http://www.devx.com/dotnet/Article/33889

Design Patterns for ASP.NET Developers, Part 2: Custom Controller Patterns

By Alex Homer
Mar 2, 2007

As you saw in the first article in this series, ASP.NET automatically implements the Model-View-Presenter (MVP) pattern when you use the built-in "code-behind" approach. However other patterns, usually referred to as "controller patterns," are also suitable for use in ASP.NET. These patterns extend the capability for displaying a single view by allowing applications to choose which view to display at runtime depending (usually) on user interaction.

The Use Case Controller Pattern
The Use Case Controller pattern coordinates and sequences interaction between the system and its users to carry out a specific process. The "Wizard" interface style is an example; users step through a sequence of screens in a defined order. They may be able to go backwards as well as forwards, or jump to a specific step, but the overall approach suggests a defined "forward movement" through the process (see Figure 1).

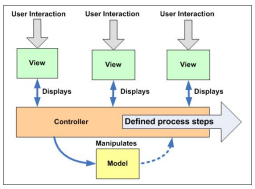


Figure 1. The Use Case Controller Pattern: The diagram shows flow control in a typical Wizard-style interaction.

In this pattern, a single Controller interacts with the Model (the data) and displays the appropriate view, depending on the user's interaction or the application's requirements. It is possible to incorporate multiple Controllers in a hierarchy for complex processes, and use configuration files or other persisted data to define the steps, which lets you alter control flow without recompiling and redeploying the application for easy maintenance and extension or alteration of the overall process. However, the underlying principles are as shown in Figure 1.

The Page Controller and Front Controller Patterns
Two other design patterns related to Use Case Controller are the Page Controller and Front Controller patterns. These implement and extend the principles of the Use Case Controller pattern in a way that specifically suits ASP.NET. These patterns allow an application to either select the content to display in a page using different partial Views, such as varying page content composition with common headers and footers, or select which View (page) to display, such as pages that present different content depending on the browser or user credentials (see Figure 2).

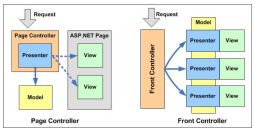


Figure 2. The Page Controller and Front Controller Patterns: These patterns are well-suited to ASP.NET, because they support partial or alternate views.

The Page Controller pattern uses a single Presenter (part of the MVP pattern implemented by the ASP.NET code-behind technology), which interacts with the Model (the data for the page). When it receives a request, the Page Controller can determine which partial View to display within the page, and then interact with that View following the MVP pattern.

In the Front Controller pattern, a separate controller examines each request and determines which page to display. Each page is a complete MVP implementation with its own View, and each Presenter interacts with the View and the Model (the data).

The Plug-in, Module, and Intercepting Filter Patterns

A series of related design patterns define how applications can use additional components or modules to extend their capabilities or perform specific functions. Typically, they allow the features or behavior of an application to change when it loads separate components that implement extra functionality

The Plug-in pattern usually relates to general-purpose software components. For examples, consider the way that Web browsers use ActiveX controls, Java applets, and the Macromedia plug-in to provide extra functionality or display Flash animations.

The Module pattern usually relates to the capability of an application to load and use custom assemblies (modules) that extend its functionality. An example is the Composite UI Application Block (CAB), which contains a feature that allows an application to load a complete interface component (such as a window with its associated Controller or Presenter and data), and integrate it seamlessly into a composite Windows Forms interface.

The Intercepting Filter pattern has a slightly different implementation. It usually consists of a component that resides within an application pipeline, such as the HTTP pipeline that ASP.NET uses to process each request. The framework calls a method within the HTTP module at a specific point during pipeline processing, allowing the module to change the behavior of the application.

Implementing the Controller Patterns in ASP.NET
You can implement both the Page Controller and the Front Controller patterns in ASP.NET. In fact, ASP.NET makes it easy to combine them if required, as shown in Figure 3.

In Figure 3, the Front Controller specifies which of three Web pages will load depending on some feature of the request. Each Web page implements the MVP pattern, though one of them also uses the Page Controller pattern to determine which partial View to display within its page. This is, in fact, the overall structure of the downloadable sample application.

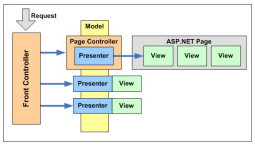


Figure 3. Combining Front Controller and Page Controller: The diagram shows how the Page Controller and Front Controller patterns can work

All three patterns implemented in this article, the Use Case Controller, Page Controller, and Front Controller, make use of the same set of three partial Views implemented as user controls. Therefore, before looking at the pattern implementations in detail, the next section examines these three user controls.

Implementing the Three Partial View User Controls
To illustrate the various controller patterns, the example application uses three very simple user controls that implement Views for several of the pages. Figure 4 shows the page from the Use Case Controller pattern example that uses the

views for several of the pages. Figure 4 shows the page from the Use Case Controller pattern example that uses the CustomerList user control (`CustomerList.ascx`). Note that the page heading, buttons, and actual URL are part of the hosting page, and not part of the user control.

in combination.

The View itself (the ASPX page) contains only a GridView control and a Label control (where any errors messages will appear). The code to populate the GridView is in the `Page_Load` event handler. It uses the CustomerModel class to get a DataSet containing the list of customers and binds the DataSet to the GridView to show the results:

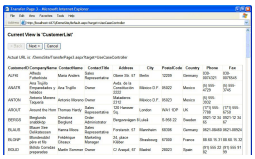


Figure 4. The Customer List User Control: In this figure, the Customer List user control doesn't include the page heading, the buttons, or the "Actual URL" label.

```
public partial class CustomerList : System.Web.UI.UserControl
{
   protected void Page_Load(object sender, EventArgs e)
   {
      try
      {
         // get DataSet of customers from CustomerModel class
         // and bind to GridView control in the page
         CustomerModel customerList = new CustomerModel();
         GridView1.DataSource = customerList.GetCustomerList();
         GridView1.DataBind();
      }
      catch (Exception ex)
      {
         Label1.Text += "USER CONTROL ERROR: " + ex.Message;
      }
      Label1.Text += "<p />";
   }
}
```

The second user control, named "CustomerDetails" (`CustomerDetails.ascx`), displays details of a specific customer (see Figure 5). To simplify the example, this is hard-coded to show the details of the customer with ID value "ALFKI", but you could of course easily modify the page to allow users to enter a customer ID in the same way as in the `Default.aspx` page you saw earlier.

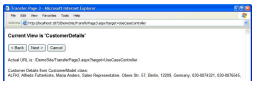


Figure 5. The Customer Details User Control: Here's how the control looks in the example application.

The code in the `Page_Load` event handler of this user control uses the `GetCustomerDetails` method of the CustomerModel class to get a DataRow containing details of the specified customer, converts this to an Object array, and then iterates through the array displaying the values:

```
public partial class CustomerDetails : System.Web.UI.UserControl
{
   protected void Page_Load(object sender, EventArgs e)
   {
      try
      {
         // use CustomerModel to get DataRow
         // for specified customer
         CustomerModel customers = new CustomerModel();
         DataRow[] details =
            customers.GetCustomerDetails("ALFKI");
         // convert row values into an array
         Object[] values = details[0].ItemArray;
         Label1.Text += "Customer Details from " +
            "CustomerModel class: <br />";
         // iterate through the array displaying the values
         foreach (Object item in values)
         {
            Label1.Text += item.ToString() + ", ";
         }
      }
      catch (Exception ex)
      {
         Label1.Text += "USER CONTROL ERROR: " + ex.Message;
      }
      Label1.Text += "<p />";
   }
}
```

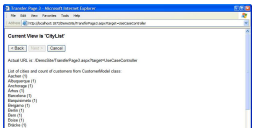


Figure 6. The City List User Control: The City List user control displays a list of cities where customers reside, shown here in the sample application.

The third user control, named "CityList" (`CityList.ascx`), displays a list of the cities where customers reside, together with a count of the number of customers in each city (see Figure 6).

The code in the `Page_Load` event handler of the "CityList" user control uses the `CustomerModel.GetCityList` method to get a System.Collections.SortedList instance containing the list of cities and the count of customers in each one. It then iterates through the list displaying the key (the city name) and the value (the number of customers) for each item:

```
public partial class CityList : System.Web.UI.UserControl
{
   protected void Page_Load(object sender, EventArgs e)
   {
      try
      {
         // use CustomerModel to get SortedList of cities
         CustomerModel customers = new CustomerModel();
         SortedList cities = customers.GetCityList();
         // iterate through the SortedList displaying
         // the values
         Label1.Text +=
```

```
            Label1.Text +=
               "List of cities from CustomerModel class: <br />";
            foreach (String key in cities.Keys)
            {
               Label1.Text += key + " ("
                  + cities[key].ToString() + ")<br />";
            }
         }
         catch (Exception ex)
         {
            Label1.Text += "USER CONTROL ERROR: " + ex.Message;
         }
         Label1.Text += "<p />";
      }
   }
```

Implementing the Use Case Controller Pattern

The example implementation of the Use Case Controller pattern uses a single ASPX page (`TransferPage1.aspx`), with a code-behind file that implements the Presenter. If the page load was caused by a postback (a user clicked one of the buttons in the page), code in the `Page_Load` event of the Presenter extracts the name of the partial View (the user control) to display from the page's ViewState, and saves this in a local variable named `viewName`. When the page load is not a postback, the code just sets `viewName` to the default value "CustomerList" and calls the method `LoadAndDisplayView` within the Presenter:

```
public partial class TransferPage1 : System.Web.UI.Page
{

   String viewName = String.Empty;

   protected void Page_Load(object sender, EventArgs e)
   {
      if (Page.IsPostBack)
      {
         // get current view name from page viewstate
         viewName = (String)ViewState["ViewName"];
      }
      else
      {
         viewName = "CustomerList";
         LoadAndDisplayView();
      }

      // display actual URL of currently executing page
      lblActualPath.Text = Request.CurrentExecutionFilePath;
      String qs = Request.QueryString.ToString();
      if (qs != null && qs != String.Empty)
      {
         lblActualPath.Text += '?' + qs;
      }
   }
...
```

The remaining code in the `Page_Load` event displays the actual URL of the current page and any query string, so that you can see the effects of the Front Controller when it redirects requests to different pages. You will see how the Front Controller works in the next article in this series.

To load and display a user control dynamically, code in the LoadAndDisplayView method creates a new instance of the control and then adds it to the Controls collection of an ASP.NET Placeholder control located in the main View (the ASPX page). After displaying the user control, the code sets the `Enabled` properties of the "Back" and "Next" buttons, depending on the current view name, and displays the name of the view in the page header element (a `<div>` control with the `runat="server"` attribute). Finally, it saves the name of the view in the page ViewState to get ready for the next postback:

```
private void LoadAndDisplayView()
{
   // load and display the appropriate view
   if (viewName != null && viewName != String.Empty)
   {
      try
      {
         UserControl view =
            (UserControl)LoadControl(viewName + ".ascx");
         viewPlaceHolder.Controls.Add(view);
      }
      catch (Exception ex)
      {
         throw new Exception(
            "Cannot load view '" + viewName + "'", ex);
      }
   }
   else
   {
      viewName = "No view specified";
   }
   // set state of buttons to match view
   btn_Back.Enabled = (viewName != "CustomerList");
   btn_Next.Enabled = (viewName != "CityList");
   // display name of current view
   pageHeaderElement.InnerText =
      "Current View is '" + viewName + "'";
   // save in page viewstate for use in postback
   ViewState["ViewName"] = viewName;
}
```

As an alternative, you could use the `Server.Execute` method to execute separate ASPX pages, each an MVP pattern implementation with its own Presenter (code-behind file) and View (ASPX page) that generates the appropriate output. This output will appear in the resulting page as a partial View, though you must remember not to include the `<html>`, `<head>`, and `<body>` elements in the partial View implementation.

However, it is likely that you will have to include the ASP.NET `<form runat="server">` section in the partial View to be able to use ASP.NET web controls in that page—which means that you can only use one partial View per hosting page. User controls are more likely to be easier to manage, and more efficient. Each can contain its own initialization code, and does not require a `<form>` section. Neither do they, by default, contain the `<html>`, `<head>`, and `<body>` elements.

The only other code the Presenter requires is button-click handlers. The event handlers for the "Back" and "Next" buttons change the value of the `viewName` local variable, and then call the `LoadAndDisplayView` method to display the current view. The event handler for the "Cancel" button just redirects the request back to the default page of the example application:

```
protected void btn_Back_Click(object sender, EventArgs e)
{
   switch (viewName)
   {
      case "CustomerDetails":
         viewName = "CustomerList";
         break;
      case "CityList":
         viewName = "CustomerDetails";
         break;
   }
   LoadAndDisplayView();
}

protected void btn_Next_Click(object sender, EventArgs e)
{
   switch (viewName)
   {
      case "CustomerList":
         viewName = "CustomerDetails";
         break;
      case "CustomerDetails":
         viewName = "CityList";
         break;
   }
   LoadAndDisplayView();
}

protected void btn_Cancel_Click(object sender, EventArgs e)
{
   Response.Redirect("Default.aspx");
}
```

To see the results, look back at Figure 4, 5, and 6. These show the three views that the Use Case Controller example displays as you click the "Back" and "Next" buttons.

Author's Note: You can open the Use Case Controller page by selecting either "Use Case Controller" or "TransferPage1.aspx" in the dropdown list at the bottom of the `Default.aspx` page of the sample application.

Implementing the Page Controller Pattern

ASP.NET automatically provides a Presenter implementation for the Model-View-Presenter pattern through a code-behind file for each ASP.NET page. However, you can extend the implementation by adding your own Page Controller in the form of a base class that the code-behind files inherit.

By default, the code-behind class for an ASP.NET page inherits from the Page class in the System.Web.UI namespace. If you create a suitable base class that inherits from Page, you can use this as the base for your own concrete implementations of the code-behind class. This is useful if some or all of your pages require common functionality.

The base class can also perform some of the tasks of a Page Controller by handling the `Page_Init` or `Page_Load` events. For example, code in the `Page_Load` event can select a View to display at runtime for a page, allowing a single page to change its content based on some external condition such as a value in the query string.

The example application uses a base class named PageControllerBase (in the `App_code` folder), which inherits from System.Web.UI.Page and exposes three values that the concrete classes that inherit from it can access. These values are a reference to the partial View (a user control) to load and display, the name of the View, and a string value to display when no View is specified:

By default, the code-behind class for an ASP.NET page inherits from the Page class in the System.Web.UI namespace.

```
// base class for TransferPage2 and TransferPage3
// implements common tasks and can call method(s)
// in the concrete page class implementations
public class PageControllerBase : Page
{
   // values that will be available in page class
   protected UserControl displayView = null;
   protected String displayViewName = String.Empty;
   protected String noViewSelectedText = "No View Selected";
   ...
```

The PageControllerBase class handles the `Page_Load` event to get the name of the View to display from the query string, and then attempts to create an instance of the appropriate user control for the concrete page code to access. At the end of the `Page_Load` event handler, the code calls a method named `PageLoadEvent`, passing in the values of the sender and EventArgs originally received as parameters:

```
protected void Page_Load(Object sender, System.EventArgs e)
```

```
{
   // get name of view (UserControl) to show in the page
   displayViewName = Context.Request.QueryString["view"];
   if (displayViewName != null && displayViewName !=
      String.Empty)
   {
      try
      {
         // load view from ASCX file
         displayView = (UserControl)
            Page.LoadControl(displayViewName + ".ascx");
      }
      catch { }
   }
   // call concrete page class method
   PageLoadEvent(sender, e);
}
```

Every page that inherits from this base class must implement the `PageLoadEvent` method. To ensure that this is the case, the PageControllerBase class declares a virtual method that each concrete page implementation must override:

```
// define method that concrete page classes must implement
// and will be called by this base class after Page_Load
virtual protected void PageLoadEvent(Object sender, System.EventArgs e)
{
   // overridden in concrete page implementation
}
```

The example application contains two pages, named `TransferPage2.aspx` and `TransferPage3.aspx` that inherit from the PageControllerBase class. The code in the overridden `PageLoadEvent` methods is the same in both pages—it is only the View (the ASPX UI) that differs. However, they could be completely different if required, although all inheriting page should make some use of the common functionality of the PageControllerBase class (or there is no point in inheriting from it!). Here's the `PageLoadEvent` code used in both pages:

```
public partial class TransferPage2 : PageControllerBase
{
   // override virtual method in PageControllerBase class
   protected override void PageLoadEvent(Object sender,
      System.EventArgs e)
   {
      // use values that were set in base class Page_Load event
      if (displayView != null)
      {
         pageTitleElement.Text = "Displaying view '" +
            displayViewName + "'";
         pageHeadingElement.InnerHtml = "Displaying view '" +
            displayViewName + "'";
         PlaceHolder1.Controls.Add(displayView);
      }
      else
      {
         pageHeadingElement.InnerHtml = noViewSelectedText;
      }

      // display actual URL of currently executing page

      lblActualPath.Text = Request.CurrentExecutionFilePath;
      String qs = Request.QueryString.ToString();
      if (qs != null && qs != String.Empty)
      {
         lblActualPath.Text += '?' + qs;
      }
   }
}
```

The preceding code displays the name of the current View in the page title and heading, and adds the user control instance created by the PageControllerBase class to the `Controls` collection of an ASP.NET Placeholder control declared in the ASPX file:

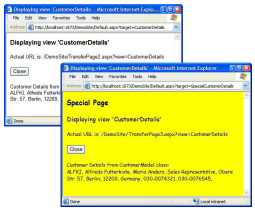


Figure 7. The CityList View : The figure shows two differing views of the same page.

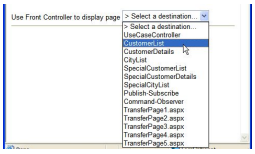
The final section of the `PageLoadEvent` method shown above displays the actual page URL so that you can see the effects of the Front Controller implementation (described in the next article in this series). Figure 7 shows the two pages described in this section displaying the "CityList" View.

> Author's Note: You can open the page `TransferPage2.aspx` with the appropriate View displayed by selecting the View name ("CustomerList," "CustomerDetails," or "CityList") in the dropdown list at the bottom of the `Default.aspx` page. To open the `TransferPage3.aspx` page with the appropriate View displayed, select "SpecialCustomerList," "SpecialCustomerDetails," or "SpecialCityList." To open the pages with no View displayed, select the page name ("TransferPage2.aspx" or "TransferPage3.aspx") in the dropdown list.

Implementing the Front Controller Pattern

The most common approach for implementing the Front Controller pattern in ASP.NET is through an HTTP Module that handles one of the ASP.NET HTTP pipeline events, and executes a `Server.Transfer` action to load the appropriate target page. This is the technique implemented in the example application. A dropdown list at the bottom of the `Default.aspx` page displays a list of possible targets, from which you can select (see Figure 8).

You may recall from the discussion of the Singleton pattern in the previous article in this series that the example application uses a Singleton class named `TransferUrlList.cs` to expose the contents of an XML file that contains the target "short names" (as displayed in the dropdown list in Figure 8) and the actual URL to load:



```
<?xml version="1.0" encoding="utf-8" ?>
<transferUrls>
```

```
  <item name="UseCaseController" url="TransferPage1.aspx" />
  <item name="CustomerList"
    url="TransferPage2.aspx?view=CustomerList" />
  <item name="CustomerDetails"
    url="TransferPage2.aspx?view=CustomerDetails" />
  <item name="CityList"
    url="TransferPage2.aspx?view=CityList" />
  <item name="SpecialCustomerList"
    url="TransferPage3.aspx?view=CustomerList" />
  <item name="SpecialCustomerDetails"
    url="TransferPage3.aspx?view=CustomerDetails" />
  <item name="SpecialCityList"
    url="TransferPage3.aspx?view=CityList" />
  <item name="Publish-Subscribe" url="TransferPage4.aspx" />
  <item name="Command-Observer" url="TransferPage5.aspx" />
</transferUrls>
```


Figure 8. Sample Application Page Targets: The figure shows the list of targets for the Front Controller example displayed in the default page.

To demonstrate the Front Controller pattern, the application contains a class named `FrontController.cs` (in the `App_Code` folder) that implements an HTTP Module. The class declaration indicates that it implements the IHttpModule interface, which means that it must declare the public methods `Init` and `Dispose`.

In the `Init` method, the class subscribes to the `PreRequestHandlerExecute` event by registering an event handler named `MyPreRequestHandler`. As the class does not use any unmanaged resources, the `Dispose` method requires no action—so it declares only an empty method:

```
public class FrontController : IHttpModule
{
   public void Init(HttpApplication context)
   {
      // register a handler for the event you want to handle
      context.PreRequestHandlerExecute += MyPreRequestHandler;
   }

   public void Dispose()
   {
      // no clean-up required
   }
   ...
```

The ASP.NET HTTP pipeline raises the `PreRequestHandlerExecute` event just before it executes the request. The code in the event handler in the FrontController module can therefore make a decision about which page to load based on some external condition. This may be something like the browser language, country, browser type, a value in the user's ASP.NET Profile, or some other value from the environment.

The example module takes simpler approach to make it easier to specify the target you want when experimenting with the application. It examines the query string for a value that corresponds to one in the list of transfer URLs exposed from the XML file, and redirects to the appropriate page. The `GetTransferUrl` method of the TransferUrlList class returns the translated URL if found in the list, or the original URL if it is not in the list. The code then executes the `Server.Transfer` method to that URL:

```
private void MyPreRequestHandler(Object sender, EventArgs e)
{
   // use features of the request (user, browser,
   // IP address, etc.) to decide how to handle the request,
   // and which page to show.
   // this example looks for specific items in the query
   // string that indicate the required target (such as
   // "CustomerList") using a dictionary of values loaded from
   // an XML disk file

   // get Singleton list of transfer URLs
   TransferUrlList urlList = TransferUrlList.GetInstance();
   // get the current request query string
   String reqTarget = HttpContext.Current.
      Request.QueryString["target"];
   if (reqTarget!= null && reqTarget != String.Empty)
   {
      // see if target value matches a transfer URL
      // by querying the list of transfer URLs
      // method returns the original value if no match
      String transferTo = urlList.GetTransferUrl(reqTarget);
      try
      {
         // transfer to the specified URL
         HttpContext.Current.Server.Transfer(transferTo, true);
      }
```

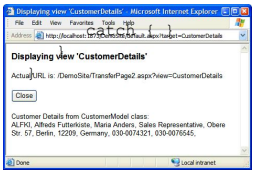


Figure 9. Effect of Selecting "CustomerDetails": When you select the "CustomerDetails" item in the dropdown list, the application loads TransferPage2, which displays the CustomerDetails View as shown here.

When you select one of the "short name" values in the dropdown list in the default page of the application, you'll see the effects of the Front Controller module. For example, Figure 9 shows the result of selecting the "CustomerDetails" option. The code in the Front Controller HTTP module translates this into the URL `TransferPage2.aspx?view=CustomerDetails`—as you can see in the Label control that displays the actual URL of the current page.

Notes on Using a Front Controller HTTP Module

You may be tempted to try using URLs for your Front Controller that include the "short names" as part of the path, rather than in the query string. This will work when you run your application within Visual Studio 2005 because all requests go to the built-in Visual Studio Web server. However, if you run your application under the IIS Web server you will get "Page Not Found" errors, because IIS passes only pages that actually exist to the ASP.NET runtime.

For example, if you use a URL such as `http://localhost/CityList`, and there is no folder in your Web site named `CityList`, IIS will return a "404 Not Found" error. It will *not* initiate the ASP.NET HTTP pipeline, and so your module will not execute. Even if the `CityList` folder does exist, IIS will return either a "403 - Forbidden" error, or (if Directory Browsing is enabled) a file listing.

Of course, you could create a folder for each "short name" and put an empty ASPX page in each of these folders so that the Front Controller module redirects to the appropriate page as required. This is better than using pages that contain redirection code (such as `Response.Redirect` or `<meta httpequiv="refresh">` elements) in each folder, because the Front Controller will intercept each request and will not actually load the empty ASP.NET page.

An alternative is to map all requests to the ASP.NET runtime DLL, rather than just the standard ASP.NET file extensions such as `aspx` and `ascx`. However, this is probably not a practical solution because it will result in an excessive processing overhead in ASP.NET.

As you are probably beginning to realize, ASP.NET makes extensive use of design patterns internally, and you can capitalize on design patterns yourself to build logical and easy-to-maintain applications. I'll cover some more advanced patterns in the final article in this series.