Chapter 14

Personalization with Profiles and Web Parts

Web applications often need to track user information over time. Sometimes, this information is mission critical, such as customer orders, shipping addresses, and so on. A site may also want to track information that is not absolutely essential to the application, but is helpful in terms of the overall user experience with the site. This type of information is often referred to as ***personalization information***. Examples of personalization information can include user preference information such as the language or theme to use in the site, the size of the font, shopping interests, and so on. Personalization can also include information that is visit oriented, but which needs to be more persistent than session state. Some examples of this type of information are a shopping cart, a wish list, or a list of the last *N* products viewed by the user.

Prior to ASP.NET 2.0, developers needed to create this functionality from scratch, typically using tables relationally linked to a user table. ASP.NET 2.0 introduced two new features that help developers manage and implement personalization in their sites. These features are the profile system and the Web parts framework and are the subject of this chapter.

ASP.NET Profiles

The profile feature is a way of easily associating and persisting application information with users. You can use profiles to track any user-related information, from personalization data to more mission-critical information such as customer names, addresses, financial information, and so on.

The profile feature is yet another ASP.NET subsystem that is built using the provider model (see Chapter 13 for more information on the provider model). As such, the actual data storage mechanism for the profile information is independent of its usage. By default, ASP.NET uses the `SqlProfileProvider`, which uses the same SQL Server database as the membership and role management features examined in the previous chapter (i.e., `ASPNETDB.MDF` in the `App_Data` folder). However, if you have an existing database of user or personalization data, you might decide instead to create your own custom profile provider.

Profile data can be easily defined within the `Web.config` file and is consumed using the familiar properties programming model. It can be used to store data of any type, as long as it is serializable. Unlike session state data, which must be cast to the appropriate type, profile data is made available via strongly typed properties.

Defining Profiles

Perhaps the best thing about the profile feature is how easy it is to set up. You define the data that is to be associated with each user profile via the `profile` element in the `Web.config` file. Within this element, you simply define the properties whose values you want to persist. Each property consists of a name, a type, and perhaps a default value as well as a flag indicating whether this property is available to anonymous users. For instance, Listing 14.1 illustrates the configuration of three string properties.

Listing 14.1 Web.config

```
<configuration>
   <system.web>
      <authentication mode="Forms" >
         <forms loginUrl="LoginSimple.aspx" timeout="2" />
      </authentication>
      <authorization>
         <deny users="?"/>
      </authorization>
      …
      <profile>
         <properties>
            <add name="BackgroundColor" type="string"
```

```
            defaultValue="White"
            allowAnonymous="false" />
         <add name="TextColor" type="string"
            defaultValue="Black"
            allowAnonymous="false"/>
         <add name="Region" type="string"
            defaultValue="NorthAmerica"
            allowAnonymous="false"/>
      </properties>
    </profile>
  </system.web>
</configuration>
```

CORE NOTE

It is important to remember that if you do not explicitly allow anonymous use, the user must be logged in before using the page's `Profile` property; otherwise, a runtime exception is generated.

Table 14.1 lists the notable configuration properties for the profile properties `add` element.

Table 14.1 Configuration for the Profile add Element

Name	Description
`allowAnonymous`	Indicates whether the property can be set by anonymous users. The default is `false`.
`customProviderData`	A string of custom data to be used by the profile provider. Individual providers can implement custom logic for using this data.
`defaultValue`	The initial default value used for the property.
`name`	The name of the property.
`provider`	The profile provider to use (if not using the default profile provider). By default, all properties are managed using the default provider specified for profile properties, but individual properties can also use different providers.
`readOnly`	Indicates whether the property is read-only. The default is `false`.

Table 14.1 Configuration for the Profile add Element *(continued)*

Name	Description
`serializeAs`	Specifies the serialization method to use when persisting this value. Possible values are described by the `SerializationMode` enumeration: `Binary`, `ProviderSpecific` (the default), `String`, and `Xml`.
`type`	The data type of the property. You can specify any .NET class or any custom type that you have defined. If you use a custom type, you must also specify how the provider should serialize it via the `serializeAs` property.

Using Profile Data

After the profile properties are configured, you can store and retrieve this data within your pages via the `Profile` property (which is an object of type `ProfileCommon`), of the `Page` class. Each profile defined via the `add` element is available as a property of this `Profile` property. For instance, to retrieve the `Region` value, you could use the following.

```
string sRegion = Profile.Region;
```

Notice that you do not have to do any typecasting, because the profile properties are typecast using the value specified by the `type` attribute for each property in the `Web.config` file.

You can save this profile value in a similar manner as shown in the following.

```
Profile.Region = txtRegion.Text;
```

This line (eventually) results in the profile value being saved by the profile provider for the current user. Where might you place these two lines of code? A common pattern is to retrieve the profile values in the `Page_Load` handler for the page and then set the profile value in some type of event handler as a result of user action.

Listings 14.2 and 14.3 illustrate a sample page that makes use of the three profile properties defined in Listing 14.1. You may recall that the `Web.config` file in Listing 14.1 requires that the user must first log in. After authentication, the page uses the default region, background color, and text color defined in Listing 14.1. The user can then change these profile values via three drop-down lists (see Figure 14.1). When the user clicks the Save Settings button, the event handler for the button changes the page's colors as well as saves the values in the profile object for the page.

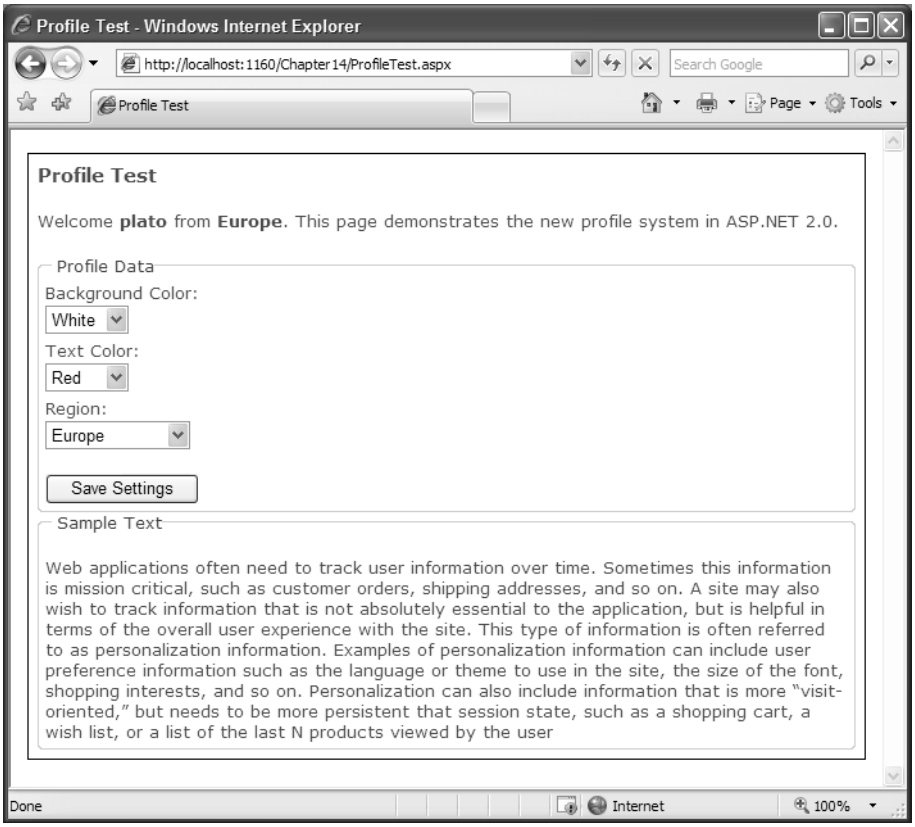


Figure 14.1 ProfileTest.aspx

Listing 14.2 ProfileTest.aspx

```
<h1>Profile Test</h1>
<p>Welcome
<asp:LoginName ID="logName" runat="server" Font-Bold="true" /> from
<asp:Label ID="labWhere" runat="server" Font-Bold="true" />.
This page demonstrates the new profile system in ASP.NET 2.0.
</p>
<asp:Panel ID="panForm" runat="server" GroupingText="Profile Data">
   <asp:label ID="labBack" runat="server"
      AssociatedControlID="drpBackColor"
      Text="Background Color:" />
   <asp:DropDownList ID="drpBackColor" runat="server" >
      <asp:ListItem>Black</asp:ListItem>
      <asp:ListItem>Green</asp:ListItem>
      <asp:ListItem>Red</asp:ListItem>
```

```
      <asp:ListItem>Yellow</asp:ListItem>
      <asp:ListItem>White</asp:ListItem>
   </asp:DropDownList><br />

   <asp:label ID="labText" runat="server"
    AssociatedControlID="drpTextColor" Text="Text Color:" />

   <asp:DropDownList ID="drpTextColor" runat="server" >
      <asp:ListItem>Black</asp:ListItem>
      <asp:ListItem>Green</asp:ListItem>
      <asp:ListItem>Red</asp:ListItem>
      <asp:ListItem>Yellow</asp:ListItem>
      <asp:ListItem>White</asp:ListItem>
   </asp:DropDownList><br />
   <asp:label ID="labRegion" runat="server"
      AssociatedControlID="drpRegion" Text="Region:" />

   <asp:DropDownList ID="drpRegion" runat="server" >
      <asp:ListItem>Africa</asp:ListItem>
      <asp:ListItem>Asia</asp:ListItem>
      <asp:ListItem>Europe</asp:ListItem>
      <asp:ListItem>North America</asp:ListItem>
      <asp:ListItem>Oceania</asp:ListItem>
      <asp:ListItem>South America</asp:ListItem>
   </asp:DropDownList>
   <p>
   <asp:Button ID="btnSave" runat="server" Text="Save Settings"
      OnClick="btnSave_Click" />
   </p>
</asp:Panel>
<asp:Panel ID="panSample" runat="server"
   GroupingText="Sample Text">

   <p>
   Web applications often need to track user information over time.
Sometimes this information is mission critical, such as customer
orders, shipping addresses, and so on. A site may also wish to track
information that is not absolutely essential to the application, but
is helpful in terms of the overall user experience with the site.
This type of information is often referred to as personalization
information. Examples of personalization information can include
user preference information such as the language or theme to use in
the site, the size of the font, shopping interests, and so on.
Personalization can also include information that is more "visit-
oriented," but needs to be more persistent than session state, such
as a shopping cart, a wish list, or a list of the last N products
viewed by the user
   </p>
</asp:Panel>
```

The code-behind for this page is reasonably straightforward. The first time the page is requested, it sets the page's colors based on the current profile values, and sets the default values of the page's controls to these same profile values. The event handler for the Save Settings button simply sets the profile properties to the new user-chosen values. Everything else is handled by the profile system and its provider, so that the next time this user requests this page, whether in five minutes or five years, it uses these saved profile values.

Listing 14.3 ProfileTest.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class ProfileTest : System.Web.UI.Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      if (! IsPostBack)
      {
         SetPageData();

         // Set initial values of controls to profile values
         drpRegion.SelectedValue = Profile.Region;
         drpBackColor.SelectedValue = Profile.BackgroundColor;
         drpTextColor.SelectedValue = Profile.TextColor;
      }
   }
   protected void btnSave_Click(object sender, EventArgs e)
   {
      // Save the profile values based on user input
      Profile.Region = drpRegion.SelectedValue;
      Profile.BackgroundColor = drpBackColor.SelectedValue;
      Profile.TextColor = drpTextColor.SelectedValue;

      SetPageData();
   }
   private void SetPageColors()
   {
      labWhere.Text = Profile.Region;
```

```
        body1.Attributes["bgcolor"] = Profile.BackgroundColor;
        body1.Attributes["text"] = Profile.TextColor;
    }
}
```

Profile Groups

You can define hierarchical data within the profile element using the `group` element. For instance, you could define the following two profile groups in the `Web.config` file.

```
<profile>
   <properties>
      <group name="UserPreferences">
         <add name="MasterPage" type="string"
            defaultValue="Main"/>
         <add name="Theme" type="string" defaultValue="Cool" />
      </group>
      <group name="Visit">
         <add name="Last5Products"
           type=
            "System.Collections.Specialized.StringCollection" />
      </group>
   </properties>
</profile>
```

You can access grouped properties using dot notation, as shown here.

```
StringCollection products = Profile.Visit.Last5Products;
Profile.UserPreferences.Theme = "CoolTheme";
```

CORE NOTE

You cannot nest profile groups within another profile group.

How Do Profiles Work?

The profile system uses the provider model introduced in Chapter 13 so that some type of profile provider handles the actual responsibility of saving and retrieving items from the data store. But how does the `Profile` property of the page "know" about the profile properties defined in the `Web.config` file?

When a page is requested from a Web application that has profile properties defined in its `Web.config` file, ASP.NET automatically generates a class named `ProfileCommon` in the `Temporary ASP.NET Files` directory. In the following

example, you can see the `ProfileCommon` class that was generated from the profiles in Listing 14.1.

```
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:2.0.50727.42
//
//   Changes to this file may cause incorrect behavior and will
//   be lost if the code is regenerated.
// </auto-generated>

using System;
using System.Web;
using System.Web.Profile;

public class ProfileCommon : System.Web.Profile.ProfileBase {

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

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

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

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

As you can see, this generated class is a subclass of the `ProfileBase` class and contains strongly typed properties for each profile property defined in the `Web.config` file. This class is available to the page via a `Profile` property that is added to the Web Form by the page parser.

Saving and Retrieving Profile Data

One of the real benefits of using the profile system is that the values for each user are automatically saved and retrieved by the system. The profile system works via an `HttpModule` named `ProfileModule`. You may recall from Chapter 2, all incoming requests pass through the events in the `HttpApplication` pipeline and, as part of this process, different HTTP modules can register their own event handlers with the events in the pipeline.

The `ProfileModule` registers its event handlers for the `AcquireRequestState` and `EndRequest` events of the pipeline, as well as registers its own `Personalization` event (which is called before the profile data is loaded and which can be handled in the `global.asax` file). The `ProfileModule` does *not* populate the profile object with data during the `AcquireRequestState` event, because this necessitates retrieving data from the profile provider for every request, regardless of whether the page used profile data. Instead, an instance of `ProfileCommon` is created and populated with data (via the `ProfileBase.Create` method) from the provider by `HttpContext` during the page's first request for the `Profile` property.

The data within the `ProfileCommon` object is potentially saved during the `EndRequest` event of the request pipeline. The data in this object is serialized and then saved by the provider if the data has been changed by the page. For the default SQL Server profile provider, simple primitive data is serialized into a string; for more complex data types, such as arrays and custom types, the data is serialized into XML. For instance, the sample profile properties from Listing 14.1 are serialized into the following string and then saved into the `PropertyValuesString` field of the `aspnet_Profile` table.

```
YellowRedEurope
```

As well, there is a `PropertyNames` field in this table that specifies how to deserialize this string data (it indicates the starting position for each property and its length).

```
BackgroundColor:S:0:6:TextColor:S:6:3:Region:S:9:6:
```

In summary, using profiles in your page potentially adds two requests to the data source. A page that uses profile data always reads the profile data at the beginning of the request. Because the profile system uses the lazy initialization pattern (see Chapter 12), if your page does not access the `Profile` object, it does not read the profile data from the data source. When the page has finished the request processing, it checks to

see if the profile data needs to be saved. If all the profile data consists of strings or simple primitive data types such as numbers or Booleans, the profile system checks if any of the properties of the page's `ProfileCommon` object are *dirty* (that is, have been modified) and only saves the data if any of them has been modified. If your profile object does contain any complex type, the page *always* saves the profile data, regardless of whether it has been changed.

Controlling Profile Save Behavior

If your profile data includes nonprimitive data types, the profile system assumes the profile data is dirty and thus saves it, even if it hasn't been modified. If your site includes many pages that use complex profile data, but few that actually modify it, your site will do a lot of unnecessary profile saving. In such a case, you may want to disable the automatic saving of profile data and then explicitly save the profile data only when you need to.

To disable autosaving, simply make the following change to the `profile` element in your application's `Web.config` file.

```
<profile automaticSaveEnabled="false" >
```

After the automatic profile saving is disabled, it is up to the developer to save the profile data by explicitly calling the `Profile.Save` method. For instance, you could change the code-behind in your example in Listing 14.3 to the following.

```
protected void btnSave_Click(object sender, EventArgs e)
{
   Profile.Region = drpRegion.SelectedValue;
   Profile.BackgroundColor = drpBackColor.SelectedValue;
   Profile.TextColor = drpTextColor.SelectedValue;
   Profile.Save();

   SetPageColors();
}
```

Using Custom Types

Profile properties can be any .NET type or any custom type that you have defined. The only requirement is that the type must be serializable. For instance, imagine that you have a custom class named `WishListCollection` that can contain multiple `WishListItem` objects. If you want to store this custom class in profile storage, both it and its contents must be serializable. You can do so by marking both classes with the `Serializable` attribute, as shown here.

```
[Serializable]
public class WishListCollection
{
```

```
  …
}

[Serializable]
public class WishListItem
{
  …
}
```

Along with allowing the use of custom types for profile properties, the profile system allows you to define the profile structure programmatically using a custom type rather than declaratively in the `Web.config` file. You can do this by defining a class that inherits from the `ProfileBase` class and then referencing it via the `inherits` attribute of the `profile` element in the `Web.config` file. For instance, Listing 14.4 illustrates a sample derived profile class. Notice that it inherits from the `ProfileBase` class. Notice as well that each property is marked with the `SettingsAllowAnonymous` attribute; this is equivalent to the `AllowAnonymous` attribute that was used in the `Web.config` example in Listing 14.1.

Listing 14.4 CustomProfile.cs

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Profile;
using System.IO;

public class CustomProfile: ProfileBase
{
   [SettingsAllowAnonymous(false)]
   public List<WishListItem> WishList
   {
      get {
         if (base["WishList"] == null)
            base["WishList"] = new List<WishListItem>();
         return (List<WishListItem>)base["WishList"];
      }
      set { base["WishList"] = value; }
   }

   [SettingsAllowAnonymous(true)]
   public string Theme
   {
      get {
         string theme = (string)base["Theme"];
```

```
        // Set up a default theme if necessary
        if (theme.Length == 0)
        {
            theme = "Cool";
            base["Theme"] = theme;
        }
        return theme;
    }
    set
    {
        string newThemeName = value;
        string path =
            HttpContext.Current.Server.MapPath("~/App_Themes");
        if (Directory.Exists(path))
        {
            // Retrieve array of theme folder names
            String[] themeFolders =
                Directory.GetDirectories(path);

            // See if the specified theme name actually exists
            // as a theme folder
            bool found = false;
            foreach (String folder in themeFolders)
            {
                DirectoryInfo info = new DirectoryInfo(folder);
                if (info.Name == newThemeName)
                    found = true;
            }
            // Only set the theme if this theme exists
            if (found)
                base["Theme"] = newThemeName;
        }
    }
}

[SettingsAllowAnonymous(true)]
public ViewedProducts Last5Products
{
    get {
        if (base["Last5Products"] == null)
            base["Last5Products"] = new ViewedProducts();
        return (ViewedProducts)base["Last5Products"];
    }
    set { base["Last5Products"] = value;  }
}

// This property is explained shortly
// in section on migrating anonymous profiles
[SettingsAllowAnonymous(false)]
```

```
    public bool MigratedAlready
    {
        get { return (bool)base["MigratedAlready"]; }
        set { base["MigratedAlready"] = value; }
    }
}
```

With this custom profile class defined, you can use it by changing your `Web.config` file as shown in the following example. Notice that you can still supply additional declarative properties to the profile if you want.

```
<profile inherits="CustomProfile">
    <properties>
        <add name="BackgroundColor" type="string" />
        …
    </properties>
</profile>
```

The main advantage of using a custom provider class is that the developer can better control what happens when the profile's accessors (i.e., the getters and setters) are used. For instance, in Listing 14.4, the setter for the `Theme` property first verifies that the specified theme actually exists as a real theme folder before setting the value. Just like we saw back in Chapter 11 with custom business objects and entities, a custom profile class allows you to implement some type of business logic with your profile data. As well, another advantage of using a custom provider class is that you can define a profile property that uses a generic collection, as illustrated in Listing 14.4.

Working with Anonymous Users

As you have seen, by default, the profile properties in ASP.NET are only available to authenticated users. However, you can make profiles available to anonymous users. This involves two steps. The first step is to mark individual properties as allowing ***anonymous access***. You can do this declaratively using the `allowAnonymous` attribute.

```
<profile>
    <properties>
        <add name="BackgroundColor" type="string"
            allowAnonymous="true" />
        <add name="TextColor" type="string"
            allowAnonymous="false" />
        …
    </properties>
</profile>
```

If you are using a custom profile class, you can change the `SettingsAllowAnonymous` attribute to each property in the class.

```
public class CustomProfile: ProfileBase
{
   [SettingsAllowAnonymous(true)]
   public string BackgroundColor
   {
     …
   }

   [SettingsAllowAnonymous(false)]
   public string TextColor
   {
     …
   }
   …
}
```

The second step is to enable the ***anonymous identification*** feature for the Web application as a whole via the `anonymousIdentification` element.

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

By default, anonymous identification is disabled. By enabling it, ASP.NET ensures that the `AnonymousIdentificationModule` is involved in the request pipeline. This module creates and manages anonymous identifiers for an ASP.NET application. When an anonymous user makes a request for a resource in an application that has anonymous identification enabled, the `AnonymousIdentificationModule` generates a ***globally unique identifier*** (this is universally referred to as a GUID). A ***GUID*** is a 128-bit integer (16 bytes) that can be used across all computers and networks wherever a unique identifier is required. Such an identifier has a very low probability of being duplicated. At any rate, after generating the GUID, the module writes it to a persistent cookie and this GUID becomes the anonymous user's name.

CORE NOTE

Although each new anonymous user session is assigned a new GUID (because the GUID is written to a cookie on the computer), if a different person accesses the site using this same computer, the site uses the same anonymous GUID issued to the first user.

Deleting Anonymous Profiles

Because every new anonymous user session generates a new GUID, a site that receives many anonymous requests may end up with many profile records for all these anonymous requests, most of which are inactive. In this case, you may need to run some type of scheduled task that deletes these inactive profiles because there is no automatic built-in mechanism for doing so. You can do this via the `ProfileManager` class. This class can be used to manage profile settings, search for user profiles, and delete user profiles that are no longer in use. Table 14.2 lists the methods of the `ProfileManager` class.

Table 14.2 Methods of the ProfileManager Class

Method	Description
`DeleteInactiveProfiles`	Deletes user profiles for which the last activity date occurred before the specified date.
`DeleteProfile`	Deletes the profile for the specified user name.
`DeleteProfiles`	Deletes the specified list of profiles.
`FindInactiveProfilesByUserName`	Retrieves a collection of `ProfileInfo` objects in which the last activity date occurred on or before the specified date and the user name for the profile matches the specified user name.
`FindProfilesByUserName`	Retrieves a collection of `ProfileInfo` objects that matches the specified user name.
`GetAllInactiveProfiles`	Retrieves a collection of `ProfileInfo` objects in which the last activity date occurred on or before the specified date.
`GetAllProfiles`	Retrieves a collection of `ProfileInfo` objects that represents all the profiles in the profile data source.
`GetNumberOfInactiveProfiles`	Gets the number of profiles in which the last activity date occurred on or before the specified date.
`GetNumberOfProfiles`	Gets the total number of profiles in the profile data source.

For instance, imagine that you have some type of administrative page in which you want to display all the authenticated profiles and all the anonymous profiles in two separate `GridView` controls. The code for this might look like the following.

```
protected void Page_Load(object sender, EventArgs e)
{
   // Get all the authenticated profiles
   grdProfiles.DataSource = ProfileManager.GetAllProfiles(
       ProfileAuthenticationOption.Authenticated);
   grdProfiles.DataBind();

   // Get all the anonymous profiles
   grdAnonProfiles.DataSource = ProfileManager.GetAllProfiles(
       ProfileAuthenticationOption.Anonymous);
   grdAnonProfiles.DataBind();
}
```

Notice that the `ProfileManager.GetAllProfiles` method uses the `ProfileAuthenticationOption` enumeration to specify which profiles to retrieve. If you want to delete the anonymous profiles in which there has been no activity for the last day, you could use the following.

```
// Delete everything created one day ago or later
DateTime dateToDelete = DateTime.Now.Subtract(new TimeSpan(1,0,0));
ProfileManager.DeleteInactiveProfiles(
   ProfileAuthenticationOption.Anonymous, dateToDelete);
```

You can also delete just a specific profile as well. For instance, if the `GridView` of authenticated users has a delete command column, you could allow the user to delete a specific authenticated user using the following.

```
protected void grdProfiles_RowCommand(object sender,
   GridViewCommandEventArgs e)
{
   if (e.CommandName == "Delete")
   {
      // Retrieve user name to delete
      int rowIndex = Convert.ToInt32(e.CommandArgument);
      string username = grdProfiles.Rows[rowIndex].Cells[1].Text;

      ProfileManager.DeleteProfile(username);
   }
}
```

CORE NOTE

In the code samples that can be downloaded for this book, there is a completed profile manager page that allows an administrator to view and delete authenticated and anonymous user profiles. Figure 14.2 illustrates how this sample profile manager appears in the browser.

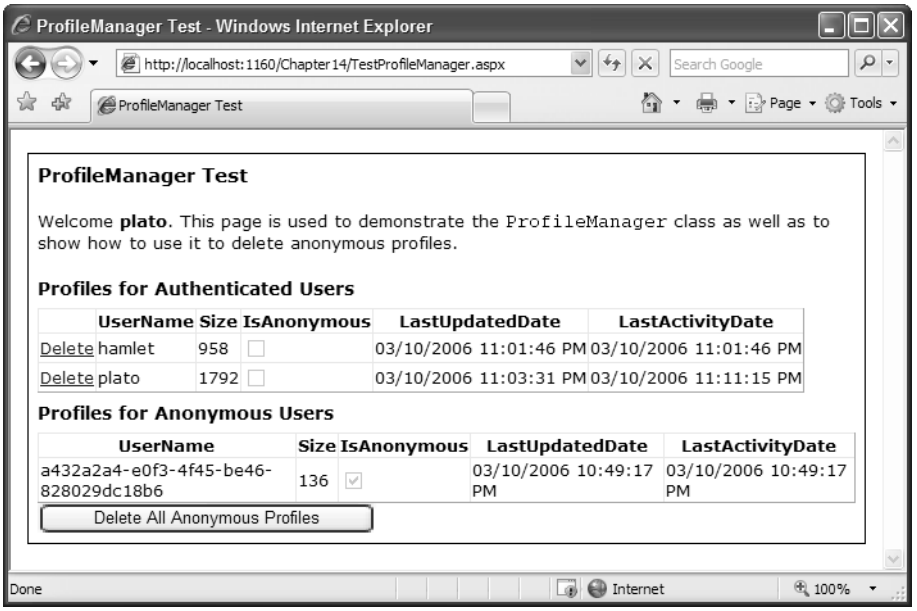


Figure 14.2 Sample profile manager

Migrating Anonymous Profiles

If your site supports profiles for both authenticated and anonymous users, you might want the ability to migrate a user's anonymous profile to his authenticated profile. That is, if an anonymous user sets some profile values, then registers and logs on, the profiles the user set while anonymous should be copied to the new profile the user has as an authenticated user.

The profile system provides a mechanism for this scenario. You can handle this situation by creating a `MigrateAnonymous` event handler for the `ProfileModule` in the `global.asax` file. This handler copies the profile properties from the anonymous profile into the profile for the just-logged-in user. However, there is a potential problem with this mechanism. Recall that before the user can log in, the system considers the user to be an anonymous user. Unfortunately, the `MigrateAnonymous`

event is triggered every time the user logs in. This means you would likely overwrite the user's actual profile with the anonymous profile each time the user logs in. One way to handle this situation is to add some type of flag to the profile to indicate whether the profile has already been migrated. The custom profile class in Listing 14.4 has such a flag property, named `MigratedAlready`.

You can set up the `MigrateAnonymous` event handler in the `global.asax` file as shown in the following example. Notice that it also deletes the anonymous profile and its cookie.

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

<script runat="server">
   void Application_Start(object sender, EventArgs e)
   {
      …
   }
   …
   public void Profile_OnMigrateAnonymous(object sender,
      ProfileMigrateEventArgs args)
   {
      // Retrieve the anonymous profile
      ProfileCommon anonymousProfile =
         Profile.GetProfile(args.AnonymousID);

      // Only migrate the values if the logged in profile hasn't
      // already been migrated
      if (!Profile.MigratedAlready)
      {
         // Copy the anonymous profile values to
         // the logged in profile
         Profile.Theme = anonymousProfile.Theme;
         Profile.WishList = anonymousProfile.WishList;
         Profile.Last5Products = anonymousProfile.Last5Products;
      }

      // Because the user is now logged on, make sure you don't
      // overwrite the profile in the future by setting the
      // MigratedAlready flag
      Profile.MigratedAlready = true;

      // Delete the anonymous profile. If the anonymous ID is not
      // needed in the rest of the site, remove the
      // anonymous cookie.
      ProfileManager.DeleteProfile(args.AnonymousID);
      AnonymousIdentificationModule.ClearAnonymousIdentifier();
   }

</script>
```

When to Use Profiles

The profile system is ideal for user-related data in which you do not have an already existing database schema. If you already have an existing database of user or personalization data, you may prefer instead to use a similar approach to that covered in Chapter 11—namely, create entity or business object classes for this data along with necessary data access classes, rather than design a custom profile provider for this existing data. Even if you do not already have an existing database schema for user information, you might want to avoid using the default profile provider for mission-critical or large amounts of data, because the profile system is not exactly the most optimized way to store data. Although the profile system can serialize its data into string or XML or binary format, serialization is never as efficient as the pure binary approach used by a database.

In my opinion, the profile system is best used for tracking user information that is generated by the application itself, and not for data that is actually entered by the user. For instance, in a site that contains forms for the user to enter her mailing addresses, credit card information, shoe size, and so on, it is best to design database tables for containing this information, and then use business objects/entities and data access classes for working with this data. This way, you can add any validation, application-logic processing, or other behaviors to these classes.

The profile system is ideal for customizing the site based on the user's preferences and the user's behavior within the site, because this type of data either necessitates little validation or logic (e.g., use preferences) or can be generated by the system itself without the user's knowledge (e.g., tracking user behaviors).

In the extended example that you can download for this chapter, you can see both types of customization demonstrated. This sample site (see Figures 14.3, 14.4, and 14.5) uses roughly the same master page-based theme selection mechanism used in the listings at the close of Chapter 6, except here it uses the profile mechanism rather than session state to preserve the user's theme choice.

The site also keeps track of (and displays) the last five books browsed by the user in the single book display page (see Figure 14.4). Both this tracking and the user's theme selection are enabled for both authenticated and unauthenticated users.

Finally, if the user is authenticated, the profile system is used to maintain the user's wish list of books (shown in Figure 14.5).

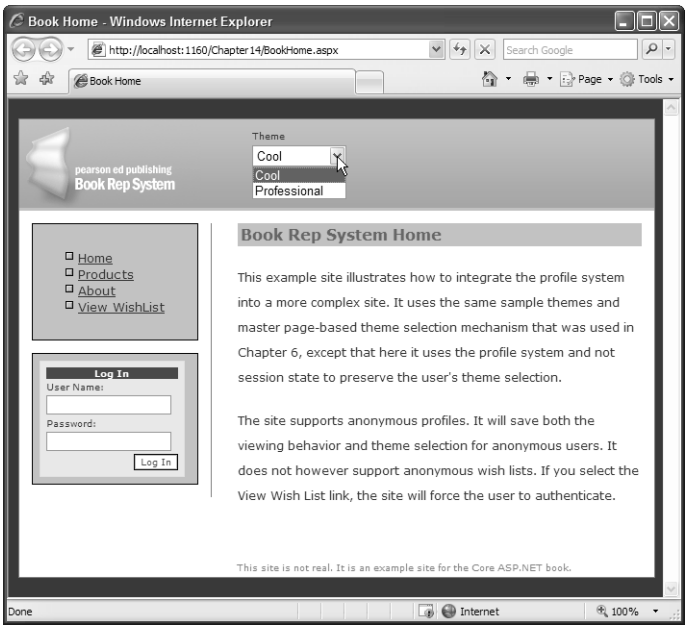


Figure 14.3 Setting the site theme

Figure 14.4 Tracking the user's page visits with the profile system

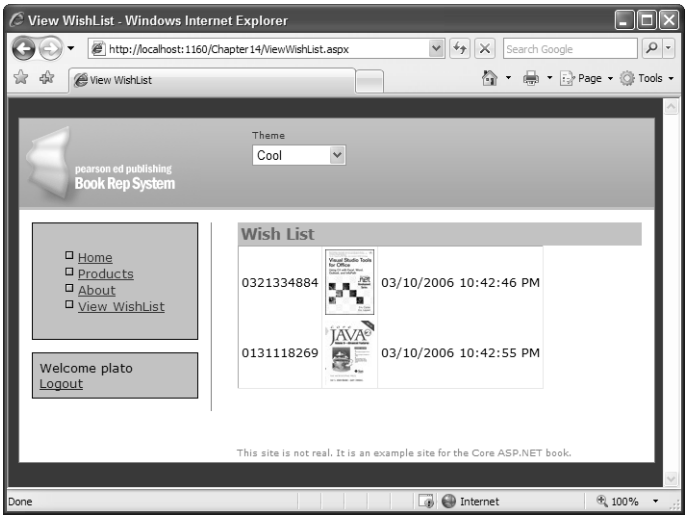


Figure 14.5 Using profile feature to maintain a user wish list

Web Parts Framework

In the last section, we examined how you can use the profile system in ASP.NET to programmatically customize Web content for specific users. In this section, we examine the Web parts framework that was introduced in ASP.NET 2.0. The ***Web parts framework*** is the name for the infrastructure and controls that allow you to create pages that provide end users with the ability to modify the content, appearance, and behavior of these pages directly within the browser. When users modify pages and controls, the settings are saved so as to retain the user's personal preferences across future browser sessions. Thus, the Web parts framework lets you let develop sites that can be customized by users at runtime without developer or administrator intervention. Figure 14.6 illustrates a sample page that is using Web parts. Each of these rectangular boxes of content can be moved into different locations on the page by the user, as well as minimized, restored, and closed.

This type of personalized Web site is sometimes called a portal. A ***portal*** is a type of Web application that displays different types of information, usually based on some type of prior user setup. A portal site may provide a unified interface to an organization's information or it may aggregate information from multiple organizations. Microsoft's My MSN and Google's Personalized Home Page are examples of very general-purpose portals.

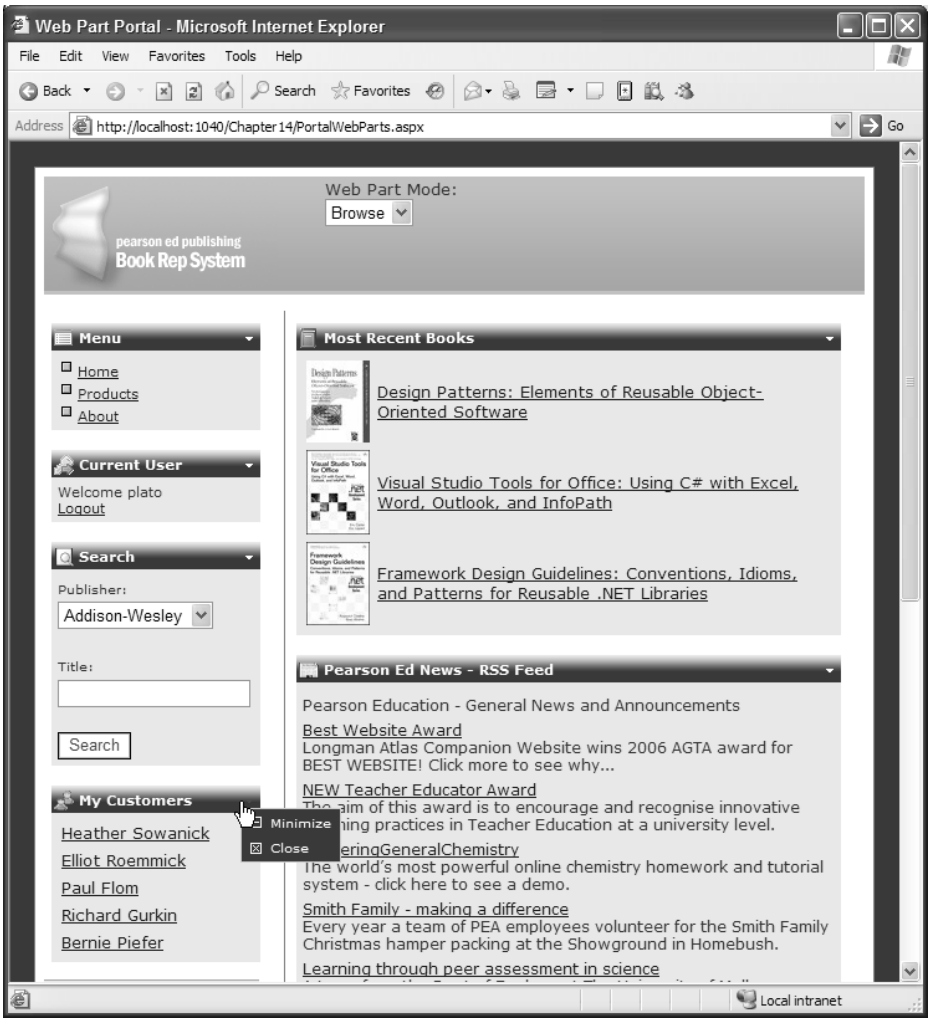


Figure 14.6 Sample page using Web parts

Microsoft has a dedicated product named SharePoint 2003 that can be used to create portal sites for managing collaboration between geographically dispersed team members. A SharePoint portal is constructed using something called SharePoint Web parts, which (in the current version) are extensions of ASP.NET 1.1 controls that can interact with the SharePoint framework. The rest of this chapter covers the ASP.NET 2.0 Web parts framework, which is different than the SharePoint Web parts framework but has a similar functionality. Finally, to muddy the waters further,

with SharePoint Service Pack 2, SharePoint Services can now use ASP.NET 2.0 Web parts; as well, future versions of SharePoint are to use ASP.NET 2.0 Web parts as their default content model.

CORE NOTE

If you require Web part-like portal functionality, you might also want to consider instead Microsoft Sharepoint or the open-source DotNetNuke.

Web Parts, Web Part Zones, and the Web Part Manager

The ASP.NET Web parts framework allows you to create user-customizable pages. These pages consist of regular ASP.NET content as well as special content containers called Web parts. A ***Web part*** is a type of Web server container control; it is a special container for both ASP.NET and regular HTML content. A Web part is thus a user-customizable "box" of displayable content that is provided with additional user interface elements such as chrome and verbs. Figure 14.7 illustrates many of the additional elements that are added to a Web part.

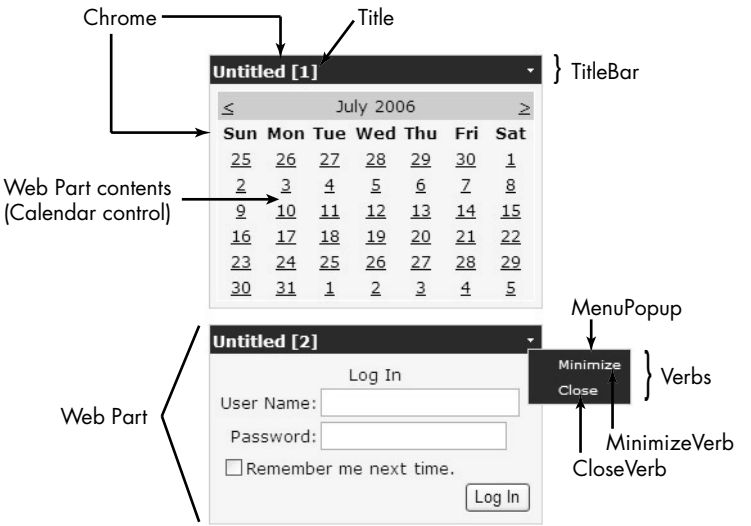


Figure 14.7 Components of a Web part

The ***verbs*** of a Web part are the actions that a user can instruct the Web part to perform. By default, the Web part framework adds the minimize/restore and close verbs to each Web part. Depending upon the display mode of the Web parts, other verbs may be available, such as edit, connect, and export. As well, it is possible to define custom verbs for a Web part.

The ***chrome*** of a Web part refers to the common user interface elements rendered around each Web part. The chrome for a Web part includes a border, a title bar, and the icons, title text, and verb menu that appear within the title bar.

Web parts are placed into one or more ***Web part zones***, which is yet another specialized container control (named `WebPartZone`). It is the Web part zone that defines the chrome and verbs for the Web parts that it contains. Figure 14.8 illustrates the relationship between parts and zones.

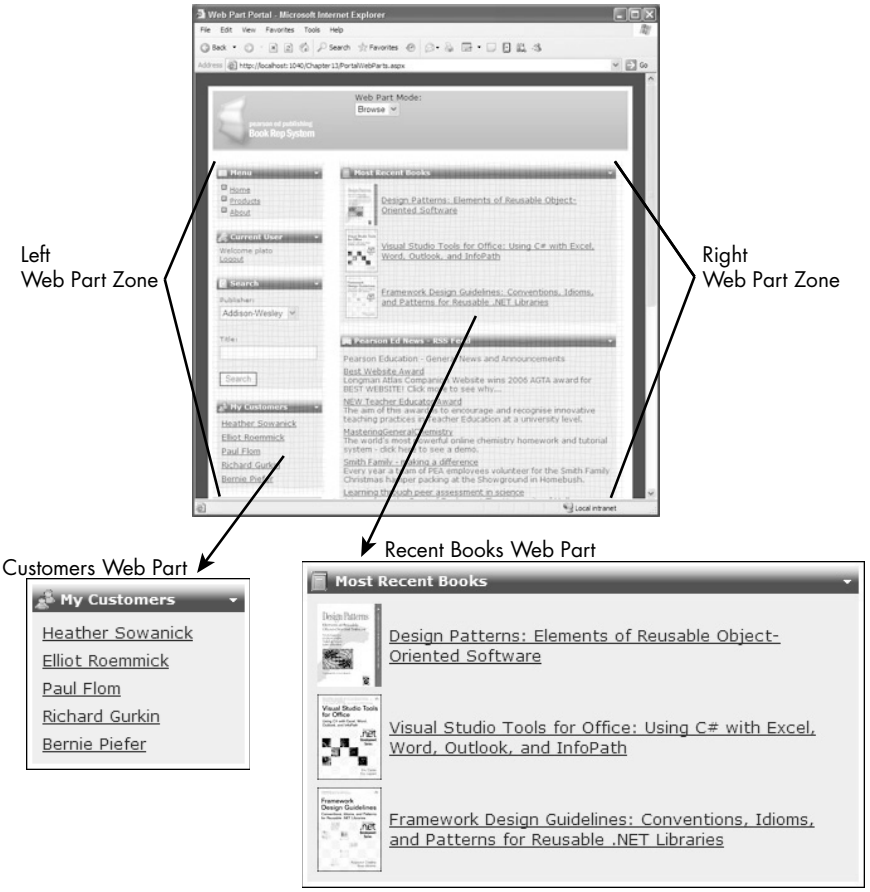


Figure 14.8 Web parts and Web part zones

Finally, there is the ***Web part manager***, which manages the entire Web part infrastructure on a page. This manager is a nonrendered control named `WebPartManager`. This control must be present on any page that uses the Web part infrastructure. The manager control can also be used to programmatically change the display mode of the Web parts or to set up communication between Web parts.

Figure 14.9 illustrates some (but not all) of the different display modes available to the Web part manager. These different display modes may also require the use of different types of zones: for instance, the `CatalogZone`, `EditorZone`, and `ConnectionsZone`.

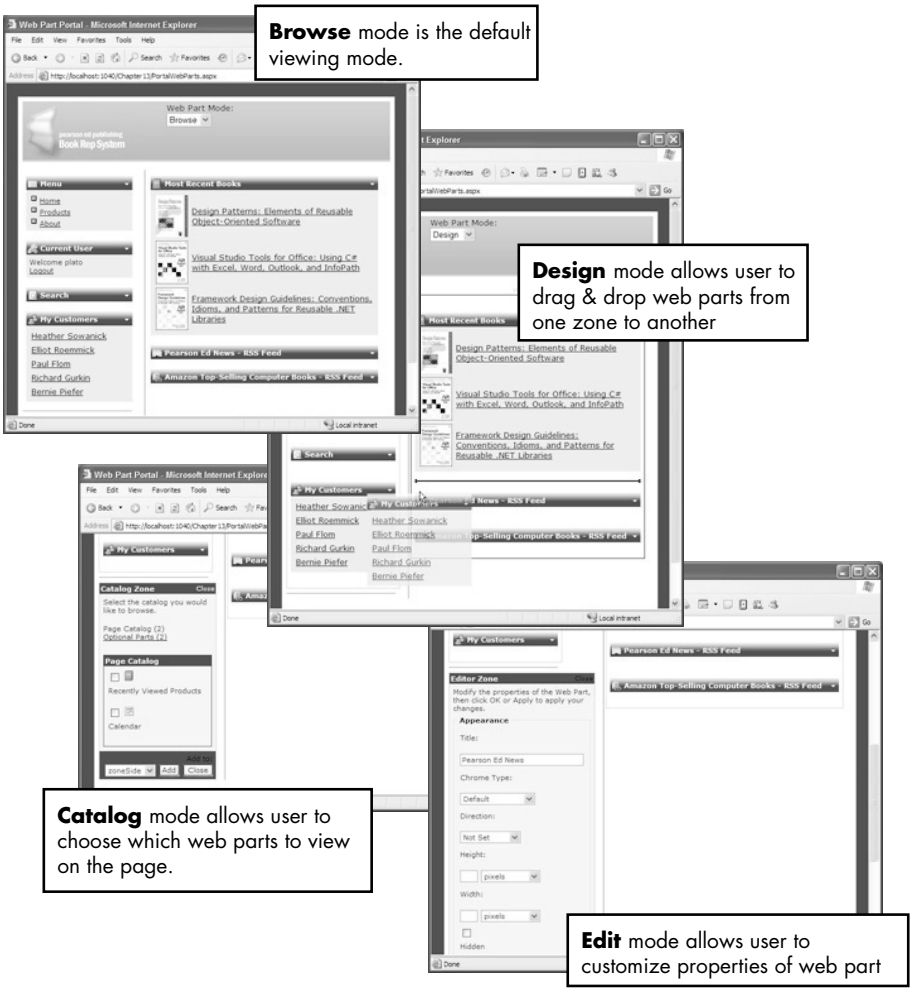


Figure 14.9 Web part manager display modes

Internally, a Web part is a server control that ultimately inherits from the `WebPart` abstract base class. As you can see from Figure 14.10, Web parts are specializations of the `Panel` server control. As can also be seen from this diagram, Web parts are one of three different part types in the Web parts framework. The editor part allows users to modify the appearance of a Web part, whereas the catalog part provides a catalog or list of Web parts that a user can add to a page.

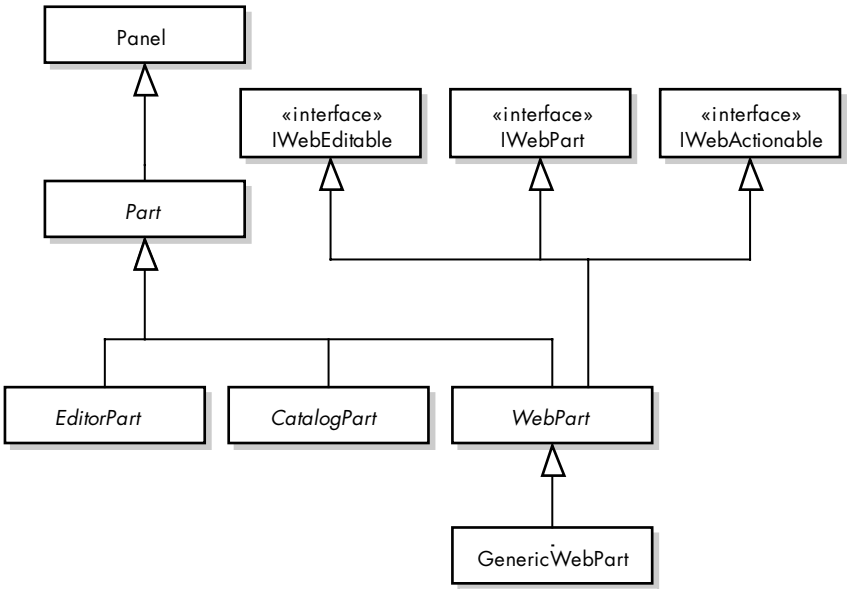


Figure 14.10 Web parts class hierarchy

Web Parts Personalization

Another feature of the Web parts infrastructure is its support for personalization. ***Web parts personalization*** refers to the capability of the current state of Web part property values to be identified with users and saved in long-term storage. It should be stressed that Web parts personalization stores user-specific state data for controls on a particular Web page only. Information that relates to the user as a person or that is intended to be used across multiple pages in a Web application should be kept in a profile.

The Web parts personalization system is based on the provider model, and by default, uses the same database as the membership and profile systems. Just as with the other provider-based subsystems in ASP.NET, it is possible to create your own customized personalization provider so as to use any data store.

Web parts personalization requires authenticated users. Although Web parts are visible for anonymous users, the personalization of Web parts requires the user to be logged in first. Normally, any personalization changes that the user makes to the Web parts in a page are saved just for that user, so that the next time the user visits the page, the previous changes are visible. In Web parts terminology, this is called ***user scope***.

The other possible scope is shared scope. With ***shared scope***, any changes that the user makes are visible to all users. By default, all users (authorized or anonymous) are denied access to shared scope. It might make sense, however, for site administrators to have shared scope so that they can make changes that are visible to all users. We illustrate how to set up shared scope later in the chapter in the section "Making the BehaviorEditorPart Appear" on page 884.

Creating and Using Web Parts

There are three basic approaches to creating a Web part:

- Use *any* Web server control or HTML server control within a `WebPartZone` control. When you add a server control to a `WebPartZone`, the runtime wraps the control as a `GenericWebPart` object, and thus is treated as a `WebPart` control and has the same consistent Web part user interface elements such as verbs, icons, a title, and a header.
- Create a user control that implements the `IWebPart` interface.
- Create a custom control that derives from the `WebPart` abstract class.

In this chapter, we begin with the first approach and then move onto the second approach, because it provides us with more control and cleaner markup. We then finish with the third approach, because it provides the highest degree of customization and control.

Because we begin by letting ASP.NET convert normal server controls into Web parts, let us examine the steps necessary for creating a page with Web parts.

1. Add a `WebPartManager` control to the page, as shown in the following.

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

2. Add one or more `WebPartZone` controls to the page. This control is the container that will hold the Web parts. Each `WebPartZone` control contains a `ZoneTemplate`, which contains the Web parts. As well, each `WebPartZone` control can contain various style elements, such as `HeaderStyle` and `PartChromeStyle`, which allow you to define and customize the appearance of the Web parts within the zone.

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

```
      …
   </ZoneTemplate>

   style elements can go here
   <PartChromeStyle … />
   …
</asp:WebPartZone>
```

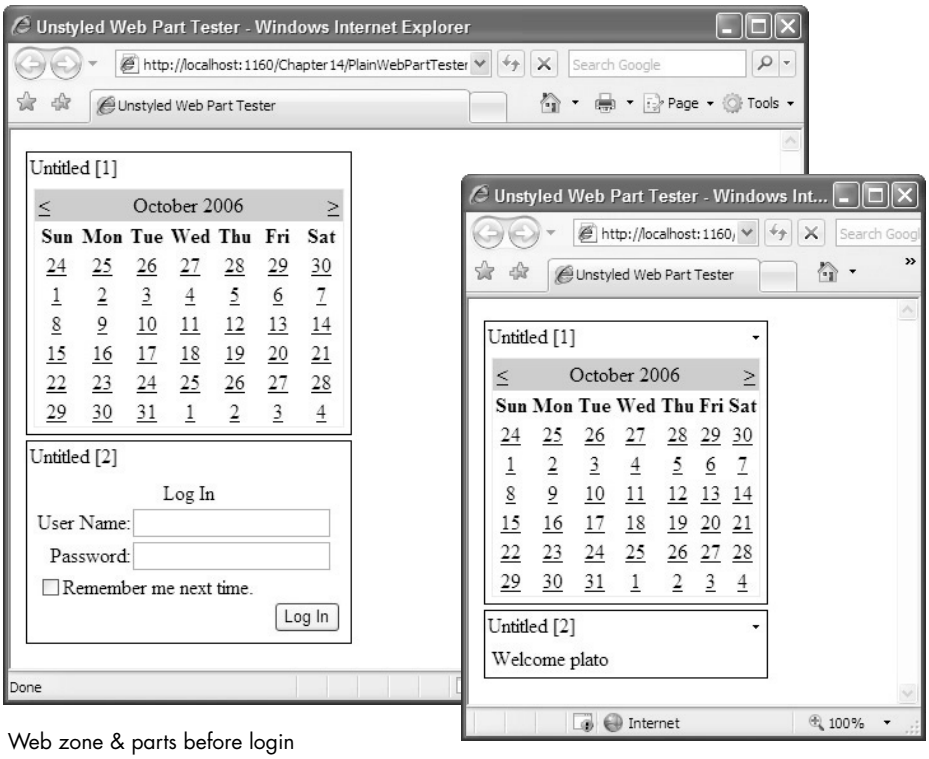
Most pages have at least two different zones, which correspond broadly to the different layout areas for the page. For instance, in Figure 14.7, the page structure contains two columns. Each of these columns contains a separate `WebPartZone` control. Some developers place different zones into different columns of an HTML `<table>` element; other developers place the different zones into different HTML `<div>` elements and then use CSS to position the zones. The latter approach is used in this chapter.

3. Add one or more Web parts to the different zones. Recall that any ASP.NET control, whether a standard control—such as a `GridView` or a `Label` control, a user control, or even a custom control—can be used as a Web part. In the following example, two Web parts (a `Calendar` control and a `LoginView` control) are added to a Web part zone.

```
<asp:WebPartZone ID="wpzOne" runat="server" >
   <ZoneTemplate>
      <asp:Calendar ID="calOne" runat="server"  />
      <asp:LoginView ID="logView" runat="server">
         <AnonymousTemplate>
            <asp:Login ID="logSignin" runat="server" />
         </AnonymousTemplate>
         <LoggedInTemplate>
            <asp:LoginName ID="logName" runat="server"
               FormatString="Welcome {0}" />
         </LoggedInTemplate>
      </asp:LoginView>
   </ZoneTemplate>
</asp:WebPartZone>
```

4. Optionally add specialized zones and parts, such as the `CatalogZone` or the `EditorZone`.

The result in the browser is shown in Figure 14.11. Notice how the chrome for the Web parts changes after the user is authenticated. Anonymous users still see the Web parts; however, they do not see the verb menu, nor are they able to view the page in anything other than the default browse display mode.



Web zone & parts before login

Web zone & parts after login

Figure 14.11 Simple Web part example

Of course, the example Web parts shown in Figure 14.11 are quite plain. You can style the Web parts by adding style elements to the `WebPartZone` (and by setting properties of the `WebPartZone` control). As well, within the Visual Studio designer, you can use the AutoFormat option to apply a prebuilt set of formatting options to a zone. Table 14.3 lists the different style elements available to the `WebPartZone`, whereas Table 14.4 lists the different properties of the `WebPartZone` control.

Table 14.3 WebPartZone Style Elements

Name	Description
`EmptyZoneTextStyle`	The style for the text that appears in an empty zone.
`ErrorStyle`	The style for any error messages that are displayed if Web part cannot be loaded or created.

Table 14.3 WebPartZone Style Elements *(continued)*

Name	Description
`FooterStyle`	The style of the text that appears in the footer of the zone.
`HeaderStyle`	The style of the text that appears in the header of the zone.
`MenuCheckImageStyle`	The style of the check mark that may appear beside a verb in the verb menu.
`MenuLabelHoverStyle`	The style of the text label for the verb menu when user positions the mouse pointer over the label.
`MenuLabelStyle`	The style of the text label for the verb menu.
`MenuPopupStyle`	The style of the pop-up verb menu for each Web part in the zone.
`MenuVerbHoverStyle`	The style of the verb menu items when user positions the mouse pointer over the text.
`MenuVerbStyle`	The style of the verb menu items.
`PartChromeStyle`	The style of the borders for each Web part in the zone.
`PartStyle`	The style of the title bar for each Web part in the zone.
`PartTitleStyle`	The style of the title in the title bar for each Web part in the zone.
`SelectedPartChromeStyle`	The style of the chrome for the selected Web part in a zone. A Web part is selected when it is in edit mode.
`TitleBarVerbStyle`	The style of the verbs in the title bar.
`VerbStyle`	The style for the verbs for each Web part in the zone.

Table 14.4 Notable WebPartZone Properties

Property	Description
`AllowLayoutChange`	Indicates whether the user is allowed to change the layout of Web parts within the zone. Default is `true`.
`BackImageUrl`	The URL of an image to display in the background of a zone.
`DragHighlightColor`	The color to be displayed around the border of a Web zone and its drop-cue when a user is dragging a Web part.

Table 14.4 Notable WebPartZone Properties *(continued)*

Property	Description
`EmptyZoneText`	The text to be displayed when a Web zone contains no Web parts.
`HeaderText`	The text to display in the header of the zone.
`LayoutOrientation`	Indicates whether the controls in a zone are to be arranged horizontally or vertically.
`MenuCheckImageUrl`	The URL of image to display when a verb is checked.
`MenuLabelText`	The text label for the pop-up verb menu in each Web part.
`MenuPopupImageUrl`	The URL for image that opens the drop-down menu in the title bar of each Web part.
`Padding`	The cell padding for the table that contains the Web parts in the zone.
`ShowTitleIcons`	Indicates whether title icons are to be displayed in the title bar of each Web part in the zone.
`TitleBarVerbButtonType`	Indicates what type of button to use (`Button`, `Image`, or `Link`) for the verbs in the title bar.
`VerbButtonType`	The type of button to use for the zone when rendered in an older browser.
`WebPartVerbRenderMode`	Specifies how to render verbs on the Web parts in the zone. Possible values are `WebPartVerbRenderMode.Menu` (verbs appear in a drop-down menu) and `WebPartVerbRenderMode.TitleBar` (verbs appear as links in title bar).

All Web parts within a zone inherit the visual appearance defined by the zone. Thus, if you move the Web part from one zone to another, its appearance may change if the styling of the zone is different.

CORE NOTE

Perhaps the easiest way to ensure that the Web parts in each zone have the same consistent appearance is to set the properties and style elements for the `WebPartZone` control in a skin file.

Verbs

Verbs are an important part of the Web part framework. They are the actions that a user can instruct a Web part to perform. When displayed with the default Browse display mode, a Web part has two verbs: Minimize/Restore and Close. When viewing a page anonymously, there is no user interface to represent these verbs. After a user is authenticated, however, the different verbs are available as either a pop-up menu or a series of text or image links in the title bar of each Web part, depending upon the value of the zone's `WebPartVerbRenderMode` property and of its verb setting elements, as shown in Figure 14.12.

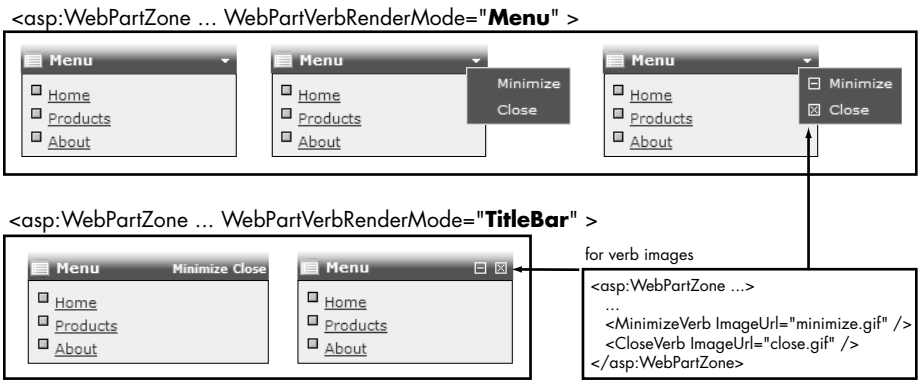


Figure 14.12 Sample verbs in a Web part

CORE NOTE

Verbs can only be displayed in the pop-up menu if the user's browser is Internet Explorer. For other browsers, the verbs are displayed in the title bar.

As shown in Figure 14.12, each zone has its own verb setting element. Table 14.5 lists the verbs that are provided by the Web part framework. You can also create your own verbs; we illustrate how to do so later when we create a custom Web part control.

Table 14.5 Web Part Verbs

Name	Description
`Close`	Closes the Web part. After a Web part is closed, the only way to bring it back is via the catalog.
`Connect`	Allows a connection to be made to another Web part.
`Delete`	Deletes a control permanently from a page. This verb is not allowed for Web parts added declaratively.
`Edit`	Allows the user to edit the Web part. When editing, the user can personalize the appearance, properties, layout, and content of the Web part.
`Export`	Allows the user to export an XML definition file for each Web part in a zone.
`Help`	Displays help for the Web part in a modal or modeless window, or by navigating to a help page.
`Minimize`	Minimizes the Web part. When minimized, only the title bar of the part is visible.
`Restore`	Restores a Web part from the minimized state.

You can customize the settings for each verb in the zone via the verb setting elements, as shown in the following.

```
<asp:WebPartZone … >
   …
   <CloseVerb Enabled="false" />
   <MinimizeVerb Text="Min" Description="Shrinks this box"/>
   <RestoreVerb Text="Res" Description="Restores this box"/>
</asp:WebPartZone>
```

Each verb setting element supports the properties listed in Table 14.6.

Table 14.6 Notable Verb Properties

Property	Description
`Checked`	Indicates whether custom state for the verb is currently active. If it is selected, a check box appears next to the verb. This property is not used with the system-provided verbs; it is only available for custom verbs.
`Description`	A short description of the verb that is displayed in the tool tip for the verb.

Table 14.6 Notable Verb Properties *(continued)*

Property	Description
`Enabled`	Indicates whether the verb is enabled.
`ImageUrl`	The URL of the image to display for the verb.
`Text`	The text label for the verb.
`Visible`	Indicates whether the verb is visible.

Making Web Parts from User Controls

You may have noticed from the example shown in Figure 14.11 that the title for each Web part begins with "Untitled." Why is this? You may remember that when you add a regular Web server control to a Web part zone, it is wrapped as a `GenericWebPart` control so that the control can exhibit the same functionality as true `WebPart` controls. As you can see from Figure 14.10, this control inherits from the `WebPart` base control, which in turn implements the `IWebPart` interface. This interface, which is described in Table 14.7, defines the minimal user interface properties needed by any Web part. If you want to change the default implementations of any of these `IWebPart` properties in your Web parts, you need to either create your own custom control for the Web part or create a user control for the Web part that implements the `IWebPart` interface.

Table 14.7 IWebPart Properties

Property	Description
`CatalogIconImageUrl`	The URL of the image used to represent the Web part in the catalog.
`Description`	The summary description of the Web part for use in tool tips and the catalog.
`SubTitle`	The string that is concatenated with the `Title` property and displayed in the title bar of the Web part.
`Title`	The title of the Web part, which is displayed in the Web part's title bar.
`TitleIconImageUrl`	The URL of the image that is displayed in the Web part's title bar.
`TitleUrl`	The URL of a page that contains more information about the Web part. When set, the title in the title bar becomes a hyperlink.

As you may recall from Chapter 6, a user control encapsulates the appearance and behavior of a part of a page's user interface. User controls have an extension of `.ascx` and their code-behind classes inherit from the `UserControl` rather than the `Page` class.

If you want to be able to display a different title as well as a title bar image—for example, the `LoginView` Web part shown earlier—you could define the following user control.

```
<%@ Control Language="C#" AutoEventWireup="true"
   CodeFile="LoginControl.ascx.cs" Inherits="LoginControl" %>

<asp:LoginView ID="logView" runat="server">
   <AnonymousTemplate>
      <asp:Login ID="logSignin" runat="server"
         DisplayRememberMe="True"
         TitleText="" TextLayout="TextOnTop"
         RememberMeText="Remember me">
      </asp:Login>
   </AnonymousTemplate>
   <LoggedInTemplate>
      <asp:LoginName ID="logName" runat="server"
         FormatString="Welcome {0}" /><br/>
      <asp:LoginStatus ID="logStatus" runat="server" />
   </LoggedInTemplate>
</asp:LoginView>
```

The code-behind for this user control might look like the following. Notice how the user control class definition must also implement the `IWebPart` interface. The implementation of this interface simply consists of properties as well as the data members to contain the state for these properties.

```
public partial class LoginControl : UserControl, IWebPart
{
   // Data members necessary for IWebPart properties
   private string _description = "";
   private string _title = "";
   private string _subtitle = "";
   private string _titleIconImageUrl = "";
   private string _catalogIconImageUrl = "";
   private string _titleUrl = "";

   // Implement IWebPart

   public string CatalogIconImageUrl
   {
      get { return _catalogIconImageUrl; }
      set { _catalogIconImageUrl = value; }
   }
```

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

    public string Subtitle
    {
        get { return _subtitle; }
    }

    public string Title
    {
        get { return _title; }
        set { _title = value; }
    }

    public string TitleIconImageUrl
    {
        get { return _titleIconImageUrl; }
        set { _titleIconImageUrl = value; }
    }

    public string TitleUrl
    {
        get { return _titleUrl; }
        set { _titleUrl = value; }
    }

    // Any additional properties or behaviors go here
}
```

After this user control is defined, it can be used within a Web part zone. Remember that to use a user control in a Web page, you must first register the tag name and prefix via the `Register` page directive. You can then declaratively add the control via this tag name and prefix as well as set any of the Web part properties defined by the control.

```
<%@ Register Src="controls/LoginControl.ascx"
    TagName="LoginControl" TagPrefix="uc" %>
…
<asp:WebPartZone ID="zoneSample" runat="server"  >
    <ZoneTemplate>
        …
        <uc:LoginControl ID="myLogin" runat="server"
            Title="Current User"
            Description=
              "Necessary to login in order to modify web parts"
            TitleIconImageUrl="images/webpartLogin.gif"
```

```
            CatalogIconImageUrl="images/webpartLogin.gif"  />
    </ZoneTemplate>
</asp:WebPartZone>
```

If your Web site is to have more than two or three Web parts created from user controls, you may find that you are uncomfortable with the fact that the code-behind classes for each of these user controls looks almost identical. That is, because a Web part user control must implement `IWebPart`, you will find that each of these user controls have the identical data members and property definitions necessary to implement this interface. A better approach is to define a custom base class that implements the `IWebPart` interface, and then use it as the base class for your Web part user controls. Listing 14.5 contains a sample implementation of this custom base class.

Listing 14.5 CustomWebPartBase.cs

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

/// <summary>
/// Base class for my user control-based Web parts
/// </summary>
public abstract class CustomWebPartBase : UserControl, IWebPart
{
    private string _description = "";
    private string _title = "";
    private string _titleIconImageUrl = "";
    private string _catalogIconImageUrl = "";
    private string _titleUrl = "";

    // IWebPart interface requires Subtitle to be read-only; thus
    // to allow subclasses to initialize it, you must make it a
    // protected data member
    protected string _subtitle = "";

    public string CatalogIconImageUrl
    {
        get { return _catalogIconImageUrl; }
        set { _catalogIconImageUrl = value; }
    }

    public string Description
    {
        get { return _description; }
```

```
      set { _description = value; }
   }

   public string Subtitle
   {
      get { return _subtitle; }
   }

   public string Title
   {
      get { return _title; }
      set { _title = value; }
   }

   public string TitleIconImageUrl
   {
      get { return _titleIconImageUrl; }
      set { _titleIconImageUrl = value; }
   }

   public string TitleUrl
   {
      get { return _titleUrl; }
      set { _titleUrl = value; }
   }
}
```

Your user control code-behind classes simply inherit from this base class and set up the default values for the properties, as shown in Listing 14.6.

Listing 14.6 LoginControl.ascx.cs

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

public partial class LoginControl : CustomWebPartBase
{
   public LoginControl()
   {
      // Set default values for properties
      Title = "Current User";
      Description =
        "Necessary to login in order to modify web parts";
```

```
        TitleIconImageUrl = "images/webpartLogin.gif";
        CatalogIconImageUrl = "images/webpartLogin.gif";

        // IWebPart interface requires Subtitle to be read-only;
        // thus, must initialize it via protected data member
        // of CustomWebPartBase
        _subTitle = "Login";
    }

    // any additional properties or behaviors go here
}
```

Making Web Parts from Custom Controls

Although creating Web parts from user controls is quite straightforward, there may be times when you want to instead create a Web part as a custom control. The advantage of using custom controls for your Web parts is that you have complete control over the rendering and behavior of the control. As well, you can add custom verbs and make it easier to create communication links between Web parts when you create Web parts from custom controls. As well, if you want to add personalization attributes to properties within the Web part (we do this later when we work with the `PropertyGridEditorPart` control), you can avoid a potential deserialization error if your property uses enumerated types by using a custom Web part control.

Creating a custom Web part control is by no means difficult. You must create a new class that inherits from the `WebPart` class. You can then add any custom properties that your Web part requires. Finally, you must provide the rendering for the control, either by overriding the `RenderContents` or the `CreateChildControls` method. You should override `RenderContents` when you want to exactly control the rendered output of the Web part; when you want to use the functionality and rendering of other controls, you should override `CreateChildControls`. In this section's example, you use the `RenderContents` approach. For an example of the `CreateChildControls` approach, you can examine Listing 14.8 later in the chapter.

The following listing illustrates the code for a custom Web part that displays the first 10 books in the books database. Eventually, you add custom verbs to the Web part so that the user can filter the list by a publisher. The code for a basic custom control is shown in Listing 14.7.

Listing 14.7 BookListViewer.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
```

```
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

using FourLayer.ApplicationLogic;
using FourLayer.BusinessEntity;

namespace Chapter14
{
   public class BookListViewer : WebPart
   {
      private Publisher _publisher = null;

      public BookListViewer()
      {
         Title = "View Books";
         Description = "Views all books";
         TitleIconImageUrl = "images/webpartBooks.gif";
         CatalogIconImageUrl = "images/webpartBooks.gif";
      }

      /// <summary>
      /// Outputs the HTML content for the control
      /// </summary>
      protected override void RenderContents(
         HtmlTextWriter writer)
      {
         EntityCollection<Book> books;
         if (PublisherToUse == null)
            books = BookCatalogLogic.GetAllBooks();
         else
            books =
              BookCatalogLogic.GetBooksByPublisher(Publisher.Id);

         int max = 10;
         if (max > books.Count) max = books.Count;

         string output = "<div id='listContainer'>";
         for (int i = 0; i < max; i++)
         {
            Book book = books[i];
            output += "<p><a href=BookDetails.aspx?id=";
            output += book.Isbn + ">";
            output += book.Title;
            output += "></p>";
         }
         output += "</div>";
```

```
            writer.Write(output);
        }

        [Personalizable, WebBrowsable]
        public Publisher PublisherToUse
        {
            get { return _publisher; }
            set {
                _publisher = value;
                Title = "View Books for " + _publisher.Name;
            }
        }
    }
}
```

Notice that this custom control inherits from the `WebPart` base class. This base class has the Web part functionality already defined. The custom control also initializes the Web part properties within its constructor. The `RenderContents` method outputs the actual HTML content that appears within the Web part. This method uses some classes from Chapter 11 to retrieve the data, loop through it, and output the appropriate HTML using the passed in `HtmlTextWriter` object. The control also defines a `Publisher` property, which we use later to allow the user to filter the books by a publisher. This property is marked with two attributes. The `Personalizable` attribute ensures that the value of this property is saved to the data store by the personalization system, whereas the `WebBrowsable` attribute makes this property visible to the `PropertyGridEditorPart` editor part (covered later in the chapter).

After this class is defined, you can then use it in a Web part page. Just as with user controls, you have to register the class in your markup through the `Register` directive.

```
<%@ Register Namespace="Chapter14" TagPrefix="ch14" %>
```

This `Register` directive is a bit different than with user controls, in that you do not specify the filename but the namespace and/or assembly name. Using the control is now simply a matter of using this tag prefix along with the class name.

```
<ch14:BookListViewer id="myViewer" runat="server" />
```

Adding a Custom Verb

To make your custom Web part control more useful, you can change it so that the user can filter the displayed books by a publisher selected by the user in the verb menu. To create a custom verb, your Web part control must override the `Verbs` property (which is defined by the `IWebActionable` interface). In your version of

`Verbs`, you create and return a collection of `WebPartVerb` objects. Each of these `WebPartVerb` objects needs to reference the handler method that runs when the user selects the verb (from the verb menu or the verb list in the title bar).

In our example, we create a verb for each publisher in the database (obviously, this might cause some visual problems if there are many publishers and our web part zone uses `WebPartVerbRenderMode="TitleBar"`). The code for creating these custom verbs is as follows.

```
public override WebPartVerbCollection Verbs
{
   get
   {
      // Create new collection of verbs
      ArrayList verbs = new ArrayList();
      // Get all the publishers
      EntityCollection<Publisher> pubs =
         BookCatalogLogic.GetAllPublishers();

      // Make each publisher a verb
      foreach (Publisher p in pubs)
      {
         WebPartVerb verb = new WebPartVerb(p.Id.ToString(),
            new WebPartEventHandler(ChangePublisher));
         verb.Text = p.Name;
         verb.Description = "See books for " + p.Name;

         // If this is the current publisher, check it
         if (PublisherToUse != null &&
            p.Name == PublisherToUse.Name)
         {
            verb.Checked = true;
         }
         // Add new verb to collection
         verbs.Add(verb);
      }
      // Create new verb collection with the new verbs added
      // to any existing verbs
      return new WebPartVerbCollection(base.Verbs, verbs);
   }
}
```

Notice that the constructor for `WebPartVerb` requires a reference to the handler method that executes when the verb is selected. All your example needs to do is set the `Publisher` property to the name of the publisher in the verb, as shown in the following.

```
public void ChangePublisher(object o, WebPartEventArgs e)
{
   WebPartVerb verb = (WebPartVerb)o;
   PublisherToUse =
      BookCatalogLogic.GetPublisherByName(verb.Text);
}
```

The result in the browser is shown in Figure 14.13.

Figure 14.13 Custom control Web part with custom verbs

Changing the Different Display Modes

If you have been following the examples so far in this chapter, you may have wondered what exactly is the benefit of web parts. If all you want is to display content within boxes, the Web parts framework might seem to be more bother than it is worth. However, the real benefit of Web parts can be realized after you allow the user to modify your Web part pages. To do this requires that the user be able to change the ***Web part display mode*** of the page via the `WebPartManager.DisplayMode` property. Table 14.8 lists the available display modes for the Web parts framework.

Table 14.8 Display Modes

Name	Description
`Browse`	Displays Web parts in the normal mode. This is the only display mode available for anonymous users.
`Catalog`	Displays the catalog user interface that allows end users to add and remove page controls. Allows dragging of controls, just as with design mode.
`Connect`	Displays the connections user interface that lets users connect Web parts.

Table 14.8 Display Modes *(continued)*

Name	Description
`Design`	Makes each Web zone visible and enables users to drag Web parts to change the layout of a page.
`Edit`	Enables the Edit verb in Web parts. Choosing the Edit verb displays editing elements and lets end users edit the Web parts on the page. Also allows the dragging of Web parts, just as with design mode.

You must provide the user interface yourself for changing the `DisplayMode` property. In the example that follows, we use a `DropDownList` (you can see the result in Figure 14.14), but you could use buttons or some other control type. You could also encapsulate the markup and event handling for this list within a user or custom control. Your definition for this list is as follows.

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

Web Part Mode:
<asp:DropDownList ID="drpWebPartMode" runat="server"
   AutoPostBack="true"
   OnSelectedIndexChanged="drpWebPartMode_SelectedIndexChanged" />
```

Notice that we have not statically populated the list with the available modes. You should not statically fill the list because the items in the list vary depending upon the user's permissions (e.g., an anonymous user should only see the Browse mode, an authenticated user might see Design and Catalog, whereas a user with the administrator role might see the Edit mode). Instead, we programmatically fill the list when the page loads.

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      // Loop through all the supported display modes
      foreach (WebPartDisplayMode mode in
         partManager.SupportedDisplayModes)
      {
         // Only display the mode if it is enabled for
         // the current user
         if (mode.IsEnabled(partManager))
         {
            // Create list item and add to list
            ListItem item = new ListItem(mode.Name);
```

```
            // If this item matches the current display mode,
            // make it the selected item in the list
            if (mode == partManager.DisplayMode)
               item.Selected = true;

            drpWebPartMode.Items.Add(item);
         }
      }
   }
}
```

The event handler for the list simply needs to change the display mode of the `WebPartManager` control on the page to that chosen by the user.

```
protected void drpWebPartMode_SelectedIndexChanged(object sender,
   EventArgs e)
{
   // Retrieve the user's selection
   string mode = drpWebPartMode.SelectedValue;

   // Retrieve the WebPartDisplayMode object for this mode
   WebPartDisplayMode display =
      partManager.SupportedDisplayModes[mode];

   // Set the display mode if this mode is enabled
   if (display != null && display.IsEnabled(partManager))
      partManager.DisplayMode = display;
}
```

Design Mode

When the Web parts page is in Design mode, each Web part zone is visible with its name and a border around it. The user can then drag and drop Web parts from one zone to another, as shown in Figure 14.14.

CORE NOTE

This drag-and-drop behavior is currently *only* available to users using Internet Explorer as their browser. For non-IE browsers, Web parts can only be moved using the Catalog Zone (covered shortly). You can also add this drag-and-drop behavior for other uplevel browsers if you use ASP.NET AJAX, which is covered in the Appendix.

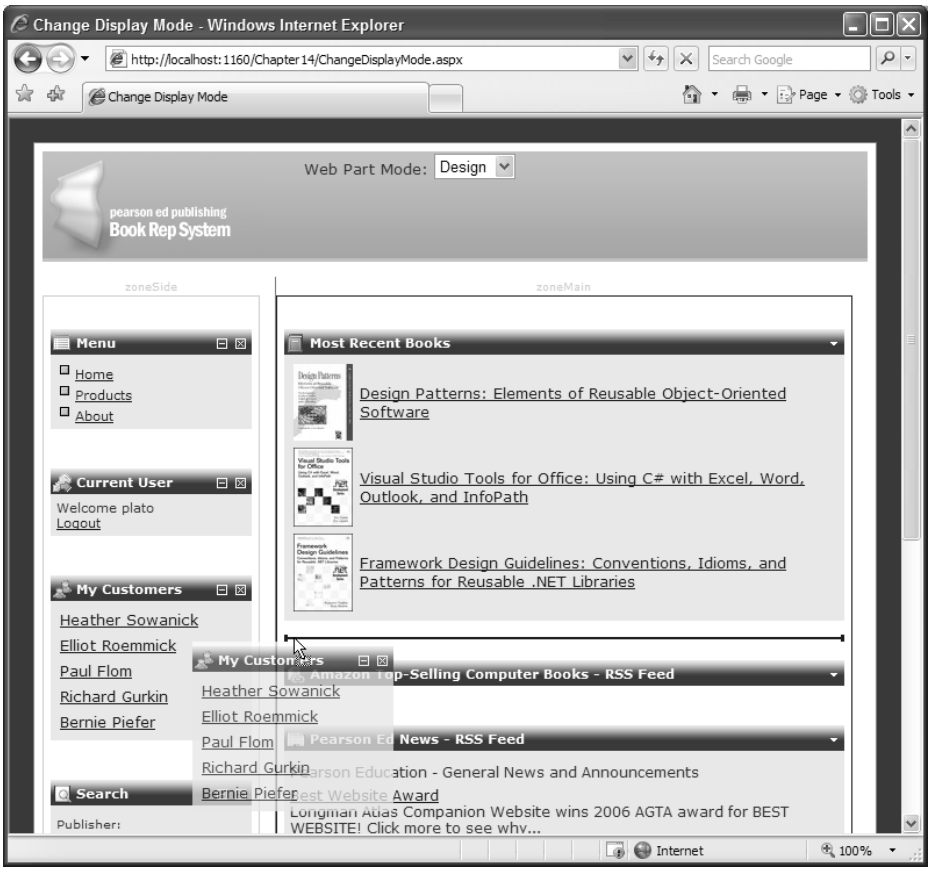


Figure 14.14 Moving Web parts while in Design mode

To provide this design functionality to your Web part page, you need a page that contains more than one `WebPartZone`. The following example illustrates the markup for the page shown in Figure 14.14.

```
<div id="sideArea">
   <asp:WebPartZone ID="zoneSide" runat="server"
         Padding="6" WebPartVerbRenderMode="TitleBar"
         PartChromeType="TitleOnly" Width="14em">
      <ZoneTemplate>
         <uc:MainMenuControl ID="myMenu" runat="server"
            Title="Menu"/>
         <uc:LoginControl ID="myLogin" runat="server"
            Title="Current User" />
         <uc:SearchControl ID="mySearch" runat="server"
```

```
            Title="Search"/>
         <uc:CustomersControl ID="myCustomers" runat="server"
            Title="My Customers" />
      </ZoneTemplate>
   </asp:WebPartZone>
</div>

<div id="mainArea">
   <asp:WebPartZone ID="zoneMain" runat="server"
         WebPartVerbRenderMode="TitleBar"
         Padding="6" PartChromeType="TitleOnly">
      <ZoneTemplate>
         <uc:RecentBooksControl ID="myRecentBooks" runat="server"
            Title="Most Recent Books" />
         <uc:RSSControl ID="myAmazon" runat="server"
            Title="Amazon Top-Selling Computer Books"
          Description="Amazon Top Computer Books"

            XmlUrl="~/amazonRSS.xml"
            XsltUrl="~/RssTransformAmazon.xsl" />
         <uc:RSSControl ID="myPearson" runat="server"
            Title="Pearson Ed News"
            Description="Pearson Ed News"
            CatalogIconImageUrl="~/images/webpartNews.gif"
            TitleIconImageUrl="~/images/webpartNews.gif"
            XmlUrl="~/pearsonRSS.xml"
          XsltUrl="~/RssTransformPearson.xsl" />
      </ZoneTemplate>
   </asp:WebPartZone>
</div>
```

Notice that the zones in this example contain no formatting markup; instead, all of the styling elements and properties are contained in a separate skin file for the theme used in the page, while the actual positioning of the zones is contained within a CSS file for the theme. Notice as well that the different web parts are all encapsulated as user controls.

Catalog Mode

Catalog mode allows the user to add Web parts to the page at runtime. It also allows the user to move parts from zone to zone in the same manner as Design mode. A ***catalog*** is simply a list of Web parts that is visible when a page is in Catalog mode.

When the page is switched to Catalog mode, the catalog user interface appears. The user interface for this catalog is provided by a `CatalogZone` control, which can be declaratively added to the page. The `CatalogZone` can only contain `CatalogPart` controls. Table 14.9 lists the available `CatalogPart` controls.

Table 14.9 CatalogPart Controls

Name	Description
`DeclarativeCatalogPart`	Provides a way for developers to define a set of possible (but not initially visible) Web parts declaratively to a catalog. A user can select controls from the list and add them to the page, which effectively gives users the ability to change the set of controls and the functionality on a page.
`ImportCatalogPart`	Provides the user interface for a user to upload a definition file (an XML file with a `.WebPart` extension that imports settings for a control) to a catalog, so that the control can be added to the page. The control must be present on the server before the definition file can be imported. The description file is not the same as the control itself. It is an XML file that contains name/value pairs that describe the state of the control.
`PageCatalogPart`	Maintains references to Web parts that have been previously added to the page but have been subsequently closed by the user. This part lets the user reopen (add back to the page) closed Web parts.

After the `CatalogZone` control is added declaratively to the page, `CatalogPart` controls can be added to the zone. For instance, in the following example, we add a catalog zone to the side area of our page that contains both a `PageCatalogPart` and a `DeclarativeCatalogPart`.

```
<div id="sideArea">
   <asp:CatalogZone ID="catZone" runat="server" Padding="6" >
      <ZoneTemplate>
         <asp:PageCatalogPart ID="pageCatalog"
            runat="server" />
         <asp:DeclarativeCatalogPart ID="decCatalog"
               runat="server" Title="Optional Parts">
            <WebPartsTemplate>
               <uc:CalendarControl ID="myCalendar" runat="server"
                  Title="Calendar"/>
               <uc:RecentViewsPartControl ID="myRecentViews"
                  runat="server"
                  Title="Recently Viewed Products"/>
            </WebPartsTemplate>
         </asp:DeclarativeCatalogPart>
      </ZoneTemplate>
   </asp:CatalogZone>
```

```
    <asp:WebPartZone ID="zoneSide" runat="server" >
        …
    </asp:WebPartZone>
</div>
```

Notice that the `DeclarativeCatalogPart` control contains two user controls. These define two optional Web parts that are not part of the initial page, but can be added later by the user via the catalog. The result in the browser is shown in Figure 14.15.

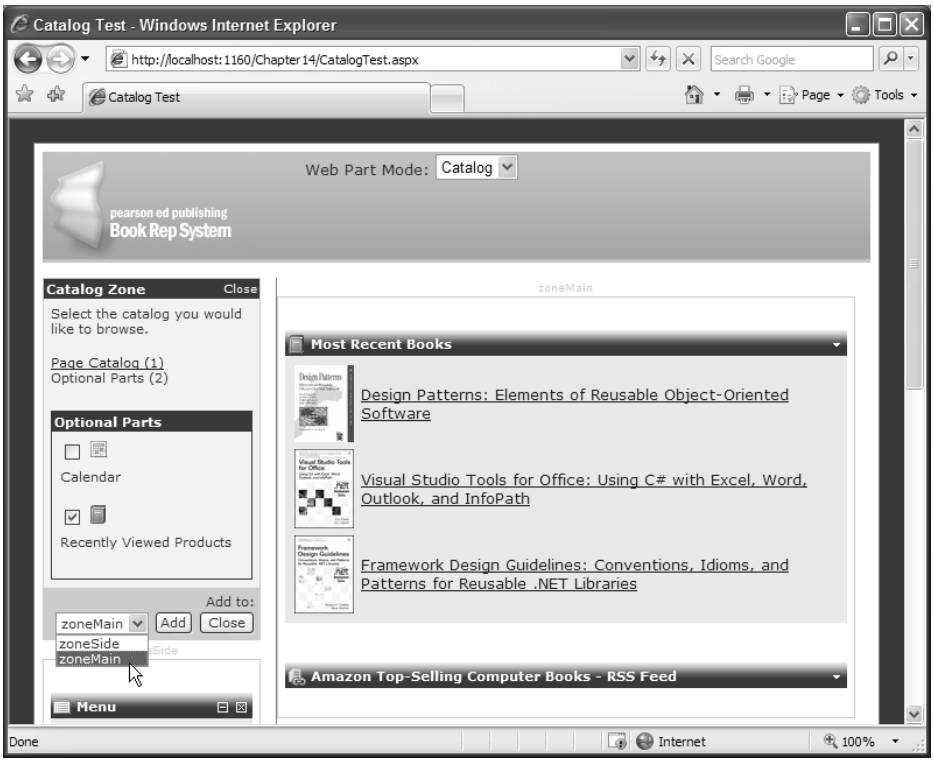


Figure 14.15 Adding parts via the catalog

The `PageCatalogPart` and a `DeclarativeCatalogPart` appear within the browser as choices within the catalog zone. For both of these parts, a list of possible Web parts appears as check boxes and labels. The user can then select a part, choose the zone, and then click the Add button. After the controls have been added to the page via the catalog, they appear in normal browse mode.

Edit Mode

The Edit design mode lets users edit or modify the properties of a Web part control. After the user switches the page to Edit mode, all Web parts in which editing is enabled display the edit verb. After the user selects the edit verb for a Web part, that Web part is then in Edit mode and the editing user interface appears for editing the selected Web part.

Similar to the approach used by Catalog mode, the user interface for editing is provided by an `EditorZone` control, which is declaratively added to the page. The `EditorZone` control can contain only `EditorPart` controls. Table 14.10 lists the available controls that are subclasses of the `EditorPart` class.

Table 14.10 EditorPart Controls

Name	Description
`AppearanceEditorPart`	Provides a user interface for editing appearance properties of a Web part.
`BehaviorEditorPart`	Provides a user interface for editing behavioral properties of a Web part.
`LayoutEditorPart`	Provides a user interface for editing the chrome state and the zone of a Web part. This part allows non-Internet Explorer users to change the layout of Web parts.
`PropertyGridEditorPart`	Provides a user interface for editing properties of a Web part that are marked in the source code with the `WebBrowsable` attribute.

Figure 14.16 illustrates how these controls might appear when rendered in the browser.

After the `EditorZone` control is added declaratively to the page, `EditorPart` controls can be added to the zone. For instance, in the following example, we have added an editor zone to the main area of our page that contains an `AppearanceEditorPart`, `BehaviorEditorPart`, and a `LayoutEditorPart`.

```
<asp:EditorZone ID="editZone" runat="server" >
   <ZoneTemplate>
      <asp:AppearanceEditorPart ID="partAppear" runat="server" />
      <asp:BehaviorEditorPart ID="partBehave" runat="server" />
      <asp:LayoutEditorPart ID="partLayout" runat="server" />
   </ZoneTemplate>
</asp:EditorZone>
```

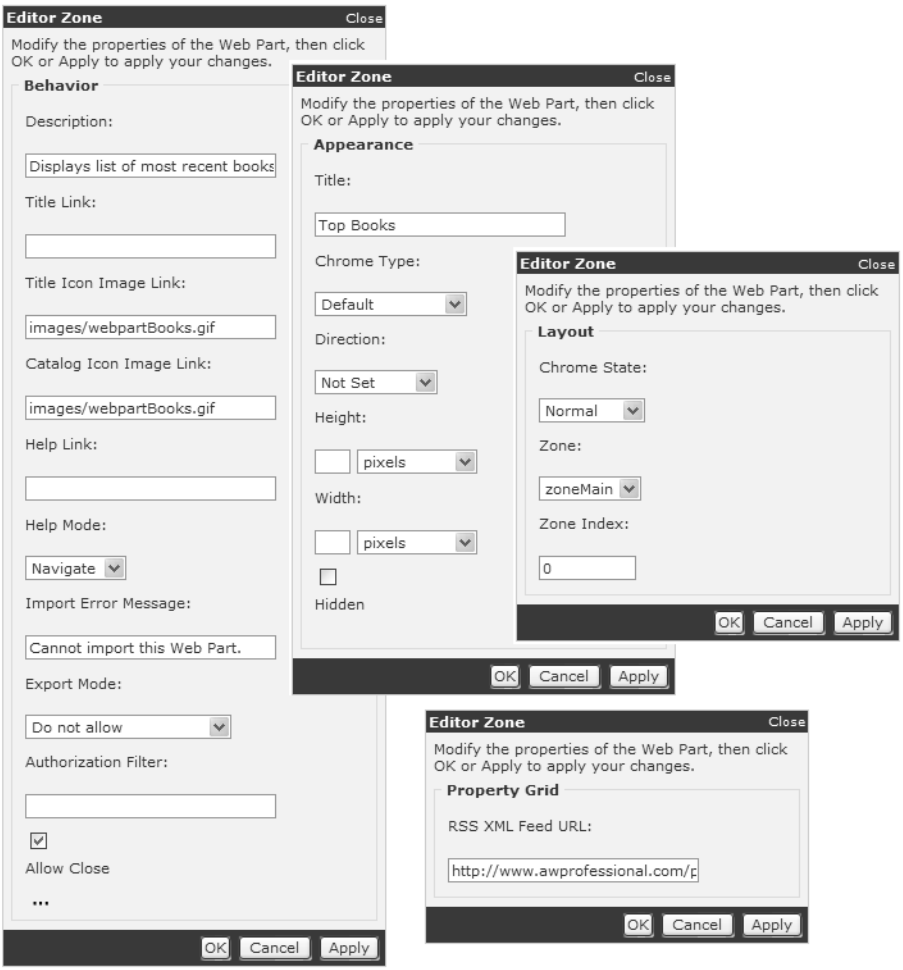


Figure 14.16 Different editor parts in the browser

Making the BehaviorEditorPart Appear

However, when you view this in the browser, only the `AppearanceEditorPart` and the `LayoutEditorPart` are actually visible. The reason for this is that changes made to the properties of a Web part via the `BehaviorEditorPart` affect not only the current user but *all* users. As such, the user must have shared scope before the `BehaviorEditorPart` control is visible.

You may recall from the earlier discussion on Web part personalization that, with shared scope, any changes that the user makes to the state of a Web part are visible to

all users. By default, all users are denied access to shared scope. It might make sense, however, for site administrators to have shared scope so that they can make universal changes to a Web part with the `BehaviorEditorPart`.

There are two steps you must follow to have the `BehaviorEditorPart` control visible in your pages. First, you must allow users in the administrator role to have access to shared scope. You do this by adding the following section to your `Web.config` file.

```
<system.web>
   …
   <webParts>
      <personalization>
         <authorization>
            <allow roles="Administrator"
               verbs="enterSharedScope"/>
         </authorization>
      </personalization>
   </webParts>
</system.web>
```

Rather than allowing shared scope on a role basis, you could instead do so on a user-by-user basis.

```
<allow users="Hamlet,Plato" verbs="enterSharedScope"/>
```

The next step is to change the personalization scope of the page from user to shared scope. Perhaps the easiest way to do this is to add the following to the `Page_Load` method of the page.

```
protected void Page_Load(object sender, EventArgs e)
{
   …

   // For users who are authorized to enter shared
   // personalization scope (that is, users who are allowed to
   // make changes that affect all users), change the page scope
   // from user scope to shared scope

   if (partManager.Personalization.Scope ==
       PersonalizationScope.User &&
       partManager.Personalization.CanEnterSharedScope)
   {
      partManager.Personalization.ToggleScope();
   }

}
```

Using the PropertyGridEditorPart

The `PropertyGridEditorPart` is a bit different from the other editor parts in that it does not present the user with a predefined set of properties. Instead, the `PropertyGridEditorPart` allows the developer to display a list of custom properties to the user. This control displays all the properties in the Web part that have been marked with the `WebBrowsable` attribute (thus, the Web part has to be a user control or a custom control). When a property is marked with this attribute, a `PropertyGridEditorPart` creates the editing user interface based on the type of the property. For instance, strings, dates, and numbers are displayed in a `TextBox`, Booleans are displayed with a `CheckBox`, whereas enumerated types are displayed in a `DropDownList`.

You can add some other attributes to the properties of your Web part class to further customize how the control displays the editing interface. These include

- `WebDisplayName`—Allows you to specify the text for the label that appears with each control in the editing interface.
- `WebDescription`—Allows you to specify a string that appears as a `ToolTip` for each control.
- `Personalizable`—Allows you to specify the personalization scope for the property. If set to `PersonalizationScope.Shared`, the property is only displayed for users in shared scope (e.g., administrators).

Listing 14.8 contains a sample Web part custom control that illustrates the use of some of these attributes. This Web part displays an RSS feed. Unlike the custom control created in Listing 14.7, this one does not override the `RenderContents` method, but instead overrides the `CreateChildControls` method. The `CreateChildControls` method allows you to programmatically create and set up existing server controls and then add them to the child control collection of the Web part; by doing so, you can let the child controls handle their own rendering. In the example shown in Listing 14.8, the `CreateChildControls` method uses an `Xml` server control along with an XSLT file to read and display several sample external RSS feeds. This control also presents two `WebBrowsable` properties: one for specifying which publisher's RSS feed to use, the other the filename of the XSLT file to use. The former is modifiable by all users, whereas the later is only available to administrators (i.e., users with shared scope).

Listing 14.8 PressReleaseRSS.cs

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
```

```
using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

namespace Chapter14
{
   /// <summary>
   /// Specifies the possible press release RSS feeds
   /// </summary>
   public enum PressReleaseRssFeeds
   {
      AddisonWesley, PeachPit, PrenticeHall, QuePublishing,
        SamsPublishing
   }

   public class PressReleaseRSS: WebPart
   {
      private string _xsltUrl =
         "~/RssTransformPressRelease.xslt";
      private PressReleaseRssFeeds _whichFeed =
          PressReleaseRssFeeds.AddisonWesley;

      // For simplicity's sake, we will hardcode the URLs for the
      // RSS feed, but it is better and more realistic
      // to extract from database or config file
      private string[] _rssUrls = {
         "http://www.awprofessional.com/press/press_rss.asp",
         "http://www.peachpit.com/press/press_rss.asp",
         "http://www.phptr.com/press/press_rss.asp",
         "http://www.quepublishing.com/press/press_rss.asp",
         "http://www.samspublishing.com/press/press_rss.asp"
      };

      public PressReleaseRSS()
      {
         Title = "Press Releases";
         Description = "Press releases from Pearson Ed";
         TitleIconImageUrl = "images/webpartNews.gif";
         CatalogIconImageUrl = "images/webpartNews.gif";
      }

      [Personalizable(PersonalizationScope.Shared),
        WebBrowsable, WebDisplayName("XSLT filename")]
      public string XsltUrl
      {
         get { return _xsltUrl; }
         set { _xsltUrl = value; }
      }
```

```
        [Personalizable(PersonalizationScope.User),
          WebBrowsable, WebDisplayName("Press Release RSS Feed")]
        public PressReleaseRssFeeds WhichFeed
        {
           get { return _whichFeed; }
           set { _whichFeed = value; }
        }

        /// <summary>
        /// Responsible for creating the content of the control
        /// </summary>
        protected override void CreateChildControls()
        {
           // Create an XML Web server control
           Xml myXml = new Xml();

           // Get URL for selected feed
           string xmlurl = _rssUrls[(int)WhichFeed];

           // Create the XPathNavigator object
           XPathDocument xpdoc = new XPathDocument(xmlurl);
           XPathNavigator xnav = xpdoc.CreateNavigator();

           // Set up the XML control and apply XSLT transformation
           myXml.XPathNavigator = xnav;
           myXml.TransformSource =
             HttpContext.Current.Server.MapPath(XsltUrl);

           // Add XML control to the Web part's collection
           // of child controls
           Controls.Add(myXml);
        }
    }
}
```

With this control defined, you can now add it to a Web part page.

```
<%@ Register Namespace="Chapter14" TagPrefix="ch14" %>

…

<ch14:PressReleaseRSS ID="myPressRelease" runat="server"
   Title="Press releases"
   Description="Press releases from Pearson Ed"
   CatalogIconImageUrl="~/images/webpartNews.gif"
   TitleIconImageUrl="~/images/webpartNews.gif"
   WhichFeed="AddisonWesley"
   XsltUrl="~/RssTransformPressRelease.xsl" />
```

Figure 14.17 illustrates how the `PropertyGridEditorPart` editor part appears for the `PublisherRSSControl` Web part depending upon the scope of the user.

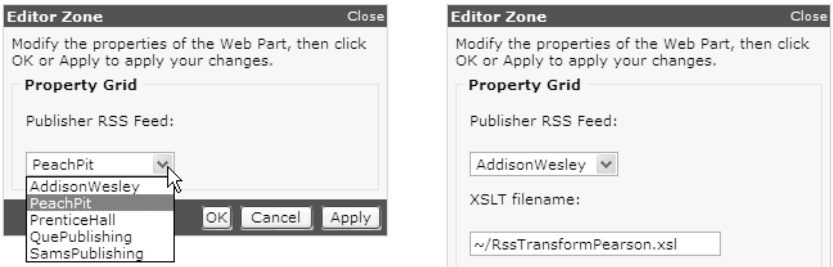


Figure 14.17 PropertyGridEditorPart in the browser

Web Part Connections

The Web parts framework allows connections to be made between Web parts. When a part is enabled for connections, a user can create dynamic communication connections between the parts at runtime, as well as static, predefined connections declared in the markup of the page. You can also declare a user interface that enables users to manage connections and to connect or disconnect Web parts at runtime.

A ***Web parts connection*** is a link between two Web parts that enables them to share data. A connection always involves exactly two parts: one is the provider of data, and the other is the consumer of the data from the provider. A ***provider Web part*** can establish connections with multiple consumers at the same time. A ***consumer Web part*** connects to only one provider at a time. Any changes made to a provider are immediately consumed by any Web part consumers. Figure 14.18 illustrates a sample usage of Web parts connections. The provider in this example is the Select Publisher Web part; whenever the user selects a different publisher, the two consumer Web parts (View Books and News) are updated and display new data based on the user's selection.

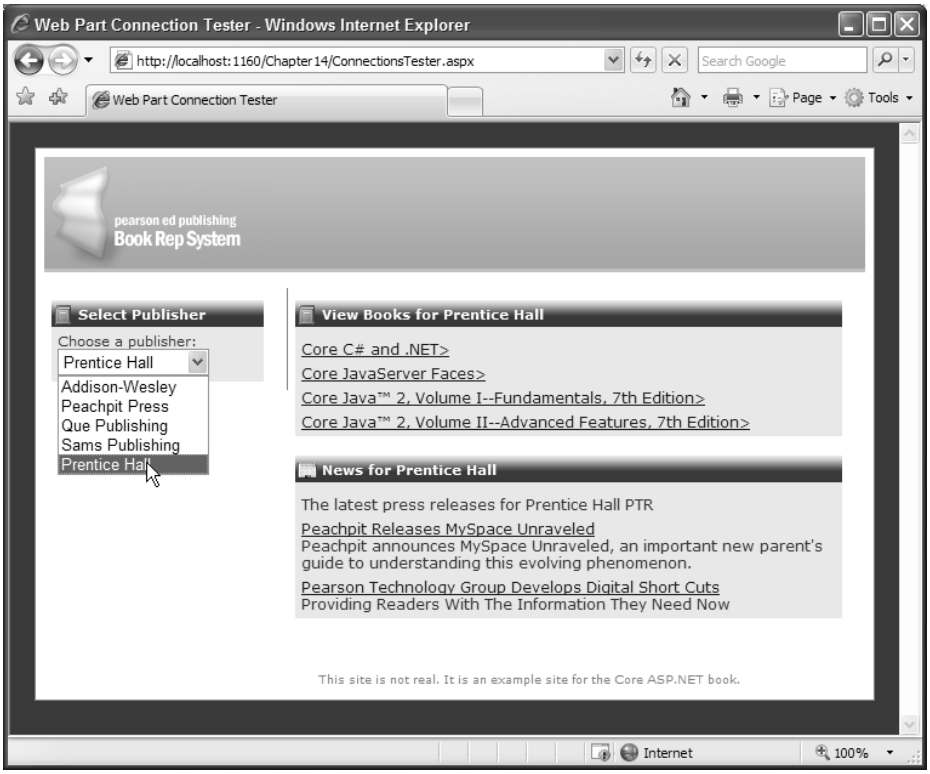


Figure 14.18 Web part connections

Creating a Static Connection

A Web part connection is either static or dynamic. ***Static connections*** are declared in the markup of the page by adding a `StaticConnection` element to the `WebPartManager`. Creating a static connection requires that you complete these four steps.

1. Define a C# `interface` that is the shared connection contract between the two Web parts. This interface defines the data that is passed between the two parts.
2. The provider Web part must implement this connection interface. As well, the provider must define a ***connection point***. A connection point is a method that returns an instance of the class that implements the connection interface; in other words, it must return the provider object itself. This method must be marked with the `ConnectionProvider` attribute.

3. The consumer Web part must also define a connection point. In this case, it is a `void` method that accepts an object of the type defined by the `interface` from step 1. This object can then be used by the consumer to retrieve data from the provider Web part. This method must be marked with the `ConnectionConsumer` attribute.
4. You must declaratively define the static connections in the markup of the Web parts page by adding a `StaticConnection` element to the `WebPartManager`.

Figure 14.19 provides a visualization of how data flows via connection points between provider and consumer Web parts.

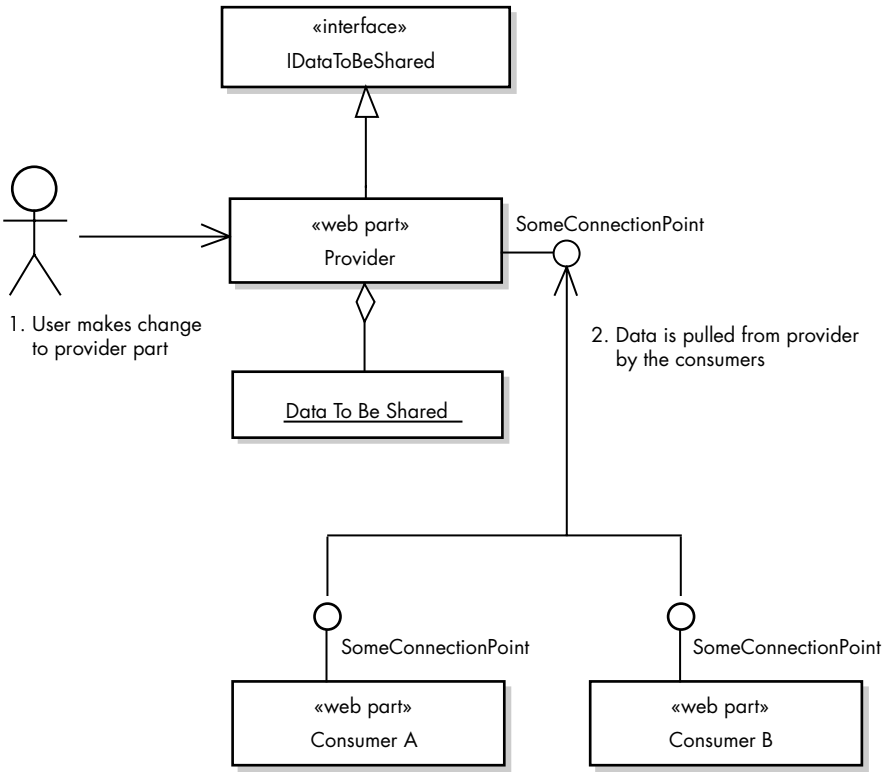


Figure 14.19 Data flow with Web part connections

Walkthroughs 14.1 through 14.3 demonstrate how to set up two Web part connections. These create the controls and the connections necessary to implement the example shown in Figure 14.18.

Walkthrough 14.1 *Defining the Interface and the Provider Web Part*

In this example, we pass a `Publisher` object that represents the selected publisher from part to part.

1. Create a new class in the `App_Code` folder named `ISelectPublisher.cs`.
2. Change the code to that shown in the following. As you can see, the `interface` defines a single property named `ChosenPublisher` that returns a `Publisher` object (defined in the `FourLayer.BusinessEntity` namespace).

```
using FourLayer.BusinessEntity;

namespace Chapter14
{
   /// <summary>
   /// Defines the shared connection contract between the
   /// SelectPublisherProvider and its consumers
   /// </summary>
   public interface ISelectPublisher
   {
      Publisher ChosenPublisher { get; }
   }
}
```

3. Create a new class in the `App_Code` folder named `SelectPublisherProvider.cs`. This custom control is the provider in your example.
4. Change the code to the following. It consists of a drop-down list of publishers in your books database. Notice that this class inherits not only from the `WebPart` class but also from the `ISelectPublisher` interface that you just defined.

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

using FourLayer.ApplicationLogic;
using FourLayer.BusinessEntity;

namespace Chapter14
{
   public class SelectPublisherProvider : WebPart,
      ISelectPublisher
   {
```

```
        DropDownList _drpPublishers;

        public SelectPublisherProvider()
        {
           Title = "Select Publisher";
           Description =
              "Select publisher for connections testing";
           TitleIconImageUrl = "images/webpartBooks.gif";
           CatalogIconImageUrl = "images/webpartBooks.gif";
        }

        protected override void CreateChildControls()
        {
           _drpPublishers = new DropDownList();
           _drpPublishers.AutoPostBack = true;

           // Get all the publishers
           EntityCollection<Publisher> pubs =
              BookCatalogLogic.GetAllPublishers();

           // Set up the drop-down list. Notice that its value
           // field is the integer publisher ID, whereas the
           // text field is the publisher name
           _drpPublishers.DataTextField = "Name";
           _drpPublishers.DataValueField = "Id";
           _drpPublishers.DataSource = pubs;
           _drpPublishers.DataBind();

           Literal lit = new Literal();
           lit.Text = "Choose a publisher: ";

           Controls.Add(lit);
           Controls.Add(_drpPublishers);
        }
     }
}
```

5. You must now add the property defined in the `ISelectPublisher` interface. This property returns a populated `Publisher` object based on the current value of the publisher drop-down list.

```
/// <summary>
/// Implementation of method defined in ISelectPublisher
/// </summary>
public Publisher ChosenPublisher
{
   get {
```

```
        // Return a publisher object for the selected ID
        int pubId = Convert.ToInt32(
           _drpPublishers.SelectedValue);
        return BookCatalogLogic.GetPublisherById(pubId);
     }
  }
```

6. Finally, you must define a connection point for this provider class. This method simply returns this instance of the provider as defined by your connection interface. It must also be marked with the `ConnectionProvider` attribute.

```
/// <summary>
/// Defines the connection point
/// </summary>
[ConnectionProvider("SelectedPublisherConnectionPoint")]
public ISelectPublisher SupplyProvider()
{
   return this;
}
```

CORE NOTE

It does not matter what name you give the connection point method in either the provider or the consumer class. It only matters that they are marked with the `ConnectionProvider` and `ConnectionConsumer` attributes, respectively.

Walkthrough 14.2 *Defining a Consumer Web Part*

This consumer Web part displays a list of books for the publisher selected by the user in the `SelectPublisher` Web part.

1. Create a new class in the `App_Code` folder named `BookListConsumer.cs`.
2. Change the code to the following. This Web part handles its own rendering. It renders a list of book title links that match the selected publisher. Notice that the selected publisher is contained in a private data member available via the `PublisherToUse` property.

```
using System;
using System.Collections;
using System.Web;
using System.Web.UI;
```

```
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

using FourLayer.ApplicationLogic;
using FourLayer.BusinessEntity;

namespace Chapter14
{
   public class BookListConsumer : WebPart
   {
      private Publisher _publisher = null;

      public BookListConsumer()
      {
         Title = "View Books";
         Description = "Views all books";
         TitleIconImageUrl = "images/webpartBooks.gif";
         CatalogIconImageUrl = "images/webpartBooks.gif";
      }

      public Publisher PublisherToUse
      {
         get { return _publisher; }
         set {
            _publisher = value;
            Title = "View Books for " + _publisher.Name;
         }
      }

      protected override void RenderContents(
         HtmlTextWriter writer)
      {
         // Retrieve books for the selected publisher
         EntityCollection<Book> books =
            BookCatalogLogic.GetBooksByPublisher(
               PublisherToUse.Id);

         // Display no more than the first 10 matches
         int max = 10;
         if (max > books.Count) max = books.Count;
         string output = "<div id='listContainer'>";
         for (int i = 0; i < max; i++)
         {
            Book book = books[i];
            output += "<p><a href=BookDetails.aspx?id=";
            output += book.Isbn + ">";
            output += book.Title;
```

```
                output += "></p>";
            }
            output += "</div>";
            writer.Write(output);
        }

    }
}
```

Notice that in the `RenderContents` method defined in the preceding, it retrieves a collection of books that match the `Id` of the `_publisher` data member (via the `PublisherToUse` property). What populates this data member? You must define a connection point that populates this member.

3. You must now define the connection point shown here for this consumer. It simply saves in a data member the reference (i.e., the interface) to your provider.

```
private ISelectPublisher _pubInterface;

[ConnectionConsumer("SelectedPublisherConnectionPoint")]
public void GetProvider(ISelectPublisher pubInterface)
{
    _pubInterface = pubInterface;
}
```

4. You now need to retrieve the `Publisher` object from the provider using this `_pubInterface` data member. When should you do this? You need to retrieve the `Publisher` object before the `RenderContents` method is invoked (because it uses this object). Perhaps the best choice is to perform this retrieval in the `OnPreRender` event of the control, which is invoked before `RenderContents`. Add this method, shown in the following, to your class.

```
/// <summary>
/// Retrieves the Publisher from the provider
/// </summary>
protected override void OnPreRender(EventArgs e)
{
    if (_pubInterface != null)
    {
        PublisherToUse = _pubInterface.ChosenPublisher;
    }
}
```

Walkthrough 14.3 *Defining Another Consumer Web Part*

This consumer Web part displays the RSS feed defined for the publisher selected by the user in the `SelectPublisher` Web part.

1. Create a new class in the `App_Code` folder named `PublisherRSS-Consumer.cs`.
2. Change the code to the following. Notice that this control is using `CreateChildControls` rather than rendering the contents itself. Instead of using an `OnPreRender` handler to retrieve the `Publisher` object from the provider, it does so directly within the `CreateChild-Controls` method.

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

using FourLayer.ApplicationLogic;
using FourLayer.BusinessEntity;

namespace Chapter14
{
   public class PublisherRSSConsumer : WebPart
   {
      private string _xsltUrl =
         "~/RssTransformPressRelease.xsl";
      private ISelectPublisher _pubInterface;

      public PublisherRSSConsumer()
      {
         Title = "News";
         Description = "News for Publisher";
         TitleIconImageUrl = "images/webpartNews.gif";
         CatalogIconImageUrl = "images/webpartNews.gif";
      }

      [Personalizable(PersonalizationScope.Shared),
        WebBrowsable, WebDisplayName("XSLT filename")]
      public string XsltUrl
      {
```

```
            get { return _xsltUrl; }
            set { _xsltUrl = value; }
        }

        protected override void CreateChildControls()
        {
            // Create an XML Web server control
            Xml myXml = new Xml();

            // Set up the XML control using the select
            // publisher provider
            if (_pubInterface != null)
            {
                // Get the publisher from the provider
                Publisher p = _pubInterface.ChosenPublisher;
                if (p != null)
                {
                    Title = "News for " + p.Name;

                    // Use the RSS URL defined for the publisher
                    XPathDocument xpdoc =
                        new XPathDocument(p.RssUrl);

                    // Set up the path navigator for this XML
                    // document
                    XPathNavigator xnav = xpdoc.CreateNavigator();
                    myXml.XPathNavigator = xnav;

                    // Specify XSLT file
                    myXml.TransformSource =
                        HttpContext.Current.Server.MapPath(XsltUrl);
                }
            }
            // Add XML control to the Web part's collection
            // of controls
            Controls.Add(myXml);
        }

        [ConnectionConsumer("SelectedPublisherConnectionPoint")]
        public void GetProvider(ISelectPublisher pubInterface)
        {
            _pubInterface = pubInterface;
        }
    }
}
```

Walkthrough 14.4 *Creating the Web Part Page with Static Connections*

Finally, you are ready to create the Web part page that uses these Web parts and which demonstrates how static connections are declared.

1. Create a new page named `ConnectionsTester.aspx`.
2. Change the markup so that it is similar to that shown here. Although you can create any layout you want, the bolded markup in the code that follows is necessary.

```
<%@ Page Language="C#" AutoEventWireup="true"
   Theme="WebPartDemo"
   CodeFile="ConnectionsTester.aspx.cs"
   Inherits="ConnectionsTester" %>

<%@ Register Namespace="Chapter14" TagPrefix="ch14" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title>Web Part Connection Tester</title>
   <link href="chapterStyles.css" type="text/css"
rel="stylesheet"  />
</head>
<body id="body1" runat="server" >
   <form id="form1" runat="server">
   <asp:WebPartManager ID="partManager" runat="server" >
   </asp:WebPartManager>
   <div id="container">
      <div id="header">
         <div id="theme"></div>
         <div id="logo">
            <asp:Image runat="server" ID="imgLogo"
               SkinID="logo" />
         </div>
      </div>
      <div id="sideArea">
         <asp:WebPartZone ID="zoneSide" runat="server">
            <ZoneTemplate>
               <ch14:SelectPublisherProvider
                  ID="mySelectPublisher"
                  runat="server" />
            </ZoneTemplate>
         </asp:WebPartZone>
      </div>
```

```
        <div id="mainArea">
           <asp:WebPartZone ID="zoneMain" runat="server">
              <ZoneTemplate>
                 <ch14:BookListConsumer id="myBookViewer"
                    runat="server"/>
                 <ch14:PublisherRSSConsmer ID="myPublisherRSS"
                    runat="server" />
              </ZoneTemplate>
           </asp:WebPartZone>
        </div>
        <div id="footer">
           <p>…</p>
        </div>
     </div>
     </form>
  </body>
  </html>
```

3. Modify the `WebPartManager` by adding the following static connection information.

```
<asp:WebPartManager ID="partManager" runat="server" >
   <StaticConnections>
      <asp:WebPartConnection ID="conn1"
         ProviderID="mySelectPublisher"
         ConsumerID="myBookViewer" />
      <asp:WebPartConnection ID="conn2"
         ProviderID="mySelectPublisher"
         ConsumerID="myPublisherRSS" />
   </StaticConnections>
</asp:WebPartManager>
```

4. Test in browser. The result should look similar to that shown in Figure 14.18 on page 890.

After examining the code for these sample provider and consumers Web parts, you may wonder when the connection between the provider and consumer is made during the page's lifecycle. It is made during the page's `LoadComplete` event. This event fires after all postback data and view-state data is loaded into the page and its controls. This means that your page cannot rely on the connection within the page's `PreInit`, `Init`, `PreLoad`, or `Load` event handlers.

Making Static Connections with Master Pages

When I introduced the Web parts framework, I mentioned that each page must have one and only one `WebPartManager`. This can be a problem for some designs, particularly those using master pages. When designing a site with master pages, it is usual

to put elements that are common to all pages within a master page. In such a situation, it might make sense to place the `WebPartManager` within the master page. How then can you define Web part connections in your content pages if the `WebPartManager` exists in a separate file?

The `ProxyWebPartManager` control exists for this particular situation in which you need to declare static connections in content pages when a `WebPartManager` control has already been declared in a master page. The `ProxyWebPartManager` control takes the place of the `WebPartManager` control in this scenario. By declaring a `ProxyWebPartManager` element instead of a `WebPartManager` element within content pages, you can still define static connections, as shown in the following example.

```
<asp:ProxyWebPartManager ID="myProxy" runat="server">
   <StaticConnections>
      <asp:WebPartConnection ID="conn3"
         ProviderID="someProvider"
         ConsumerID="someConsumer" />
   </StaticConnections>
</asp:ProxyWebPartManager>
```

At runtime, the connections in the `ProxyWebPartManager` control are simply added to the `StaticConnections` collection of the `WebPartManager` control and treated like any other connection.

Creating a Dynamic Connection

Although static connections make sense for most situations, there are times when a page is unable to create static connections. For instance, recall that it is possible for the user to add a Web part that requires a connection via the Web part catalog. In such a case, you are unable to statically predefine the connection because static connections can only be made between Web parts that are declared within `WebPartZone` elements. Another situation in which you might need to use ***dynamic connections*** is a page whose Web parts have been programmatically added.

To allow the user to make dynamic connections, you must still follow the first three steps indicated in the section on setting up static connections. Instead of adding the declarative `StaticConnections` element, however, all you must do here as a developer is add a `ConnectionsZone` control to the page. This control can be styled like the other Web parts controls. After this control is added to the page, the user must follow these steps for each connection.

1. Change the page to Connect mode (of course, this presumes that the page contains some mechanism for doing so).
2. From the consumer Web part, choose the Connect verb. This displays the Connection Zone part.

3. In the Connection Zone part, the user must choose the provider connection from the list of available provider connections for the page and click the Connect button. This connects just the two parts.
4. The user can optionally disconnect the connection or close the Connection Zone.

Figure 14.20 illustrates how this process appears in the browser.

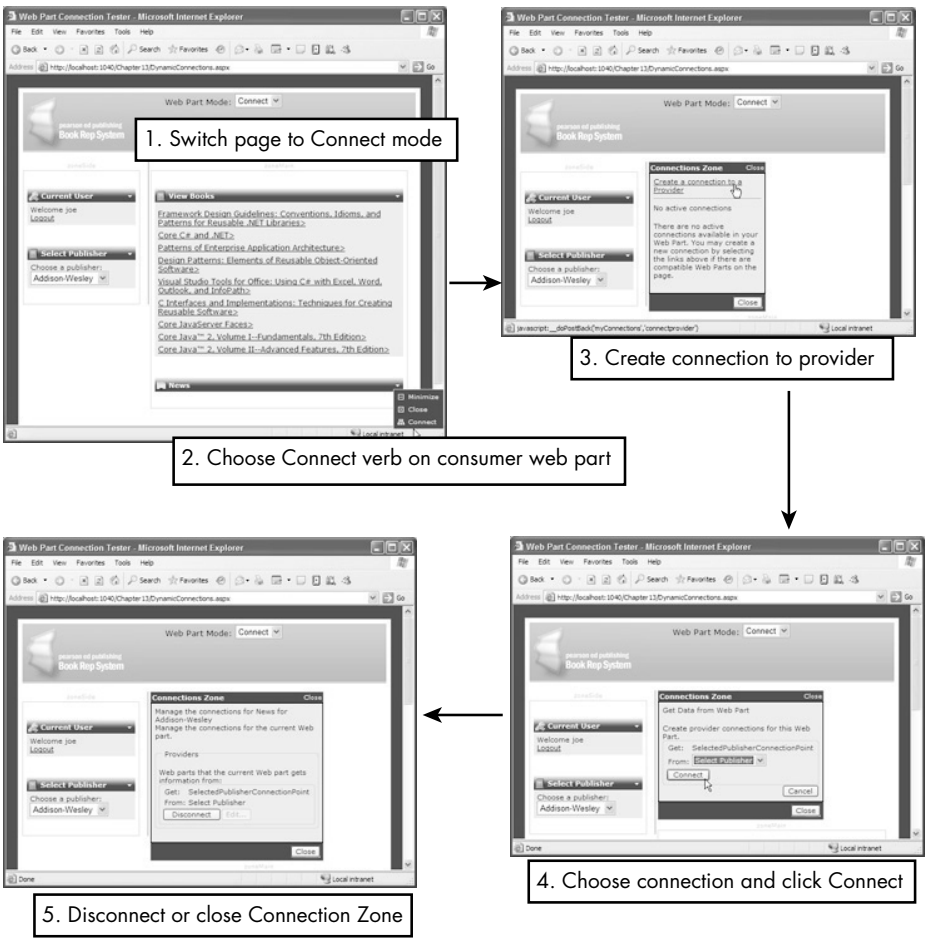


Figure 14.20 Making a dynamic connection

Summary

This chapter examined two mechanisms in ASP.NET 2.0 for integrating user personalization into a Web application. The profile system allows an application to persist user information across time and across multiple pages. The Web part framework provides the developer with a mechanism for creating pages in which the user can customize the placement, appearance, and possibly the behavior of the page's content.

The next chapter examines Web services, a powerful feature that has been an integral part of ASP.NET since its introduction. Web services are often used in conjunction with Web parts.

Exercises

The solutions to the following exercises can be found at my Web site, `http://www.randyconnolly.com/core`. Additional exercises only available for teachers and instructors are also available from this site.

1. Create a profile-based shopping cart. This requires a products listing page, a single product page in which the user can add to the cart, and a view cart page in which the user can view and remove items from the cart. You can use the book catalog database as the data source.
2. Create a page that allows administrators to view, edit, or delete anonymous and authenticated profiles. This page should allow the user to filter the profiles by some type of user-entered date range.
3. Make a new RSS user control that displays one of the Amazon RSS feeds (see `www.amazon.com/exec/obidos/subst/xs/syndicate.html`). This control should have a property named `RssFeed`, which contains the RSS URL and can be changed by the user via a `PropertyGridEditorPart`. Integrate this control with the Web part page created in Walkthrough 14.4.

The next exercises use the `ModernEyeCatalog` database.

4. Create a Web part page that displays information from the `ModernEyeCatalog` database. It must consist of the following Web parts that you must create (using either user or custom controls): a Menu part, a Login part, an Artist List part, a List of the Top 10 Keywords part, and a List of the Top 10 Works of Art part. Finally, this page should contain a catalog.
5. Create a Select Keyword part and a Art Words By Keyword part. Implement a static connection between the two parts, with the Select Keyword part the connection provider.

Key Concepts

- Anonymous access
- Anonymous identification
- Catalog
- Chrome
- Connection point
- Consumer Web part
- Dynamic connections
- Globally unique identifier (GUID)
- Personalization information
- Portal
- Provider Web part
- Shared scope
- Static connections
- User scope
- Verbs
- Web part display mode
- Web part manager
- Web parts
- Web parts connection
- Web parts framework
- Web parts personalization
- Web part zone

References

Allen, K. Scott. "Profiles in ASP.NET 2.0." `http://www.odetocode.com`.

Esposito, Dino. "Personalization and User Profiles in ASP.NET 2.0." *MSDN Magazine* (October 2005).

Onion, Fritz. "Asynchronous Web Parts." *MSDN Magazine* (July 2006).

Pattison, Ted. "Introducing ASP.NET Web Part Connections." *MSDN Magazine* (February 2006).

Pattison, Ted and Onion, Fritz. "Personalize Your Portal with User Controls and Custom Web Parts." *MSDN Magazine* (September 2005).

Walther, Stephen. "Introducing the ASP.NET 2.0 Web Parts Framework." `http://msdn.microsoft.com`.