3

Working with Web Parts

Most Web sites, and portals in particular, make a point of showing large amounts of content. This can be a feast for users, but in the long run it can also be a source of confusion. Recent studies on Web usability have concluded that personalization is a key factor in successful Web sites. Giving users the tools to build a personalized view of the content can mean the difference between an average and a superior Web site. You could say that cutting-edge Web sites are rich in content, have a consistent and modular design, and allow users to personalize the content.

How do you build such a Web site? Admittedly, rich content is not a programmatic issue, but being able to handle a wide range of content is a crucial design issue. As a page developer, you are responsible for building a modular site and making it personalizable. Web Parts and the Personalization API are the tools in Microsoft ASP.NET 2.0 that make building modular and customizable Web sites easier, and even pleasant.

In this chapter, we'll take a tour of the ASP.NET 2.0 Web Parts framework and build a small but highly personalizable Web site. We'll cover the Personalization API in Chapter 4.

Building Pages with Web Parts

ASP.NET Web Parts provide an infrastructure for creating Web applications that can handle rich content as well as large amounts of content. You can use Web parts to build sites that enable users to select and receive only the content they want. Web parts are container components that aggregate different types of content. As such, they are particularly useful for creating portal pages.

What Are Web Parts, Anyway?

Figure 3-1 is taken from the My MSN Web site. The page is an aggregation of different blocks, each presenting a particular type of information.

Figure 3-1 The My MSN page of a registered user is composed of all the blocks of information the user selected.

No block displayed in the page is there by chance. By using the Add Content and Change Details links, the user can select which blocks to display and their graphical layout. Each block in the page can be obtained with a Web part in an ASP.NET 2.0 application. Each Web part corresponds to an instance of the *WebPart* control.

Content of a Web Part

You can think of a Web part as a window of information available within the page. Users can close, minimize, or restore that window. The *WebPart* control is essentially a container filled with the usual HTML stuff—static text, link, images, and other controls, including user controls and custom controls. By combining Web Parts and the Personalization API, page developers can easily create pages and controls tailored to the individual user's needs.

The content of a Web part can be any information that is useful to all users, a subset of users, or one user in particular. Most commonly, the content of a Web part is provided by external Web sites—for example, financial, weather, and sports news. The content of a Web part can range from a list of favorite links to a gallery of photos to the results of a search.

Layout of a Web Part

From the developer's perspective, a Web Part component is a sort of smart *Panel* control. Like a panel, a Web part can contain any valid ASP.NET content. The *WebPart* class is actually derived from the *Panel* class.

What makes the Web part more powerful than a simple panel is the support it gets from the Web Part manager (represented by the aptly named *WebPartManager* control) and the extra visual elements it renders. The layout of a Web part mimics that of a desktop window. It has a caption bar with a title, as well as links to common verbs such as minimize, close, and restore.

Although the Web part can act as a container for information from external sites and pages, it is quite different from a frame. A frame points to a URL, and the browser is responsible for filling the area with the content downloaded from the URL. A Web part, on the other hand, is a server-side control that is served to the browser as plain HTML. You can still populate a Web part with the content grabbed from external sites, but you are responsible for retrieving that content, either using HTML scraping or, better yet (when available), Web services.

Introducing the Web Parts Framework

The *WebPart* control is the central element in the Web Parts infrastructure, but it is not the only one. A page employing Web parts uses several components from the Web Parts framework, each performing a particular function. Table 3-1 details these components.

Table 3-1 Components of the Web Parts Framework

Component	Description
WebPartManager	The component that manages all Web parts on a page. It has no user interface and is invisible at run time.
WebPart	Contains the actual content presented to the user. Note that *WebPart* is an abstract class; you have to create your own *WebPart* controls either through inheritance or via user controls.
WebPartZone	Wraps one or more *WebPart* controls and provides the overall layout for the *WebPart* controls it contains. A page can contain one or more zones.
CatalogPart	The base class for catalog *WebPart* controls that present a list of available Web parts to users. Derived classes are *ImportCatalogPart*, *DeclarativeCatalogPart*, and *PageCatalogPart*.
CatalogZone	A container for *CatalogPart* controls.
ConnectionsZone	A container for the connections defined between any pair of Web parts found in the page.

Table 3-1 Components of the Web Parts Framework

Component	Description
EditorPart	The base class for all editing controls that allow modifications to Web parts. An editor part presents its own user interface to let users set properties.
EditorZone	A container for *EditorPart* controls.

Aside from the *WebPartManager* class, the Web Parts infrastructure is made up of three types of components, known as *parts*: Web parts, catalog parts, and editor parts.

A Web part defines the content to show; an editor part lets users edit the structure and the settings of a particular Web part. For example, suppose you have a weather Web part that shows weather information for a few selected cities. An editor part for this Web part would provide a friendly user interface for users to add or remove cities and decide whether to see the temperature in Celsius or Fahrenheit. Based on the weather applet in the My MSN Web site, you enter in edit mode either by clicking the Select Your Cities button of the page or the Edit button in the caption bar. The specific properties you edit differ in each case, but you end up editing the contents of the Web part, as you can see in Figure 3-2.

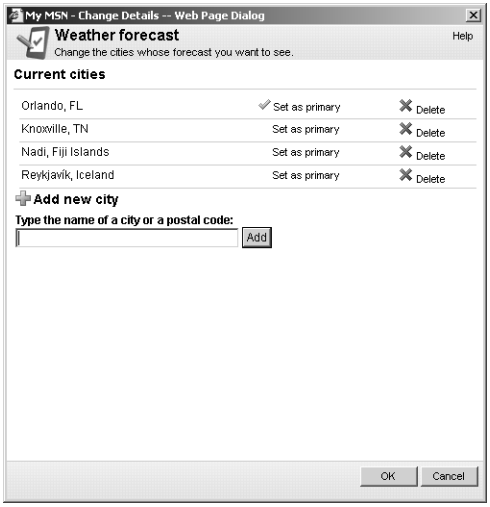

Figure 3-2 Editing the weather content block of the My MSN Web site

The Web Parts infrastructure enables users to choose a personalized set of parts to display on a page and specify their position. The list of available parts is provided by a catalog part control. In this way, users can add parts dynamically. The catalog also acts as a store for the parts that the user has removed

from the page by executing the *Close* verb represented by the Delete menu item in Figure 3-1. Removed parts can be restored if a catalog is specified for the page. We'll cover catalog and editor parts in greater depth later in the chapter.

Web Part Zones

The zone object provides a container for parts and provides additional user interface elements and functionality for all part controls it contains. A Web page can contain multiple zones, and a zone can contain one or more parts. Zones are responsible for rendering common user interface elements around the parts it contains, such as titles, borders, and verb buttons, as well as a header and footer for the zone itself.

Each type of part requires its own zone object. The *WebPartZone* is the container for *WebPart* controls. It hosts all the Web part content through a collection property named *WebParts*. In addition, it provides free drag-and-drop functionality when the Web parts are switched to design mode. The design mode is one of the display modes supported by the Web part manager and applies to all Web parts in the page. When you're in design mode, you can modify the layout of the page by moving Web parts around. The drag-and-drop facility is provided by the zone component. The *WebPartZone* control allows you to define a few style properties, such as *PartStyle* (style of the contents), *PartVerbStyle* (style of the action verbs, such as *Minimize* and *Close*), and *PartTitleStyle* (style of the caption bar).

The two other zone types are more specialized. The *EditorZone* is used to contain editor controls to configure existing *WebPart* controls. The *CatalogZone* is used to display the catalog of available *WebPart* controls the user can choose from.

You probably can't wait to see some markup code to illustrate all of this. Here's a quick example:

```
<%@ register tagprefix="x" tagname="News" src="News.ascx" %>
<%@ register tagprefix="x" tagname="Favorites" src="Favorites.ascx" %>

<%@ page language="C#"%>

<html>
<head runat="server">
    <title>WebParts--Headstart</title>
</head>
<body>
    <form runat="server">
    <h1>Demonstrating WebParts zones</h1>
    <div>
      <asp:WebPartManager ID="WebMan" runat="server" />
      <asp:WebPartZone ID="WebPartZone1" runat="server" width="600px"
```

```
            HeaderText="This is Zone #1"
            PartChromeType="TitleAndBorder">
        <PartTitleStyle Font-Size="10pt" Font-Bold="True"
            BackColor="#E0E0E0" Font-Names="verdana" />
        <PartStyle BackColor="#FFFFC0" />
        <PartVerbStyle Font-Size="X-Small" Font-Names="verdana" />
        <CloseVerb Enabled="False" />
        <ZoneTemplate>
            <x:News runat="server" id="News" />
            <x:Favorites runat="server" id="Favs" />
        </ZoneTemplate>
      </asp:WebPartZone>
    </div>
    </form>
</body>
</html>
```

The form contains a *WebPartManager* control that governs the execution and rendering of all child parts. The form also contains one Web zone that contains a couple of parts. The Web part zone is configured to show a title and a border around its content. The <*ZoneTemplate*> tag includes all the Web parts defined in the zone.

The simplest way to incorporate a Web part control is through a user control—an ASCX file. Figure 3-3 shows the page in action.

Figure 3-3 A simple WebPart component listing the latest news and useful links

If you click the Minimize button, the page posts back and only the title bar displays on return, as shown in Figure 3-4.

Figure 3-4 When a Web part is minimized, a Restore button comes up to let you restore the original panel later.

If you click the Close button (which is programmatically disabled in the previous example), the Web part is hidden from view. It is not destroyed, though. If the page includes a catalog Web part, all closed Web parts are listed in the catalog and users can restore them later.

The *WebPartManager* Class

The *WebPartManager* is a nonvisual control that manages all zones and part controls on a Web page. In particular, the manager maintains a collection of zones and parts and tracks which parts are contained in each zone. Only one *WebPartManager* can be contained in a Web form. In the simplest cases, your interaction with the manager control is limited to adding it to a page:

```
<asp:WebPartManager runat="server" id="MyWebMan" />
```

However, the Web part manager is responsible for more advanced functions that require a bit of coding. For example, it tracks the display mode of the page and notifies zones and parts of any change in the display mode. Depending on the display mode, parts and zones render differently. The default display mode is Normal, which means that catalog and editor zones are hidden and only Web parts are displayed with their own titles and borders. You can access the list of zones via the *Zones* collection.

Finally, the *WebPartManager* is responsible for initiating communication between two part controls. Two part controls within the same page can communicate and exchange information using a special channel represented by a *Connection* object. You can define a *Connection* object for a Web part manager using declarative syntax, as in the following example:

```
<asp:webpartmanager runat="server">
   <StaticConnections>
      <asp:connection runat="server" ID="MyConnection"
           ConsumerID="MyConsumerPart" ProviderID="MyProviderPart" />
   </StaticConnections>
</asp:webpartmanager>
```

Communication between parts is made possible through the use of custom interfaces. A Web part that is intended to provide some information to others would implement the provider's set of interfaces. In this way, a consumer Web part can access properties and methods in a consistent fashion.

We'll look at how connection objects connect to Web parts later in this chapter.

The *WebPart* Class

WebPart is an abstract class that is used only for referencing an existing Web part object. You define the contents of your pages using either a user control or

a custom control derived from *WebPart*. Note that if you use a user control, the ASP.NET runtime wraps it into an instance of the *GenericWebPart* control, which provides basic Web Parts capabilities. You can't use the *GenericWebPart* component alone. *GenericWebPart* wraps only one server control. To generate a user interface that aggregates multiple controls, you have two options:

- Create a user control
- Create a new control that inherits *WebPart*

If you use a user control, you can't specify *WebPart*-specific properties such as *Title*. A possible workaround is to implement the *IWebPart* interface in your user control.

Table 3-2 lists the main properties of the *WebPart* class and gives you an idea of the programming power of the Web part controls. The class indirectly inherits *WebControl* and *Panel*, so it also features quite a few extra visual properties, such as *BackColor* and *BackImageUrl* (not listed in the table).

Table 3-2 Properties of the *WebPart* Class

Property	Description
AllowClose	Indicates whether a Web part can be removed from the page.
AllowEdit	Indicates whether a Web part allows you to edit its properties.
AllowHide	Indicates whether a Web part can be hidden on a Web page.
AllowMinimize	Indicates whether a Web part can be minimized.
AllowZoneChange	Indicates whether a Web part can be moved within its zone only or between zones.
Caption	Indicates the caption of the Web part.
ChromeState	Indicates the state of the Web part: normal or minimized.
ChromeType	Indicates the type of the frame for the Web part: border, title, title and border, or none.
Description	The Web part description used in the catalog and as a ToolTip in the title bar.
Direction	Indicates the direction of the text: left-to-right or right-to-left.
HelpMode	Indicates which kind of Help user interface is displayed for a control.
HelpUrl	The URL to a topic in the Web part's help.

Table 3-2 Properties of the *WebPart* Class

Property	Description
Hidden	Indicates whether the Web part is displayed on a Web page.
IsShared	Indicates whether multiple users share the Web part.
ProviderConnectionPoints	Gets the collection of *ConnectionPoint* objects that are associated with the Web part. The *ConnectionPoint* class defines a possible connection in which the Web part is either the provider or the consumer.
Title	Indicates the string used as the first string in the title of a Web part. (See the description of the *Caption* property.)
TitleStyle	A style object used to render the title bar of the Web part.
TitleUrl	Indicates the URL where you can access additional information about the Web part control. If specified, the URL appears in the title bar of the part.
Verbs	The set of verbs associated with the Web part.
Zone	The zone that currently contains the Web part.
ZoneID	ID of the zone that currently contains the Web part.
ZoneIndex	The index of the Web part in the zone it lives in.

The programming interface of the *WebPart* class is all in the properties listed in the table. The class has no methods and events, aside from those inherited from base server controls.

A Sample Web Part Component

Let's build a sample Web part and experiment with its properties. As mentioned, a Web part is a pseudo-window nested in the Web page that contains some sort of information. Like a window, it can be moved around and its content can be configured to some extent. The Web Parts infrastructure provides drag-and-drop facilities for moving the control around and changing the page layout. The programmer is responsible for building any logic and user interface elements for editing and cataloging the content.

The primary goal of a Web part control is delivering information to users. The information can be retrieved in a variety of ways, according to the characteristics of both the Web part and the hosting application. In a portal scenario, the Web part shows through the user's personalized page some content grabbed over the Web and possibly provided by external Web sites.

The BookFinder Web Part

The sample Web part we'll build grabs information about books and authors using a given search engine. You can use the Google or the Amazon API, or you can write your own engine that searches a local or remote database. In this example, I simply want to get information about books written by a certain author. All I need is the author's name. A call to the search engine consists of the following code:

```
private string _dataSource;
_dataSource = SearchEngine.DownloadData(author);
```

The *SearchEngine* class can internally use any search technology you like, as long as it returns an XML string. It goes without saying that the XML schema is totally arbitrary. In the next example, the XML string is transformed in a *DataSet* object and used to populate a *Repeater* control.

The Web Parts component is wrapped into a user control. The body of the user control looks like the following:

```
<div style="overflow:auto;height:280px;margin:3;">
  <asp:TextBox runat="server" id="AuthorName" Text="Dino Esposito" />
  <asp:button runat="server" id="btnGo" Text="Go" onclick="OnSearch" />

  <asp:repeater runat="server" id="Presenter">
    <headertemplate>
        <table style="font-family:Verdana;font-size:8pt;">
    </headertemplate>
    <itemtemplate>
        <tr>
          <td><img src='<%# Eval("ImageUrlSmall") %>' /> </td>
          <td><b><a href='<%# Eval("Url") %>'><%# Eval("ProductName") %>
              </a></b><br />
              <i><%# Eval("Manufacturer") %></i><br />
              <%# Eval("ReleaseDate") %><br /></td>
        </tr>
    </itemtemplate>
    <footertemplate>
        </table>
    </footertemplate>
</asp:repeater>
</div>
```

The data-bound expressions used in the code use the new *Eval* keyword—a more compact replacement for the *DataBinder.Eval* method used in ASP.NET 1.x. We'll cover *Eval* in detail in Chapter 5.

The *Repeater* is bound to a *DataTable* object created from the XML string retrieved by the search engine. The field names used in the previous example are assumed to correspond to columns in the *DataTable* object. The following code snippet shows how to bind data to the *Repeater* control:

```
void BindData() {
   StringReader reader = new StringReader(_dataSource);
   DataSet ds = new DataSet();
   ds.ReadXml(reader);

   // Assume the DataSet contains a table named "Details"
   // (This is the case if you use the Amazon API)
   Presenter.DataSource = ds.Tables["Details"];
   Presenter.DataBind();
}
```

The Web part is inserted in a Web part zone using the following code:

```
<asp:WebPartZone ID="WebPartZone1" runat="server" width="600px"
     HeaderText="This is Zone #1"
     PartChromeType="TitleAndBorder" Height="286px" >
   <zonetemplate>
     <x:bookfinder runat="server" id="Books" />
   </zonetemplate>
</asp:WebPartZone>
```

Figure 3-5 shows the BookFinder Web part in action.

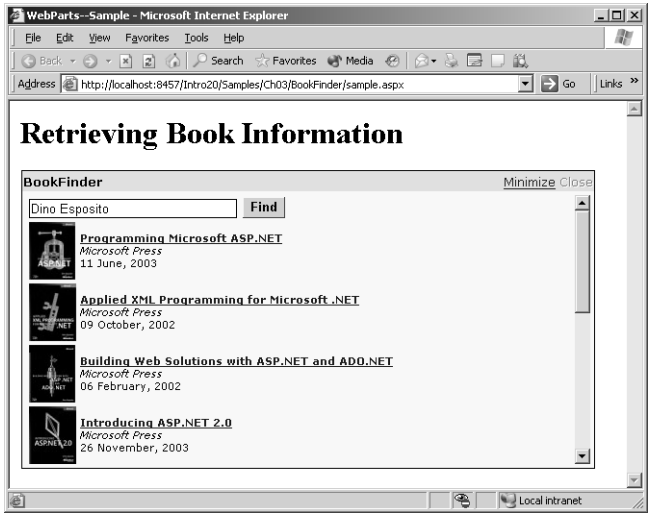

Figure 3-5 The BookFinder Web part retrieves all the books written by Dino Esposito.

> **Note** To implement a real-world book-finder Web part, you must rely on a good search engine. Two of the most popular engines, Amazon and Google, were created with different goals but both expose their services through Web services. If you're interested in the Amazon Web Services API, have a look at the *http://www.amazon.com/gp/aws/landing.html* URL and follow the steps described. Basically, you must download your free developer's kit, get your personal token to issue calls to the methods, and write your application. A similar procedure is required if you want to leverage the services of the Google engine. In this case, you get started at *http://api.google.com*.

Styling the Web Zone

The Web part can embellish its output through styles and visual properties. Note that the actual style of the Web part might depend on the settings of the Web part zone. For example, fonts, colors, and borders are inherited from the zone and apply to all parts in the zone. However, each Web part can override those settings.

The Web zone supports quite a few style properties for customizing the look and feel of the zone and its constituent parts. Table 3-3 lists the supported zone styles.

Table 3-3 Style Properties of the Web Zone

Style	Description
EmptyZoneTextStyle	Defines the style of an empty zone during the change-layout phase.
FooterStyle	Defines the style of the zone's footer.
HeaderStyle	Defines the style of the zone's header.
MenuLabelStyle	Defines the style of the zone's menu.
PartChromeStyle	Defines the style of the Web part's main frame (title, border).
PartStyle	Defines the overall style of the Web part.
PartTitleStyle	Defines the style of the title bar of the Web part.
PartVerbStyle	Defines the style of the verbs in the zone's title bar.

The following code snippet shows the styles that make up the Web part shown earlier in Figure 3-5. You can also use CSS classes defined in a separate stylesheet file.

```
<EmptyZoneTextStyle BackColor="lightyellow" Font-Size="8pt"
      Font-Names="verdana" />
<PartTitleStyle Font-Size="10pt" Font-Bold="True"
      BackColor="#E0E0E0" Font-Names="verdana" />
<PartStyle BackColor="#FFFFC0" />
<PartVerbStyle Font-Size="X-Small" Font-Names="verdana" />
```

You can customize the verbs in the title bar of Web parts. Verbs identify actions users can take on the Web part as a whole—for example, minimizing, restoring, editing, or closing the component. Verbs are enabled using the various *AllowXXX* properties on the Web part. Their style is controlled through *XxxVerb* tags. For example, you can disable the Close button by using the following code:

```
<CloseVerb Enabled="False" />
```

One of the style properties you can set is the text displayed. You can use little bitmaps, too, as shown here:

```
<closeverb imageurl="images/CloseVerb.gif"
    text="Close" description="Closes the WebPart" />
<restoreverb imageurl="images/RestoreVerb.gif"
    text="Restore" description="Restores the WebPart" />
<minimizeverb imageurl="images/MinimizeVerb.gif"
    text="Minimize" description="Minimizes the WebPart" />
```

Note that verb properties belong to the zone, not to the individual Web part. This means that all Web parts in a given zone share the same verb settings. This isn't true of other title attributes, such as the background and colors.

A verb is represented by a *WebPartVerb* object and features a few properties, including *Description* (the tooltip displayed), *Text* (the text or alternative text if an image is used), and *ImageUrl* (the image to render). In addition, you can define click handlers for both the client (*ClientClickHandler*) and the server (*ServerClickHandler*).

Changing the Zones Layout

Let's now consider a sample page that includes more zones and Web parts. We'll define two zones—the Information zone and the Miscellaneous zone. The zones occupy two cells in the same row of a table that spans the whole page.

By default, the Information zone contains the BookFinder Web part, and the Miscellaneous zone contains two other sample Web parts—MyFavorites (mentioned earlier) and MsNbcWeather.

The MyFavorites Web part reads a list of favorite links out of a server-side XML file and displays them through its user interface. The *DataSetDataSource* class (see Chapter 6) is used to load the XML data and bind it to the user interface.

```
<%@ control language="C#" classname="MyFavorites"%>
<asp:datasetdatasource runat="server" id="Source" readonly="False"
    datafile="Favorites.xml" />
<asp:datalist id="DataList1" runat="server" datasourceid="Source">
    <headertemplate>
        <b>My Current Favorite List</b><hr size="1">
    </headertemplate>
    <itemtemplate>
        <table>
          <tr>
             <td valign="top" style="WIDTH: 80%"><b>
               <a runat="server" id="TitleLabel" href='<%# Eval("Url") %>'>
               <%# Eval("Title") %></a></b>
             </td></tr>
          <tr>
             <td colspan="2">
               <asp:label id="DescriptionLabel" runat="server"
                    font-names="verdana" font-size="8pt"
                    text='<%# Eval("Description") %>'></asp:label>
             </td></tr>
        </table>
    </itemtemplate>
</asp:datalist>
```

The MsNbcWeather Web part gets weather information about a U.S. city (identified by a ZIP Code) and displays that in the page. Weather information is provided by the MSNBC Web site. In this case, the information is not retrieved through a Web service call. A ready-to-use page on the *http://www.msnbc.com* site returns a JavaScript object filled with weather information. The Web part invokes this URL and then processes the JavaScript code in the client-side *onload* event of the window.

The sample portal page we're building using these three Web parts is shown in Figure 3-6.

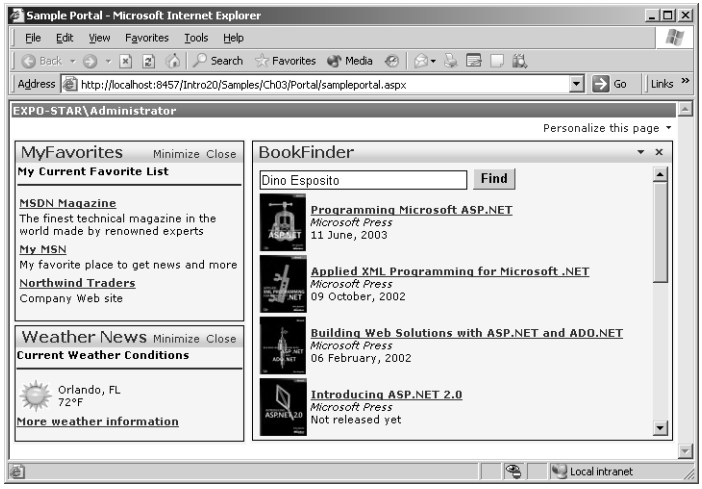

Figure 3-6 The sample portal home page

Note that the two zones have different settings for verbs. In particular, the rightmost zone uses bitmaps for the minimize and close verbs. Two other things in the user interface of the page are worth noting: the login name and the personalization link at the top of the page.

The login name has no meaning other than to serve as a reminder that a portal page is user-specific by design. The sample page uses the new *LoginName* security control to display the name of the current user. (See Chapter 10 for more details about ASP.NET 2.0 security controls.)

When you build portal pages, you should figure out a way to store personalized settings on a per-user basis. The Personalize This Page link button starts a procedure that lets you change the layout of the zones. You attach the personalization link button through the *WebPartPageMenu* control.

```
<asp:WebPartPageMenu ID="WebPartPageMenu1" Runat="server"
    Font-Names="Verdana" Font-Size="8pt"
    Text="Personalize this page">
    <MenuStyle BorderColor="Blue" BorderStyle="Solid"
        Font-Names="Verdana" Font-Size="8pt" BorderWidth="1px" />
</asp:WebPartPageMenu>
```

The control represents a pagewide menu and can be placed anywhere in the page. The page menu does not depend on zones or *WebPart* controls. When clicked, the link button displays a list of options, as shown in Figure 3-7.

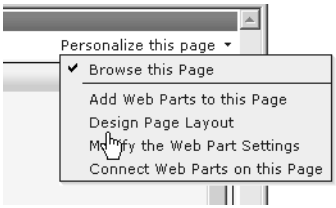

Figure 3-7 The pagewide WebPart menu in its default configuration

The menu items, as well as their state and text, can be modified programmatically. Each menu item represents a verb enabled on the defined Web parts—browse, catalog, design, edit properties, and connect to other Web parts. Each verb is characterized by a property through which you can enable, hide, or label the menu items. The verb properties are *BrowseModeVerb*, *CatalogModeVerb*, *DesignModeVerb*, *EditModeVerb*, and *ConnectModeVerb*. The default mode is Browse.

When you're in design mode (as you can see in Figure 3-8), drag-and-drop facilities let you move Web parts from one zone to another. Once dropped onto a new zone, the Web part inherits the currently active graphical settings, including the title bar and verb settings.

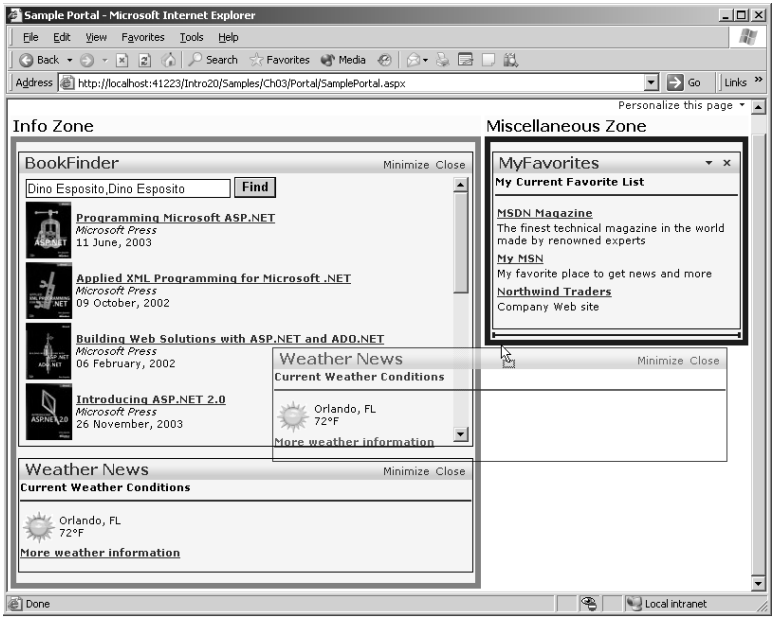

Figure 3-8 In design mode, users can move Web parts around zones using drag-and-drop.

In design mode, each zone sports a border and displays its title text so users can easily spot what Web parts are available for moving. When users finish moving Web parts around, they can click the *Browse* verb to persist the changes.

Persisting Layout Changes

As mentioned, a typical portal page is inherently user-specific. So if the user changes the layout of the zones, the new layout must be stored and used whenever that page is visited. In ASP.NET 2.0, this doesn't require much coding work on your part. All you have to do is configure the application so that it supports personalization. We'll delve into personalization in the next chapter.

In addition to enabling personalization, you must supply (or better yet, declare) a data store. The data store is a Microsoft SQL Server or Microsoft Access database that contains user-specific settings related to personalization. To make the layout changes persistent across application invocations, you run the ASP.NET Configuration applet from the Website menu of Visual Studio 2005. Once in the applet, you click on the Security tab and start the Security Setup Wizard to choose and configure the personalization data store.

When a Web part–driven application finishes the personalization step, modified zone indexes are automatically stored and used to draw the page upon next access, as you can see in Figure 3-9.

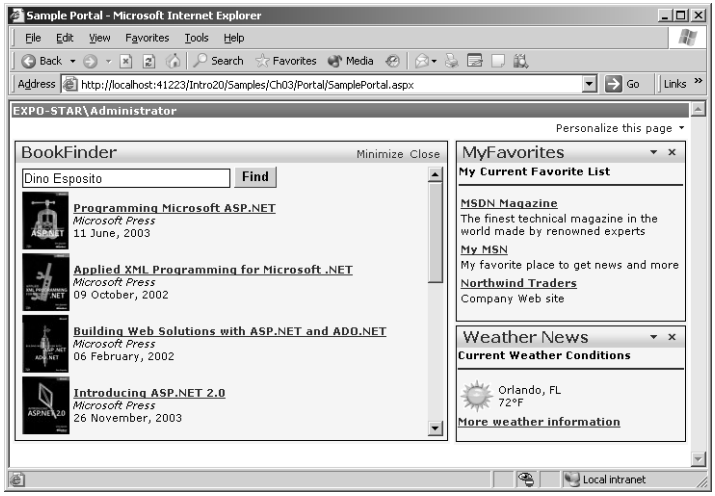

Figure 3-9 The weather Web part has been moved to the Miscellaneous zone and inherits the title and verb local attributes.

Editing and Listing Web Parts

In addition to allowing users to move Web parts from one zone to another, you can allow users to edit various characteristics of the Web parts, including their appearance, layout, and basic behavior. You can also provide users with a list of all available parts and have them choose which ones to activate.

The Web Parts framework provides basic editing and listing functionality for all Web parts. You enable in-place editing by placing one or more editor parts in the page. Listing is enabled by means of a catalog part component. As mentioned, both editing and listing are controlled and activated through the Web part page menu.

Creating an Editor Zone

The first step in enabling dynamic editing on your Web part–driven page is defining an editor zone. The tag to use is *<asp:editorzone>*. You need one editor zone per page. ASP.NET 2.0 supports quite a few types of editors, each designed to edit a particular aspect of Web parts. There are editors to change the values of public and Web browsable properties, the overall behavior of the part, and its layout and appearance.

The Edit Mode

To define an editor zone in a page, you use the following code:

```
<asp:EditorZone runat="server" HeaderText="Enter Your Changes">
    <InstructionTextStyle Font-Names="Verdana" Font-Italic="True" />
    <HeaderStyle BackColor="Blue" ForeColor="White" />
    <ZoneTemplate>
       <asp:AppearanceEditorPart runat="server" />
       <asp:LayoutEditorPart runat="server" />
       <asp:PropertyGridEditorPart runat="server" />
    </ZoneTemplate>
</asp:EditorZone>
```

In addition to specifying some optional style information, you have to create an *<asp:editorzone>* tag and place a *<zonetemplate>* in it. The zone template lists the editors you want to use. The editor zone shows up only in edit mode and appears at the exact position where you defined it in the page. For this reason, you should choose an appropriate placement that doesn't obstruct the editing process.

The Web Parts framework provides a default layout and visual settings for the editor zones. You can change the default settings, though—including the title and the style of the buttons on the footer.

You enter in edit mode, selecting the Modify The Web Parts Settings option on the page menu. When this happens, all Web parts in the page show a little bitmap that represents the edit menu. You click on the menu to make the editors appear, as shown in Figure 3-10.

Figure 3-10 Click on the Edit menu to display all referenced Web Parts editors.

Figure 3-11 shows what happens when the Weather Web part is switched to edit mode.

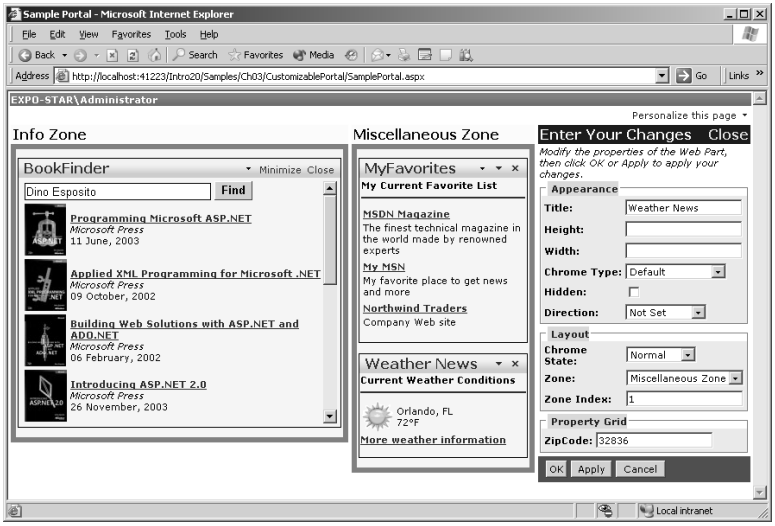

Figure 3-11 When the editors are running, users can change some visual settings for the selected Web part.

The Editor Part Components

Table 3-4 details the editor parts. You can select more than one editor in the same zone. Editors are displayed in the specified order within the editor zone.

Table 3-4 Editor Parts

Editor	Description
AppearanceEditorPart	Lets you edit visual settings such as width, title, direction of the text, and border type.
BehaviorEditorPart	Lets you modify some behavioral settings, such as whether the Web part supports personalization, editing, and minimization. The editor part also lets you edit help and title links.
LayoutEditorPart	Lets you edit the frame style (normal or minimized) and the zone the part belongs to. You can also modify the index of the part within the selected zone.
PropertyGridEditorPart	Lets you edit the custom properties of the Web part component. A custom property is a public property defined on a *WebPart*-derived class marked with the *Personalizable* and *WebBrowsable* attributes.

The footer of the editor zone has a standard toolbar with buttons for saving and exiting (the OK button), saving and continuing (the Apply button), and exiting without saving (the Cancel button). Any change applied during the edit phase is stored in the personalization data store. This feature is provided by the ASP.NET 2.0 framework and requires no additional coding.

> **Important** For the property grid editor to show up, the Web part must have publicly browsable and personalizable properties. The properties must be exposed by the Web part control, not any of its constituent controls. Even though user controls can be employed as Web parts, they have no browsable properties. Adding a browsable property to, say, the .ascx won't work because the property must be exposed by a *WebPart*-derived class. You have to create your own Web part class to be able to edit custom properties through the property grid editor. We'll look at a custom Web part class later in the chapter.

Adding Web Parts Dynamically

A Web part can show a variety of verbs in its title bar, including Close. If you click that button, the Web part is closed and hidden from view, and there's not much you can do to view the Web part again. But is the Web part gone forever? Of course not. Or, more exactly, not if you have designed the Web part page appropriately.

The Catalog Zone

The catalog zone is a Web part component that allows users to add Web parts to the page at run time. A Web parts catalog contains the list of Web parts you want to offer to users. At a minimum, though, the catalog acts as a store for Web parts that the user has removed from the page. The catalog guarantees that no inadvertently closed Web part is lost. You bring up the catalog of a page by choosing the Add Web Parts To This Page menu item.

The following code demonstrates a simple but effective catalog:

```
<asp:CatalogZone runat="server" headertext="Catalog Zone">
    <HeaderVerbStyle Font-Size="8pt" />
    <InstructionTextStyle Font-Italic="True" Font-Size="8pt" />
    <FooterStyle cssclass="EditorZoneFooter" /
>
    <CatalogItemStyle Font-Size="8pt" />
    <PartLinkStyle Font-Names="verdana" Font-Size="8pt" />
    <VerbStyle cssclass="EditorZoneVerb" />
    <HeaderStyle Font-Bold="True" BackColor="Blue" ForeColor="White" />
    <ZoneTemplate>
        <asp:PageCatalogPart runat="server" Title="Available Parts" />
    </ZoneTemplate>
</asp:CatalogZone>
```

Aside from the visual styles that adorn the HTML output of the catalog zone, the only piece of code that really matters is the *<asp:PageCatalogPart>* element. It is the container that will list the available Web parts at run time.

The catalog lists the Web parts that have been closed and gives users a chance to check and add them to one of the existing zones, as you can see in Figure 3-12.

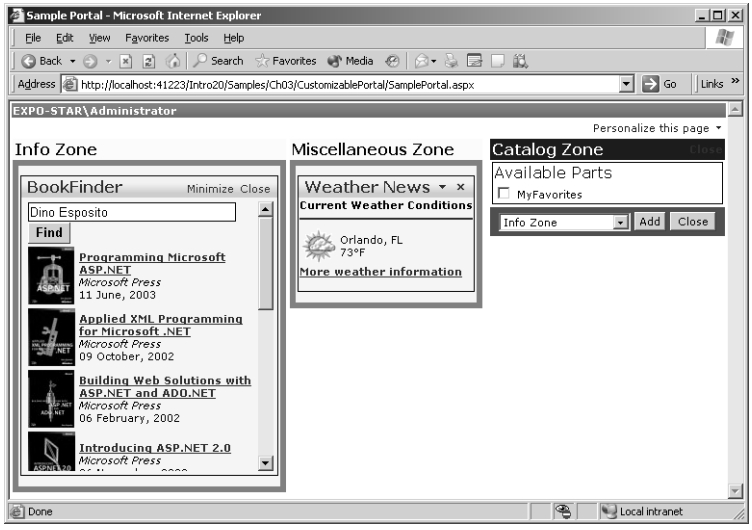

Figure 3-12 The page catalog in action in the sample page

> **Note** The Web Parts framework provides a lot of functionality and a variety of built-in Web dialog boxes. The page catalog and the various editor parts are a few examples. Note that in their native format, these dialog boxes have no visual style set. You are responsible for setting borders, fonts, and colors to give them a professional look.

The Catalog Part Components

The catalog zone can contain two types of catalog parts: a *PageCatalogPart* control and a *DeclarativeCatalogPart* control. As mentioned, the former is a sort of placeholder for the Web parts that the user removes from the page. The *DeclarativeCatalogPart* control contains the list of Web parts that users can add to their page. The Web parts managed by the page catalog are those statically declared in the .aspx source file. The Web parts managed by the *DeclarativeCatalogPart* are not instantiated and managed until they are explicitly added to the page.

The following code defines new externally available Web parts:

```
<ZoneTemplate>
  <asp:declarativecatalogpart runat="server"
       title="Other Parts">
     <webpartstemplate>
         <x:Sample1 Runat="server" id="sample1" />
         <x:Sample2 Runat="server" id="sample2" />
     </webpartstemplate>
   </asp:declarativecatalogpart>
</ZoneTemplate>
```

These Web parts are listed in the catalog side by side with the Web parts declared in the page. You are provided with links to switch between groups of parts. Figure 3-13 shows the appearance of the catalog part.

> **Important** You can also set and control the display mode programmatically. The Web part page menu is helpful if you need to create a fully customizable page, but it forces you to play by its rules and, more importantly, requires advanced browser support. In fact, the menu is displayed through client-side script code based on Dynamic HTML features. You can place simple link buttons in the page and attach some server-side code like the following:
>
> ```
> MyWebPartManager.DisplayMode = WebPartManager.CatalogDisplayMode;
> ```
>
> The code sets the display mode of the Web parts to catalog.

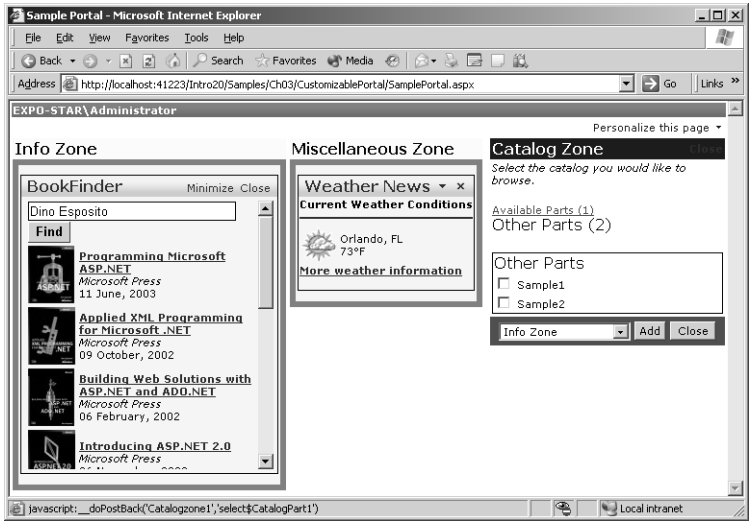

Figure 3-13 The catalog zone lists groups of Web parts that are available to the Web page.

Connecting to Other Web Parts

Web part controls can communicate with other Web parts on the same Web page and exchange data. For this feature to work, each Web part must implement the appropriate interfaces. The communication is one-way and relies on the services of a connection object. The Web part connection object establishes a channel between a Web part control that acts as a provider and a Web part that acts as a consumer.

Two connected Web parts operate in a publisher/subscriber fashion. Any change in the values exposed by the provider are immediately reflected by the consumer. As you can imagine, this model lends itself well to representing master/detail models of data.

The Connection Model

The Web part connection model consists of two interoperating entities—a connection and a connection point. A connection connects two points, one from the provider control and one from the consumer. The connections available in the page are managed by the Web part manager. Web part controls can communicate with more than one other part.

Enabling Web Parts Connectivity

The following code illustrates the key step in a connection-enabled Web page that supports Web parts:

```
<asp:WebPartManager runat="server" id="MyWebPartManager">
    <StaticConnections>
        <asp:connection
             ProviderID="emp"
             ProviderConnectionPointIDname="EmployeeIDProvider"
             ConsumerID="ord"
             ConsumerConnectionPointID="EmployeeIDConsumer" />
    </StaticConnections>
</asp:WebPartManager>
```

All the necessary connection objects are declaratively listed in the body of the Web part manager. A connection is identified by a provider and a consumer object. For both objects, you specify an ID and the name of the corresponding connection point. The *ProviderID* and *ConsumerID* properties must match the ID of existing Web parts. The *ConsumerConnectionPointID* and *ProviderConnectionPointID* properties must match the name of a connection point defined within the Web parts.

> **Note** The Web Parts framework supports static and dynamic connections. Static connections are defined within the body of the *WebPartManager* object and are available to users as soon as they open the page. Dynamic connections enable users to connect and disconnect two Web parts using code.

Connection Points and Interfaces

A connection point defines a possible connection for a *WebPart* control. A connection point doesn't guarantee communication—it simply provides a way for the *WebPartManager* object to establish a communication channel between two parts. A connection point can act as a provider or as a consumer. In the former case, the Web part exposes information through the connection channel that other registered Web parts consume. A consumer connection point, on the other hand, receives incoming data exposed by a provider.

The communication between providers and consumers is defined by a communication contract. The contract set between a provider and a consumer consists of an interface implemented in the provider that the consumer needs to know. This interface can contain properties, events, or methods that the consumer can use once the communication is established. The consumer doesn't need to implement any interface, but it must be aware of the interfaces that its provider supports.

Building a Master/Detail Schema

Let's apply the Web part connection model to a couple of custom Web part controls that inherit from the *WebPart* base class. The provider Web part is named EmployeesWebPart; it exposes the value of employee ID. In addition, the control displays some information about the specified employee.

The consumer component is the *OrdersWebPart* control; it displays all the orders issued by a particular employee. The ID of the employee can be set directly through the programming interface of the component, or it can be automatically detected when the provider Web part signals a change in its state. This link creates a master/detail relationship between the two Web parts.

Provider Web Part Components

When you create a provider Web part, the first thing you define is the communication contract for the connection points. The contract is defined as an interface. The Web part component is a custom ASP.NET control derived from *WebPart* that implements the contract interface.

```
interface IEmployeeInfo
{
   int EmployeeID { get; set; }
}
public class EmployeesWebPart : WebPart, IEmployeeInfo
{
   private int _empID;
   public int EmployeeID
    {
       get { return _empID; }
       set { _empID = value; }
    }
    ⋮
}
```

To make *EmployeeID* show up in the property grid editor, you mark it as browsable and personalizable.

```
[Personalizable(true), WebBrowsable(true)]
public int EmployeeID
{
   get { return _empID; }
   set { _empID = value; }
}
```

To give the Web part a user interface, you can override the *RenderContents* method. Aside from the few features described so far, writing a custom Web part is not much different from writing a custom control.

The next step is creating a provider connection point. You define a function that returns an instance of the current class, and you mark it with the *[ConnectionProvider]* attribute. This function creates the connection point for the data based on the *IEmployeeInfo* interface:

```
[ConnectionProvider("EmployeeIDProvider", "EmployeeIDProvider")]
private IEmployeeInfo ProvideEmployeeInfo()
{
    return this;
}
```

Notice that the name of the connection point must match the *ProviderName* or the *ConsumerName* property of the *<asp:connection>* tag, depending on whether the connection point is for a provider or a consumer.

> **Note** When the *WebPart* provider control implements just one provider interface, as in this case, there's no need to explicitly mention the interface in the connection provider attribute. When multiple interfaces are supported, you must add a third parameter to the *[ConnectionProvider]* attribute to indicate the contract on which the connection is based.
>
> ```
> [ConnectionProvider["Prov", "Prov", typeof(IMyInterface)]
> ```

The sample *EmployeesWebPart* control retrieves and displays some information about the specified employee in SQL Server's Northwind database. Figure 3-14 shows its user interface.

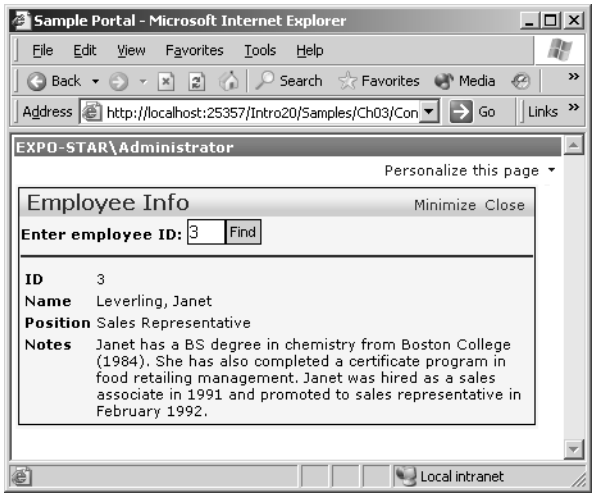

Figure 3-14 The *EmployeesWebPart* control in action

Consumer Web Part Components

A Web part that acts as a consumer is even simpler to write than a provider. Besides generating its own user interface, the Web part has only one duty—creating a consumer connection point for the specified interface.

```
[ConnectionConsumer("EmployeeIDConsumer", "EmployeeIDConsumer")]
private void GetEmployeeInfo(IEmployeeInfo empInfo)
{
   if (empInfo != null)
   {
      _empID = empInfo.EmployeeID;
      FindEmployeeInfo();
   }
   else
      throw new Exception("No connection data found.");
}
```

The ASP.NET runtime creates a consumer connection point that corresponds to a method marked with the *[ConnectionConsumer]* attribute. The method marked with the attribute is taken as the callback to invoke when anything on the specified interface changes.

The user interface is composed using a *DataGrid* control. The grid is filled with the results of a query run against the Orders table in the Northwind database.

Putting It All Together

The two custom Web parts that support connection points must be added to the page using a custom prefix, just like any other custom control. First you compile the two files to an assembly, and then you link it to the page using the *@Register* directive:

```
<%@ Register tagprefix="x" Namespace="Samples" Assembly="MyWebParts" %>
```

If, for some reason, the two classes belong to different namespaces, you use two different prefixes. The code that inserts the Web parts looks like the following:

```
<x:EmployeesWebPart runat="server" id="emp"
     Title="Employee Info" />
<x:OrdersWebPart runat="server" id="ord"
     Scrollbars="Auto" Height="200px"
     Title="Orders 1997" />
```

Let's briefly review the markup code that defines a Web part connection object within the page.

```
<StaticConnections>
    <asp:connection
       ProviderID="emp" ProviderConnectionPointID="EmployeeIDProvider"
       ConsumerID="ord" ConsumerConnectionPointID="EmployeeIDConsumer" />
</StaticConnections>
```

This declaration can be read as a connection set between a Web part with an ID of *emp* and a Web part named *ord*. The former acts as the provider through a connection point name *EmployeeIDProvider*. The latter plays the role of the consumer through a connection point named *EmployeeIDConsumer*.

As a result, any change in any of the properties exposed by the provider results in an internal field-changed event that is resolved, invoking the consumer's callback. The consumer retrieves and displays the orders for the specified employee. The two Web parts work perfectly in sync, as Figure 3-15 shows.

The provider Web part also defines a public and browsable *EmployeeID* property. If you set the *EmployeeID* property on the *EmployeesWebPart* control (the provider), the change is immediately reflected by the consumer, as you can see in Figure 3-16.

Note that, by design, the connection model is a one-way model—to keep the controls completely in sync, you need a second connection in which the provider and consumer roles are swapped. For example, suppose you add a public, browsable *EmployeeID* property to the Orders Web part. To reflect any property changes to the Employees Web part, you must create a second pair of connection points that are completely independent from the first pair.

Figure 3-15 A master/detail relationship set using two independent but communicating Web parts

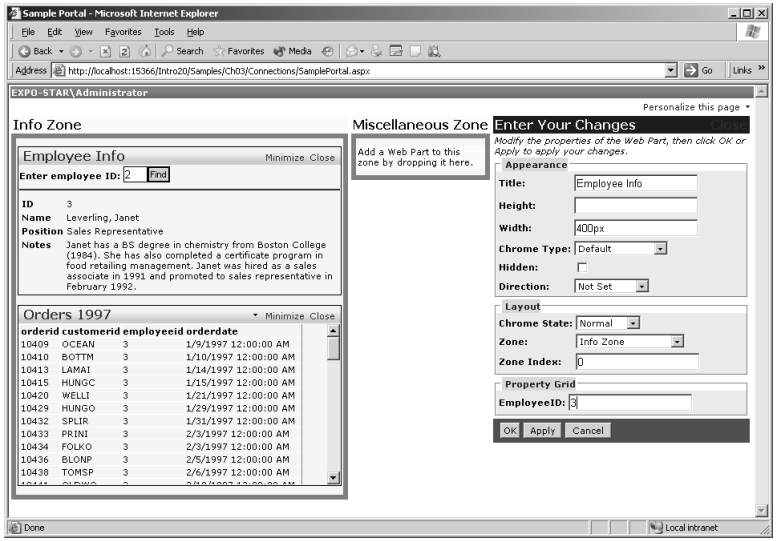

Figure 3-16 Changes to public properties tied to a connection point make the consumer refresh its user interface.

Summary

The Web Parts framework provides a simple and familiar way for ASP.NET developers to create modular Web applications that support end-user personalization. A Web part is a panel-like server control that displays some user interface elements. Like any other server control, it is configurable through properties, methods, and events.

Web parts are integrated into a framework aimed at composing pages with components that are smarter and richer than traditional controls. The surrounding Web Parts framework provides all the magic (or hard infrastructure code, if you will). In particular, you can partition the surface of your Web page into zones and bind one or more parts to each zone. Each Web part is automatically given a frame, a title bar, and some verbs (such as minimize, edit, and close). Overall, a Web part looks like a traditional window of a desktop application.

The Web Parts framework supports a variety of working modes, including design, edit, and catalog. In design mode, users can use drag-and-drop and move parts around, changing the layout of the page. In edit mode, users can also change visual and behavioral properties. The user interface of the editors is provided, free of programming charge, by the Web Parts framework. Finally, in catalog mode the framework lists all available Web parts, including those that the user might have previously closed.

When a user reconfigures the Web parts on a page, the user's settings are automatically persisted. The next time the user visits the page, the last Web parts configuration is restored. Web parts settings are persisted using page personalization. The only requirement is that page personalization be enabled for the page. No code is required to store the user settings, but the page personalization engine must be configured offline. You'll see a lot about page personalization in the next chapter.