Building Dynamic Web Portals

ASP.NET 2.0 Web Parts IN ACTION

SAMPLE CHAPTER

Darren Neimke

Foreword by Andres Sanabria

MANNING

ASP.NET 2.0
Web Parts in Action
by Darren Neimke
Chapter 2

brief contents

CHAPTER 2

Web parts: the building blocks of portals

2.1 INTRODUCTION

You may already be acquainted with web parts and web part controls, and if you've worked with products such as SharePoint you have already used them quite extensively. (If these topics are not familiar, please dip into chapter 1 for a quick refresher.) Before you start this chapter, I recommend that you whet your appetite by seeing real examples of how web part controls allow users to customize the look and feel of web applications. To do so, visit http://Start.com.This is a web-based portal created by Microsoft Research. Notice how it provides a variety of methods to customize the page, by adding web parts and configuring them. For example, while at that page you could add a web part to display the current news from one of a dozen news providers, or display the weather for any city in the world. If you're like me, you could lose many hours configuring that page to exactly suit your fancy. And guess what—once you've done it, it's *your* page! Every time you visit that page, all the web parts you added, configured, and moved around will be there for you—just the way you left them.

In chapter 1 we learned that web parts are an integral aspect of a portal application. This chapter supplies a more exhaustive understanding of web parts, in particular how they are implemented in ASP.NET 2.0 portal applications.

This chapter is the first step in a journey through the properties, methods, and operations of web parts. In it we'll learn how to add custom verbs to web parts and how to customize their look and feel through themes and other customization techniques. We'll also delve a little deeper into the subject to see how web parts implement various interfaces that expose their behavior to the rest of the ASP.NET web portal framework. By the end of this chapter we'll be well on our way to understanding how the http://Start.com website achieves its magic. Let's begin!

2.2 EXPLORING WEB PARTS

Web parts are informational components, such as news updates or comics, that are added to web pages; and as such, web parts can be considered the primary building blocks of a portal application that displays dynamic content. In ASP.NET 2.0 we are provided with the `WebPart` server control for working with web parts. The `WebPart` server control comes pre-packaged with many properties and methods needed to use it in a variety of ways to show dynamic content to users.

The composition of web parts

A web part is generally rendered with a title bar, a border, and a body for displaying its dynamic content. The web part is manipulated by a web control that allows a user to work with and customize the web part. For example, users can set the web part's height or width and provide it with a title and description. Other manipulations of web parts are accessed through small menu-like items known as verbs. These operations include performing tasks such as closing the web part, minimizing it, or connecting it to other web parts on the page. Figure 2.1 shows the basic elements that make up a standard web part control.

In figure 2.1 we can see the various sub-elements that combine to form a web part. Throughout this chapter we will learn more about each of these elements and see how to use the `WebPart` class to modify or affect each different element. Understanding how to gain access to each of these elements will take us a long way down the path towards having full control over how our web parts are displayed to the end-users of our portal.

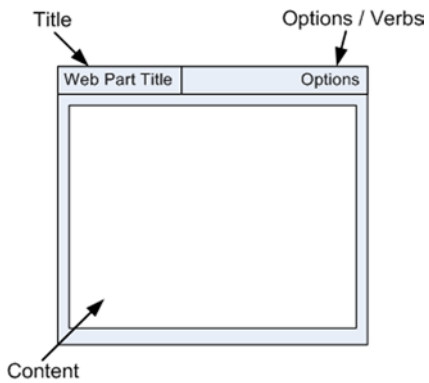

Figure 2.1 The sub-elements of a WebPart control.

Categorizing web parts

So far you've bumped into lots of buzzwords and phrases about web parts; but aside from a minor example, we have yet to learn exactly what they are. Are they simply common ASP.NET server controls? Or are they more like ASP.NET user controls? Do they ship as part of the core set of ASP.NET server controls? Answering these questions is logically the best place to begin our exploration of the ASP.NET `WebPart` control.

The simple example application that we built in chapter 1 showed that when we added standard ASP.NET server controls, the `Calendar` and the `Label`, to a web part zone, a transformation magically occurred—they became web parts. We saw that suddenly those controls had verbs, they now had titles, and they had properties that could be edited at runtime via the editor parts. To answer the question of "what is a web part?" we need to do some investigative work and discover more of the magic that turned those standard server controls into web parts!

2.2.1 Discovering the GenericWebPart control

In order to understand what happened to the `Label` and `Calendar` controls in the previous chapter, we're going to need to put our debugging skills to the test and determine how all these web part controls interact with each other. To do that we'll set up a web page and add web part controls to it. After that we'll add code to the page that will access those parts. Finally, when we attach the debugger to the code and run it we'll have a clear view of the state of those objects at runtime. Completing those steps will give us a much clearer view of how the transformation occurred.

NOTE This book is designed to be very hands-on, and as such we will frequently be writing small pages to try new things as we learn about them. Therefore, it would be a good idea to create a new ASP.NET project in Visual Studio that can be used to create ad-hoc web pages for testing. For the remainder of this book I'll refer to this test project as "your test project."

Creating a web page

Open your test project and create a new web page named `GenericWebPartTest.aspx`. This page will be used to learn how these web controls are turned into web parts. As with all web pages that contain web parts, we must start by adding a `WebPartManager` control at the top of the page. Finally, we can add a `WebPartZone` and add a `Label` control to it. With that, our code should look like the following snippet:

```
<asp:WebPartManager ID="WebPartManager1" runat="server" />

<asp:WebPartZone ID="WebPartZone1" runat="server">
  <ZoneTemplate>
    <asp:Label id="ctl1" runat="server" Text="I'm a label" />
  </ZoneTemplate>
</asp:WebPartZone>
```

All that we know at this stage is that at some time between now and when this page is displayed in a browser, the `Label` control that we added to the web part zone starts looking and behaving like a web part. We know this because we saw that the label gained two characteristics common to web parts—a Title and Verbs—when it was contained within the web part zone in the previous chapter. Next we'll write code that allows us to view the state of the page at runtime. Switch into the code section of the web page, enter the code, and set the debugging breakpoints that are displayed in figure 2.2.

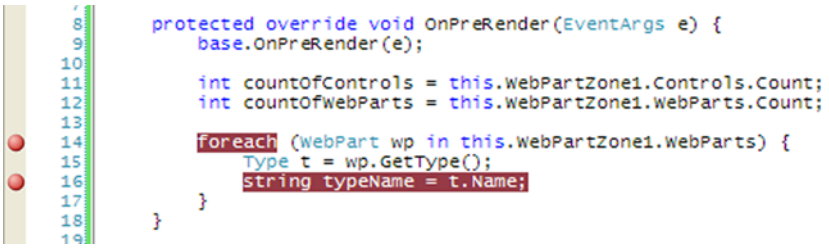

Figure 2.2 Code and debugging breakpoints are used to inspect the state of controls on our page during the pre-rendering phase of the page lifecycle.

After an ASP.NET has been processing a good while, it enters into a phase that is referred to as the pre-rendering phase of the page. During the pre-rendering phase of the page, the `OnPreRender` event handler in the page is called by the ASP.NET engine, which causes code in our method to be executed. At this time, page execution will halt at the breakpoints that we've set. When the execution halts at runtime, we can inspect the state of the controls on the page at runtime. The control that we are interested in is `WebPartZone1`.

A `WebPartZone` control has two properties that are of particular interest to us because they will provide information regarding where our `Label` control is, and what type of control it is. These properties are called "Controls" and "`WebParts`" and, as you may guess from their names, they contain collections of controls that are contained by the web part zone. Logic would have us believe that the `Label` control will turn up in the `WebPartZone`'s "Controls" collection because it is a web control but, from what we've observed, it's pretty easy to expect that it will show up in the "`WebParts`" collection instead. Let's run the page and see. From the Debug menu in Visual Studio, press Start Debugging to run the page and attach the debugger to it.

Debugging the page

The page runs and stops at the breakpoints, allowing us to place our mouse over the breakpoints and inspect the state of each variable. Hovering over the `countOfWebParts` variable that we declared indeed does prove that the zone now contains one web part control as shown in figure 2.3.

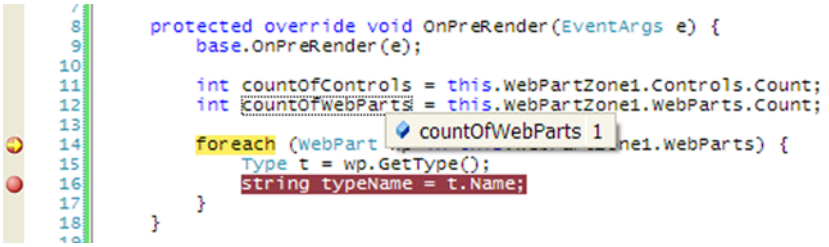

Figure 2.3 While stepping through code in debug mode you can place the mouse cursor above variables to display their state.

Similarly, hovering over the `countOfControls` variable will display zero for its value. This tells us that the web part zone now believes that it contains no web controls. Hover over the last breakpoint and you'll see that the name of the Type of the web part is `GenericWebPart` as displayed in figure 2.4.

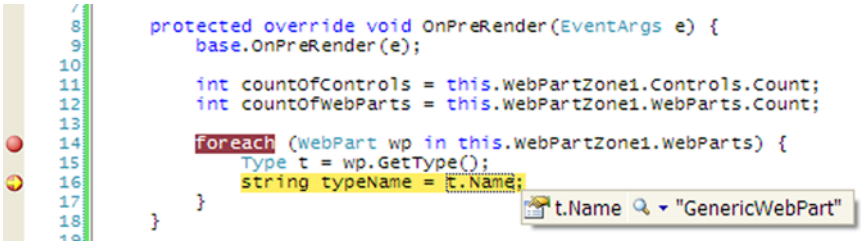

Figure 2.4 Displaying the Type of the web part at runtime shows that it is no longer a Label but is now a GenericWebPart control.

With the page still in debug mode, right-click on the `wp` variable and choose the "Quick Watch" menu option from the resulting context menu. In response, the Quick Watch dialog is displayed for the `wp` variable allowing us to see the values for all of its properties. Figure 2.5 displays the Quick Watch dialog with many of the properties for the `wp` variable shown.

In the Quick Watch dialog we can see that the `GenericWebPart` has a large number of properties that do not belong to the `Label` class but, instead are members of the `WebPart` class. These properties include: "`IsClosed`," "Title," "`CatalogIconImageUrl`," and "`ChromeState`" to name just a few. We see that there is also a property named "`WebBrowsableObject`" and that it is currently displaying a value of {Text = "I'm a label"}—the very same text value that we assigned to the original label in the mark-up for the page! This is quite a significant discovery, because it tells us that the very same `Label` control that we added to our `WebPartZone` earlier has been replaced with a new Type of control and has been wrapped by the `WebBrowsableObject` property. We can actually still get at the underlying `Label` control via the `ChildControl` property of the `GenericWebPart`—as demonstrated by the

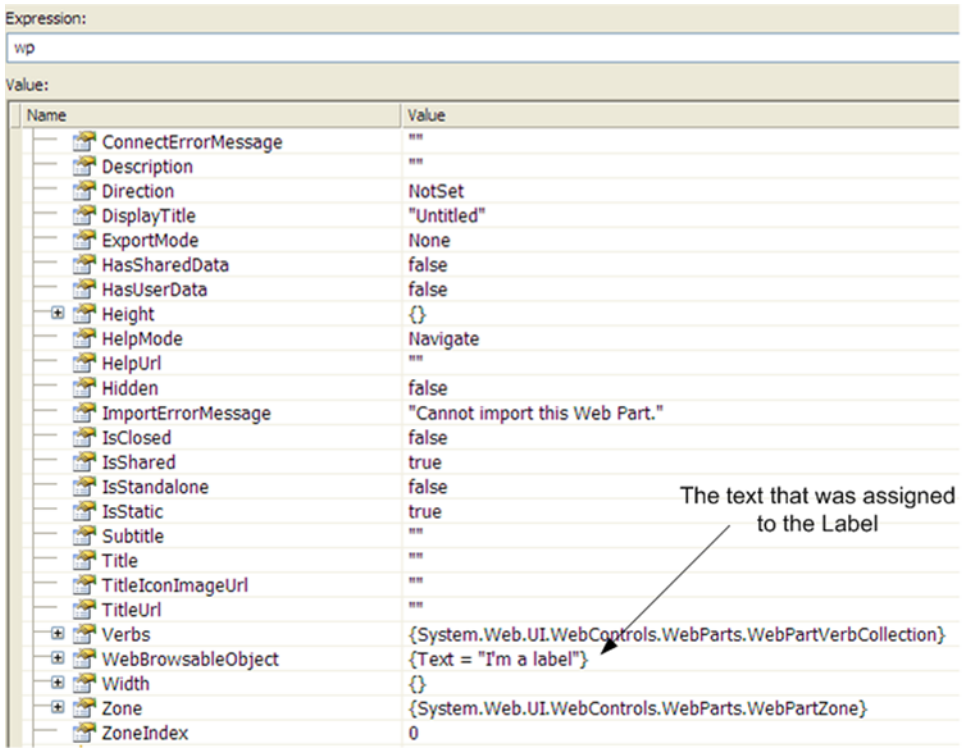

Figure 2.5 The Visual Studio 2005 Quick Watch window displays the runtime values of objects.

code in listing 2.1. Why would we want to get at the underlying control of the generic web part? One reason might be that we are using a control such as a `GridView` as the child control and we need to access the `GridView` after a page postback to re-bind it to a data source control.

Listing 2.1 The ChildControl property of the GenericWebPart provides access to the underlying web control that is wrapped by the GenericWebPart.

```
protected override void OnPreRender(EventArgs e) {
    base.OnPreRender(e);

    int countOfControls = this.WebPartZone1.Controls.Count;
    int countOfWebParts = this.WebPartZone1.WebParts.Count;

    foreach (WebPart wp in this.WebPartZone1.WebParts) {
        if (wp is GenericWebPart) {
            Type t = ((GenericWebPart)wp).ChildControl.GetType();
```

Cast the web part to a GenericWebPart to get at the ChildControl property

```
            string typeName = t.Name;  <------ Returns "Label"
        }
    }
}
```

Having viewed the page at runtime, we saw that the portal framework elevated the label to the status of a web part by enclosing it within a `GenericWebPart` wrapper before adding it to the zone. This occurs for all non-web part controls that are added to web part zones, including user controls. Being able to create user controls and have them treated as web parts makes it possible to create web parts very rapidly and easily compared to the alternative—which is to create web parts directly by inheriting from the `WebPart` class. To get a feel for some of the differences, let's take a look at what's involved when we create web parts by inheriting from the `WebPart` class. In doing so we'll better understand how to work directly with the `WebPart` class and we'll also get to see how web parts are rendered.

2.3 UNDERSTANDING THE WEBPART CLASS

Up until now, all the web parts that we've seen have been created by simply adding server controls—such as the `Calendar` and `Label`—to zones within the page. Now it's time to learn about another kind of web part control—a custom server control that directly inherits from the abstract `WebPart` base class. In this section we will create a web part by inheriting directly from the `WebPart` class and then learn how to add our custom web part to a web page by registering our custom web part class with the page.

Dragging user controls onto zones and having them treated as web parts is fine when the web parts do not need to be shared outside a single application; but part of the power of web parts is that, by their very nature, they lend themselves well to being reused in more than just a single portal application. For reuse, user controls cannot surpass custom controls, because custom controls can be compiled into very specific assemblies and easily shared between applications. If you need to share web parts between your own applications, or indeed, package them for reuse by third parties, then you will want to create custom controls so that you can take advantage of the packaging of assemblies.To do this you will need to create custom classes that derive directly from the `WebPart` class.

The `WebPart` class lives in the `System.Web.UI.WebControls.WebParts` namespace and serves as the base class for all web part controls. In fact, if you look at the definition of `GenericWebPart`, you will see that it does in fact inherit from the `WebPart` class. The `WebPart` class itself inherits from a base class named `Part`. The `Part` class has the basic properties that are relevant for all web part "parts"—including editor parts and catalog parts such as description, title and a few others.

2.3.1 Using custom controls

We've just seen that, when creating web parts, we have two options. The first option is to create a class that derives directly from the `WebPart` class, and the second option

is to drop user controls or server controls onto a web zone and have the framework place those controls within a generic wrapper known as a `GenericWebPart`. Let's now take a look at how to create web parts using each of these two methods.

The first web part that we'll build is a custom server control that inherits directly from the `WebPart` class. This will allow us to see for the first time what's involved in getting a web part up and running using that method. The web part that we will build will be a weather web part to display the weather for a variable number of days. It will have a property that allows us to specify how many days of weather to display. This is a good example of the sort of informational web part that would appear on a typical portal. Figure 2.6 displays what the weather web part control will look like when displayed in a browser.

Figure 2.6 This custom web part control displays random weather information for a variable number of days.

Creating a weather web part

To start creating the weather web part, open your test web project and add a new class file named "CustomWeatherPart.cs" to the `WebPart` folder and then derive that class from `WebPart`. At this point your class should look similar to the following snippet:

```
namespace WebPartTests {

    public class CustomWeatherPart : WebPart {

                public CustomWeatherPart() { }
    }
}
```

To allow users to set the number of days of weather to be displayed, we'll add a property named `NumberOfDays`. By default we'll set it to a value of 4 so that at least that number of days of weather will be displayed—even before the user has had time to configure it to be some other value. We can store the value for this property in `ViewState` so that it is persisted even after multiple trips between the browser and the web server (postbacks). The last bit of logic to be added ensures that the user cannot accidentally set the number of days to a value greater than 10 or less than 1. The code for our property is shown in listing 2.2, with all of its logic in place.

Listing 2.2 The NumberOfDays property contains the number of days of weather that should appear in the web part.

```
public int NumberOfDays {
        get {
                if (ViewState["NumberOfDays"] == null) {
                        return 4;
                } else {
```

```
                return (int)ViewState["NumberOfDays"];
            }
        }
        set {
            if (value < 1 || value > 10) {
                ViewState["NumberOfDays"] = 4;
            } else {
                ViewState["NumberOfDays"] = value;
            }
        }
}
```

The `NumberOfDays` property can now be used to specify how many days of weather our control will display; and the constraints that were added to the property ensure that there will always be some number of days to display weather for.

Rendering custom controls

When you work with custom server controls, one of the problems you face is that user interface elements are created by using code, as opposed to standard HTML and markup. Displaying a control in code requires creating every facet of it, including its style information in code. This is not the case when using user controls because you can use Visual Studio's design-time tools to simply drag controls from the toolbox onto the surface of the user control. What's more, once the controls are on the design surface, the developer can use design time tools, such as property editors and other wizards that exist within Visual Studio, to develop and maintain the properties of the control.

You create user interface elements for custom server controls by writing code that runs during the control's `Render` method. The `Render` method is a method that is common to all web controls and is called by the ASP.NET runtime to display every web control. In the `Render` method you write your user interface elements directly into an `HtmlTextWriter` object that is passed in as a parameter to that method by the ASP.NET Framework.

NOTE The `Render` method is a virtual (overridable) member of the `System.Web.UI.Control` class. This is a class that all server controls inherit from.

For our custom weather web part control, we'll use some logic to randomly produce a weather result for a variable number of days, and then create a weather image for each day of weather. In reality you would likely have some back-end process—such as a web service—that would return actual weather results. Add the code shown in listing 2.3 to your class and build the project to see that everything compiles.

Listing 2.3 The Render method for the weather web part displays a series of random weather images.

```
private enum WeatherType {                          <-- Create enum to use in code
    Sunny = 0,
    Rainy = 1,
    Cloudy = 2,
    Unknown = int.MaxValue
}


protected override void Render(HtmlTextWriter writer) {

    Random rand = new Random();


    for (int i = 0; i < this.NumberOfDays; i++) {

        int weatherValue = rand.Next(1000) % 3;     <-- Create random number between 0 and 3
        WeatherType todaysWeather = (WeatherType) weatherValue;

        Image img = new Image() ;                   <-- Choose one of 4 images based on result of random number
        img.ImageUrl =
            string.Format("~/images/{0}.gif", todaysWeather.ToString()) ;
        img.AlternateText = "Today's weather";

        writer.AddStyleAttribute(HtmlTextWriterStyle.TextAlign, "center");
        writer.RenderBeginTag(HtmlTextWriterTag.Div);
        img.RenderControl(writer);                  <-- Render weather image
        writer.WriteBreak();
        writer.Write(todaysWeather.ToString());
        writer.RenderEndTag(); // end Div
    }
}
```

In the `Render` method we have constructed a simple loop that will run for as many days as we have weather to display and, within each loop, a weather picture is produced and rendered.

NOTE The `HtmlTextWriter` that we've used to render our weather web part is a customized text writer that simplifies the task of writing HTML and is also capable of rendering specific output based on the device that is targeted. For example, when the page is visited by an older browser, the `HtmlTextWriter` will automatically emit down-level markup.

Adding custom controls

That completes the creation of the custom web part; all that remains is to create a web page to contain and display it—and from now on we'll refer to web pages that contain web parts as "web part pages," to distinguish them from ordinary web pages.

Add a new web page to your project named `CustomWeather.aspx` and, as with all web part pages, add a `WebPartManager` to it. You must also declare the server control to the page by using a Register directive at the head of the page as shown here:

```
<%@ Register TagPrefix="wp" Namespace="WebPartTests" %>
```

When you have registered your controls to the page you can then reference your custom web part within a `ZoneTemplate`—making sure to use the same tag prefix as declared in the register directive:

```
<asp:WebPartZone ID="WebPartZone1" runat="server">
        <ZoneTemplate>
                <wp:CustomWeatherPart
                        ID="CustomWeatherPart1"
                        runat="server"
                    Title="Weather Forecast" />
        </ZoneTemplate>
</asp:WebPartZone>
```

As we saw in figure 2.6, when this page is displayed in a browser, it will render the weather web part with the default four days' worth of weather visible. To change the number of days that are displayed you can just add the `NumberOfDays` property in the markup of the server control or set the value in code. The following snippet shows the property being set within the markup to display an entire week's worth of weather.

```
<wp:CustomWeatherPart
        ID="CustomWeatherPart1"
        runat="server"
        NumberOfDays="7"
        Title="Weather Forecast" />
```

2.3.2 Creating web parts with user controls

We've seen labels and calendars that magically morph into `GenericWebParts` and custom server controls that derive from the `WebPart` class being used to create web parts; but we haven't as yet seen a web part created using a `UserControl`. Being able to use user controls as web parts allows developers to create user interfaces employing exactly the same techniques they applied when creating web pages. This includes having the ability to drag-and-drop controls from the Toolbox onto the design surface. For this reason, user controls may also be easier to understand for someone who is relatively new to ASP.NET and who would benefit from the better design time experience that they would get when creating user controls in Visual Studio 2005. One advantage includes being able to drag controls directly from the Visual Studio 2005 Toolbox onto the surface of user controls as opposed to having to work solely in code.

The weather web part that we built had a very simple user interface and therefore the rendering code and logic were not overly complex; but as the amount of presentation code that is required for a control increases, custom server controls can become quite difficult to create and maintain because you tend to end up writing many lines of code to create the user interface layout.

Displaying calendar appointments

In this next example a user control will be used to create a web part that displays the current date and time. The web page will also display information to the users about upcoming meetings from their calendars. In our example, however, we'll again use hard-coded sample data for simplicity rather than writing the code that would be required to connect to a real calendar. Figure 2.7 shows how this web part will appear when it's complete.

From within your test web project, add a new user control file named `MyCalendar.ascx` to the project. To create the user interface elements necessary to display our control, add the markup that is displayed in listing 2.4 to the control.

Figure 2.7 A user control web part is used to display the current date and time. It also displays information about upcoming meetings for the logged-in user.

Listing 2.4 The HTML to display the calendar user interface

```
<h3>Current Date and Time</h3>
<div>
    <span style="width: 140px">Date: </span>
    <%= DateTime.Now.ToShortDateString() %>
</div>
<div>
    <span style="width: 140px">Time: </span>
    <%= DateTime.Now.ToShortTimeString() %>
</div>

<h3>Upcoming Meetings</h3>
<asp:Repeater ID="rptMeetings" runat="server">
    <ItemTemplate>
        <p>
            <b> <%# Eval("MeetingName") %></b>
            <br />
            <span style="font-size: smaller; font-style: italic;">
                <%# Eval("MeetingDateTime") %>
            </span>
        </p>
    </ItemTemplate>
</asp:Repeater>
```

The Date and Time interface elements

The Calendar interface elements

Listing 2.4 creates the presentation layout for the web part. As you can see, there is a Repeater server control named `rptMeetings` that binds to some fields named "MeetingName" and "MeetingDateTime."

Binding dynamic data

I mentioned before that, in a real world situation, the data we are displaying would be coming from a live backend, line-of-business application such as a Customer Relationship (CRM) database that contains information about the contacts and customers of a business. However, in our example, those fields are going to come from some sample data held in a data table. To create the data and bind it to the repeater, switch to source code view and add the code displayed in listing 2.5 to the form:

Listing 2.5 Data is created and is bound to the user interface elements of our Calendar web part control

```
protected override void OnLoad(EventArgs e) {

    base.OnLoad(e);

    DataTable dt = new DataTable("MeetingData");
    dt.Columns.Add("MeetingName", typeof(string));
    dt.Columns.Add("MeetingDateTime", typeof(DateTime));

    DataRow row1 = dt.NewRow();
    row1["MeetingName"] = "AGM";
    row1["MeetingDateTime"] = DateTime.Now.AddDays(.2);
    dt.Rows.Add(row1);

    DataRow row2 = dt.NewRow();
    row2["MeetingName"] = "Lunch with CEO";
    row2["MeetingDateTime"] = DateTime.Now.AddDays(1.3);
    dt.Rows.Add(row2);

    this.rptMeetings.DataSource = dt;
    this.rptMeetings.DataBind();

}
```

Add columns to a DataTable to store data

Add some rows of data to the DataTable

Add some rows of data to the DataTable

Bind the DataTable to the user interface

The code in listing 2.5 shows that a `DataTable` is created and two columns are added to it that will contain the information about meetings. Next, dummy data is appended to the table before we finally bind the table to our repeater control, which contains the user interface logic to display the data to the user.

That completes the code for the user control. Now we can create a web page to display it in. Add a page to your test web project and, as with all web part pages, add a `WebPartManager` and a `WebPartZone` to the page.

With the page in design mode, drag the user control that we just created from the Server Explorer onto the web part zone. Listing 2.6 shows how the page appears when displayed in source view. You can see that the designer has added a Register directive for the user control and also added the correct mark-up for the user control into the body of the `ZoneTemplate`. Build and run your page in a browser to view the results. They should appear as they did in figure 2.7.

Listing 2.6 The MyCalendar user control web part is declared within the web part zone

```
<%@ Register Src="MyCalendar.ascx" TagName="MyCalendar" TagPrefix="uc1" %>

<asp:WebPartZone ID="WebPartZone1" runat="server">
        <ZoneTemplate>
                <uc1:MyCalendar
                        id="MyCalendar1"
                        runat="server"
                        Title="My Calendar" />
        </ZoneTemplate>
</asp:WebPartZone>
```

Now you've seen both options for creating web part controls. That is, you can either create them by using custom server controls and by inheriting from the base `WebPart` class, or you can use user controls. In a very short time we've actually managed to build web parts by using both custom server controls and user controls. Ideally, as you've been working through these samples you've started thinking of all the different types of web parts that a real business might want to have displayed on its portal—employee information, sales data, production figures, profit and loss data, and so on. As we move further into the book, you'll learn that the portal framework comes complete with a catalog to store all these parts. Then you will see that having too many web parts never presents a problem, because they can all be stored and easily retrieved from within the catalog gallery, ready to be displayed on a user's page at any time.

2.4 UNDERSTANDING WEB PART INTERNALS

So far we've seen and created custom controls by deriving from the `WebPart` class and also created `GenericWebPart` controls by adding server controls and user controls directly to web part zones. Now it's time to zoom in on the `WebPart` class. What are the interfaces and properties that the `WebPart` class supports? How do these interfaces and properties allow other portal framework components to interact with it?

An important feature of the `WebPart` class is that it implements interfaces that allow it to describe its properties and behaviors to other members of the portal framework as described in table 2.1.

Table 2.1 The WebPart class implements three interfaces

Interface	Description
IWebPart	Describes the core properties of a web part such as its Title, Description, Height, and Width
IWebActionable	Describes how a web part provides verbs
IWebEditable	Describes a web part that provides custom editor parts for managing some of its properties

Because the `WebPart` class implements these three interfaces, each of the different members of the portal framework can interact with all web parts. For example, when a page is first displayed, each web part is handed to the web part manager. The manager then uses these interfaces to determine what capabilities a web part has. Therefore, when the manager was handed our weather web part, it didn't have to know anything about what properties we had given it, but by virtue of the fact that the weather web part is a web part, the web part manager knew that the control would have certain distinguishing features such as a Title and Verbs, etc.

In the sections that follow we'll spend time going over each of those three interfaces described in table 2.1 to gain a better understanding of how they are used by the portal framework, and also how we can use them to extend and customize the behavior of web parts that we create.

2.4.1 IWebPart

The `IWebPart` interface defines the properties that are common to all web parts. The following is a list of each of the properties that are exposed by the `IWebPart` interface.

- *CatalogIconImageUrl*—the URL of an image that is displayed for a web part when that part is displayed in a catalog of web parts.
- *Description*—Descriptive text about a web part that is displayed for a web part when that part is displayed in a catalog of web parts. This property is also used to display tooltip information about a web part.
- *Subtitle*—Combines with the Title property to form the complete title for a web part control.
- *Title*—the title of a web part control.
- *TitleIconImageUrl*—the URL of an image that is displayed in the web part's title bar.
- *TitleUrl*—a URL to a link containing additional information that is related to the web part.

Implementing the common web part properties

When you create a custom web part by inheriting directly from the `WebPart` class you will have access to each of the properties listed above in your code because the `IWebPart` interface is already implemented for you. However, when using user controls as web parts you will need to implement the interface for yourself to be able to code against these properties from within your control. The reason for this is that the user control does not inherit from the `WebPart` class, and therefore it does not have these properties associated with it. The code shown in listing 2.7 creates a user control that implements the `IWebPart` interface and provides a custom implementation for each property of that interface.

Listing 2.7 Implementing the IWebPart interface from within a user control

```
public partial class SampleWebPart : UserControl, IWebPart {

        string CatalogIconImageUrl {
                get { return "~/images/CatalogImage.png"; }
                set { return; }
        }

        string Description {
                get { return "This is the description."; }
                set { return; }
        }

        string Subtitle {
                get { return "Sub-Title."; }
        }

        string Title {
                get { return "Web Part Title"; }
                set { return; }
        }

        string TitleIconImageUrl {
                get { return "~/images/Globe.gif"; }
                set { return; }
        }

        string TitleUrl {
                get { return "~/Default.aspx"; }
                set { return; }
        }
}
```

Figure 2.8 shows how those properties would appear when rendered in a browser. Notice that the text in the title bar is represented as a clickable link that would take the user back to the `Default.aspx` page. This is due to the fact that we specified a value for the `TitleUrl` property. Notice as well, that when the mouse is placed over the title text or the image that is displayed alongside it in the title bar, the description

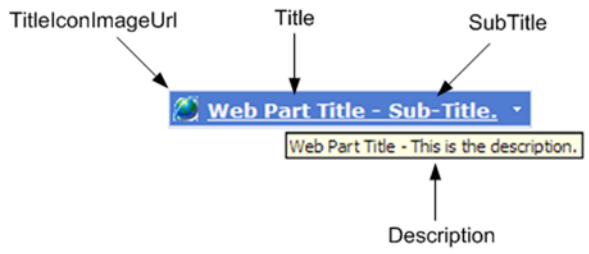

Figure 2.8 IWebPart members displayed on a web part.

is displayed as a tooltip. Each of the elements you see in the figure 2.8 web part is a direct result of the values we assigned to the interface properties in listing 2.7.

We've just seen the `IWebPart` interface in action. The `IWebPart` interface is the first of the three interfaces that are implemented by web parts. Throughout this section we've learned about the members of this interface and how they affect the look and feel of a web part at runtime. Now we'll look at the next interface that is implemented by all web parts—the `IWebActionable` interface—and see how it is used to add verbs to custom web part controls.

2.4.2 IWebActionable

Earlier in this chapter we saw that each web part has a menu, containing menu items. These menu items are referred as "verbs," and they allow users to perform operations such as closing or minimizing the web part. Every web part has a default set of verbs assigned to it by the zone in which it appears. These verbs are known as *zone verbs* and are common to all web parts—they are Close, Minimize, Restore, Delete, Edit, and Export. Which zone verbs are visible at any given time depends on the current display mode of the web page, the current user, and also the current state of the web part itself. For example, it would not make sense to display the Close verb when the web part is already closed or Restore when it is open.

Displaying custom verbs

In addition to the standard set of zone verbs that each web part is assigned, additional verbs can also be added to the web part's menu. These additional verbs would generally be for the purpose of allowing users to perform custom actions associated with the web part. This can be achieved by using the `WebPartVerb` class to create a verb, and adding that verb to the existing collection of verbs for the part. For example, we could add a verb named Copy Text to a web part that would enable users to copy text from within a web part to another control elsewhere on the page. Verbs are the perfect choice for adding these kinds of discrete operations to web parts because they are conveniently hidden away from the main user interface section of the web part, yet readily accessible to the user.

The `IWebActionable` interface is used to define a property named Verbs that appears on every web part. The Verbs property is responsible for returning all the verbs that belong to a web part, and it is here that we have the opportunity to add custom verbs to the existing collection. Let's demonstrate this by extending the `SampleWebPart` we just created, so that it implements the `IWebActionable` interface. This will allow us to add in our own custom verbs for that web part. The following snippet of code shows how to implement the interface on our `SampleWebPart` class.

```
public partial class SampleWebPart : UserControl, IWebPart, IWebActionable
{

        protected WebPartVerbCollection IWebActionable.Verbs {
```

```
            get {  }
        }
}
```

NOTE If you are using custom controls for your web parts you will not have to directly implement these interfaces because they are already implemented on the base `WebPart` class. In that case, your code will differ because you will be overriding an existing implementation of the `IWebPart` and `IWebActionable` members, as opposed to implementing one from scratch.

Next we will add code to the Verbs property to create two custom verbs for our class. The first verb will allow the user to click on it to display the current time in a label. A second verb will allow the user to clear the text of the label. The verbs that we add are also associated with corresponding server-side event handlers named `DisplayTime` and `ClearTime`. These handlers are the methods that will be called when a user clicks on the verb. The full code for our custom Verbs implementation can be seen in listing 2.8, while figure 2.9 shows the two verbs being displayed in a browser at runtime.

Listing 2.8 The Verbs property is used to associate custom verbs with a web part.

```
WebPartVerbCollection IWebActionable.Verbs {
    get {

        WebPartVerb timeVerb = new WebPartVerb(
             "TimeVerb1",
            new WebPartEventHandler(DisplayTime)
            );

        timeVerb.Text = "Change Display Text";
        timeVerb.ImageUrl = "~/images/event.gif";

        WebPartVerb clearVerb = new WebPartVerb(
           "ClearVerb1",
           new WebPartEventHandler(ClearTime)
           );

        clearVerb.Text = "Clear Display Text";


        return new WebPartVerbCollection(
new WebPartVerb[] { timeVerb, clearVerb }
);
    }
}
```

Create verbs using the WebPartVerb class

Associate any text and images to display for the verb

Add the verbs to a collection and return them

Figure 2.9 Our custom web part verbs appear in the web part's menu along with any existing zone verbs.

The verbs are created and their attributes set. This includes setting the `ImageUrl` and assigning the text that will be displayed to the user. Finally, the verbs are added to a `WebPartVerbCollection` and returned to the portal framework where they are then added to the web part for display.

When the web part is run in a browser, we can see that our custom verbs are added to the verb list along with any zone verbs. Clicking on either of our custom verbs causes the web page to post back to the web server where the associated handler method will be called. A nice enhancement might be to actually disable the Clear Display Text verb if the text has already been cleared from the display.

The WebPartEventHandler delegate

We saw that the `WebPartVerb` class is used to create verbs. When the `WebPartVerb` class was constructed, two pieces of information were passed as its arguments. This is shown in the following snippet of code:

```
WebPartVerb clearVerb = new WebPartVerb(
    "ClearVerb1",
    new WebPartEventHandler(ClearTime)
    );
```

The first argument specifies what ID to use for the verb control, and the second argument is a `WebPartEventHandler` delegate. The `WebPartEventHandler` delegate is used to associate a method with the click event for the verb, and enforces that the specified method implements the `WebPartEventHandler` interface.

> **NOTE** A delegate is a special Type in .NET that allows us to specify that methods which handle events must implement a specific interface.

When the verb that we've created is clicked by a user, a postback occurs and a method named `ClearTime` is called to handle the click event. The following piece of code shows the code for the `ClearTime` method:

```
protected void ClearTime(object sender, WebPartEventArgs e) {
    this.lblText.Text = string.Empty;
}
```

Note that the `ClearTime` method implements the `WebPartEventHandler` by taking an object and an instance of the `WebPartEventArgs` class as its arguments.

Handling events in the browser

In addition to handing the click events for verbs on the server, the `WebPartVerb` class also provides a way to specify that a client-side event handler is used. Specifying a client-side event handler for verbs allows us to handle the click event for the verb in the browser, and means that there is no postback to the web server, so no page refresh occurs.

To use client-side event handlers we can simply specify the name of the client-side function when creating the verb. This process is almost identical to the previous process when we used a server-side event handler, except that the name of the handling method is passed into the verb's constructor as a string literal instead of being passed as an instance of the `WebPartEventHandler` class. Listings 2.9 and 2.10 display the code for a Verbs property, which adds a verb that is handled by a client-side JavaScript function named `ClientClickHandler`. You can see that the ID of the control is passed to the function.

Listing 2.9 Verbs can also be associated with client-side event handlers that do not require the item to post back to the web server.

```
WebPartVerbCollection IWebActionable.Verbs {
        get {
                WebPartVerb verb = new WebPartVerb(
                        "Verb1",
                        "ClientClickHandler('" + this.ClientID + "')"
                        );

                verb.Text = "Display Web Part ID";
                verb.ImageUrl = "~/images/event.gif";

                return new WebPartVerbCollection(new WebPartVerb[] {verb});
        }
}
```

Listing 2.10 A client-side JavaScript function is written to handle the verb's click event in the browser

```
<script language="javascript" type="text/javascript">

function ClientClickHandler(webPartID) {
        alert( "You clicked the following web part: " + webPartID + "." ) ;
}

</script>
```

The use of client-side event code can help to create applications that are more dynamic and interactive, because the user is not left waiting for his operations to run while the browser performs a postback to the web server. Performing a server postback requires the entire payload of the page to be round-tripped each time that server communication is required.

Ajax, which stands for Asynchronous JavaScript and XML, is a technique that is being used increasingly by developers to create sites that are highly interactive. By using Ajax, a developer is able to send smaller packets of data through to XML web services without requiring a complete page postbox. The response from these web service calls is then processed in the browser by using client-side JavaScript. The result is twofold; first, only the part of a page that needs to be refreshed is communicated between the server and the user's browser; and second, the web pages appear more responsive because the user is not stalled while waiting for a full page refresh to occur.

I'll defer a fuller discussion about Ajax and web parts until chapter 10, when we explore ways to take advantage of this technique with the portal framework.

2.4.3 IWebEditable

The last of the three interfaces that are implemented by all web parts is `IWebEditable`. This interface allows developers to associate custom editing controls with their web parts. We'll look at this interface in detail in chapter 5 and again, in even greater detail in chapter 8. For now we'll just look at an example to show why you would need to use this interface. Remember from chapter 1 that we created the simple portal example and that we added an `AppearanceEditorPart` to an `EditorZone` to allow some of the properties of the web part to be edited by users at runtime. Figure 2.10 shows us the editor zone with the `AppearanceEditorPart` displayed.

At runtime, users of our portal can access this `AppearanceEditorPart` to directly manipulate the appearance of web parts on the page.

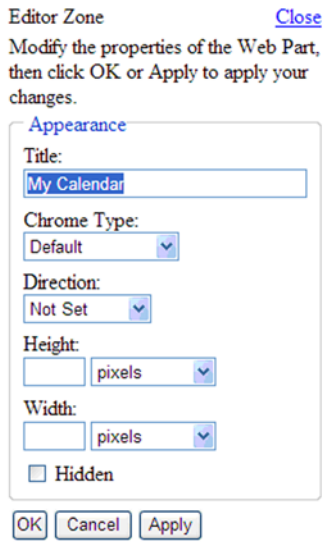

Figure 2.10 The AppearanceEditorPart provides user interface elements that allow users to manage the appearance of web parts at runtime.

Creating custom EditorParts

As you can see, the appearance editor provides us with user interface elements for managing the appearance of the web part—such as its Title, its Height, and Width. But what if you need special controls to manage a property of your web part? An example might be if we wanted to provide users of our weather web part with a map that allowed them to select their weather region visually. This is the kind of scenario that the `IWebEditable`

interface is designed to manage, as it provides us with a way of assigning custom editor controls with our web part. To implement the `IWebEditable` interface, two members must be implemented:

- *The `CreateEditorParts` method*—which allows you to return a collection of all custom editor parts that you want to associate with your control
- *The `WebBrowsableObject` property*—which provides a way to return a reference to the underlying control that you wish to expose to your custom editor part.

To associate a custom map editor part with our weather web part, we start by adding code to the `CreateEditorParts` method which returns the custom editor control that we need to manage our zip code property. The following code shows what is required to return a custom editor part named `ZipCodeSelector` from the `CreateEditorParts` method:

```
public override EditorPartCollection CreateEditorParts() {

    EditorPartCollection editorParts = base.CreateEditorParts();

    editorParts.Add(new ZipCodeSelector());
    return editorParts;
}
```

In the `ZipCodeSelector` editor part, we would include the rendering logic that we want to display when the part is displayed in the editor zone. As with all server controls, this is accomplished by writing some custom code in the Render method. In this case we'll add some code to display an icon and a link that the user can click to launch a larger map selection dialog. How do you do that? Figure 2.11 shows how the editor part would display in the browser when the code in listing 2.11 is executed.

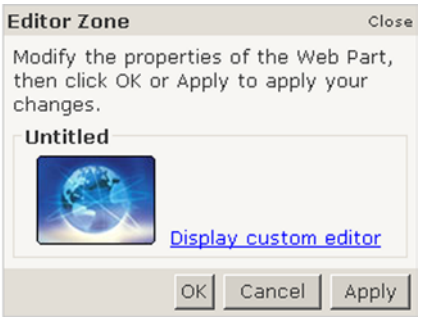

Figure 2.11 The completed editor part is displayed in the editor zone whenever the associated web part is edited.

Figure 2.11 shows a custom editor section appearing in the `EditorZone`. The custom editor section has a title of "Untitled" and contains an icon—a picture of the globe—and a link that allows users to launch a completely custom dialog for selecting zip codes. The custom dialog that is launched could display an interactive map that allowed users to select postcodes in a more visual manner than the standard textbox control normally offered for entering postcodes.

Listing 2.11 The ZipCodeSelector editor part includes code for rendering its display when shown in the editor zone

```
protected override void Render(HtmlTextWriter writer) {
    base.Render(writer);
    Image img = new Image();
    img.ImageUrl = "~/Images/Globe.jpg";
    img.BorderStyle = BorderStyle.None;
    img.Style.Add("margin", "0px 10px");

    writer.Write(@"<span  style=""font-size:0.8em;"">") ;
    img.RenderControl(writer);
    writer.Write(
    ➥@"<a href=""javascript: void(0);""
    ➥onclick=""LaunchMapEditor() ;"">"
    ➥);
    writer.Write("Display custom editor");
    writer.Write("</a>");
    writer.Write(@"</span>");
}
```

Create image to display as an icon

Write HTML to display for the control

We could set this up so that when the user clicks on the "Display custom editor" link, a dialog is displayed that provides the user with a simple way to make region selections—such as a control that allows the user to view a map of the world and gives him a way to make selections by clicking on areas of the map.

Throughout this discussion on web part internals, we've seen that by implementing certain interfaces and behaviors, our web parts are able to work together with the other components of the portal framework cohesively. We've also seen that having different pieces of the framework communicating via interfaces provides for a high level of extensibility. This extensibility was demonstrated when we were able to easily pass our own custom editor parts for use in the `EditorZone` by simply implementing the `IWebEditable` interface.

Although the discussion in this chapter has focused thus far on extending the functionality of controls, there is one last topic that we should cover before moving on to more work on the Adventure Works Portal. That topic is themes. While themes are not a specific feature of web parts, our discussion on web parts would not be complete without mentioning them. Themes are common to all controls in ASP.NET 2.0, and offer a very flexible way to create websites that, in turn, offer flexible visual styles and layouts.

2.5 APPLYING THEMES AND STYLES

Prior to ASP.NET 2.0, we used Cascading Style Sheets (CSSs) to create sites with highly flexible styles and layouts. We used CSS to create styles and associate them with the HTML elements in your application. This approach works well when we

know exactly the sort of HTML we are producing them for. The problem that arises when using ASP.NET 2.0 is that most of the time we are no longer working directly with raw HTML, but instead are working through the abstractions of server controls. For example, consider a standard server control tag for presenting the user with an `EditorZone` such as the one shown here:

```
<asp:EditorZone ID="EditorZone1" runat="server">
    <ZoneTemplate>
        <asp:PropertyGridEditorPart
            id="PropertyGridEditorPart1"
            runat="server" />
    </ZoneTemplate>
</asp:EditorZone>
```

At runtime, this `EditorZone` server control is expanded into nearly a hundred lines of HTML markup, representing several hundred individual HTML elements. The difficulty for ASP.NET developers in the past lay in determining exactly what HTML was produced by server controls, and then mapping that to the relevant CSS styles.

Adding styles to controls

In ASP.NET 2.0, the themes feature provides a new way of creating styles for server controls. You can think of themes as style sheets for server controls. It's a good idea when developing any ASP.NET server controls—not just web parts—to extract all the style-related information contained in server controls and move it into a theme file. Having the style information separate from the presentation information provides us with the same level of flexibility that we had when using CSS alone in the past. For example, consider the Google site. It's a very simple site; but when you visit it on holidays or special occasions, it is customized for the occasion. If you go there on Easter, you are likely to see the Easter Bunny hovering just above the search textbox with his basket of Easter eggs at the ready. By grouping the style information regarding images, colors, and so forth into themes, you can easily achieve this kind of customization in your own applications. By creating several themes and applying different themes on different occasions, you could make your web parts look like pumpkins on Halloween!

Visual Studio 2005 supports the themes feature by offering new designer support for it. As shown in figure 2.12, nearly all server controls offer the developer support via the new Common Tasks feature. This feature reveals itself by offering a menu of additional features that can be accessed directly from a control when that control is viewed in design mode. In figure 2.12 we see the common tasks menu being displayed for the `WebPartZone` control with a single menu item that allows the user to Auto Format that control. Clicking on the menu item displays a dialog that allows the user to apply one of four named formats to that control. Figure 2.13 shows four zones, each with a different format applied to it. The name of the format is shown in the title of the web part.

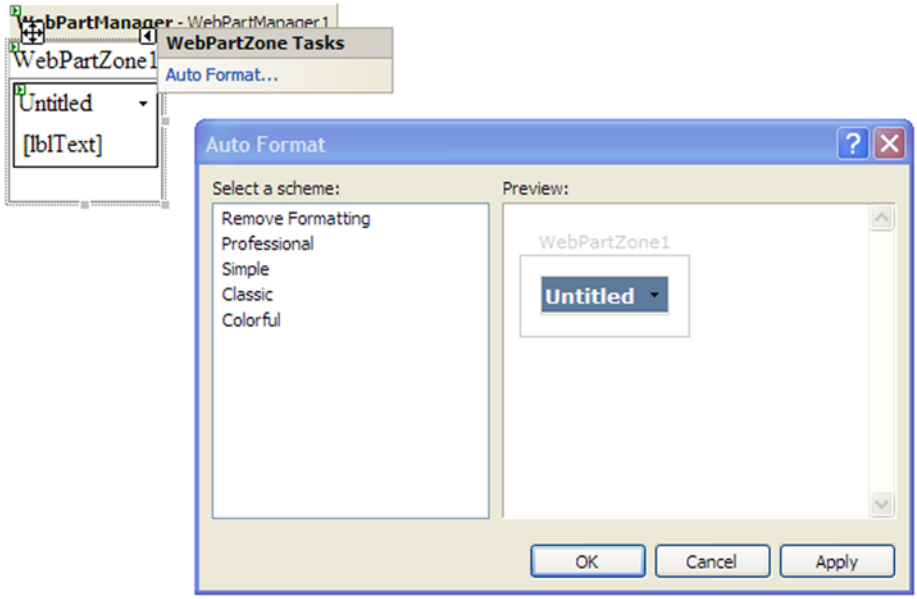

Figure 2.12 The common tasks dialog for the WebPartZone allows a developer to format that control.

When the web part zone is formatted, additional information is added to that control's declaration to describe to the ASP.NET runtime how the styles are to be applied. Listings 2.12 and 2.13 show the markup for a `WebPartZone` before and after it has style information added to it.

Figure 2.13 The four standard themes that come with ASP.NET 2.0.

Listing 2.12 A WebPartZone server control without any style-related attributes

```
<asp:WebPartZone ID="WebPartZone1" runat="server" />
```

Listing 2.13 A WebPartZone server control after applying a default theme

```
<asp:WebPartZone ID="WebPartZone1" runat="server"
  BorderColor="#CCCCCC" Font-Names="Verdana"
Padding="6">
        <PartChromeStyle BackColor="#EFF3FB" BorderColor="#D1DDF1"
          Font-Names="Verdana" ForeColor="#333333" />
        <MenuLabelHoverStyle ForeColor="#D1DDF1" />
        <EmptyZoneTextStyle Font-Size="0.8em" />
        <MenuLabelStyle ForeColor="White" />
```

```
        <MenuVerbHoverStyle BackColor="#EFF3FB" BorderColor="#CCCCCC"
        BorderStyle="Solid"
        BorderWidth="1px" ForeColor="#333333" />
        <HeaderStyle Font-Size="0.7em" ForeColor="#CCCCCC"
        HorizontalAlign="Center" />
        <MenuVerbStyle BorderColor="#507CD1" BorderStyle="Solid"
        BorderWidth="1px" ForeColor="White" />
        <PartStyle Font-Size="0.8em" ForeColor="#333333" />
        <TitleBarVerbStyle Font-Size="0.6em" Font-Underline="False"
        ForeColor="White" />
        <MenuPopupStyle BackColor="#507CD1" BorderColor="#CCCCCC"
        BorderWidth="1px" Font-Names="Verdana"
        Font-Size="0.6em" />
        <PartTitleStyle BackColor="#507CD1" Font-Bold="True" Font-
        Size="0.8em" ForeColor="White" />
</asp:WebPartZone>
```

As we can see, when the web part zone is formatted it becomes quite verbose because it contains so many style sub-elements. The biggest problem with having all that style information embedded in the page is that, if you decide to change the base look-and-feel for that control throughout your entire site, you have to go through all your pages and change every occurrence. If, however, you have used themes to style your controls, all style information for each of your server controls can be managed from a single place.

Creating themes

Now that we've seen the effect of having all of our style information embedded in pages, we are going to learn how to centralize this style information into a single location by using features known as Themes and Skin files. ASP.NET's feature, Themes, allows us to package style information such as style elements, images, and CSS files into folders. These folders contain

- *Skin Files*—Contain the style information for server controls—such as the style sub-elements shown in listing 1.13
- *Images*—Images that are associated with a specific theme—such as images of pumpkins for a theme named Halloween
- *CSS Files*—CSS information that compliments the colors and styles of the theme

These folders are then stored underneath a new, specially named folder called App_Themes within the application, and the names of the folders created underneath the App_Themes folder become the name of the theme. For example, we might want to create a theme for Valentine's Day, which has images of hearts and style information that is predominantly red. We could create a theme folder named "Valentine" to store the images and style information and images that are required to create the look-and-feel for that theme. This would include the style information for server controls, images that are associated with the theme, as well as any CSS files that we wish to use. Figure 2.14 shows the folder structure for a site that contains a blue Valentine.

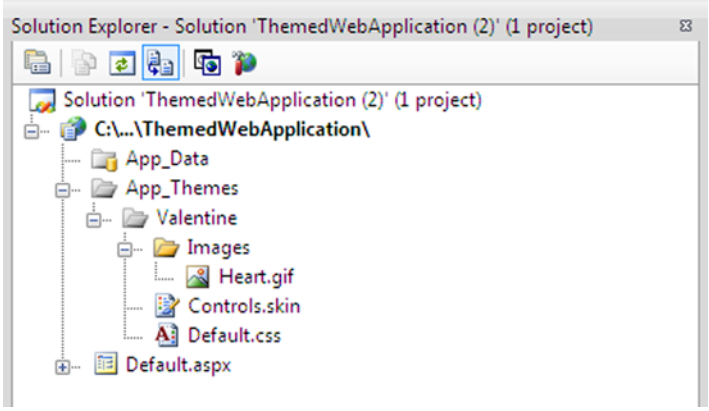

Figure 2.14 This web application contains a theme named "Valentine."

To remove the style information from the web part zone and move it into a theme folder named Valentine, follow these steps:

1. In your test web project, right-click on the solution folder and choose Add Folder, then select Theme Folder as the folder type.
2. Add a folder underneath the theme folder and name it "Valentine."
3. Right-click on the new Valentine theme folder and choose "Add Item" to add the file that will contain the style information for your server controls, and name the skin file "Valentine.skin".

The beauty of skin files is that they contain the definition of a server control in practically the same fashion as the definition would appear in a normal page, except that the theme definitions do not have an ID attribute. By the time all that style information for our web part zone is moved into the skin file, its control definition will be stripped back to its original state as shown in listing 2.12. After moving the style information into the skin file, it will as appear as it is displayed in listing 2.14:

Listing 2.14 Skin files store control definitions containing all of the style information for server controls.

```
<asp:WebPartZone runat="server" BorderColor="#CCCCCC" Font-Names="Verdana"
Padding="6">
        <PartChromeStyle BackColor="#EFF3FB" BorderColor="#D1DDF1" Font-
        Names="Verdana" ForeColor="#333333" />
        <MenuLabelHoverStyle ForeColor="#D1DDF1" />
        <EmptyZoneTextStyle Font-Size="0.8em" />
        <MenuLabelStyle ForeColor="White" />
        <MenuVerbHoverStyle BackColor="#EFF3FB" BorderColor="#CCCCCC"
        BorderStyle="Solid"
        BorderWidth="1px" ForeColor="#333333" />
```

```
        <HeaderStyle Font-Size="0.7em" ForeColor="#CCCCCC"
        HorizontalAlign="Center" />
        <MenuVerbStyle BorderColor="#507CD1" BorderStyle="Solid"
        BorderWidth="1px" ForeColor="White" />
        <PartStyle Font-Size="0.8em" ForeColor="#333333" />
        <TitleBarVerbStyle Font-Size="0.6em" Font-Underline="False"
        ForeColor="White" />
        <MenuPopupStyle BackColor="#507CD1" BorderColor="#CCCCCC"
        BorderWidth="1px" Font-Names="Verdana"
        Font-Size="0.6em" />
        <PartTitleStyle BackColor="#507CD1" Font-Bold="True" Font-
        Size="0.8em" ForeColor="White" />
</asp:WebPartZone>
```

As we can see, this skin file definition for the `WebPartZone` control looks almost identical to the `WebPartZone` control definition that we saw in listing 2.13. However, the difference is that this information is now stored in a single place—the skin file—and that all `WebPartZone` controls can now use this style without having to each have their own embedded style sub-elements. One more step is required to apply a theme within an application; you must configure the application so that it knows which theme to use. This configuration can be constructed either at page level or at application level in the `web.config` file. Both of these options are shown in listing 2.15.

Listing 2.15 Configuration entries for themes can be set at either page or configuration file level.

```
<%@ Page Language="C#" Theme="Blue" %>
```

Declaring a theme at page level

```
<system.web>
        <pages theme="Blue" />
</system.web>
```

Declaring a theme at configureation file level

The benefit of using the web configuration file to declare themes is that you are required to declare it in only one place, whereas declaring it in each page would result in many declarations. Having the theme declared in many different places makes the code more difficult to maintain, because a developer would have to locate each place that it was declared when making changes to the theme.

Now that you better understand web parts and have worked with them a bit, let's apply your new skills to the Adventure Works Cycles business.

2.6 ADDING WEB PARTS TO THE ADVENTURE WORKS SOLUTION

In chapter 1 we created a data layer so that we could connect to SQL Server 2005 and retrieve information about the Adventure Works Cycles business. In this chapter we'll start putting that data to good use as we build the beginnings of a portal application

based on the Adventure Works business. The portal that we will build throughout the course of this book will be built with small incremental steps. At the end of each chapter we'll apply a concept we've learned by integrating an implementation of that concept into the portal. While each step may, in itself seem small, by the end of the book we will have created a portal that is filled with the features that clients have come to expect of portal-style applications.

To implement the concepts you've learned in this chapter, let's get back to your job at Adventure Works. Today the HR department has asked you to develop a small website that displays some of their line of business data—such as a list of employees, departments, and information about the latest job candidates. They've specified that initially the portal should be able to display the following data:

- A listing of all departments with employee numbers listed against each one
- A listing of all employees for a given department

As an applications developer for Adventure Works you've got ASP.NET 2.0 installed and you are all geared up and ready for the task.

After discussions with the users, it is clear to you that while they have an immediate need for just these few features, their longer-term requirements are likely to be much larger. For this reason, you make the decision to use the web portal framework to build features as standalone components. Over time there will be the ability to harness the extensibility of the framework through features such as web part connections and verbs, to leverage components that we build today into tomorrow's features.

NOTE To complete this exercise you will need to create a project for the Adventure Works application. If you are comfortable with project creation and some of the new ASP.NET 2.0 features such as master pages, themes, etc., then you might just want to grab the project from the resources for this chapter that come with the book. If you would like to create the project for yourself to see how to implement these new features, you can complete the walkthrough titled "Creating the Adventure Works Project" in the appendix.

Displaying all departments

We'll address the first feature request, which was to create a web part that displays a listing of all departments from the Adventure Works database. Open the Adventure Works project and create a new folder named `WebParts` and add to it a new user control file named `DepartmentListingPart.ascx`.

With the user control in design mode, add a `GridView` server control from the toolbox by dragging it onto the design surface. From the associated `GridView` tasks, choose the `<New Data Source>` option so that we can configure a data source to return the data that we need. At this time the Data Source Configuration Wizard starts up and the Choose a Data Source Type screen is displayed as shown in figure 2.15. We've already created a data access layer to perform our data operations, so from this screen we choose the Object data source type and press OK.

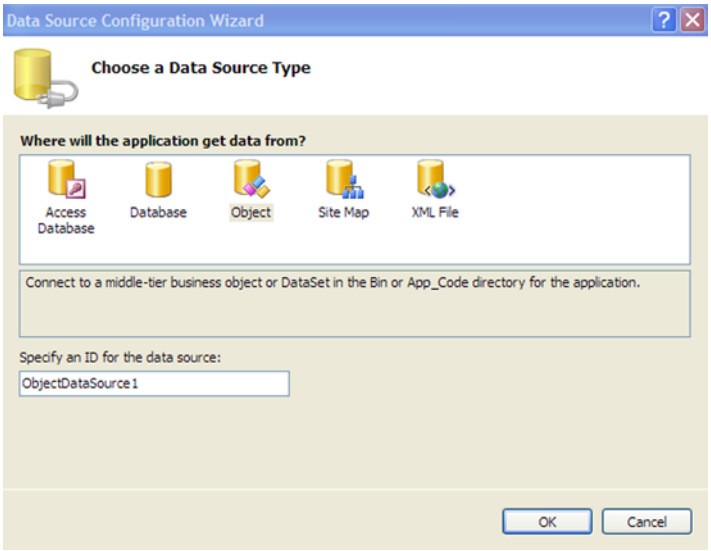

Figure 2.15 The first step in the data source configuration wizard is to specify what type of data source we are binding to.

The next screen in the wizard displays a listing of classes, allowing us to choose which business object contains the method to bind to the `GridView` control. This screen is displayed in figure 2.16. Select the `AW.Portal.Data.DataLayer` class and then press OK.

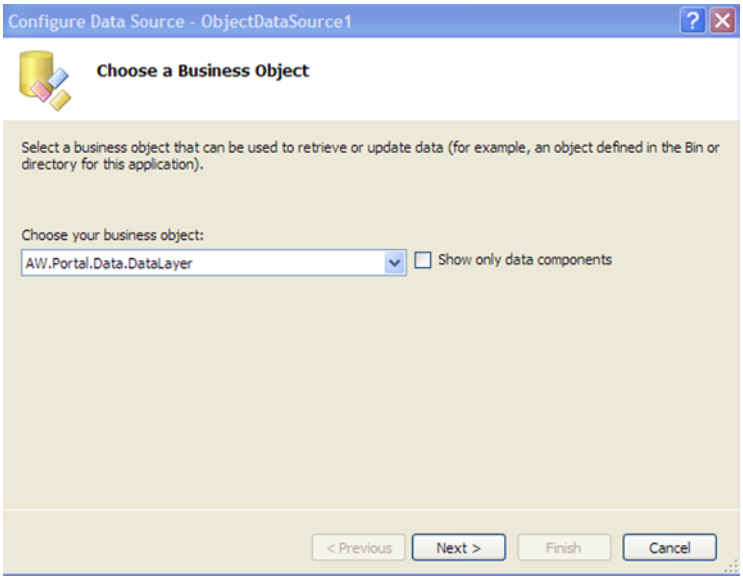

Figure 2.16 When using the object data source control, we get to specify which class will provide the data.

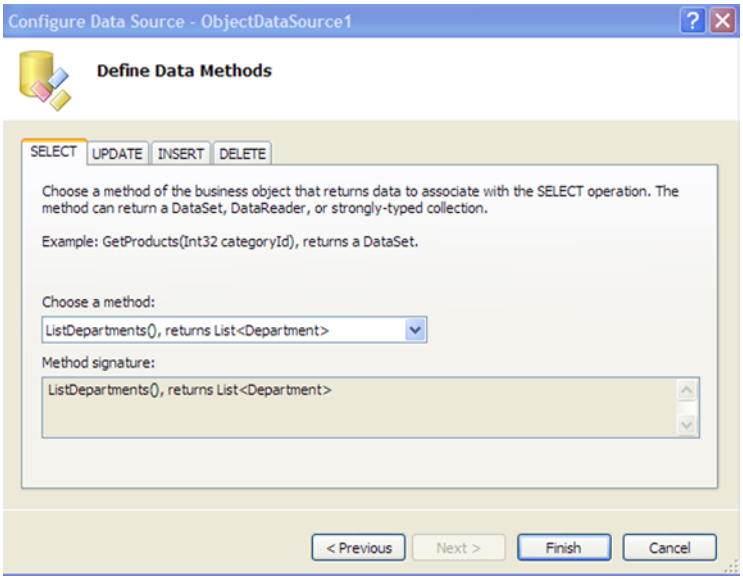

Figure 2.17 The wizard allows us to choose which methods of the object data source will perform data operations such as Select, Update, Insert, and Delete.

With the business object chosen, all that remains is to choose which method of the object will provide us with the data. The last screen of the wizard that we'll be using allows us to select the `ListDepartments` method and press Finish. This last screen is displayed in figure 2.17.

Now the wizard has all of the information it needs to create a data source control that can be bound to the `GridView`, and we can create a page in which to display our control. To add this web part to the web part page we created earlier, switch the `Default.aspx` page into design mode and drag the `DepartmentListingPart` user control from the Server Explorer on to the `WebPartZone`. Listing 2.16 shows that by adding the user control, Visual Studio has automatically registered the control with the page and added the correct markup for the user control into the zone template for us.

Listing 2.16 The DepartmentListingPart

```
<%@ Register Src="WebParts/DepartmentListingPart.ascx"
  TagName="DepartmentListingPart" TagPrefix="uc1" %>

...

<asp:WebPartZone ID="WebPartZone1" runat="server">
       <ZoneTemplate>
```

Added Register declaration for user control

```
                <uc1:DepartmentListingPart
                        ID="DepartmentListingPart1"
                        runat="server"
                        Title="Departments" />
        </ZoneTemplate>
</asp:WebPartZone>
```

Added user control to zone template

The register tag that was added by Visual Studio is known as the @ Register directive. This directive is included in ASP.NET web pages so that a tagname and tagprefix can be associated with user controls and custom server controls. You can see that the tagname (`DepartmentListingPart`) and tagprefix (`uc1`) could then be useful in declaring the department listing user control within the page.

We can now run the application and see that the web part is displayed with a listing of departments as shown in figure 2.18. To do this, right-click on the `Default.aspx` file and choose "View in Browser."

Departments

Name	Employees
Executive	2
Facilities and Maintenance	7
Finance	10
Human Resources	6
Information Services	10
Purchasing	12
Shipping and Receiving	6
Production	179
Production Control	6
Document Control	5
Quality Assurance	6
Engineering	6
Research and Development	4
Tool Design	4
Marketing	9
Sales	18

Figure 2.18 The DepartmentListing web part shows a listing of the departments within the Adventure Works business with the number of employees shown against each department.

Creating an Employees web part

Now that we have a listing of all departments, we may turn our attention to the second requirement we were given—to display a listing of employees for a given department. We'll again create this listing as a web part and again be using the `GridView` control to display the data in a list. Right-click on the `WebParts` folder and add to it a new user control file named `EmployeeListingPart.ascx`.

With the user control in design mode, add a `GridView` control and choose the <New Data Source> option. Walk through the Data Source Configuration wizard, in a manner similar to the steps we took for the previous control, and bind the `GridView` to the `ListEmployees` method of the data layer class. Finally, add the web part to the `RightZone` in the web part page.

When viewed in the designer, your page should be similar to the page shown in figure 2.19. You can now view the portal by right-clicking on the `Default.aspx` file and choosing "View in Browser."

The full source code for the portal at this stage can be found in the resources that accompany this book.

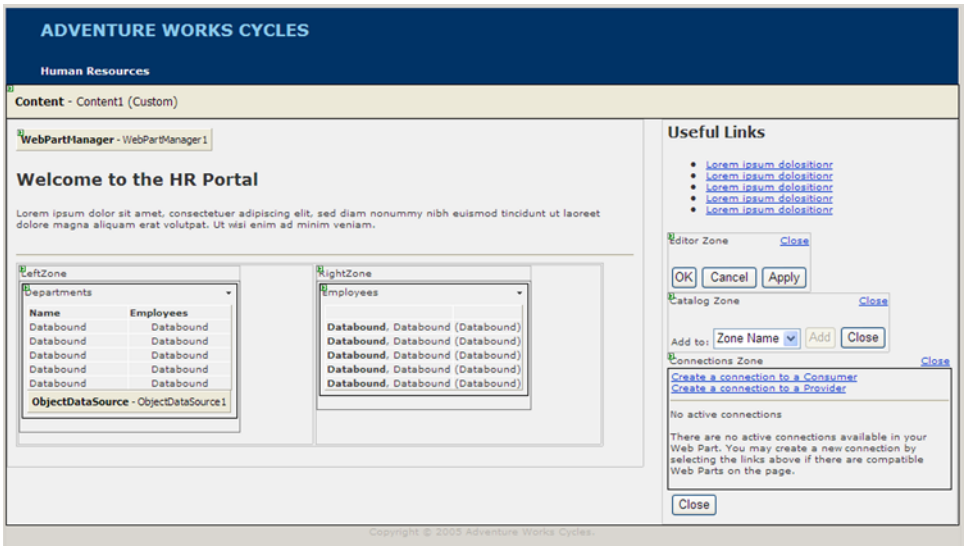

Figure 2.19 The portal when viewed in design mode within Visual Studio 2005

2.7 SUMMARY

In this chapter we've covered a stretch of important ground in learning about these fundamental portal components; but perhaps even more important is the fact that we now have our portal up and running. As we move through the book, in each chapter we will add small additional touches to the portal as we learn new concepts. By the end of the last chapter you will see that the sum of all these small additions is an interesting portal with many useful features.

Throughout this chapter we covered some fundamental lessons about web parts and, in particular, the `WebPart` server control, and we then added them to our own portal. We have demystified some of what happens when controls are added to web zones and learned about the special `GenericWebPart` control. We also saw how to use verbs to link additional operations to our web parts.

Finally, we started work on the Adventure Works portal application by creating the Visual Studio 2005 project and then adding our first web parts for the HR department. These web parts were simple, but we're not finished with them yet. The next chapter dips into web part connections. With this knowledge we'll be positioned to connect our two web parts and have the department part act as a filter for the employee part. This will allow us to complete the requirement that the employees be viewable by department.

WEB DEVELOPMENT

ASP.NET 2.0 Web Parts IN ACTION

Darren Neimke Foreword by Andres Sanabria

The static Web is going out of style. Its click-and-wait user experience is giving way to dynamic personalized content and intuitive interactions. With ASP 2.0, a web developer can compose a page out of separate working parts—"Web Parts"—that independently communicate with the server to produce rich interactive portals like Yahoo!, Google/ig, and Live.com. The new Web Parts API makes it easy to centrally manage a portal's parts.

ASP.NET 2.0 Web Parts in Action is packed with annotated code, diagrams, and crystal-clear discussions. You'll develop a sample project from design to deployment, adding content zones, personalization, and a custom look-and-feel. Since any website is invariably a work-in-progress, you'll appreciate learning how to upgrade your portals on the fly.

Along the way you'll pick up handy code instrumentation techniques and a few tricks to help your portals manage themselves. As an added bonus, the book introduces the Microsoft Ajax Library ("Atlas") and shows how you can add Ajax to a web part. You'll even create a Live.com gadget.

This book is for web developers familiar with ASP.NET.

What's Inside

- Effective portal design strategies
- Add personalization features
- Create user-friendly controls
- Develop custom themes and WebPartChrome
- Automate site health monitoring
- Techniques for graceful error recovery

Darren Neimke, MCAD, is a senior consultant with Readify in Australia, a Microsoft MVP in ASP.NET and a member of the prestigious ASP Insiders group.

"A must for every ASP.NET developer using Web Parts."
—Scott Guthrie
General Manager
Microsoft Developer Division

"Squeezes the full potential out of ASP.NET Web Parts."
—Andres Sanabria
from the Foreword

"Great book, great author, great style—there's nothing even close."
—Paul Wilson
ASP.NET MVP and ASPInsider

"Impressive detail. Well done!"
—Joe Litton
Microsoft Certified Professional

"Brisk and to the point."
—Stuart Caborn, ThoughtWorks

www.manning.com/neimke

ISBN 1-932394-77-X

$44.99 US/$58.99 Canada