise to prompt the user before migrating her anonymous data, since that user may have profile data already associated with her account and might elect to not have the data she entered as an anonymous client overwrite the data that was stored previously on her behalf.

Managing Profile Data

Once you start using Profile in a live site, you will quickly discover that the number of entries in your profile database can grow without bound, especially if you have enabled anonymous storage. To deal with this, there is a class called ProfileManager which you can use to periodically clean up the profile database. Listing 4-27 shows the core static methods of this class, which tie into the current Profile provider.

Listing 4-27: The ProfileManager class

```
public static class ProfileManager
{
  public static int DeleteInactiveProfiles(
         ProfileAuthenticationOption authenticationOption,
         DateTime userInactiveSinceDate);
  public static bool DeleteProfile(string username);

  public static ProfileInfoCollection FindProfilesByUserName(...);
  public static ProfileInfoCollection GetAllProfiles(...);
  public static int GetNumberOfInactiveProfiles(...);
  public static int GetNumberOfProfiles(...);

    //...
}
```

This class is accessible both in an ASP.NET Web application as well as in any .NET application that links to the System.Web.dll assembly. You can use the static methods in this class to build an administrative page in your site that lets the administrator clean up the profile database from time to time, perhaps giving her the option of specifying an inactive lower bound above which all profiles should be deleted (using the last parameter to DeleteInactiveProfiles method). If you prefer to automate the process, you could also write a Windows service that ran continuously on the server, deleting inactive profiles periodically, or perhaps a command line program that was run as part of a batch script periodically. Whichever technique you use is unimportant, but making sure you have a plan to clear out unused profile data from time to time is critical, especially if you allow anonymous clients to store data.

Storing Profile Data

The default Profile provider in ASP.NET 2.0 stores profile data in a local SQL Server 2005 Express database located in the App_Data directory of

your application. For most production sites, this will be insufficient, and they will want to store the data in a full SQL Server database along with the rest of the data for their application. You can change the default database used by the Profile provider class by changing the value of the LocalSqlServer connection string in your web.config file. Prior to doing this, you must ensure that the target database has the profile and membership tables installed, which you can do using the aspnet_regsql.exe utility. Running this utility without any parameters brings up a user interface which walks you through installing the schema into an existing database, or creating a new default database, aspnetdb, to store all of ASP.NET 2.0's application services (membership, roles, profiles, Web part personalization, and the SQL Web event provider).

You can also use the command line parameters to install the database in an automated fashion (for example, if you are writing an install script for your application). For instance, to install all of the ASP.NET 2.0 application services into a new database named aspnetdb on the local machine (using Windows credentials to access the database), you would run the command:

```
aspnet_regsql -A all -C server=.;database=aspnetdb;trusted_connection=yes
```

Then, to change your ASP.NET application to use this new database to store Profile data (along with all other Application Service data), you would remove the LocalSqlServer connection string and then add it with a connection string pointing to your new database, as shown in Listing 4-28.

LISTING 4-28: Changing the default database for Profile storage

```
<configuration>
 <connectionStrings>
  <remove name="LocalSqlServer" />
  <add name="LocalSqlServer"
       connectionString=
               "Server=.;Database=aspnetdb;trusted_connection=yes"/>
 </connectionStrings>
 <!--...-->
```

Serialization

As you saw earlier, the default serialization for properties stored in Profile is to write them out as strings, storing the property names and substring

indices in the PropertyNames column. You can control how your properties are serialized by changing the serializeAs attribute on the add element in web.config. This attribute can be one of four values:

```
Binary
ProviderSpecific
String
Xml
```

The default is ProviderSpecific, which might better be called TypeSpecific since the type of the object will determine the format of its serialization. ProviderSpecific with the default SQL Provider implementation writes the property as a simple string if it is either a string already or a primitive type (int, double, float, etc.). Otherwise it defaults to XML serialization, which is a natural fallback because it will work with most types (even custom ones) without any modification to the type definition itself. So what ProviderSpecific really means is StringForPrimitiveTypesAndStringsOtherwiseXml, which is obviously quite a mouthful, so it's understandable they went with something shorter. This can lead to some confusing behavior if you're not aware of it, however. For example, consider the two Profile property definitions shown in Listing 4-29.

Listing 4-29: Sample Profile property definitions with invalid default values

```
<add name="TestCode" type="System.Net.HttpStatusCode"
     defaultValue="OK" /> <!-- defaultValue invalid -->
<add name="TestDate" type="DateTime"
     defaultValue="1/1/2006"/> <!-- defaultValue invalid -->
```

After using integer and string profile properties, adding an enum and a DateTime in this manner seems reasonable. Because the default serialization is ProviderSpecific, we now know that these two types will be serialized with the XmlSerializer, so specifying default values as simple strings is not going to fly (as you will find out quickly once you try accessing the properties). You have two ways of dealing with this problem. One is to specify the XML-serialized value directly in the configuration file (taking care to escape any angle brackets), as shown in Listing 4-30.

Listing 4-30: Specifying XML-serialized default values

```
<add name="TestCode" type="System.Net.HttpStatusCode"
     defaultValue=
        "&lt;HttpStatusCode&gt;OK&lt;/HttpStatusCode&gt;" />
```

```
<add name="TestDate" type="DateTime"
     defaultValue=
            "&lt;dateTime&gt;2006-01-01&lt;/dateTime&gt;" />
```

The other (and perhaps more appealing) option is to change the serialization from ProviderSpecific (which we know turns into XML) to String. String serialization only works for types that have TypeConversions defined for strings, which in our case is true since both enums and the DateTime class have string conversions defined (we discuss how to write your own string conversions in the next section). If you look carefully at Listing 4-22, you will notice that it specifies a serializeAs attribute of String for the HttpStatusCode so that a simple string default value of "OK" could be used, as shown in Listing 4-31.

LISTING 4-31: Specifying string default values

```
<add name="TestCode" type="System.Net.HttpStatusCode"
     serializeAs="String"
     defaultValue="OK" />
<add name="TestDate" type="DateTime"
     serializeAs="String"
     defaultValue="2006-01-01" />
```

The other option for serialization is Binary, which will use the BinaryFormatter to serialize the property. With the default SQL provider, this will write the binary data into the database's PropertyValuesBinary column. This is a useful option if you want to make it difficult to tweak the profile values directly in the database, or if you are storing types whose entire state is not properly persisted using the XmlSerializer (classes with private data members that are not accessible through public properties fall into this category, for example). Before you can use the binary option, the type being stored must be marked with the Serializable attribute or must implement the ISerializable interface. Keep in mind that selecting the binary serialization option makes it impossible to specify a default value, so it is typically used only for complex types for which a default value doesn't usually make sense anyway. If you ever do need to specify a default value for binary serialization, it is technically possible by base64 encoding a serialized instance of the type and using the resulting string in the defaultValue property.

User-Defined Types as Profile Properties

One of the advantages of the Profile architecture is that it is generic enough to store arbitrary types and, as we have seen, it supports several different persistence models. This means that it is quite straightforward to write your own classes to store user data and then store the entire class in Profile. Suppose, for example, we wanted to provide a shopping cart for users to let them collect items to purchase. We might write a class to store an individual item containing a description and a cost, and another class that keeps a list of all of the items in the current cart as well as exposing a property that calculates the total cost of all items in the cart, as shown in Listing 4-32.

Listing 4-32: Sample ShoppingCart class definition

```
namespace PS
{
    [Serializable]
    public class ShoppingCart
    {
        private List<Item> _items = new List<Item>();

        public Collection<Item> Items
        {
            get { return new Collection<Item>(_items); }
        }

        public float TotalCost
        {
            get
            {
                float sum = 0F;
                foreach (Item i in _items)
                    sum += i.Cost;
                return sum;
            }
        }
    }

    [Serializable]
    public class Item
    {
        private string _description;
        private float  _cost;

        public Item() : this("", 0F) { }

        public Item(string description, float cost)
```

```
        {
            _description = description;
            _cost = cost;
        }

        public string Description
        {
            get { return _description;  }
            set { _description = value; }
        }

        public float Cost
        {
            get { return _cost;  }
            set { _cost = value; }
        }
    }
}
```

Note that our classes are marked with the [Serializable] attribute in anticipation of using binary serialization (although the XmlSerializer will work fine with these classes as well, so we have both options at our disposal). We can then add a profile property of type ShoppingCart to our collection, and we have a fully database-backed per-client persistent shopping cart implemented!

```
<profile enabled="true">
  <properties>
    <add name="ShoppingCart" type="PS.ShoppingCart"
         allowAnonymous="true" />
  </properties>
</profile>
```

Using the shopping cart in our application is as simple as accessing the ShoppingCart property in Profile and adding new instances of the Item class as needed (the sample available for download has a complete interface for users to shop using this class as the storage mechanism).

```
Profile.ShoppingCart.Items.Add(
         new Item("Chocolate covered cherries", 3.95F));
```

Optimizing Profile

You may be wondering at this point what sort of cost is incurred by leveraging Profile to store your per-client data, especially if you start using

complex classes like the ShoppingCart, which may end up storing significant amounts of information on behalf of each user. Those of you who have taken advantage of the SQL Server-backed Session state feature introduced in ASP.NET 1.0 may be especially leery, since by default each request for a page incurred two round trips to the state database to retrieve and then flush session from and to the database. The good news is that by default, the profile persistence mechanism is reasonably efficient. Unlike out-of-process Session state, it performs lazy retrieval of the profile data on behalf of a user (loading on demand only), and only writes the profile data back out if it has changed.

Unfortunately, if you are storing anything besides strings, DateTime classes, or primitive types, it becomes impossible for the ProfileModule to determine whether the content has actually changed, and it is forced to write the profile back to the data store every time it is retrieved. This is obviously true for custom classes as well, so be aware that adding any types beside string, DateTime, and primitives will force Profile to write back to the database at the end of each request that accesses Profile. Internally there is a dirty flag used to keep track of whether a property in Profile has changed or not. You can explicitly set the IsDirty property for a profile property to false. If you do this for all properties associated with a specific provider instance, then when that provider instance is asked to save the profile data, it will see that all the properties passed to it are not dirty and it will skip communication with the database. This approach relies on knowledge of the underlying SettingsBase, SettingsProperty, and SettingsPropertyValue types (all in System.Configuration). For a profile property called Nickname, you could force it to not be considered dirty, as shown in Listing 4-33.

Listing 4-33: Setting the IsDirty attribute for a property in a custom class

```
Profile.PropertyValues["Nickname"].IsDirty = false;
```

Note that you can disable automatic profile saving using the automaticSaveEnabled attribute on the <profile/> element in the configuration file (this attribute defaults to true). You can set automaticSaveEnabled to false to stop ProfileModule from storing the Profile on your behalf automatically. It is then up to you to call Profile.Save if you want to store data back to the database. Alternatively, you can hook the ProfileModule's ProfileAutoSaving

event. If you set the ContinueWithProfileAutoSave property on the event argument to false, then the ProfileModule will not call Profile.Save.

As you saw earlier, it is possible to specify String, Binary, or Xml as the serialization mechanism for your properties. If you are storing your own custom classes like our ShoppingCart example, you can take steps to reduce the amount of space used to store instances of your class in one of two ways: writing your own TypeConverter for the class to support conversion to string format, or implementing the ISerializable interface to control the format of the binary data used by the BinaryFormatter. Listing 4-34 shows the default serialization of our ShoppingCart class with four items in it in XML format. The equivalent binary serialization occupies 678 bytes of space.

Listing 4-34: XML-serialized shopping cart with four items (590 characters)

```
<?xml version="1.0" encoding="utf-16"?>
<ShoppingCart xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Items>
<Item>
  <Description>Chocolate covered cherries</Description>
  <Cost>3.95</Cost>
</Item>
<Item>
  <Description>Toy Train Set</Description>
  <Cost>49.95</Cost>
</Item>
<Item>
  <Description>XBox 360</Description>
  <Cost>399.95</Cost>
</Item>
<Item>
  <Description>Wagon</Description>
  <Cost>24.95</Cost>
</Item>
  </Items>
</ShoppingCart>
```

By default, you cannot use the serializeAs="String" option for custom types, since there is no way to convert the types to and from a string format in a lossless way. You can provide such a conversion yourself by implementing a TypeConverter for your class. This involves creating a class that inherits from TypeConverter, implementing the conversion methods, and then annotating your original class with the TypeConverter attribute that

associates it with your conversion class. You must also decide on how to persist your class as a string (and then parse it from a string), which can be a nontrivial task, so make sure it's worth the effort before taking this approach. As an example, here is a TypeConverter class for the Item class that represents items in our shopping cart. In this case I chose to use a non-printable character as a delimiter, and since the Item class consists of two pieces of state which are easily rendered as strings, the parsing becomes trivial using the Split method of the string class. The converter class is then associated with the Item class using the TypeConverter attribute, both of which are shown in Listing 4-35.

Listing 4-35: Writing a custom type converter

```
public class ItemTypeConverter : TypeConverter
{
    private const char _delimiter = (char)10;

    public override object
         ConvertFrom(ITypeDescriptorContext context,
                CultureInfo culture, object value)
    {
        string sValue = value as string;
        if (sValue != null)
        {
            string[] vals = sValue.Split(_delimiter);
            return new Item(vals[0],
                        float.Parse(vals[1]));
        }
        else
            return base.ConvertFrom(context,
                                culture, value);
    }

    public override object
           ConvertTo(ITypeDescriptorContext context,
                       CultureInfo culture,
                    object value, Type destinationType)
    {
        if (destinationType == typeof(string))
        {
            Item i = value as Item;
            return string.Format("{0}{1}{2}",
                    i.Description, _delimiter, i.Cost);
        }
        else
        {
```

```
            return base.ConvertTo(context, culture,
                    value, destinationType);
        }
    }

    public override bool CanConvertFrom(
                        ITypeDescriptorContext context,
                        Type sourceType)
    {
        if (sourceType == typeof(string))
            return true;
        else
            return base.CanConvertFrom(
                         context, sourceType);
    }

    public override bool CanConvertTo(
                        ITypeDescriptorContext context,
                        Type destinationType)
    {
        if (destinationType == typeof(string))
            return true;
        else
            return base.CanConvertTo(
                        context, destinationType);
    }
}

[Serializable]
[TypeConverter(typeof(ItemTypeConverter))]
public class Item
{
  ...
```

With these classes in place, our Item class can be used with string serialization in a profile property. Note that for our shopping cart to be completely serializable as a string, we also need to write a type converter for our ShoppingCart class, a sample of which can be found in the downloadable samples for this book. The advantage of controlling the persistence in this way is that the serialization of the same shopping cart filled with four items now only takes 79 characters!

Going the Custom Route

Any time you find yourself spending a lot of time trying to make an architecture do what you want, it is important to step back and make sure that

the work necessary to customize the architecture to do what you want is less than what it would take to do it entirely yourself. Profile is a great example of a feature that is convenient and easy to use but that may be too constraining as your design evolves. Let's look at what features Profile specifically gives us.

- Support for anonymous and authenticated clients
- Anonymous users identified through a new cookie (or alternatively through embedded id with URL mangling, including support for autodetect cookieless mode)
- Arbitrary type storage, strongly typed through configuration file
- Per-client persistent data store
- Management class for cleaning up unused profile data

One of the drawbacks to using Profile to store client data is that it stores all of the data in one column (or two columns if you are using both string and binary serialization) of the database table. This means that it is practically impossible to make modifications to the profile data without going through the profile API. It's also impractical to generate any reports from the data or otherwise collect information from the database directly.

If you find yourself wanting more control over the storage of per-client state in your application, you have two choices: build a custom Profile provider or forget Profile and just write data yourself. Building a custom Profile provider gives you the ability to retarget where Profile actually writes the data, but because of the nature of the provider interface, it doesn't really make it any easier to write property values to specific columns in a table. For more information and samples on building custom Profile providers, take a look at the ASP.NET provider model toolkit (http://msdn.microsoft.com/asp.net/downloads/providers/default.aspx).

If you decide to forget Profile and just write the serialization of client data yourself, be aware that you can still leverage the identification features of Profile even if you aren't using the storage features. Specifically, there is a UserName property on the ProfileBase class that will contain either the name of the current authenticated user or the GUID that was generated for an anonymous user. You can use this UserName property as a unique index into a custom database table of your own construction to

easily store and retrieve user data. Just make sure that Profile and anonymousIdentification are enabled in your application, and you can use the same client identification mechanism as Profile.

```
<anonymousIdentification enabled="true"/>
        <profile enabled="true" />
```

By writing your own client persistence backend using the unique identifier provided by Profile, you gain several unique advantages over the generic profile implementation.

- The ability to write stored procedures against client data
- The ability to retrieve only the portions of data you need for a client at any given time (instead of relying on Profile to just load the whole chunk into memory)
- The ability to cache per-client data across requests for efficiency
- Complete control over the serialization, and the ability to map onto existing tables instead of creating new data stores that you may already have in place

The sample available for download contains an alternate implementation of the shopping cart described earlier, using a custom database table to store cart items and leveraging the unique client identifier available through the ProfileBase class. In general, you may even consider starting out by using Profile for convenience to get things started, and then later migrate some of the profile data into custom tables with a separate data access layer. In this sense, Profile fills a convenient role as an easy way to store per-client data, with an obvious path forward to factoring data out into a more strongly typed schema.

SUMMARY

With the reintroduction of cross-page posting and the introduction of Profile and the Wizard, View, and MultiView controls to the ASP.NET developer's toolbox, ASP.NET 2.0 should make the discussion of where to store client state in Web applications even more interesting. Cross-page posting brings back the common technique of parsing the POST request from a

source page in a different target page. The Wizard, MultiView, and View controls provide an easy-to-use implementation of a common technique of showing and hiding parts of a page as the client interacts with it. Profile gives developers a complete solution for persisting client data across sessions, for both authenticated and anonymous users.