ASP.NET Presentation Patterns

http://i.msdn.microsoft.com/Global/Images/clear.gif Contents

[Inside an ASP.NET Event Handler](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080003)   
[The Original MVC Pattern](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080053)   
[Model2: A Web Variation of MVC](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080069)   
[ASP.NET MVC Framework vs. Manual MVC](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080076)   
[The MVP Pattern](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080083)   
[The Page Controller Pattern](http://msdn.microsoft.com/en-us/magazine/dd252940.aspx#id0080092)

In a layered Web app, the presentation layer is just as important as navigation logic, business logic, and data access. While you should try to keep the presentation layer (mostly the presentation logic) independent from the UI technology and platform, it's not always easy to do. Design patterns can help.

In this month's column, I'll examine some design patterns for building an ASP.NET presentation layer. I'll start with the Model-View-Controller (MVC) pattern—the root of all patterns specifically targeted to the UI—and discuss its applicability to ASP.NET beyond the ASP.NET MVC Framework.

**Inside an ASP.NET Event Handler**

An ASP.NET presentation layer is composed mostly of .aspx pages that are orchestrated by the HTTP runtime environment. A typical .aspx page, at some point, may place an HTTP request following a certain user's action (such as a button click or a list selection). In ASP.NET Web Forms programming, these events are usually handled by methods—event handlers—written in the codebehind class of the page. At first glance, it would appear that there's a direct connection between the user action and the system's reaction. While this may be true in desktop applications, it is not the case in ASP.NET.

In ASP.NET, a lot of things happen in the time between a user click and the display of an updated page—regardless of whether or not you're using AJAX. The flow of information is shorter, however, in ASP.NET applications that you build with the ASP.NET MVC Framework.

Let's consider a button click event. As a developer, you handle it by writing some code in the Button control's Click event handler. The code goes in the page's codebehind, like so:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl02_code');" \o "Copy Code)

void Button1\_Click(object sender, EventArgs e)

{

// Perform any required action

}

You could place all the code for the required action here. Or, you could group all of this code in a static or instance method exposed by an object:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl03_code');" \o "Copy Code)

void Button1\_Click(object sender, EventArgs e)

{

// Static method bound to the user action

ActionManager.Button1Clicked();

}

You could also organize the code that responds to the user's activity in controllers and actions:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl04_code');" \o "Copy Code)

void Button1\_Click(object sender, EventArgs e)

{

// Static method bound to the user action

ThisPageController controller = new

ThisPageController();

controller.Button1Clicked();

}

I'm assuming that each page has its own controller component and that each controller has a list of public methods logically bound to a possible user action.

How would you let the controller know the state of the page and how the new view is generated? Button1\_Click is invoked on the server after the user has clicked the button and the ASP.NET runtime has received and processed the corresponding HTTP request. The Button1\_Click method is exposed by a class derived from System.Web.UI.Page. This class has access to UI controls.

Here's a simple change to the controller's signature that would enable the controller to access any public element in the page:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl05_code');" \o "Copy Code)

void Button1\_Click(object sender, EventArgs e)

{

// Static method bound to the user action

ThisPageController controller = new ThisPageController(this);

controller.Button1Clicked();

}

The controller's signature now receives a reference to the current ASP.NET page—the view. The controller method performs its work and then simply updates server controls. The standard page lifecycle of ASP.NET will do the rest and produce the HTML for the browser.

**The Page Lifecycle**

**Figure 1** shows the main steps in the page lifecycle. Whenever an ASP.NET request hits the Web server and an HTTP handler is found to service it, the steps outlined in the figure are performed. As you can see, the code for a simple click of a button is only a small fraction of the whole ASP.NET infrastructure. You'll see in a moment that the page lifecycle represents a fraction of the work that the ASP.NET runtime executes to service each request.

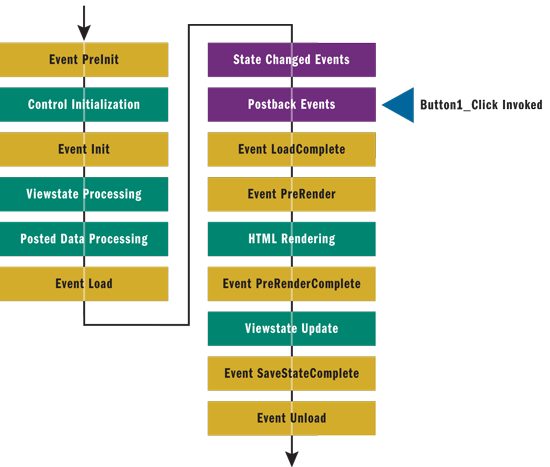


Figure 1 **The ASP.NET Page Lifecycle**

Before the code in Button1\_Click can execute, the ASP.NET runtime creates an instance of the Page class that holds the expected behavior for the requested URL. The Page class processes the request according to the steps in **Figure 1** and uses your event handlers where appropriate.

In the Button1Clicked method in a controller class, you should be able to access any of the controls in the page with the same syntax you would use in a plain codebehind event handler. But members that reference UI controls (say, TextBox1) are not public members on the Page class. How can you make them accessible to a controller? You can implement a custom interface on the codebehind class that includes public properties for the UI controls you intend to manipulate from within the controller. Here's an example:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

public interface IThisPageView

{

TextBox TextBox1Property;

...

}

// In the codebehind class

public TextBox TextBox1Property

{

get {return TextBox1;}

}

**Figure 2** provides an example of the sample controller. In Button1Clicked, you do your regular ASP.NET Web Forms programming and determine the next view by simply updating the state of server controls. The logic of the page is now encapsulated in a separate class that can be developed separately from the page.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 2 A Sample Class Acting as the Controller for a Page

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22_ctl00_ctl00_code');" \o "Copy Code)

public class ThisPageController

{

protected IThisPageView thePage;

public ThisPageController(IThisPageView page)

{

thePage = page;

}

public void Button1Clicked()

{

// Get the text currently stored in control TextBox1

string text = thePage.TextBox1Property.Text;

// Do something: i.e., turn to upper case

string newText = text.ToUpper();

// Update the text in control TextBox1

\_thePage.TextBox1Property.Text = newText;

}

}

This solution, though, is still tightly coupled to the controls in the page. Let's consider a refinement that refers to the Model-View-Presenter (MVP) pattern. I'll briefly recall the basics of the MVC and MVP patterns.

**The Original MVC Pattern**

The MVC pattern was devised back in 1979 as an approach to move away from catch-all, autonomous front ends. An autonomous view is a class that displays content, maintains state information for the view, and incorporates the full logic to handle any user actions from start to finish.

This code snippet lays the groundwork for an autonomous view:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl10_code');" \o "Copy Code)

void Button1\_Click(object sender, EventArgs e)

{

// Perform any required action. All the code you

// need to handle the event goes here. A page

// built in this way is an autonomous view.

// MVC fights autonomous views.

}

With such a monolithic view, you can hardly build a testable presentation layer that's separate from the underlying layers. Applying the concept of Separation of Concerns (SoC) helps you to get the right combination of low coupling and high cohesion for any component, because it causes you to keep the various functionalities in their own layers rather than intermingled. Second, SoC makes it easier to implement a navigational workflow to decide which page comes next.

The original paper describing the MVC approach, "[How to Use Model-View-Controller](http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html)," is available online. If you've never read it, I suggest you have a look at it regardless of your familiarity with the MVC philosophy. Today, MVC is considered a pattern for building a presentation layer, but it was originally meant for building complete applications. It's helpful to look at it in this light.

MVC is a loosely defined pattern that gives the architect much discretion over implementation details. This is probably why so many variations of MVC exist.

MVC splits the app into three parts—the model, the view, and the controller. The model refers to the application data, manages the application's functionalities, and notifies the view of state changes. The view is concerned with the content to display to users. Finally, the controller maps user gestures to actions on the model and updates the current view or selects the next view. These three actors are often referred to as the MVC triad. **Figure 3** provides a graphical comparison between autonomous views and MVC applications.

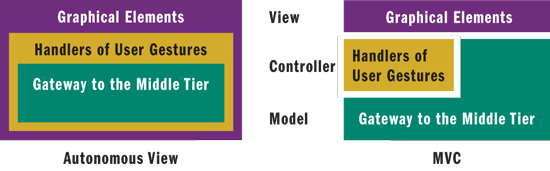


Figure 3 **From Autonomous View to MVC**

In MVC, the user interacts with the view, and actions (such as a button click) are captured by the view and forwarded to the controller. The controller decides what to do and does it through an interaction with the model. The model is essentially the business layer plus some extra capabilities. In particular, the model notifies the view about changes that may require an update of the UI.

The model doesn't know any details of the view, but between model and view there's an "observer" relationship. In other words, the view holds a reference to the model and registers itself with the model to receive a notification of changes. When the notification is received, the view gets fresh data from the model and updates the UI. **Figure 4** illustrates a sequence diagram of an MVC interaction. Note that in some [MVC implementations](http://msdn.microsoft.com/library/ms978748), it's the controller that notifies the view of the changes.

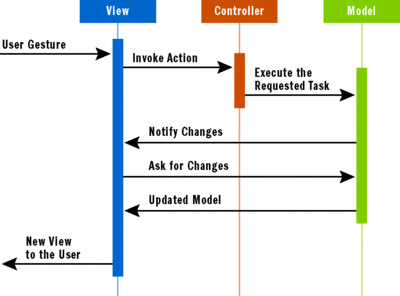


Figure 4 **A Standard MVC Interaction** (Click the image for a larger view)

**Model2: A Web Variation of MVC**

MVC was devised to target desktop applications, but because of its relatively loose formulation, it was easily adapted for the Web. A popular Web variation of the MVC pattern is Model2. Model2 is the historical name of the pattern that fuels the ASP.NET MVC Framework today.

The Model2 pattern exposes the view to users through the browser. User actions are captured by the browser and transformed into a new HTTP request. In this way, requests flow from the browser to a special component (referred to as the front controller) sitting within the Web server. The front controller coordinates all of the requests that are made to the Web application.

In ASP.NET, the front controller takes the form of the URL-routing HTTP module. The HTTP module captures the request and routes it to the appropriate controller for the requested action to take place. As the action method returns, the controller orders the view to refresh and passes fresh data for the new UI. The output of the view is captured by the front controller HTTP module and sent back to the browser.

**Figure 5** shows the sequence diagram for a typical Model2 interaction. You should note that the diagram includes only the general steps. For example, in the ASP.NET MVC implementation of the pattern, the URL routing component does some work in addition to using the controller.

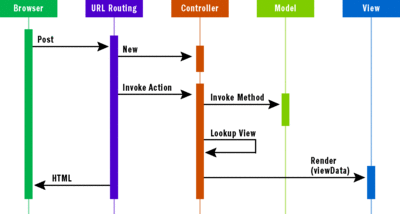


Figure 5 **The Model2 Interaction as in the ASP.NET MVC Framework** (Click the image for a larger view)

A few differences exist between the original MVC and Model2. As you can see, there's no contact between the view and the model as in the MVC original formulation based on the aforementioned "observer" relationship. The user action is not captured and handled by the view. The controller renders the view and explicitly passes display data to it.

In ASP.NET, when you say MVC you should specify whether you mean Model2 or a manual implementation of MVC, as briefly explained in the beginning of this column. (If you'd like to read more on the *MSDN Magazine* article [ASP.NET MVC Framework](http://msdn.microsoft.com/magazine/cc337884).

**ASP.NET MVC Framework vs. Manual MVC**

Is there any difference at all between using the ASP.NET MVC Framework and rolling your own implementation of the MVC pattern in order to have a controller class for each page (or for a group of pages)? In terms of SoC, I don't see any significant differences. In both cases, you have a controller class that is neatly separated from the view and the model that represents the gateway to the business or service layer.

In terms of testability, the Model2 pattern is preferable over the original formulation of the MVC pattern because of the neat separation it forces between view and model and because of an extremely thin view. This aspect alone makes code written for the ASP.NET MVC Framework extremely effective to test. Furthermore, in Model2 there's a sort of loose contract that can be established between the view and the controller. In the ASP.NET MVC Framework, this contract is represented by the ViewData container object or, better yet, by the ViewPage<T> base class for view classes.

Compared with ASP.NET Web Forms, a Model2 implementation is also faster to execute the requested action. In a Model2 implementation, all you need is a component that processes the HTTP request and invokes a controller. There's no need to create page classes dynamically; the lookup for the controller is based on a much simpler algorithm than the lookup of an HTTP handler in a Web Forms scenario. Likewise, you don't need the page lifecycle, the viewstate, or server controls. In a Model2 implementation, such as the ASP.NET MVC Framework, you have a much more agile runtime environment. Let's explore it further.

In an MVC scenario implemented over the Web Forms programming model, any user action originates an HTTP post. The Web server captures these requests and maps them to a page class. The Page class goes through its own lifecycle, as shown in **Figure 1**. Next, the Page class figures out the right controller and invokes a method. The execution of the method modifies the model. The remainder of the page lifecycle involves refreshing the view.

In a Model2 scenario implemented with the ASP.NET MVC Framework, the generation of the next view is a much simpler process. The URL routing module parses the URL, determines from it the controller to use, and invokes an action on it. Then the controller collects new data for the new view and passes this information on. The view assembles some HTML that is then returned to the browser as the response to the captured request. The view is a passive subject that needs virtually no testing. Instead, you focus your testing effort on the controller.

Model2 is an excellent alternative for improving the testability of an application. However, what if you still need the Web Forms runtime environment with its support for Session, Cache, server controls, and even viewstate that you get with the MVC pattern? Does this mean that you have to abandon SoC and testability? Certainly not. You can opt for the other variation of MVC that I mentioned at the outset—the MVP pattern. It applies to Web applications without requiring an ad hoc runtime.

**The MVP Pattern**

The original formulation of MVC has one key weakness: the mechanism that updates the view. The flow of information you observe in **Figure 4** shows that the view communicates the user gesture to the controller but receives a notification of changes from the model. Next, the view needs to leverage its intimate knowledge of the model to grab updated data and refresh itself. MVP is a variation of MVC that separates the elements of the triad more cleanly. **Figure 6** displays the typical interaction between the actors in an MVP model.

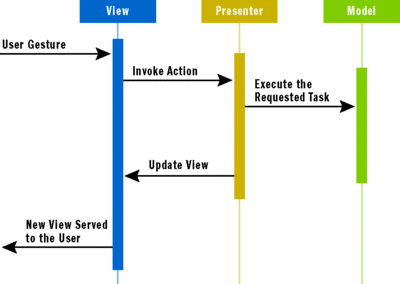


Figure 6 **The MVP Pattern in Action** (Click the image for a larger view)

In MVP, the view forwards user input to the presenter, and from the presenter it receives fresh data for updates. In turn, the presenter services the request by interacting with the model.

In the original MVC, there's no explicit contract that states which data the view needs. In MVC, the view holds a reference to the model and runs its own logic to select the data it needs and massage it into UI elements. Equipped with this logic, the view is not as passive as it should be for testing purposes. In addition, the view depends, to some extent, on the underlying UI platform.

In MVP, the presenter is essentially the mediator between end users and applications. The presenter has the power to render the view and interact with the model. As a result, most of the presentation logic lives within the presenter. Because the presenter is a plain class with no UI, it is an inherently more testable class. You can see a practical implementation of the MVP pattern in ASP.NET in the August 2006 [*MSDN Magazine* Design Patterns column](http://msdn.microsoft.com/magazine/cc188690). MVP is natively supported by the Microsoft Web Client Software Factory, which is a collection of ASP.NET-related application blocks. An [MVP QuickStart](http://msdn.microsoft.com/library/cc304754) is available on MSDN.

MVP is a general UI pattern that, unlike Model2, is not specifically designed for the Web. Model2 (the ASP.NET MVC Framework) and MVP are nearly the same thing. Interestingly, neither was originally designed to be a general-purpose pattern.

Technically speaking, there's just one difference between Model2 and MVP: the contract between the controller (called the presenter in MVP) and the view. The contract is loosely defined in Model2 and formally defined in MVP.

In MVP, the view implements an interface, and the presenter talks to the view using only the members on the interface. So, for example, the presenter reads input data in the view using methods and getters in the interface and sets new values for the view using methods and/or setters in the interface. In Model2, you use a strongly typed container.

With MVP, the presentation logic gains independence from the UI platform, so it's easier to reuse the same presenter across different platforms. In addition, the same presenter can work with different views of the same application thus enabling Software as a Service (SaaS) scenarios. Finally, with MVP you have a component—the presenter—that may store some logic for navigation between pages. Whether you implement the logic internally or use a Windows Workflow Foundation (WF) component is up to you.

**The Page Controller Pattern**

MVC, Model2, and MVP are patterns external to the design of the ASP.NET runtime. The Web Forms programming model of ASP.NET is based on yet another design pattern—the Page Controller pattern. The Page Controller pattern entails the creation of a central object (the dynamically created class that inherits from your codebehind class) that handles all HTTP requests directed at a specific Web page (see **Figure 7**).

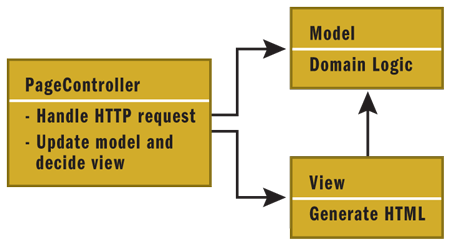


Figure 7 **The Page Controller Structure**

If you customize this pattern, you may get some benefits in terms of logic reuse across pages, including the navigation logic. You can customize it by simply creating a hierarchy of Page classes and incorporating navigation and presentation logic in the classes higher up in the hierarchy to make that logic available to derived classes. A customized page controller may be a good way to improve the reuse of presentation and navigation logic without switching to a full new pattern.

Applying MVP means essentially designing the codebehind class of pages in a different way by implementing an interface on the Page class and using a separate component—the presenter—to handle user gestures. No changes occur to the runtime environment, as the request is routed to the codebehind class in the usual way.

Creating an ASP.NET MVC Framework project means entering changes to the runtime environment to enable a front controller—the URL routing module—to resolve the request in terms of an action on a controller. Both options have an impact on the development of the application.

The first issue to assess is whether you need viewstate and server controls. There might be situations where you prefer not to deal with them. As an example, if you have no server controls, styling the UI using CSS is much easier. Using AJAX with server controls is currently easier than with an ASP.NET MVC Framework solution. Applications with a lot of data entry or complex table-based views are not so easy to build without rich server controls. So it is ultimately a trade-off. The key question is whether you need server controls and viewstate. There's no obvious answer.

If you decide you'd better stick to Web Forms and still want some SoC, then MVP is for you. MVP is an important UI pattern that may be a bit expensive to implement in relatively simple applications. On the other hand, MVP shines in enterprise-class applications where you really need to reuse as much presentation logic as possible, across multiple platforms and in SaaS scenarios.