[ASP.NET PAGE OBJECT MODEL:]

Summary: Learn about the eventing model built around ASP.NET Web pages and the various stages that a Web page experiences on its way to HTML. The ASP.NET HTTP run time governs the pipeline of objects that transform the requested URL into a living instance of a page class first, and into plain HTML text next. Discover the events that characterize the lifecycle of a page and how control and page authors can intervene to alter the standard behavior. (6 printed pages)

Introduction: Each request for a Microsoft® ASP.NET page that hits Microsoft® Internet Information Services (IIS) is handed over to the ASP.NET HTTP pipeline. The HTTP pipeline is a chain of managed objects that sequentially process the request and make the transition from a URL to plain HTML text happen. The entry point of the HTTP pipeline is the HttpRuntime class. The ASP.NET infrastructure creates one instance of this class per each AppDomain hosted within the worker process (remember that the worker process maintains one distinct AppDomain per each ASP.NET application currently running).

The HttpRuntime class picks up an HttpApplication object from an internal pool and sets it to work on the request. The main task accomplished by the HTTP application manager is finding out the class that will actually handle the request. When the request is for an .aspx resource, the handler is a page handler—namely, an instance of a class that inherits from Page. The association between types of resources and types of handlers is stored in the configuration file of the application. More exactly, the default set of mappings is defined in the <httpHandlers> section of the machine.config file. However, the application can customize the list of its own HTTP handlers in the local web.config file. The line below illustrates the code that defines the HTTP handler for .aspx resources.

```
<add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
```

An extension can be associated with a handler class, or more in general, with a handler factory class. In all cases, the HttpApplication object in charge for the request gets an object that implements the IHttpHandler interface. If the association resource/class is resolved in terms of a HTTP handler, then the returned class will implement the interface directly. If the resource is bound to a handler factory, an extra step is necessary. A handler factory class implements the IHttpHandlerFactory interface whose GetHandler method will return an IHttpHandler-based object.

How can the HTTP run time close the circle and process the page request? The IHttpHandler interface features the ProcessRequest method. By calling this method on the object that represents the requested page, the ASP.NET infrastructure starts the process that will generate the output for the browser.

The Real Page Class

The type of the HTTP handler for a particular page depends on the URL. The first time the URL is invoked, a new class is composed and dynamically compiled to an assembly. The source code of the class is the outcome of a parsing process that examines the .aspx sources. The class is defined as part of the namespace ASP and is given a name that mimics the original URL. For example, if the URL endpoint is page.aspx, the name of the class is ASP.Page_aspx. The class name, though, can be programmatically controlled by setting the ClassName attribute in the @Page directive.

The base class for the HTTP handler is Page. This class defines the minimum set of methods and properties shared by all page handlers. The Page class implements the IHttpHandler interface.

Under a couple of circumstances, the base class for the actual handler is not Page but a different class. This happens, for example, if code-behind is used. Code-behind is a development technique that insulates the code necessary to a page into a separate C# or Microsoft Visual Basic® .NET class. The code of a page is the set of event handlers and helper methods that actually create the behavior of the page. This code can be defined inline using the <script runat=server> tag or placed in an external class—the code-behind class. A code-behind class is a class that inherits from Page and specializes it with extra methods. When specified, the code-behind class is used as the base class for the HTTP handler.

The other situation in which the HTTP handler is not based on Page is when the configuration file of the application contains a redefinition for the PageBaseType attribute in the <pages> section.

<pages PageBaseType="Classes.MyPage, mypage" />

The PageBaseType attribute indicates the type and the assembly that contains the base class for page handlers. Derived from Page, this class can automatically endow handlers with a custom and extended set of methods and properties.

The Page Lifecycle

Once the HTTP page handler class is fully identified, the ASP.NET run time calls the handler's ProcessRequest method to process the request. Normally, there is no need to change the implementation of the method as it is provided by the Page class.

This implementation begins by calling the method FrameworkInitialize, which builds the controls tree for the page. The method is a protected and virtual member of the TemplateControl class—the class from which Page itself derives. Any dynamically generated handler for an .aspx resource overrides FrameworkInitialize. In this method, the whole control tree for the page is built.

Next, ProcessRequest makes the page transit various phases: initialization, loading of view state information and postback data, loading of the page's user code and execution of postback server-side events. After that, the page enters in rendering mode: the updated view state is collected; the HTML code is generated and then sent to the output console. Finally, the page is unloaded and the request is considered completely served.

During the various phases, the page fires a few events that Web controls and user-defined code can intercept and handle. Some of these events are specific for embedded controls and subsequently can't be handled at the level of the .aspx code.

A page that wants to handle a certain event should explicitly register an appropriate handler. However, for backward compatibility with the earlier Visual Basic programming style, ASP.NET also supports a form of implicit event hooking. By default, the page tries to match special method names with events; if a match is found, the method is considered a handler for the event. ASP.NET provides special recognition of six method names. They are Page_Init, Page_Load, Page_DataBind, Page_PreRender, and Page_Unload. These methods are treated as handlers for the corresponding events exposed by the Page class. The HTTP run time will automatically bind these methods to page events saving developers from having to write the necessary glue code. For example, the method named Page_Load is wired to the page's Load event as if the following code was written.

this.Load += new EventHandler(this.Page_Load);

The automatic recognition of special names is a behavior under the control of the AutoEventWireup attribute of the @Page directive. If the attribute is set to false, any applications that wish to handle an event need to connect explicitly to the page event. Pages that don't use automatic event wire-up will get a slight performance boost by not having(51-aspx) to do the extra work of matching names and events. You should note that all Microsoft Visual Studio® .NET projects are created with the AutoEventWireup attribute disabled. However, the default setting for the attribute is true, meaning that methods such as Page_Load are recognized and bound to the associated event.

The execution of a page consists of a sequence of stages listed in the following table and is characterized by application-level events and/or protected, overridable methods.

Table 1. Key events in the life of an ASP.NET page

Stage	Page Event	Overridable method
Page initialization	Init	
View stateloading		LoadViewState
Postback data processing		LoadPostData method in any control that implements the IPostBackDataHandler interface
Page loading	Load	
Postback change notification		Raise51PosAspxtDataChangedEvent method in any control that implements the IPostBackDataHandler interface
Postback event handling	Any postback event defined by controls	RaisePostBackEvent method in any control that implements the IPostBackEventHandler interface
Page pre-rendering phase	PreRender	
View state saving		SaveViewState
Page rendering		Render
Page unloading	Unload	

Some of the stages listed above are not visible at the page level and affect only authors of server controls and developers who happen to create a class derived from Page. Init, Load, PreRender, Unload, plus all postback events defined by embedded controls are the only signals of life that a page sends to the external world.

Stages of Execution

The first stage in the page lifecycle is the initialization. This stage is characterized by the Init event, which fires to the application after the page's control tree has been successfully created. In other words, when the Init event arrives, all the controls statically declared in the .aspx source file have been instantiated and hold their default values. Controls can hook up the Init event to initialize any settings that will be needed during the lifetime of the incoming Web request. For example, at this time controls can load external template files or set up the handler for the events. You should notice that no view state information is available for use yet.

Immediately after initialization, the page framework loads the view state for the page. The view state is a collection of name/value pairs, where controls and the page itself store any information that must be persistent across Web requests. The view state represents the call context of the page. Typically, it contains the state of the controls the last time the page was processed on the server. The view state is empty the first time the page is requested in the session. By default, the view state is stored in a hidden field silently added to the page. The name of this field is __VIEWSTATE. By overriding the LoadViewState method—a protected overridable method on the Control class—component developers can control how the view state is restored and how its contents are mapped to the internal state.

Methods like LoadPageStateFromPersistenceMedium and its counterpart SavePageStateToPersistenceMedium can be used to load and save the view state to an alternative storage medium—for example, Session, databases, or a server-side file. Unlike LoadViewState, the aforementioned methods are available only in classes derived from Page.

Once the view state has been restored, the controls in the page tree are in the same state they were the last time the page was rendered to the browser. The next step consists of updating their state to incorporate client-side changes. The postback data-processing stage gives controls a chance to update their state so that it accurately reflects the state of the corresponding HTML element on the client. For example, a server TextBox control has its HTML counterpart in an <input type=text> element. In the postback data stage, the TextBox control will retrieve the current value of <input> tag and use it to refresh its internal state. Each control is responsible for extracting values from posted data and updating some of its properties. The TextBox control will update its Text property whereas the CheckBox control will refresh its Checked property. The match between a server control and a HTML element is found on the ID of both.

At the end of the postback data processing stage, all controls in the page reflect the previous state updated with changes entered on the client. At this point, the Load event is fired to the page.

There might be controls in the page that need to accomplish certain tasks if a sensitive property is modified across two different requests. For example, if the text of a textbox control is modified on the client, the control fires the TextChanged event. Each control can take the decision to fire an appropriate event if one or more of its properties are modified with the values coming from the client. Controls for which these changes are critical implement the IPostBackDataHandler interface, whose LoadPostData method is invoked immediately after the Load event. By coding the LoadPostData method, a control verifies if any critical change has occurred since last request and fires its own change event.

The key event in the lifecycle of a page is when it is called to execute the server-side code associated with an event triggered on the client. When the user clicks a button, the page posts back. The collection of posted values contains the ID of the button that started the whole operation. If the control is known to implement the IPostBackEventHandler interface (buttons and link buttons will do), the page framework calls the RaisePostBackEvent method. What this method does depends on the type of the control. With regard to buttons and link buttons, the method looks up for a Click event handler and runs the associated delegate.

After handling the postback event, the page prepares for rendering. This stage is signaled by the Pretender event. This is a good time for controls to perform any last minute update operations that need to take place immediately before the view state is saved and the output rendered. The next state is SaveViewState, in which all controls and the page itself are invited to flush the contents of their own ViewState collection. The resultant view state is then serialized, hashed, Base64 encoded, and associated with the __VIEWSTATE hidden field.

The rendering mechanism of individual controls can be altered by overriding the Render method. The method takes an HTML writer object and uses it to accumulate all HTML text to be generated for the control. The default implementation of the Render method for the Page class consists of a recursive call to all constituent controls. For each control the page calls the Render method and caches the HTML output.

The final sign of life of a page is the Unload event that arrives just before the page object is dismissed. In this event you should release any critical resource you might have (for example, files, graphical objects, database connections).

Finally, after this event the browser receives the HTTP response packet and displays the page.

Summary

The ASP.NET page object model is particularly innovative because of the eventing mechanism. A Web page is composed of controls that both produce a rich HTML-based user interface and interact with the user through events. Setting up an eventing model in the context of Web applications is challenging. It's amazing to see that client-side generated events are resolved with server-side code, and the output of this is visible as the same HTML page, only properly modified.

To make sense of this model it is important to understand the various stages in the page lifecycle and how the page object is instantiated and used by the HTTP run time.