

Because event handling requires a round-trip to the server, ASP.NET offers a smaller set of events in comparison to a totally client-based event system. Events that occur very frequently, such as mouse movement or drag-and-drop events, are not really supported by ASP.NET (although it is still possible to use client-side event handlers for these types of events in Javascript, or to use an AJAX-based API such as ASP.NET AJAX or AJAX.NET).

Postback

The first and foremost thing to learn about the ASP.NET event system is the concept of **postback**. In ASP.NET, postback is the process by which the browser posts information back to itself (i.e., posts information back to the server by requesting the same page). Postback in ASP.NET only occurs within Web Forms (i.e., within a `form` element with `runat=server`), and only server controls postback information to the server.

Figure 2.2 illustrates the basic and simplified postback interaction for a simple Web page.

More specifically, the simplified processing cycle for a Web Form is as follows.

1. The user requests an ASP.NET Web Form. In this example, the request uses the HTTP GET method.
2. On the server, the page is run, doing any preliminary processing such as compilation (if necessary), as well as calling other handlers as part of the page and application lifecycle (covered later in the chapter).
3. The `Page_Load` method of the page is called.
4. The rest of the page executes, which ultimately results in a generated HTML response, which is sent back to the browser. Part of this generated HTML is the view state (discussed shortly) information contained within a hidden HTML input element. As well, the `action` and `method` attributes of the `<form>` element are set so that the page will make a postback request to the same page when the user clicks the Enter button.
5. Browser displays the HTML response.
6. The user fills in the form, then causes the form to post back to itself (perhaps by clicking a button). If the user clicks a link that requests a different page, the following steps are not performed, because with the new page request, we would return back to step 1.
7. The page is posted back to the server, usually using the HTTP POST method. Any form values along with the view state information are sent along as HTTP form variables.
8. On the server, the page is run (no compilation is necessary because it will have already been compiled). The ASP.NET runtime recognizes that this page is being posted back due to the view state information. All user input is available for programmatic processing.

9. The `Page_Load` method is called.
10. Any invoked control event handlers are called. In this case, a click event handler for the button will be called.
11. The generated HTML is sent back to the browser.
12. Browser displays the HTML response.

These steps continue as long as the user continues to work on this page. Each cycle in which information is displayed and then posted back to the server is sometimes also called a *round trip*.

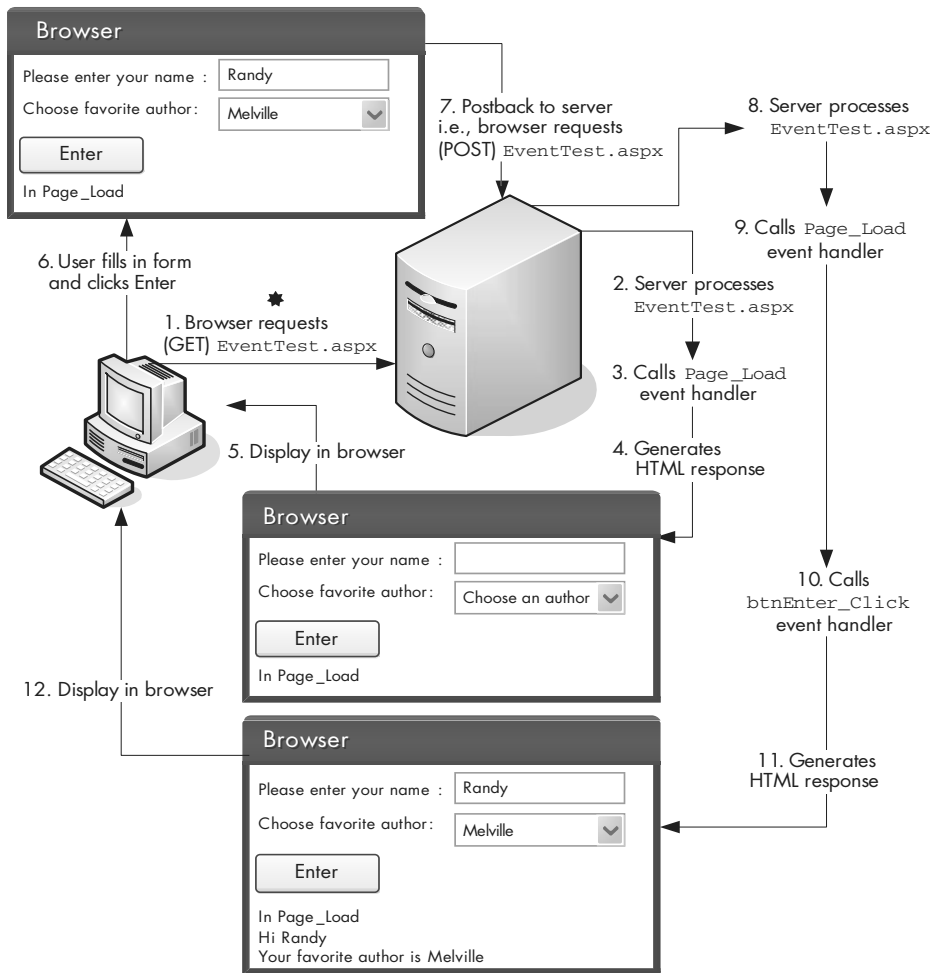


Figure 2.2 Postback flow

Page and Control Events

As can be seen from Figure 2.2, there are two different event types in ASP.NET: **page events** and **control events**. When a page request is sent to the server, a specific series of page events is always triggered in a specific order. Control events are associated with particular controls and are fired in certain circumstances. There are some standard events that all controls share; as well, most controls have unique events particular to that control. For instance, the `DropDownList` Web server control has an event that is triggered when the user selects a new list item.

CORE NOTE

We will see later in the chapter that the `Page` class ultimately is a subclass of the `Control` class. As a result, Web pages and server controls share many of the same events.



View State and Control State

View state is one of the most important features of ASP.NET. It is a specially encoded string that is used to retain page and form information between requests and is stored within a hidden HTML `<input>` element. All page elements not posted back via the standard HTTP `POST` mechanism are stored within this string. The view state thus provides the mechanism for preserving display state within Web Forms. Recall that HTTP is by nature stateless. This means that after the server responds to a request, it no longer preserves any data used for that request. Nonetheless, Web applications very frequently need to retain state on a page between requests. For instance, imagine a page that contains a user registration form. After the user clicks the Submit button, the code running on the server must check first to ensure that this user information doesn't already exist. If it does, it must return to the browser with the appropriate error message. Unless you want the user to curse and be frustrated with your form, you should also redisplay the form so that it contains the previously entered data.

This is an example of the need to maintain state in a Web application. Implementing this kind of state has typically involved cookies or form parameters and was often a real hassle in an older environment such as ASP classic. Although view state is not in fact used by ASP.NET to restore control values in a form, view state is ASP.NET's solution to the general problem of HTTP statelessness.

View state is generated after all the page code has executed but before the response is rendered. The value of each Web server control on the page is serialized into text as a number of Base64-encoded triplets, one of which contains a name-value pair. This view state string is then output to the browser as a hidden

`<input>` element named (as we have already seen in Chapter 1) `__VIEWSTATE`. When the form is posted back, ASP.NET receives the view state (because it was contained in a form element), deserializes this information, and restores the state of all the controls prior to the post. ASP.NET updates the state of the controls based on the data that has just been posted back, and then calls the usual page and control event handlers.

Because the details of encoding and decoding values from the view state are handled by the ASP.NET runtime, you can generally ignore the view state and simply revel in its benefits. However, sometimes, a developer may want to turn off the view state for a page. For instance, if a very large data set is being displayed, the view state also is quite large, which may significantly lengthen the time it takes the browser to download and render the page (this is especially an issue for mobile browsers). If a page is not going to post back to itself, you can improve page performance by disabling the view state for the page within the `Page` directive, as shown here.

```
<%@ Page ... EnableViewState="false" %>
```


The view state can also be programmatically manipulated within the code-behind class. It is a dictionary object that uses keys to locate and store objects. This use of view state, along with other issues involved in using it, is covered in Chapter 12.

Control state is a new feature in ASP.NET 2.0. It allows the developer to store custom control data between round trips, similar to the view state; unlike the view state, the control state can never be turned off by the developer. It is typically used when creating custom controls that will be used by other developers. Because control state is always available, it can safely be used even when view state is disabled.

Page Lifecycle

Page and control events occur in a certain order, which is called the *page lifecycle*. The precise order and number of events in this lifecycle are shown in Figure 2.3 and described in all their exhaustive glory in Table 2.1.

CORE NOTE



If you want to examine the setup of this pipeline yourself, you can do so by examining the private `ProcessRequestMain` method of `Page`. One way to do so is to use Lutz Roeder's .NET Reflector (available from <http://www.aisto.com/roeder/dotnet>) to examine the code in the `System.Web.UI` assembly.

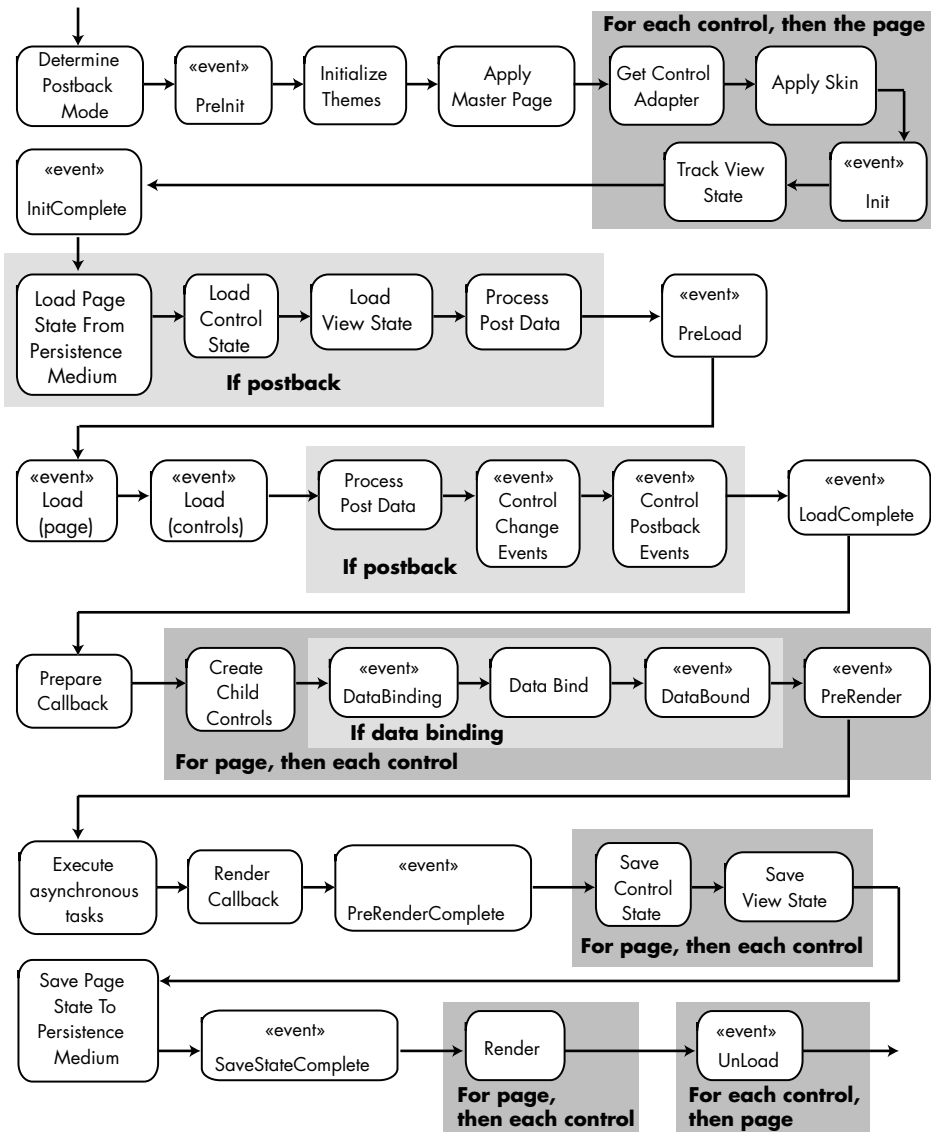


Figure 2.3 Page lifecycle

Table 2.1 Page Lifecycle

Event	Description
Determine postback mode	The <code>IsPostBack</code> property is set based on the presence of the view state in the page request. Can be handled by overriding the <code>DeterminePostBackMode</code> method.
<code>PreInit</code>	Occurs at the beginning of page initialization. You can use this event to set the master page or theme dynamically.
Initialize themes	The page theme is set and initialized. The private method <code>InitializeThemes</code> method of the <code>Page</code> class is called.
Apply master page	The page's master page is applied. The private method <code>ApplyMasterPage</code> method of the <code>Page</code> class is called.
Get control adapter (controls, then page)	Control adapters allow a developer to change the markup produced by server controls. This step gets any control adapter defined in the <code>App_Browsers</code> folder for the control.
Apply skin (controls, then page)	Applies any skin defined for the control.
<code>Init</code> (controls, then page)	You cannot access other server controls yet as there is no guarantee that it has been initialized yet. View state information cannot be used yet.
<code>InitComplete</code>	Raised after page initialization is complete. All declared controls on the page are initialized; they do not, however, contain data entered by the user. View state information can still not be used yet.
Page state is loaded from persistence medium	If this is a postback request, load any saved view state information from hidden <code><input></code> element. You can implement your own custom state mechanism by overriding the <code>LoadPageStateFromPersistenceMedium</code> method of the <code>Page</code> class.
Control state loaded (controls)	If this is a postback request, load any control state information. Control state exists in a postback request even if view state has been disabled.
View state loaded (page, then controls)	If this is a postback request, load any relevant view state information for this page or control.
Process post data	Loads post data back into any controls. Implemented by the private <code>ProcessPostData</code> method of the <code>Page</code> class.
<code>PreLoad</code>	Occurs after any view state is restored but before the <code>Load</code> event.
<code>Load</code> (page, then controls)	Used to perform most of the processing steps for the page and controls that are to occur for each page request.

Table 2.1 Page Lifecycle (*continued*)

Event	Description
Process post data	Loads post data back into any controls. This is attempted again in order to populate any dynamic controls added in any Load handlers.
Control change events	All control change events (such as <code>TextChanged</code> for a <code>TextBox</code>) are triggered.
Control postback events	All control postback events (such as <code>Click</code> for a <code>Button</code>) are triggered
<code>LoadComplete</code>	Occurs after all Load events. You can use this event for any task that requires all controls to be loaded.
Prepare callback	If an asynchronous call back is defined, it is raised.
Create child controls (page, then controls)	If the control contains any child controls, they are created.
Data binding	Data binding (and its events) occurs for any controls that have a <code>DataSourceID</code> property set. Data binding is covered in Chapter 8.
<code>PreRender</code> (page, then controls)	This event is the last chance to affect the page or controls before they are rendered.
Execute any asynchronous tasks	Starts the execution of any asynchronous tasks that have been defined for the page using the <code>PageAsyncTask</code> class and registered using the <code>RegisterAsyncTask</code> method.
Render callback	Renders any client-script callback.
<code>PreRenderComplete</code>	Indicates that all content for the page has been pre-rendered.
Page state is saved to persistence medium	Saves the page view state and control state to persistence medium. You can implement your own custom state mechanism by overriding the <code>SavePageStateFromPersistenceMedium</code> method of the <code>Page</code> class.
<code>SaveStateComplete</code>	Occurs after the page state has been saved.
Render (for page, then controls)	The page's <code>Render</code> method is called and then the <code>Render</code> method for each control and its children is called. Rendering finally outputs the HTML and other text to be sent to the client.
Unload (for controls, then page)	First each control and then the page triggers this event just before calling the <code>Dispose</code> method for the control or page. Can be used to perform any final cleanup of resources used by the controls or pages.

This is no doubt a very imposing list of steps. Fortunately, for most development tasks, you can remain blithely ignorant of most of these steps. We can get by initially instead simply by understanding the five general stages in the page lifecycle:

- **Page initialization**—During this stage, the page and its controls are initialized. The page determines if it is a new request or a postback request. The page event handlers `Page_PreInit` and `Page_Init` are called. As well, the `PreInit` and `Init` methods of any server controls are called. Any themes (covered in Chapter 6) are then applied.
- **Loading**—If the request is a postback, control properties are loaded with information recovered from special page state containers called the view state and the control state. The `Page_Load` method of the page, as well as the `Page_Load` method of its server controls are called.
- **Postback event handling**—If the request is a postback, any control postback event handlers are called.
- **Rendering**—During page rendering, the view state is saved to the page, and then each control along with the page renders themselves to the output response stream. The `PreRender` and then the `Render` method of the page and the controls are called. Finally, the result from the rendering is sent back to the client via the HTTP response.
- **Unloading**—Final cleanup and disposal of resources used by the page occurs. The `Unload` methods of the controls and the page are called.

Within each of these stages, the ASP.NET page raises events that you can handle in your code. For the vast majority of situations, you only need worry about one page event (`Page_Load`) and certain unique control events. Because page events always happen, you only need write the page event handler in your code using the appropriate naming convention (if `AutoEventWireup` is enabled). The naming convention is `Page_XXXX` where `XXXX` is the event name.

Control events, on the other hand, need to be explicitly wired—that is, you must explicitly bind the handler method to the event. This can be done declaratively in the control definition in the markup, or programmatically in the code-behind. Prior to Visual Studio 2005, the Designer in Visual Studio always used the programmatic approach. The current version of Visual Studio now uses the declarative approach.

To declaratively bind a handler to a control event, you use the appropriate `OnXXXX` attribute of the control, where `XXXX` is the event name. For instance, to bind the event handler method `btnSubmit_Click` to the `Click` event of a `Button` Web server control, you would use

```
<asp:Button id="btnSubmit" runat="server"
    OnClick="btnSubmit_Click" />
```

There must now be an event handler named `btnSubmit_Click` defined in the code for this page. As mentioned earlier, all event handlers in the .NET Framework

have a common method signature. The appropriate event handler would thus look like the following:

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    // Do something
}
```

CORE NOTE

You can get Visual Studio to create the event handler and add the appropriate attribute to the control by double-clicking the appropriate event entry within the Properties window while within Design view, as shown in Figure 2.4.

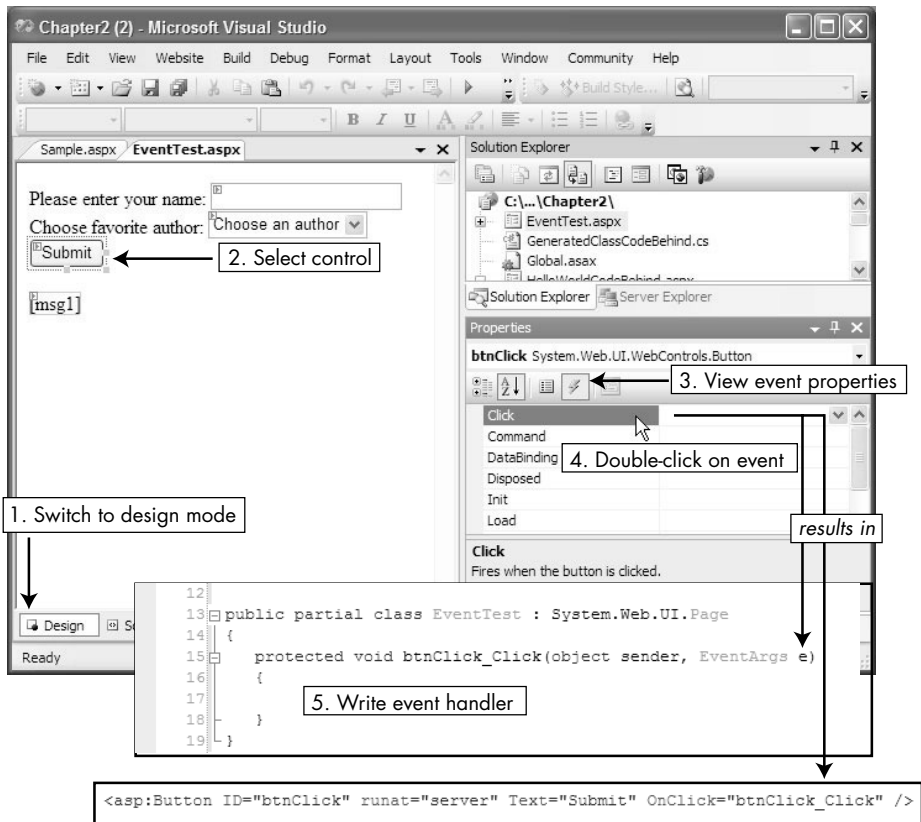


Figure 2.4 Adding an event handler in Visual Studio



CORE NOTE

Many developers (as well as Visual Studio itself) use the `objectname_eventname` naming convention for event handler methods. Using a consistent naming convention ultimately makes your Web application easier to understand and maintain.

You can also do the equivalent event binding using programming. Conceptually, this approach is quite a bit more complicated. It requires that you hook up a **delegate** to the appropriate event of the control. A delegate is an object that encapsulates a method (similar to a function pointer in C and C++) in a type-safe manner. The analogous code for programmatically wiring the button's `Click` event using a delegate would be

```
btnSubmit.Click += new EventHandler( this.btnSubmit_Click );
```

Because an event can be handled by multiple event handlers, the preceding code had to add the delegate to its list of delegates for that event (using the `+=` operator).

To better understand how to work with event handlers, Walkthrough 2.1 demonstrates how to implement the example shown in Figure 2.2.

Walkthrough 2.1 *Event-Handling Example*

1. In Visual Studio, create a new Web site called `Chapter2`.
2. Use the `Website → Add New Item` menu option (or right-click the project container in the Solution Explorer and choose the `Add New Item`).
3. Choose the `Web Form` template and change the filename to `Event-Test.aspx`.
4. Place the following markup within the `<form>` element. Notice the `OnClick` attribute for the `Button` control.

Please enter your name:

```
<asp:TextBox ID="name" runat="server" />  
<br />
```

Choose favorite author:

```
<asp:DropDownList ID="myList" runat="server">  
  <asp:ListItem>Choose an author</asp:ListItem>
```

```
<asp:ListItem>Atwood</asp:ListItem>
<asp:ListItem>Austin</asp:ListItem>
<asp:ListItem>Hawthorne</asp:ListItem>
<asp:ListItem>Melville</asp:ListItem>
</asp:DropDownList>
<br />
<asp:Button ID="btnEnter" Text="Enter" runat="server"
    OnClick="btnEnter_Click" />
<p><asp:Label ID="msg1" runat="server" /></p>
```

5. Switch to the code-behind file for EventTest by right-clicking EventTest.aspx and select the View Code menu command or by pressing **F7**.
6. Modify the Page_Load method as shown in the following.

```
protected void Page_Load(object sender, EventArgs e)
{
    msg1.Text = "In Page_Load<br/>";
}
```

7. Add the following method.

```
protected void btnEnter_Click(object sender, EventArgs e)
{
    if (myList.SelectedIndex > 0)
    {
        msg1.Text += "Hi " + name.Text + "<br/>";
        msg1.Text += "Your favorite author is ";
        msg1.Text += myList.SelectedItem;
    }
}
```

This method checks if the user has selected one of the authors from the DropDownList (i.e., it is not still on the Choose an Author list item, which has a SelectedIndex of 0), and if so, it displays a message containing the selected author in the Label control.

8. Save your changes.
9. Use the View in Browser menu command to test the page. Enter something in the textbox, select an item from the list, and select the Enter button. The result should look like that shown in Figure 2.5.

Notice that the Page_Load method is called the first time the page is requested, as well as after the button is clicked. The EnterBtn_Click method is called only after the button is clicked.

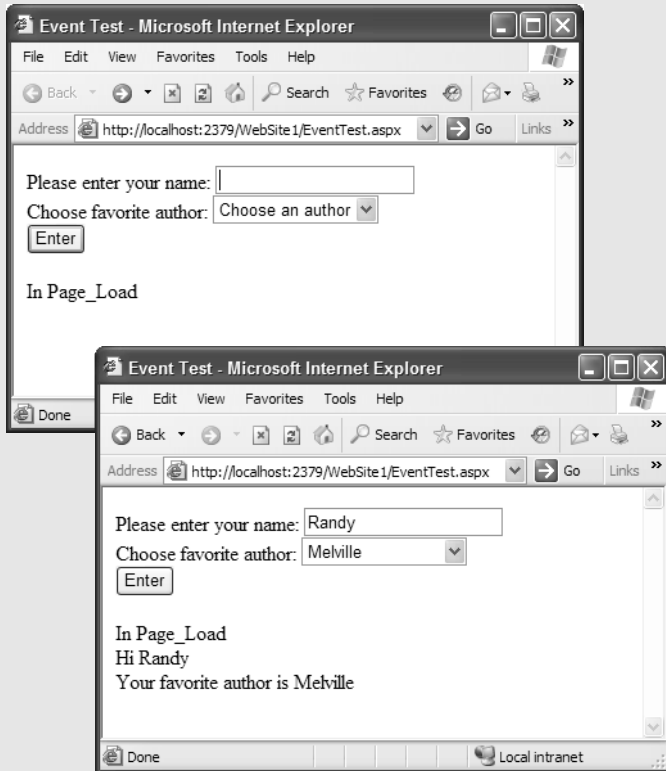


Figure 2.5 EventTest.aspx in Web browser

- Examine the generated HTML in the browser via the browser's View Source menu option. It looks something like that shown below. Notice that the action attribute on the form element is `EventTest.aspx`. This means that when you click the Submit button, you are making a POST request to the same page.

```
<!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><title>Event Test</title></head>
<body>
<form name="form1" method="post" action="EventTest.aspx"
  id="form1">
<div>
```

```

<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
  value="/wEPDwUJmZu4OTQyMTQyD2QWAgIDD2QWAgIHDw8WAh4EVGv4
dAU+SW4gUGFnZV9Mb2FkPGJyLz5IaSBSYw5keTxicl8+ww91ciBmYXZ
vcml0ZSBhdXRob3IgaXMgTWVsdmlsbGVkZGQ0UuquBlGDmIzZ2VVPg2a
Kmu+a81g==" />
</div>
<div>
  Please enter your name:
  <input name="name" type="text" value="Randy" id="name" />
  <br />
  Choose favorite author:
  <select name="myList" id="myList">
    <option value="Choose an author">
      Choose an author</option>
    <option value="Atwood">Atwood</option>
    <option value="Austin">Austin</option>
    <option value="Hawthorne">Hawthorne</option>
    <option selected="selected" value="Melville">
      Melville</option>
  </select>
  <br />
  <input type="submit" name="btnEnter" value="Enter"
    id="btnEnter" />
  <p><span id="msg1">In Page_Load<br/>
  Hi Randy<br/>Your favorite author is Melville</span></p>
</div>
<div>
  <input type="hidden" name="__EVENTVALIDATION"
    id="__EVENTVALIDATION"
    value="/wEWCALi/s/8DAL7uPQdAuuUwb8PAqmgr5IGArOL1q0PA
uny6fcEAoDL04ICape3wPgBDkQ091kDsBMvJvAv4m1IZpXZsi4=" />
</div></form>
</body>
</html>

```

Detecting Postback

There are times when you may want your page to behave differently the very first time it is requested. One typical example is that you want to read and display values from a database in a list only the first time the page is requested. In subsequent postbacks, the data is preserved by the view state so there is no need to re-read the database.

ASP.NET provides the developer with the ability to test if it is being requested for the first time via the `IsPostBack` property of the `Page` class (recall that this is the base class for all code-behind classes). This property is equal to `false` if the page is being requested for the first time. Thus, you can perform different processing the first time the page is requested using code similar to the following in the `Page_Load` method.