

ADVISOR**ADVISOR.ZONE****BOOK CHAPTER**

Beginning ASP.NET 1.0 with Visual Basic .NET: Event-Driven Programming and Postback

This book excerpt shows you how the ASP.NET programming model passes back information to users in response to generated events, letting you create efficient Web forms that work in a logical manner.

By Rob Birdwell, Matt Butler, Ollie Cornes, Chris Goode, Gary Johnson, John Kauffman, Ajoy Krishnamoorthy, Juan T. Llibre, Christopher Miller, Neil Raybould, David Sussman, and Chris Ullman

One of the most important features of ASP.NET is the way it allows you to use an event-driven programming model. This isn't a new thing as the Windows interface itself is an event-driven environment. What this means is that nothing happens under Windows unless it is in response to something (an event) that Windows has detected. Example of such events could be a user clicking on a desktop icon, pressing a key, or opening the Start menu. ASP.NET works in a similar manner.

In ASP.NET web forms rely on events to trigger the running of code housed in subroutines. As we've just mentioned, this is nothing new, it's been possible to run small sections of client-side code on the user's browser with HTML for a long time. What ASP.NET is doing differently is using postback, where additional information is sent back to the server for processing, rather than doing it entirely on the client-side browser. This postback means that information can be sent back to the server whenever an event is triggered by the user. This is a very powerful concept, as it means we can do things like remembering which option in a list was selected, or what information a user typed into a textbox, between page submissions. We've already seen these features in action in Chapter 3, but so far we've only hinted at what has been going on - now we're going to look at it in more detail.

ARTICLE INFO**MICROSOFT .NET ADVISOR**

Doc # 13357
1 December 2003
Length 26 pages

[Back to standard article layout](#)

This article is an excerpt from the book *Beginning ASP.NET 1.0 with Visual Basic .NET* from Wrox Press. For more information, visit <http://www.wrox.com>.



However, the event-driven nature of ASP.NET doesn't just stop there. It allows us to completely modularize our code into separate functions and subroutines, and only use those sections when a specific situation requiring them has arisen. We're going to see how ASP.NET can pass back information to the user, in response to particular events being generated, in a whole new way; and how this will enable us to create web forms that not only look different from the ones we have seen in previous chapters, but are faster, more efficient and work in a more logical way.

In this chapter we'll look at:

- What an event is
- What event-driven programming is
- ASP.NET 'generic' events
- HTML events
- ASP.NET server control events
- The ASP button control
- Server-side processing of events
- How the event-driven programming model changes the way in which we program

What is an Event?

Let's start by defining exactly what an event is in the real world. For example, a fictional employee, Joe Public, sits in Marketing doing his Sales job, talking to contacts. He can receive information in two ways, one is via the phone, to say things like "We need to restock ten cans of jumping beans". When the phone rings, this is an event that he can respond to. He'll answer the phone and take appropriate action, such as sending out an extra ten cans of beans. So an event is just something that happens that can be detected and reacted to.

Extending our example further, Joe Public can also receive news via e-mails, such as a new policy from the boss, meaning he has to put \$1 on all canned goods. This is a different event. Joe Public can also react to the event in any way he chooses. He can phone the stores and say "I'm afraid, I've got to charge you \$1 extra for the cans of jumping beans", or he could say, "Stuff it, boss! We charge too much for our beans already, I'm quitting". So the way that events are responded to is variable, and determined by the individual person, or object.

So, you can break down our event-driven environment into three chronological sections:

- An events occurs - for example the phone rings
- The event is detected by the system - Joe Public hears the phone
- The system reacts to the event - Joe Public answers the phone

In an operating system such as Windows events occur in a very similar manner. A user clicks the mouse. The mouse sends a message to the Operating System. The Operating System receives the message from the mouse, and generates an onclick event. This event is detected by the Operating System and appropriate action taken, such as displaying a menu or highlighting a piece of text.

What is Event-driven Programming?

Event-driven programming fundamentally changes the nature of the programming model. We leave behind the idea of sequential pages being processed on a server, and look instead at proper event-driven programming, where the server responds to events triggered by the user. This may all sound like gobbledygook at the moment, but bear with us. All we mean by this is that before event-driven programming your programs would execute from top to bottom, like this:

```
Line 1  
Line 2  
Line 3  
Line 4
```

Broadly speaking 'traditional' programming languages (those over ten years old) will start with the first line of your code, process it, move on to the second line, process that, and then move on to the third. Even when functions and subroutines are used it doesn't change the order of execution a great deal - as one procedure calls another, which calls another, and so on. So there is still an ordered sequence of execution.

The concept of event-driven programming changes all of this - with events, this sequential way of doing things is no longer appropriate. Consider the Windows operating system, again. Windows doesn't execute in a sequential fashion. If you click on a menu, you expect the menu to appear instantly, if you double click on an icon, then you expect the corresponding program will run immediately, you don't want to have to wait for Windows to finish whatever it is doing first. Indeed, you don't have to because the Windows interface is event-driven and able to react to an event to create a new thread and thereby perform things as soon as the user clicks on an icon or choose a menu selection. Under the covers, Windows is waiting for an event to occur. As soon as one does it will take the appropriate action to deal with that event.

Without events a lot of Windows programs couldn't run. Windows doesn't run Word, Notepad or the Calculator by itself, as part of a sequential series of executions, it only loads them when the user requests them. This same principle of programming is being used in ASP.NET. Typically the code in your web page (not HTML code, but client-side scripting) will be run by the browser, and will react to your mouse-click by itself. However with ASP.NET, the processing of events is shifted to the server.

ASP.NET Events

Everything in ASP.NET comes down to objects, and in particular the page object we mentioned in Chapter 2. Each web form you create is a page object in its own right. You can think of the entire web form as being like an executable program whose output is HTML. Every time a page is called the object goes through a series of stages - initializing, processing, and disposing of information. Because the page object performs these stages each time the page is called they happen every time a round trip to the server occurs. Each one of these stages can generate events, in the same way that clicking a mouse button can in Windows.

What happens in ASP.NET is that as you view your ASP.NET web form, a series of events

are generated on your web server. The main events that are generated are as follows:

- **Page_Init** - occurs when the page has been initialized. You can use the subroutine `Sub Page_Init()` associated with it to run code before .NET does certain automatic actions, such as displaying controls on the page. It works in the same manner as `Page_Load`, but occurs before it.
- **Page_Load** - occurs when the whole page is visible for the first time (that is, when the page has been read into memory and processed), but after some details about some of the server controls may have been initialized and displayed, by `Page_Init`.
- **Control Events** are dealt with - once the ASP.NET server controls have loaded they can respond to click events, the changing of a selection in a list or checkbox, or when a control is bound to a data source. They are only dealt with when the form is posted back to the server - we look at this later in the chapter.
- **Page_Unload** - occurs once the control events in a page have been dealt with, and is an ideal place to shut down database connections, and the like. It isn't compulsory to do this, as .NET will perform these operations for you, but it is good practice to tidy up your code, and the subroutine attached to this event is the best place to do it.

In addition, there is a set of events that the **Page** object supports which are not always fired, such as **PreRender**, which is generated just before the information is written to the browser and allows updates to be made to the page. There is also the **Error** event that occurs whenever an error is generated on the page and the **CommitTransaction** and **AbortTransaction** events which are fired whenever a transaction is about to be started or cancelled.

We've already said that when a page object is first instantiated it is completely blank. Every time you submit a page, you instantiate a new version of the page object, even if you continually submit the same page. This is because of the statelessness of the HTTP protocol. In Chapter 3 we talked about this fact, and how it means that the browser makes a connection to the server with a request for the page, the web server searches for the page and either returns a page or appropriate status message and then shuts down the connection. Everything you wish to do with the page must be performed within this series of stages, before the connection is closed.

What we're getting at here, is each Page object is entirely separate, each time you refresh details or submit information to the same web page, as far as the web server is concerned it is a brand new page and is treated as such. There is no persistence of information between pages, and this has to be handled using other means that we will see in later chapters. However ASP.NET is able to use certain aspects of the ASP.NET server controls to remember the state of information contained within the controls.

To keep things simple, we're going to leave looking at Control Events until a little later in the chapter, and first concentrate on getting a solid understanding of the three events that ASP.NET will always call in the lifetime of a page. As we mentioned before, they are `Page_Init`, `Page_Load` and `Page_Unload`.

As we've already mentioned, if you want code to execute on your page before anything else occurs, then you need to associate it with the `Page_Init` event:

```
<script language="vb" runat="server">
Sub Page_Init()
... code here ...
```

```
End Sub
</script>
```

Alternatively, if you want to perform actions, after the page has been loaded, but before the user has had a chance to submit any information, then the Page_Load() event should be used:

```
<script language="vb" runat="server">
Sub Page_Load()
... code here ...
End Sub
</script>
```

We've used the Page_Load() subroutine for all of our examples so far in this book to make sure that the code that we are writing gets run. Although we could just as easily have placed it in the Page_Init() subroutine.

To clarify this concept, once again, Page_Load is an event that is triggered every time your web form is loaded, and it calls the Sub Page_Load() subroutine. So what's really happening is when the page is loaded, the event is triggered, ASP.NET then detects the event and runs the contents of the subroutine associated with it.

If we look at the following code, we can see:

```
<script runat="server" language="vb">
Sub Page_Load()
  Message1.text = "This is text that is only displayed once the Page_Load()
event has been triggered."
End Sub
</script>
<html>
<head>
<title>Event Driven Page</title>
</head>
<body>
<asp:label id="Message1" runat="server" />
</body>
</html>
```

This ASP.NET code is only triggered in response to the Page_Load() event occurring. What happens if we take our code and put it into a different subroutine? The answer is that the code will now only run when another event is triggered and calls the new subroutine, or if the subroutine is called from elsewhere in our ASP.NET code.

For example, you could move the code that displays the text into a separate subroutine, called Subroutine_1() instead.

```
<script runat="server" language="vb">
Sub SubRoutine_1()
  Message1.text = "This is text that is only displayed once the Page_Load()
event has been triggered."
End Sub
</script>
...
```

Now, if you went back to the example, and changed the code in this way it wouldn't work properly anymore, as this code would no longer be run. This is because the subroutine is not associated with any event and isn't called from anywhere else. Therefore the code it

contains will never be run, and so nothing is displayed:

So, we've looked at `Page_Init` and `Page_Load`, but that leaves us still with the `Page_Unload` event. Like the other two, this will occur every time a page is sent, but the main difference is this event occurs after all of the ASP.NET code in the page has been executed and has completed its tasks. If there are specific actions you wish to take place before the user moves to another page, but after the page has been loaded, then this is the place to do it. The `Page_Unload` event might occur after all other tasks have been performed, but remember as these events are performed as a single discrete action on the server, the `Page_Unload` will already have been performed by the time the user gets to see the page. In other words it is performed as soon as the page has finished being displayed and not when you might expect, when the user moves to another web page.

The only function of this final event, is to give you the chance to perform housekeeping actions, such as closing down objects, before the page is sent back to the user. As `Page_Unload` is performed after the rest of the ASP.NET has been completed, it is not possible to use controls or ASP.NET statements to display code in the associated `Page_Unload()` subroutine either. Typically not much else is done other than the previously mentioned housekeeping and tidying tasks in `Page_Unload()` so we won't be discussing it in further detail in this chapter.

To summarize - we've already said that it's not just `Page_Load` that ASP.NET can respond to. There are three main events that always occur, regardless of the Custom controls triggered:

- `Page_Init()`
- `Page_Load()`
- `Page_Unload()`

Events in HTML

Before we launch into more detail of how these events work in ASP.NET, it helps to look at the way HTML deals with these events.

For those used to writing HTML pages, to understand the advantages of using ASP.NET event handlers, you first need to see how HTML does it. In a form, HTML will usually use the `<input>` tag with the `type` attribute being set to `Submit` to create a submit button, to send the form's data. However, if you wanted to create a button but didn't wish to send the contents of the form to the server for processing, you'd need to use the HTML `<input>` tag again, but this time setting the `type` attribute to `Button`. This button wouldn't do anything at all at the moment; to get it to react to an event you'd need to add some more code to the `<input>` tag. Typically HTML will use a combination of a form and a small piece of client-side script to achieve this. The client-side script is placed in a special event attribute of an HTML tag and the tag is placed inside the form as usual.

This is best demonstrated with an example. We're going create a page with a button that displays a message when the user presses it. Don't worry, you don't have to know any client-side script for this example. But you must be using a modern browser, IE4 and upward, Netscape 6 and upward, or Opera 5 and upward.

Try It Out - Handling an Event in HTML

1. Open up your web page editor and type in the following:

```

<html>
<head>
<title>HTML Event Example</title>
</head>
<body>
<form>
<input type="button" value="Click Me" onclick=
                                "alert('You have raised an
event! ')" ">
</form>
</body>
</html>

```

2. Save it as htmlevent.htm (note the HTM suffix, this isn't an ASP.NET web form).

3. View it in your browser and click on the button:



How It Works

This is an example of a built-in HTML event, known as an intrinsic event. These events are supported by nearly every HTML tag you care to mention, from `<body>` to `` and of course, as in this example `<input>`. Below is a list of some of the events that it is possible for your browser to react to:

- `onmouseup` - occurs when a mouse button is released while clicking over an element
- `onmousedown` - occurs when a mouse button is pressed and held while clicking over an element
- `onmouseover` - occurs when a mouse is moved over an element
- `onmousemove` - occurs when a mouse moves over an element
- `onclick` - occurs when a mouse is clicked over an element
- `ondblclick` - occurs when a mouse is double-clicked while hovering over an element
- `onkeyup` - occurs when a key is released over an element
- `onkeypress` - occurs when a key is pressed and released over an element
- `onkeydown` - occurs when a key is pressed and held down while over an element

In our extremely simplistic program we have just created a form with a single button:

```
<form>
<input type="button" value="Click Me" onclick=
                                "alert('You have raised an
event!')">
</form>
```

The <input> tag is set to have a type of button, so when the button is clicked it isn't automatically submitted. Next, we use onclick. Inside this event attribute there is a little bit of JavaScript code that brings up a dialog box with a message (alert) in it. The code inside the onclick attribute is known as an event handler. An event handler is simply a section of code that manages a suitable response to the event, just like the Sub Page_Load() subroutine does.

We could change the code so that our button reacts to a different event; here we've changed it to the onmouseover event:

```
<form>
<input type="button" value="Click Me" onmouseover=
                                "alert('You have generated an
event!')">
</form>
```

If you go back and change your code, it will now react if you move your mouse over the button, rather than having to click on it. The event handling code is just the same. You could also amend your code so that your button can respond to several events:

```
<form>
<input type="button" value="Click Me" onclick=
                                "alert('You have generated a click
event!')"
                                onmouseover=
                                "alert('You have generated an event by
hovering!')">
</form>
```

The reason we're talking about how HTML does this, is because the ASP.NET way is very similar, and it provides a good introduction. We'll go on to look at how ASP.NET handles these kinds of events now.

Server Control Events in ASP.NET

ASP.NET, like HTML, allows you to use events, such as onclick and onload within your server-side controls, and then write ASP.NET code that 'reacts' to them in any .NET language. One of the main advantages is that you don't have to rely on a modern browser to get these events to work. Because, while the browser used to deal with the events in HTML, in ASP.NET the web server can deal with them and sends pure HTML back to the browser.

You can add events to ASP.NET controls, in exactly the same way that you would add them in HTML to client-side controls. In ASP.NET an onclick event for a button control might look like this:

```
<asp:button id="button1" text="Click me" onclick=
                                "ClickEventHandler"
runat="server" />
```

Here we supply the name of a subroutine in the onclick event attribute. This subroutine name will be mapped directly onto a subroutine with the same name found in your script code:

```
<script language="vb" runat="server">
Sub ClickEventHandler(Sender As Object, E As EventArgs)
... ASP.NET code here...
End Sub
</script>
```

If you haven't provided a subroutine with the same name, then nothing will happen when the event is triggered. The code will only be run when the name of the subroutine in the server control attribute matches the name of a subroutine in your code. You can call the subroutine whatever name you want, as long as you are consistent in using the same name in the server control and within the <script> tags.

The arguments we provide for this event handler are used to let the calling event pass information to the handler. The first of these - Sender - provides a reference to the object that raised the event, while the second - E - is an event class that captures information regarding the state of the event being handled and passes an object that's specific to that event.

ASP.NET server controls have a reduced set (compared to HTML) of events that can be added to controls as extra attributes. These are as follows (note that these event names aren't case sensitive):

Event name	Description
onload	Occurs when the control has loaded into the window or frame
onunload	Occurs when a control has been removed from a window or frame
onclick	Occurs when a mouse button (or similar) is clicked when hovering over the <asp: button> control
oninit	Occurs when the web page is first initialized
onprerender	Occurs just before the control is rendered

In addition to this, we also have the following events that can't be handled by the user in an event handler, but that occur in ASP.NET and cause actions within ASP.NET:

selectindexchanged checkchanged	These two occur when the contents of a control have been altered, such as a checkbox being clicked, or a list item being selected. These only apply to the appropriate controls, such as list items and checkboxes.
------------------------------------	---

The large difference between HTML controls and ASP.NET server controls is the way they're dealt with. With HTML form controls, when the event is raised, the browser deals with it. It will sit around waiting for a user to click on the page or move their mouse. But, with server-side controls, the event is raised by the browser, but instead of being dealt with by it, the client raises a trigger telling the server to handle the event. In fact the server only generates a single form postback event, which occurs when you click on the

ASP.NET button control. Anything else that might have been done in the page is packaged up within this single event. So if you've checked a radio button and chosen an item from a listbox, this information is recorded when you send the form.

It doesn't matter what kind of event was raised, the client will always return this single postback event to the server. However, some events are impossible to deal with on the server, such as key presses or mouseovers, so there are no equivalent ASP.NET events for these. They will not be passed onto the server and have to be handled by the client. The reasons for this are simple.

With HTML, your browser will wait for you to click on a control. It can sit around all day waiting for you, if you keep the same page open. This does not happen in ASP.NET. As mentioned previously, HTTP only allows you to open a connection, get a page, and close that connection. Similarly, the server cannot keep a copy of page object open on the server, on the off chance that you might submit details to the page at a later point. No, the whole set of events, corresponding to the changes you have made to the page, have to be dealt with in a single discrete action.

The page object is initialized; the page loaded, then the control events are all bundled up into a single postback event and dealt with, then the page is unloaded. This happens in a matter of seconds or milliseconds. And explains why only a select few events can be dealt with by your server-side code.

The advantage of passing events to the server is that it adds better structure to your code, because you can separate out your event handling code in your ASP.NET applications entirely. There are some disadvantages too - wherever possible it's a good idea to do client processing, to offload work from central CPUs to improve performance.

Now let's look at an example of how events work with the ASP.NET server controls. Before we do this we need to introduce another new server control.

The ASP.NET Button Server Control

There is one ASP.NET server control in particular, that we didn't look at in the chapter on server controls, that can be used to react to events. It performs the equivalent functionality of setting the `<input>` tag to type `Button` in HTML. This is the button control, and the reason we haven't mentioned it before is that you need to understand events to be able to make any decent use of it.

The button control has the following format:

```
<asp:button id="id_name" event="event_handler_name" runat="server"/>
```

To get it to work you need to specify an event. The ASP.NET button control supports only the following events:

- oninit
- onprerender
- onload
- onclick
- onunload

If we're going to get our page to respond dynamically, we need to use the only event

capable of doing this, the onclick one. Let's take a look at an example now.

This example contains three ASP.NET HTML server controls. The first is a checkbox to register a user's choice, the second is a button that will raise the event and the third a label control. As we've mentioned earlier in the book, you can't render dialog boxes directly in ASP.NET - to display information dynamically you use the label control.

The example asks the user for some information about whether they wish to receive further information from a company. It will display a message appropriate to the user's response depending on whether they checked the checkbox or not.

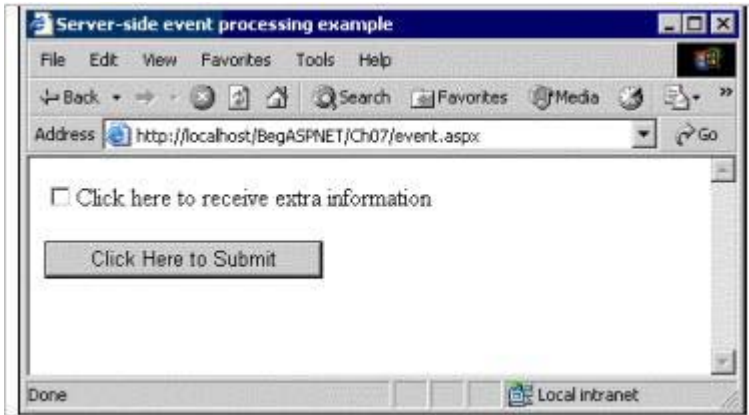
Try It Out - Using the ASP.NET Button Server Control

1. Open up your text editor and type in the following code:

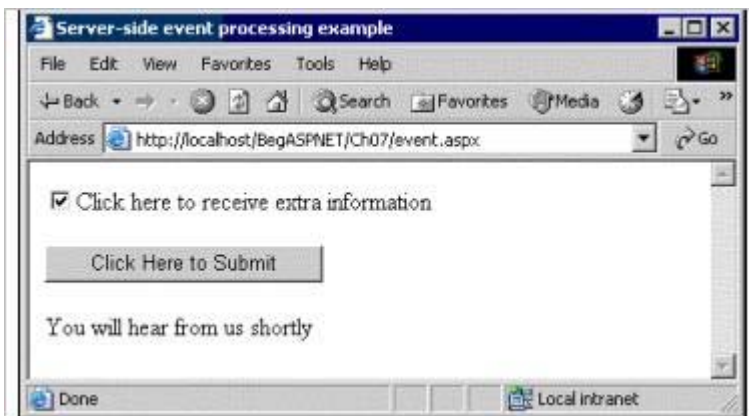
```
<script language="vb" runat="server">
  Sub ClickHandler(Sender As Object, E As EventArgs)
    If ExtraInfoBox Is "On" then
      Message.Text = "<br /><br />You will hear from us shortly"
    Else
      Message.Text = "<br /><br />You will not receive any further" & _
                    " information from
us"
    End If
  End Sub
</script>
<html>
  <head>
    <title>Server-side event processing example</title>
  </head>
  <body>
    <form runat="server">
      <asp:CheckBox id="ExtraInfoBox" Text=
        "Click here to receive extra information" Runat="server" />
      <br /><br />
      <asp:Button id="Button1" Text="Click Here to Submit"
        onclick="ClickHandler"
runat="server"/>
      <asp:Label id="Message" runat="server"/>
    </form>
  </body>
</html>
```

2. Save this as event.aspx.

3. Run the code in your browser.



4. Check the box and click on the "Click Here to Submit" button. An appropriate message is displayed:



How It Works

We're simply taking the event raised when a user clicks on the button and running a short passage of ASP.NET code. Our subroutine ClickHandler is the event handler specified in the onclick event attribute. This event handler examines the checkbox in the example and displays a different message in the label control depending on whether or not the checkbox has been selected. This is our event handler/subroutine code:

```
<script language="vb" runat="server">
Sub ClickHandler(Sender As Object, E As EventArgs)
  If ExtraInfoBox Is "On" then
    Message.Text = "<br /><br />You will hear from us shortly"
  Else
    Message.Text = "<br /><br />You will not receive any further " & _
                                     " information from us"
  End If
End Sub
</script>
```

This code will display an appropriate message depending on whether the contents of the ExtraInfoBox checkbox control has been activated or not. The crucial difference from all of the other examples we have looked at so far is that, instead of the automatically generated Page_Load event, we are using our own customized event. This means our code

is only run when the click event happens. If nobody clicks on the button then nothing happens.

Another important point to notice about the click event handler code placed at the top of the page is that it requires two parameters. Up to now our Page_Init and Page_Load events have required no parameters, because we didn't need to pass information between them. You can use parameters here in just the same way as you would with normal subroutines, but you should be aware that there are special reasons why events require parameters.

Each event that is generated also generates information about itself, such as which control it was generated by. To be able to use this information within your subroutine you have to pass in two parameters. The first of these has type Object and is named Sender, the second is of type EventArgs and is named E. Both of these parameters are passed to any server-side event that can be raised by ASP.NET. They're standard parameters, which all event procedures are passed. If you don't specify them when using server control events, then you will cause an error. They're important because they pass information to ASP.NET about which HTML element generated an event, what type of event was generated and other informative details about the event, and allow you to use and manipulate these values within your ASP.NET code.

When an event is generated it is a corresponding object that is passed as the Sender parameter. This contains information about which object is sending the information about the event. The second parameter E contains a set of related information - such as data about which class the event is a member of. It's still too early to deal completely with what is happening here, as you won't understand it without a thorough understanding of objects (which we cover in Chapter 8), but it should be a little clearer when we come to talk about it again in the next couple of chapters. All you need to know is that parameters pass details to a subroutine about the specific event that has been generated. Every subroutine/event handler works in the same way, whether raised by ASP.NET or by the programmer, and must be passed these 'generic' parameters.

But this isn't the only information that is passed. The data contained within the form is also passed by ASP.NET, at the same time when an event subroutine is called with the ASP button control:

```
asp:Button id="Button1" Text="Click Here to Submit" onclick="ClickHandler()"
runat="server"/>
```

In this model, when the user submits the form. The form data is sent, not using the <Input type="submit">, but by the ASP Button server control. This control generates an event, which calls our subroutine as an event, and by passing two parameters (Sender As Object, E As EventArgs) you are able to use all of the form data directly on your page. This is postback.

Now we're going to move on and consider the process of postback in more detail, the process whereby all this information is parceled up and sent back to the server.

Event-Driven Programming and Postback

A big issue throughout this book so far has been the postback architecture. You may not

have noticed this, as we've not focused on it or explained it until now. Postback is the process by which the browser posts information back to the server telling the server to handle the event, the server does so and sends the resulting HTML back to the browser again. Postback only occurs with web forms, and it is only server controls that post back information to the server.

ASP.NET doesn't look after the processing of all events, as it is still necessary to handle some events (such as onmouseover) on the client-side, as the server couldn't possibly react to them in time.

But the great advantage of processing some events on the server is that you don't have to rely on a particular browser to recognize your client-side script, and you can send back extra information, for example the state of a particular control, to help influence your decision. This is something fundamentally different to what happens in HTML. Let's just demonstrate this now with an example:

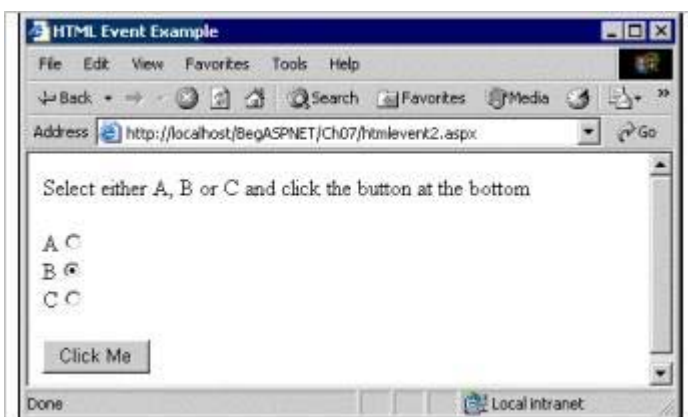
Try It Out - Demonstrating HTML's Lack of Postback

1. Open your web page editor and type in the following:

```
<html>
<head>
  <title>HTML Event Example</title>
</head>
<body>
<form method="get">
Select either A, B or C and click the button at the bottom<br/>
<br/>
A<input type="radio" value="a" name="test"><br />
B<input type="radio" value="b" name="test"><br />
C<input type="radio" value="c" name="test"><br /><br />
<input type="submit" value="Click Me" onclick="
                                alert('You have generated an
event!') ">
</form>
</body>
</html>
```

2. Save this as htmlevent2.htm, making sure you remember the HTM suffix again.

3. View this in your browser and click on a choice:



4. Now click on the Click Me button - after receiving a dialog saying that you have

generated an event you will see the following:



Your selection has disappeared!

How It Works

There isn't anything unusual happening in our HTML form. Though you should note that, as we've omitted the action attribute of the form, it will post the data to back to itself:

```
<form method="get">
Select either A, B or C and click the button at the bottom<br/>
<br/>
A <input type="radio" value="a" name="test"><br />
B <input type="radio" value="b" name="test"><br />
C <input type="radio" value="c" name="test"><br /><br />

<input type="submit" value="Click Me" onclick="alert('You have generated an
event!')">
</form>
```

As you can see from the URL, a querystring has been appended to submit the information:

<http://localhost/5040/ch07/htmlEvent2.htm?test=b>

However, when we go back to the page, our selection has disappeared. This is the normal behavior for HTML. However ASP.NET can improve upon this, as we shall see. Let's adapt our previous example to use as ASP.NET server control instead:

Try It Out - Using Postback

1. Type the following into your page editor:

```
<html>
<head>
<title>Postback Event Example</title>
</head>
<body>
<form runat="server">
Select either A, B or C and click the button at the bottom<br/>
<br/>
<asp:radiobuttonlist id="test" runat="server">
<asp:listitem id="option1" value="a" runat="server" />
<asp:listitem id="option2" value="b" runat="server" />
<asp:listitem id="option3" value="c" runat="server" />
```

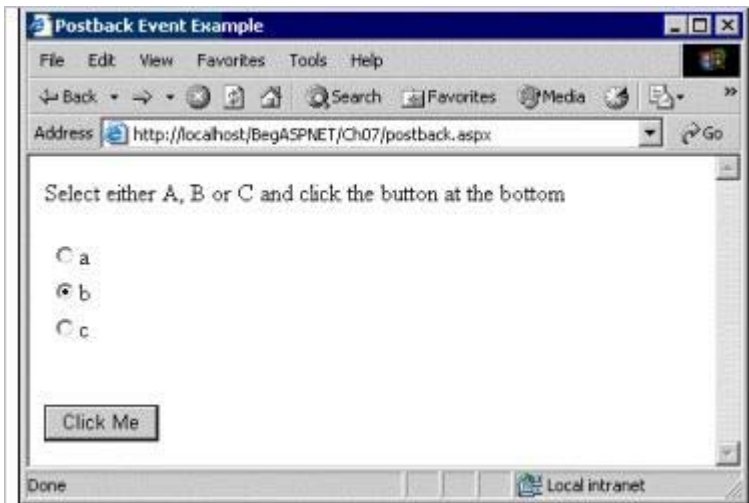
```

</asp:radiobuttonlist>
<br /><br />
<input type="submit" value="Click Me" onclick=
                "alert('You have generated an
event!') ">
</form>
</body>
</html>

```

2. Save this as postback.aspx.

3. View this in your browser and select a choice and click on the button:



This time your choice is remembered.

How It Works

This is an example of the concept of postback in action. Click on View Source in your browser to view your code - because of differences between machines and browsers your code won't look exactly like this, but it will be pretty similar:

```

<html>
<head>
  <title>Postback Event Example</title>
</head>
<body>
<form name="ctrl0" method="post" action="postback.aspx" id="ctrl0">
<input type="hidden" name="__VIEWSTATE" value="dDw0MDk5MTgxNTU7Oz4=" />

```

Select either a, b or c and click the button at the bottom


```

<br/>
<table id="test" border="0">
  <tr>
    <td>
      <span value="a">
        <input id="test_0" type="radio" name="test" value="a" />
        <label for="test_0">a</label>
      </span>
    </td>
  </tr><tr>
    <td>

```

```

<span value="b">
  <input id="test_1" type="radio" name="test" value="b"
                                     checked="checked" />
  <label for="test_1">b</label>
</span>
</td>
</tr><tr>
<td>
  <span value="c">
    <input id="test_2" type="radio" name="test" value="c" />
    <label for="test_2">c</label>
  </span>
</td>
</tr>
</table>
<br /><br />
<input type="submit" value="Click Me" onclick=
                                     "alert('You have generated an
event!') ">
</form>
</body>
</html>

```

The `<asp:radiobuttonlist>` control has been turned into its equivalent `<input>` controls and has been formatted as part of a table. Also the event attribute is no longer specified in the input tag. However, the largest change is the addition of the hidden VIEWSTATE control:

```
<input type="hidden" name="__VIEWSTATE" value="dDw0MDk5MTg5NTU7Oz4=" />
```

Information about the state of the form is sent back in an associated hidden control, called `_VIEWSTATE`. This information is generated by ASP.NET. In fact the information contained within `_VIEWSTATE` control in the value attribute is actually an encrypted version of the old state of the form. ASP.NET is able to look at the old version and compare it to the current version. From this it knows how to persist the state of ASP.NET server controls between page submissions. If there are differences, such as a different radio button being selected, then internal events are generated by ASP.NET in response to this and it deals with it appropriately, to create a 'current' version of the form. This `_VIEWSTATE` control is crucial to ASP.NET being able to remember the state of controls between page submissions, without actually maintaining a page object or HTTP connection throughout.

ASP.NET has used this postback information about the state of the page, to create a new version of it, with the state of the corresponding controls being remembered. This can be used by the browser in future iterations of the page, but is never displayed again. So postback works by using only HTML tags and controls, which contain information that can be interpreted by ASP.NET.

The IsPostBack Test

Before we leave postback though, let's go back to something we introduced in Chapter 3, the `IsPostBack` test. This was used to see whether a user had returned a form, together with data, or whether it was the first time the form had been displayed. Now you can see that our `IsPostBack` test is just checking to see whether any postback information has been generated by an event. If it hasn't then this test generates false, otherwise it generates true. This test has to be used in conjunction with a particular page, so the

following syntax:

```
If Page.IsPostBack then
```

becomes necessary. This is only used with ASP.NET forms but it makes the process of forms posting back to themselves much simpler.

Changing the Way we Program on the Web

Throughout this chapter, we've been referring to Sub Page_Load() as just another subroutine, and indeed this is all it is. But it is slightly different in that ASP.NET automatically calls this subroutine for us. As virtually all ASP.NET code is placed in subroutines or functions, we rely on the event associated with this code to kick-start our applications.

In order for ASP.NET to handle other events, you'll need to create your own event handlers, using your own subroutines. These subroutines will be called by event attributes of the HTML server controls, such as onclick. When you call these subroutines you are able to pass information about the event by which they were called using the event parameters we discussed earlier.

In fact all programming in VB .NET is event-driven and object-oriented, it's just a case of whether:

- ASP.NET raises the event, such as Page_Load() or Page_Init()
- The user of the web page raises it by clicking on a button and raising an associated handler

Whichever is the case, they both operate in the same way, causing the program to break from whatever it was doing and move to the requested function or subroutine.

We're going to look at an example that draws all of the things we've looked at in the chapter together and utilizes this new way of doing things. It contains two textboxes to take information from the user, and will perform a calculation depending on whether the user selects an add, subtract, divide, or multiply button.

Try It Out - Event-driven Example

1. Open up your web page editor and type in the following:

```
<script runat="server" language="vb">

Sub Add(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) + Cdbl(tbxInput2.text)
End Sub

Sub Subtract(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) - Cdbl(tbxInput2.text)
End Sub

Sub Factor(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) * Cdbl(tbxInput2.text)
```

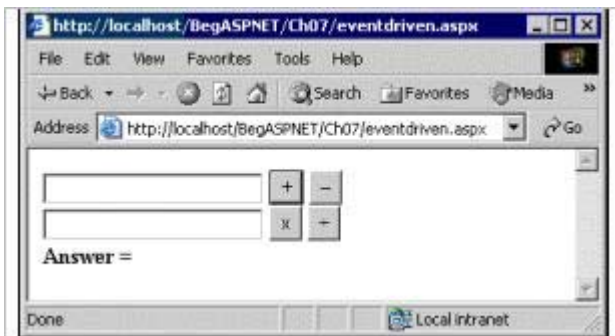
```
End Sub
```

```
Sub Ratio(sender as object, e As EventArgs)
    lblAnswer.Text = CDbl(tbxInput1.text) / CDbl(tbxInput2.text)
End Sub
```

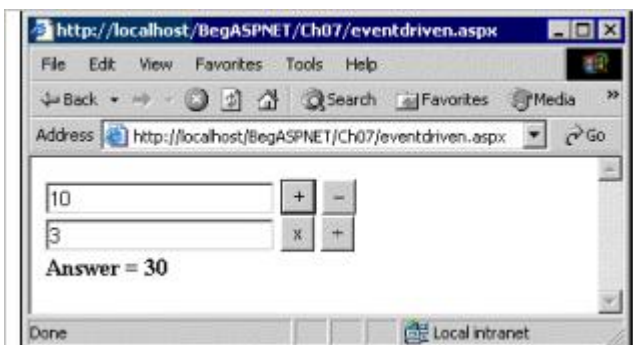
```
</script>
<html>
  <form runat="server">
    <asp:textbox id="tbxInput1" runat="server" />
    <asp:button id="btnAdd" runat="server" text=" + " Onclick="Add" />
    <asp:button id="btnSubtract" runat="server" text=" - "
Onclick="Subtract" />
    <br/>
    <asp:textbox id="tbxInput2" runat="server" />
    <asp:button id="btnFactor" runat="server" text=" x " Onclick="Factor" />
    <asp:button id="btnRatio" runat="server" text=" ÷ " Onclick="Ratio" />
    <br/>
    <b>Answer = <asp:Label id="lblAnswer" runat="server" /></b>
  </form>
</html>
```

2. Save this as eventdriven.aspx.

3. View it in your browser:



4. Enter two numbers into the text fields and select the multiply button to perform a calculation:



5. You will see the answer displayed for your particular operation. Go back alter the numbers and click a different button, such as subtract:



Only the operation corresponding to the button clicked was performed.

Please note that if you don't enter a number into one or other of the boxes before selecting an operator, you will cause an error. We'll leave this for now, but look at error handling in Chapter 17.

How It Works

This example works by using event-driven programming, as we outlined previously. We have created four separate subroutines, Add, Subtract, Factor and Ratio.

```
<script runat="server" language="vb">

Sub Add(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) + Cdbl(tbxInput2.text)
End Sub

Sub Subtract(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) - Cdbl(tbxInput2.text)

End Sub

Sub Factor(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) * Cdbl(tbxInput2.text)
End Sub

Sub Ratio(sender as object, e As EventArgs)
    lblAnswer.Text = Cdbl(tbxInput1.text) / Cdbl(tbxInput2.text)
End Sub

</script>
```

These subroutines are called by the four ASP.NET button controls we have created:

```
...
<asp:button id="btnAdd" runat="server" text=" + " Onclick="Add" />
<asp:button id="btnSubtract" runat="server" text=" - " Onclick="Subtract" />
...
<asp:button id="btnFactor" runat="server" text=" x " Onclick="Factor" />
<asp:button id="btnRatio" runat="server" text=" ÷ " Onclick="Ratio" />
```

Each button has a symbol corresponding to its equivalent mathematical operation, and it

calls that relevant subroutine. All four subroutines work in an identical way. Let's look at Sub Add more closely:

```
Sub Add(sender as object, e As EventArgs)
    lblAnswer.Text = CDbl(tbxInput1.text) + CDbl(tbxInput2.text)
End Sub
```

This is called by the onclick attribute of btnAdd, and is passed the two generic parameters. These parameters allow it to access the two textboxes, which contain the numbers the user entered:

```
<asp:textbox id="tbxInput1" runat="server" />
...
<asp:textbox id="tbxInput2" runat="server" />
...
```

The first textbox is called tbxInput1, so we can reference it as tbxInput1, and reference its contents by referring to its text attribute. The second textbox is tbxInput2 and can be referenced by referring to its text attribute also.

As the text attribute returns its information as a string, we have to use CDbl, which converts the data type from a string to a double. Then we can perform a mathematical operation on our two pieces of data, effectively saying:

```
tbxInput1 + tbxInput2
```

Then we store our data in the <asp:label> control, lblAnswer. Once again, we can access its text attribute, but instead of getting the information, here we are setting it. So we are effectively saying:

```
lblAnswer.text = tbxInput1 + tbxInput2
```

This information is displayed on our screen. The other three subroutines work in the same way, and will only be executed in the event that the particular button associated with them is pressed.

In this way our ASP.NET can react in a different way to each of the buttons on the page, while making use of the same data. We'll be coming across events a lot more throughout the book and expanding on our understanding of them in later chapters, but for now this gives you the necessary foundations on which to base your understanding of them.

Summary

We've kept this chapter short, as we want to focus on the main points that are central to the event-driven way in which ASP.NET works. It generates a set of events, which occur whenever a web page is initialized, loaded and unloaded on the server. In the duration of this sequence, ASP.NET can also react to other events that may have been generated by the server controls. The code needed to react to these events is placed in event handlers. By passing two generic parameters to these event handlers, you are able to use all of the form data.

Event-driven programming is something utilized in ASP.NET and something that fundamentally alters the way in which pages/web forms are created. You might create a web form with several sections of code, some of which are never executed, while others will be executed constantly. When web forms are created you now need to think, which

events will occur, and how do I make my pages react to them?

In the next chapter we move on the concept of objects and see how everything in ASP.NET, including events, are in fact just objects. This will fundamentally alter, yet again, the way in which we create our ASP.NET web forms.

SIDEBAR

Exercises

1. Explain why event-driven programming is such a good way of programming for the Web.

2. Run the following HTML code in your browser (remember to save the page with an .htm suffix). Now translate it into a set of ASP.NET server controls so that the information entered into the form is retained when the Submit button is clicked. Add a function to the button to confirm that the details were received:

```
<%@ Page Language="VB" runat="server" %>

<script runat="server">
  Sub ClickHandler(Sender As Object, E As EventArgs)
    message.text = "Details received."
    questions.visible = False
  End Sub
</script>

<html>
<head>
  <title>ASP.NET</title>
</head>
<body>

  <asp:label id=message runat=server />
  <form id=questions runat="server">

    <h4>Please enter your name:</h4>
    <asp:textbox id=name runat="server" /><br /><br />

    <h4>What would you like for breakfast?</h4>
    <asp:checkboxlist id=food runat="server">
      <asp:listitem value="Cereal"/>
      <asp:listitem value="Eggs"/>
      <asp:listitem value="Pancakes"/>
    </asp:checkboxlist>

    <h4>Feed me:<h4>
    <asp:radiobuttonlist id=when runat="server">
      <asp:listitem value="Now"/>
      <asp:listitem value="Later"/>
    </asp:radiobuttonlist>

    <asp:button type="submit" id="btnSubmit" onclick="ClickHandler"
text="Thank you!" runat="server" />
```

```
</form>
```

```
</body>
```

```
</html>
```

3. Add a Page_Load event handler to the ASPX code you've just created, to confirm the selections made in the following format:

Thank you very much _____

You have chosen _____ for breakfast, I will prepare it for you _____.

4. Create a very basic virtual telephone using an ASPX file that displays a textbox and a button named Call. Configure your ASPX file so that when you type a telephone number into your textbox and press Call, you are:

- Presented with a message confirming the number you are calling
- Presented with another button called Disconnect that, when pressed, returns you to your opening page, leaving you ready to type another number

5. Using the Switch construct, associate three particular telephone numbers with three names, so that when you press the Call button, your confirmation message contains the name of the person you are 'calling' rather than just the telephone number.

[What do YOU think? CLICK HERE to add a comment to this article.](#)

No reader comments yet.

[Printer-friendly page layout](#)

Portions of ADVISOR NETWORK and this Web site copyright ©1983-2004, this article copyright ©2003 ADVISOR MEDIA, Inc. unless otherwise stated. All Rights Reserved. ADVISOR® is a registered trademark of ADVISOR MEDIA, Inc. in the United States and other countries. For ADVISOR NETWORK Terms of Use see <http://AdvisorMedia.com/Legal>.

ADVISOR®