

## **UNIT 3 : State Management and Advanced Concepts**

- Introduction
  - View State : example, making view state secure, retaining member variables, storing custom objects
  - Transferring information, custom cookies, session state, session state configuration, application state
  - The Global.asax Application file
  - Login Controls
  - Validation Controls
  - Site Navigation and Site Maps
- 

## **Introduction**

The most significant difference between programming for the web and programming for the desktop is *state management*—how you store information over the lifetime of your application. This information can be as simple as a user's name, or as complex as a full shopping cart for an e-commerce store.

In a traditional Windows application, there is little need to think about state management. Memory is always available, and you only need to worry about a single user. In a web application, thousands of users can simultaneously run the same application on the same computer (the web server), each one communicating over a stateless HTTP connection. These conditions make it impossible to design a web application like a traditional Windows program.

Understanding these state limitations is the key to creating efficient web applications. You have different storage options, including view state, session state, and custom cookies. You have also consider how to transfer information from page to page using cross-page posting and the query string.

## **View State**

One of the most common ways to store information is in *view state*.

View state uses a hidden field that ASP.NET automatically inserts in the final, rendered HTML of a web page. It's a perfect place to store information that's used for multiple postbacks in a single web page.

For example, if you change the text of a label, the Label control automatically stores its new text in view state. That way, the text remains in place the next time the page is posted back.

### **Features Of View State**

These are the main features of view state,

- Retains the value of the Control after post-back without using a session.
- Stores the value of Pages and Control Properties defined in the page.
- Creates a custom View State Provider that lets you store View State Information in a SQL Server Database or in another data store.

### **The ViewState Collection**

The ViewState property of the page provides the current view state information. This property is an instance of the StateBag collection class. The StateBag is a dictionary collection, which means every item is stored in a separate “slot” using a unique string name.

For example, consider this code:

```
Me.ViewState["Counter"] = 1
```

This places the value 1 into the ViewState collection and gives it the name Counter. If currently no item has the name Counter, a new item will be added automatically. If an item is already stored under the name Counter, it will be replaced.

When retrieving a value, you use the key name. You also need to cast the retrieved value to the appropriate data type using the casting syntax. This extra step is required because the ViewState collection stores all items as basic objects

For Example, Here the code that retrieves the counter from view state and converts it to an integer:

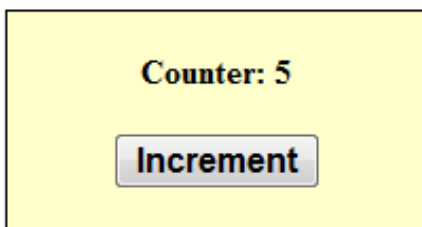
```
TextBox1.Text = ViewState["Counter"].ToString();
```

## A View State Example

The following example is a simple counter program that records how many times a button is clicked.

```
Protected Sub cmdIncrement_Click(ByVal sender As Object, ByVal e As EventArgs) Handles cmdIncrement.Click
    Dim Counter As Integer
    If ViewState("Counter") Is Nothing Then
        Counter = 1
    Else
        Counter = CType(ViewState("Counter"), Integer) + 1
    End If
    ViewState("Counter") = Counter
    lblCount.Text = "Counter: " & Counter.ToString()
End Sub
```

Output : *A simple view state counter*



## Making View State Secure

As you add more information to view state, this value can become much longer. Because this value isn't formatted as clear text, many ASP.NET programmers assume that their view state data is encrypted.

## Tamperproof View State

If you want to make view state more secure, you have two choices.

First, you can make sure the view state information is tamperproof by instructing ASP.NET to use a *hash code*. A hash code is sometimes described as a cryptographically strong checksum. The idea is that ASP.NET examines all the data in view state, just before it renders the final page. It runs this data through a hashing algorithm (with the help of a secret key value). The hashing algorithm creates a short segment of data, which is

the hash code. This code is then added at the end of the view state data, in the final HTML that's sent to the browser.

When the page is posted back, ASP.NET examines the view state data and recalculates the hash code using the same process. It then checks whether the checksum it calculated matches the hash code that is stored in the view state for the page. If a malicious user changes part of the view state data, ASP.NET will end up with a new hash code that doesn't match. At this point, it will reject the postback completely.

Hash codes are actually enabled by default, so if you want this functionality, you don't need to take any extra steps. Occasionally, developers choose to disable this feature to prevent problems in a web farm where different servers have different keys. To disable hash codes, you can use the `enableViewStateMac` attribute of the `<pages>` element in the `web.config` or `machine.config` file, as shown here:

```
<configuration>
  <system.web>
    <pages enableViewStateMac="false" />
    ...
  </system.web>
</configuration>
```

### Private View State

Even when you use hash codes, the view state data will still be readable by the user. In many cases, this is completely acceptable after all, the view state tracks information that's often provided directly through other controls. However, if your view state contains some information you want to keep secret, you can enable view state *encryption*.

You can turn on encryption for an individual page using the `ViewStateEncryptionMode` property of the `Page` directive:

```
<%@Page ViewStateEncryptionMode="Always" %>
```

Or

you can set the same attribute in a configuration file:

```
<configuration>
  <system.web>
    <pages viewStateEncryptionMode="Always" />
    ...
  </system.web>
</configuration>
```

You have three choices for your view state encryption setting

- (1) always encrypt (Always),
- (2) never encrypt (Never), or
- (3) encrypt only if a control specifically requests it (Auto).

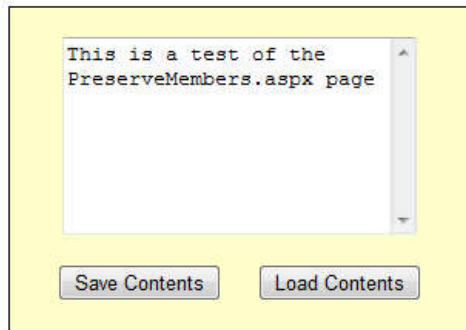
The default is Auto, which means that the page won't encrypt its view state unless a control on that page specifically requests it.

### Retaining Member Variables

You have probably noticed that any information you set in a member variable for an ASP.NET page is automatically abandoned when the page processing is finished and the page is sent to the client. Interestingly, you can work around this limitation using view state.

The basic principle is to save all member variables to view state when the Page.PreRender event occurs and retrieve them when the Page.Load event occurs. Remember, the Load event happens every time the page is created. In the case of a postback, the Load event occurs first, followed by any other control events.

For Example, The page provides a text box and two buttons. The user can choose to save a string of text and then restore it at a later time. The Button.Click event handlers store and retrieve this text using the Contents member variable. These event handlers don't need to save or restore this information using view state, because the PreRender and Load event handlers perform these tasks when page processing starts and finishes.



**Figure. :** A page with state

using System;

```
public partial class PreserveMembers : System.Web.UI.Page
{
    private string Contents;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.IsPostBack)
            Contents = System.Convert.ToString(System.Web.UI.Control.ViewState["Text"]);
    }

    protected void Page_PreRender(object sender, EventArgs e)
    {
        System.Web.UI.Control.ViewState["Text"] = Contents;
    }

    protected void cmdSave_Click(object sender, EventArgs e)
    {
        Contents = txtValue.Text;
        txtValue.Text = "";
    }

    protected void cmdLoad_Click(object sender, EventArgs e)
    {
        txtValue.Text = Contents;
    }
}
```

The logic in the Load and PreRender event handlers allows the rest of your code to work more or less as it would in a desktop application. If you store unnecessary information in view state, it will enlarge the size of the final page output and can thus slow down page transmission times. Another disadvantage with this approach is that it hides the low-level reality that every piece of data must be explicitly saved and restored.

## Storing Custom Objects

You can store your own objects in view state just as easily as you store numeric and string types. However, to store an item in view state, ASP.NET must be able to convert it into a stream of bytes so that it can be added to the hidden input field in the page. This process is called *serialization*. If your objects aren't serializable (and by default they're not), you'll receive an error message when you attempt to place them in view state.

To make your objects serializable, you need to add a `Serializable` attribute before your class declaration. For example, here's an exceedingly simple `Customer` class:

```
<Serializable(>
public class Customer
{
    private string _firstName;

    public string FirstName
    {
        get
        {
            return _firstName;
        }

        set
        {
            _firstName = Value;
        }
    }

    private string _lastName;

    public string LastName
    {
        get
        {
            return _lastName;
        }

        set
        {
            _lastName = Value;
        }
    }

    public Customer(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Because the `Customer` class is marked as serializable, it can be stored in view state:

```
{
    Customer cust = new Customer("Marsala", "Simons");
    ViewState["CurrentCustomer"] = cust;
}
```

Remember, when using custom objects, you'll need to cast your data when you retrieve it from view state.

```
{
    Customer cust;
    cust = (Customer)ViewState["CurrentCustomer"];
}
```

Once you understand this principle, you'll also be able to determine which .NET objects can be placed in view state. You simply need to find the class information in the Visual Studio Help. The easiest approach is to look

the class up in the index. For example, to find out about the FileInfo class, look for the index entry “FileInfo class.” In the class documentation, you’ll see the declaration for that class, which looks something like this:

```
<Serializable>  
<ComVisible(True)>  
Public NotInheritable Class FileInfo  
    Inherits FileSystemInfo
```

If the class declaration is preceded with the Serializable attribute, instances of this class can be placed in view state. If the Serializable attribute isn’t present, the class isn’t serializable, and you won’t be able to place instances in view state.

### Data Objects That Can be Stored in View state

- String
- Boolean Value
- Array Object
- Array List Object
- Hash Table
- Custom type Converters

### Advantages of View State

- Easy to Implement.
- No server resources are required: The View State is contained in a structure within the page load.
- Enhanced security features: It can be encoded and compressed or Unicode implementation.

### Disadvantages of View State

- Security Risk: The Information of View State can be seen in the page output source directly. You can manually encrypt and decrypt the contents of a Hidden Field, but It requires extra coding. If security is a concern then consider using a Server-Based state Mechanism so that no sensitive information is sent to the client.
- Performance: Performance is not good if we use a large amount of data because View State is stored in the page itself and storing a large value can cause the page to be slow.
- Device limitation: Mobile Devices might not have the memory capacity to store a large amount of View State data.
- It can store values for the same page only.

### When We Should Use View State

- When the data to be stored is small.
- Try to avoid secure data.

## Transferring Information Between Pages

One of the most significant limitations with view state is to transfer information to a specific page. If the user navigates to another page, this information is lost.

There are two basic techniques to transfer information between pages: cross-page posting and the query string.

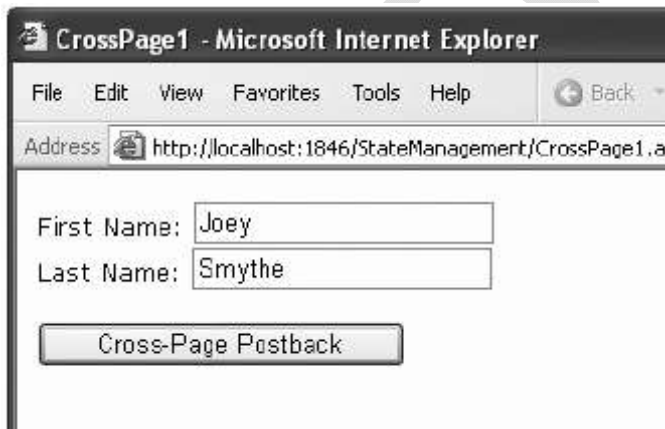
### Cross-Page Posting

A cross-page postback is a technique that extends the postback mechanism so that one page can send the user to another page, complete with all the information for that page.

The infrastructure that supports cross-page postbacks is a new property named `PostBackUrl`, which is defined by the `IButtonControl` interface and turns up in button controls such as `ImageButton`, `LinkButton`, and `Button`. To use cross-posting, you simply set `PostBackUrl` to the name of another web form. When the user clicks the button, the page will be posted to that new URL with the values from all the input controls on the current page.

For Example, A page named `CrossPage1.aspx` that defines a form with two text boxes and a button. When the button is clicked, it posts to a page named `CrossPage2.aspx`.

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="CrossPage1.aspx.vb" Inherits="CrossPage1" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>CrossPage1</title>
</head>
<body>
    <form id="form1" runat="server" >
        <div>
            First Name:
            <asp:TextBox ID="txtFirstName" runat="server"></asp:TextBox>
            <br />
            Last Name:
            <asp:TextBox ID="txtLastName" runat="server"></asp:TextBox>
            <br />
            <br />
            <asp:Button runat="server" ID="cmdPost"
                PostBackUrl="CrossPage2.aspx" Text="Cross-Page Postback" /><br />
        </div>
    </form>
</body>
</html>
```



**Figure :** *The source of a cross-page postback*

Now if you load this page and click the button, the page will be posted back to `CrossPage2.aspx`. At this point, the `CrossPage2.aspx` page can interact with `CrossPage1.aspx` using the `Page.PreviousPage` property. Here's an event handler that grabs the title from the previous page and displays it:

```
using System;
```

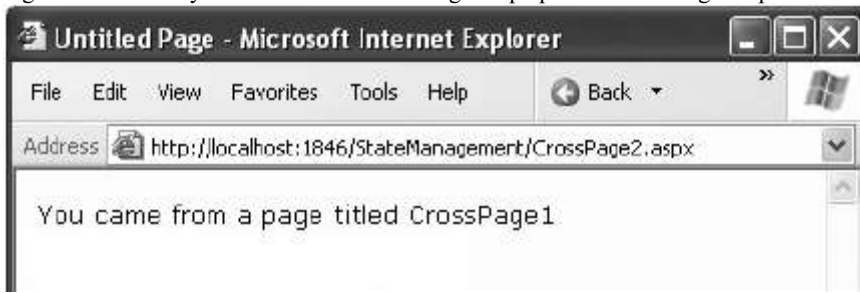
```
public partial class CrossPage2 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (System.Web.UI.Page.PreviousPage != null)
            lblInfo.Text = "You came from a page titled " + System.Web.UI.Page.PreviousPage.Title;
    }
}
```

```

}
}

```

Figure shows what you will see when CrossPage1.aspx posts to CrossPage2.aspx.



**Figure :** *The target of a cross-page postback*

### Getting More Information from the Source Page

For example, if you want to expose the values from two text boxes in the source page, you might add properties that wrap the control variables. Here are two properties you could add to the CrossPage1 class to expose its TextBox controls:

```

class SurroundingClass
{
    public TextBox FirstNameTextBox
    {
        get
        {
            return txtFirstName;
        }
    }

    public TextBox LastNameTextBox
    {
        get
        {
            return txtLastName;
        }
    }
}

```

A better choice is to define specific, limited methods or properties that extract just the information you need. For example, you might decide to add a FullName property that retrieves just the text from the two text boxes. Here's the full page code for CrossPage1.aspx with this property:

```

partial class CrossPage1 : System.Web.UI.Page
{
    public string FullName
    {
        get
        {
            return txtFirstName.Text + " " + txtLastName.Text;
        }
    }
}

```

This way, the relationship between the two pages is clear, simple, and easy to maintain.

Now you can rewrite the code in CrossPage2.aspx to display the information from CrossPage1.aspx:



```
protected void Page_Load(object sender, EventArgs e)
{
    if (PreviousPage != null)
    {
        lblInfo.Text = "You came from a page titled " + PreviousPage.Title + "<br />";
        CrossPage1 prevPage;
        prevPage = PreviousPage as CrossPage1;
        if (prevPage != null)
            lblInfo.Text += "You typed in this: " + prevPage.FullName;
    }
}
```

The target page (CrossPage2.aspx) can access the Title property and FullName property to get the information.

Figure 7-5 shows the new result.



**Figure 7-5.** Retrieving specific information from the source page

## The Query String

Another common approach is to pass information using a query string in the URL. This approach is commonly found in search engines. For example, if you perform a search on the Google website, you'll be redirected to a new URL that incorporates your search parameters. Here's an example:

```
http://www.google.ca/search?q=organic+gardening
```

The query string is the portion of the URL after the question mark. In this case, it defines a single variable named *q*, which contains the string *organic+gardening*.

The advantage of the query string is that it's lightweight and doesn't give any kind of burden on the server. However, it also has several limitations:

- Information is limited to simple strings, which must contain URL-legal characters.
- Information is clearly visible to the user and to anyone else who cares to eavesdrop on the Internet.
- The enterprising user might decide to modify the query string and supply new values, which your program won't expect and can't protect against.
- Many browsers impose a limit on the length of a URL (usually from 1KB to 2KB). For that reason, you can't place a large amount of information in the query string and still be assured of compatibility with most browsers.

To store information in the query string, you need to place it there yourself. Typically, this means using a special HyperLink control or a special Response.Redirect() statement such as the one shown here:

```
Response.Redirect("newpage.aspx?recordID=10")
```

You can send multiple parameters as long as they're separated with an ampersand (&):

```
Response.Redirect("newpage.aspx?recordID=10&mode=full")
```

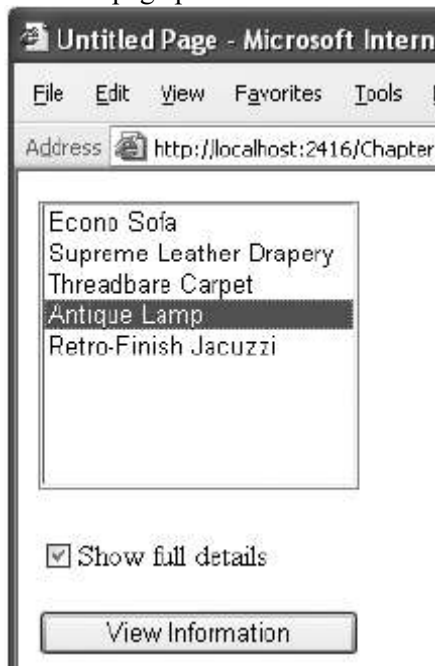
The receiving page has an easier time working with the query string. It can receive the values from the QueryString dictionary collection exposed by the built-in Request object:

```
string ID = Request.QueryString("recordID");
```

Note that information is always retrieved as a string, which can then be converted to another simple data type. Values in the QueryString collection are indexed by the variable name. If you attempt to retrieve a value that isn't present in the query string, you'll get a null reference (Nothing).

### A Query String Example

The first page provides a list of items, a check box, and a submission button



**Figure :** *A query string sender*

Here's the code for the first page:

```
using System;

public partial class QueryStringSender : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // Add sample values.
            lstItems.Items.Add("Econo Sofa");
            lstItems.Items.Add("Supreme Leather Drapery");
            lstItems.Items.Add("Threadbare Carpet");
            lstItems.Items.Add("Antique Lamp");
            lstItems.Items.Add("Retro-Finish Jacuzzi");
        }
    }

    protected void cmdGo_Click(object sender, EventArgs e)
    {
        if (lstItems.SelectedIndex == -1)
```

```

        lblError.Text = "You must select an item.";
    else
    {
        // Forward the user to the information page,
        // with the query string data.
        string Url = "QueryStringRecipient.aspx?";
        Url += "Item=" + lstItems.SelectedItem.Text + "&";
        Url += "Mode=" + chkDetails.Checked.ToString();
        System.Web.UI.Page.Response.Redirect(Url);
    }
}
}
}

```

Here's the code for the recipient page

using System;

```

public partial class QueryStringRecipient : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        lblInfo.Text = "Item: " + System.Web.UI.Page.Request.QueryString["Item"];
        lblInfo.Text += "<br />Show Full Record: ";
        lblInfo.Text += System.Web.UI.Page.Request.QueryString["Mode"];
    }
}

```

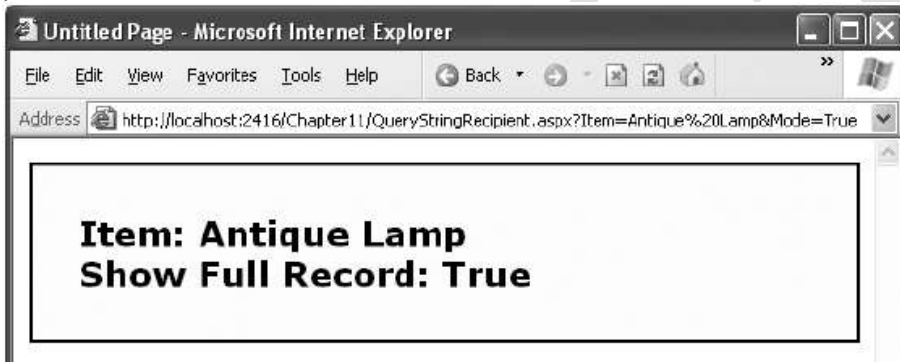


Figure : A query string recipient

## Cookies

*Cookies* provide another way that you can store information for later use. Cookies are small files that are created on the client's hard drive.

One advantage of cookies is that they work transparently without the user being aware that information needs to be stored. They also can be easily used by any page in your application and even be retained between visits, which allows for truly long-term storage.

They suffer from some of the same drawbacks that affect query strings—namely, they're limited to simple string information, and they're easily accessible and readable if the user finds and opens the corresponding file. These factors make them a poor choice for complex or private information or large amounts of data.

Some users disable cookies on their browsers, which will cause problems for web applications that require them. Also, users might manually delete the cookie files stored on their hard drives.

Cookies are easy to use. Both the Request and Response objects provide a Cookies collection. The important trick to remember is that you retrieve cookies from the Request object, and you set cookies using the Response object.

To set a cookie, just create a new HttpCookie object. You can then fill it with string information and attach it to the current web response:

```
// Create the cookie object.
HttpCookie cookie = new HttpCookie("Preferences");

// Set a value in it.
Cookie["LanguagePref"] = "English";

// Add another value.
Cookie["Country"] = "US";

// Add it to the current web response.
Response.Cookies.Add(cookie);
```

A cookie added in this way will persist until the user closes the browser and will be sent with every request. To create a longer-lived cookie, you can set an expiration date:

```
' This cookie lives for one year.
cookie.Expires = DateTime.Now.AddYears(1);
```

You retrieve cookies by cookie name using the Request.Cookies collection:

```
Dim cookie As HttpCookie = Request.Cookies["Preferences"];

' Check to see whether a cookie was found with this name.
' This is a good precaution to take,
' because the user could disable cookies,
' in which case the cookie will not exist.

String language;
If (cookie!=Nothing)
    language = cookie["LanguagePref"];
```

The only way to remove a cookie is by replacing it with a cookie that has an expiration date that has already passed. This code demonstrates the technique:

```
HttpCookie cookie=new HttpCookie("LanguagePref");
cookie.Expires = DateTime.Now.AddDays(-1);
Response.Cookies.Add(cookie);
```

### Advantages

1. Cookies do not require any server resources since they are stored on the client.
2. Cookies are easy to implement.

### Disadvantages:

- 1) Cookies can be disabled on user browsers
- 2) Cookies are transmitted for each HTTP request/response causing overhead on bandwidth
- 3) No security for sensitive data

### Cookie Limitations:

1. Most browsers support cookies of up to 4096 bytes(4kbytes)

2. Most browsers allow only 20 cookies per site; if you try to store more, the oldest cookies are discarded.
3. Browser supports 300 cookies towards different websites.
4. Complex type of data not allowed (eg: dataset), allows only plain text (ie, cookie allows only string content)
5. Cookies are browser specific (ie, one browser type[IE] stored cookies will not be used by another browser type[firefox]).

## Session State

Session state management is one of ASP.NET's premiere features. It allows you to store any type of data in memory on the server. The information is protected, because it is never transmitted to the client, and it's uniquely bound to a specific session. Every client that accesses the application has a different session and a distinct collection of information. Session state is ideal for storing information such as the items in the current user's shopping basket when the user browses from one page to another.

### Session Tracking

ASP.NET tracks each session using a unique 120-bit identifier. ASP.NET uses a proprietary algorithm to generate this value, thereby guaranteeing (statistically speaking) that the number is unique and it's random enough that a malicious user can't reverse-engineer or "guess" what session ID a given client will be using. This ID is the only piece of session-related information that is transmitted between the web server and the client.

When the client presents the session ID, ASP.NET looks up the corresponding session, retrieves the objects you stored previously, and places them into a special collection so they can be accessed in your code. This process takes place automatically.

For this system to work, the client must present the appropriate session ID with each request. You can accomplish this in two ways:

- *Using cookies*: In this case, the session ID is transmitted in a special cookie (named ASP.NET\_SessionId), which ASP.NET creates automatically when the session collection is used. This is the default, and it's also the same approach that was used in earlier versions of ASP.
- *Using modified URLs*: In this case, the session ID is transmitted in a specially modified (or *munged*) URL. This allows you to create applications that use session state with clients that don't support cookies.

Session state doesn't come for free. Though it solves many of the problems associated with other forms of state management, it forces the server to store additional information in memory. This extra memory requirement, even if it is small, can quickly grow to performance-destroying levels as hundreds or thousands of clients access the site.

### Using Session State

You can interact with session state using the System.Web.SessionState.HttpSessionState class, which is provided in an ASP.NET web page as the built-in Session object. The syntax for adding items to the collection and retrieving them is basically the same as for adding items to a page's view state.

For example, you might store a DataSet in session memory like this:

```
Session["InfoDataSet"] = dsInfo;
```

You can then retrieve it with an appropriate conversion operation:

```
dsInfo = CType(Session["InfoDataSet"], DataSet);
```

Session state is global to your entire application for the current user. However, session state can be lost in several ways:

- If the user closes and restarts the browser.
- If the user accesses the same page through a different browser window, although the session will still exist if a web page is accessed through the original browser window. Browsers differ on how they handle this situation.
- If the session times out due to inactivity. More information about session timeout can be found in the configuration section.
- If your web page code ends the session by calling the `Session.Abandon()` method.

In the first two cases, the session actually remains in memory on the web server, because ASP.NET has no idea that the client has closed the browser or changed windows. The session will linger in memory, remaining inaccessible, until it eventually expires.

Table describes the methods and properties of the `HttpSessionState` class.

**Table :** *HttpSessionState Members*

Member	Description
Count	Provides the number of items in the current session collection.
IsCookieless	Identifies whether the session is tracked with a cookie or modified URLs.
IsNewSession	Identifies whether the session was created only for the current request. If no information is in session state, ASP.NET won't bother to track the session or create a session cookie. Instead, the session will be re-created with every request.
Mode	Provides an enumerated value that explains how ASP.NET stores session state information. This storage mode is determined based on the web.config settings discussed in the "Session State Configuration" section later in this chapter.
SessionID	Provides a string with the unique session identifier for the current client.
Timeout	Determines the number of minutes that will elapse before the current session is abandoned, provided that no more requests are received from the client. This value can be changed programmatically, letting you make the session collection longer when needed.
Abandon()	Cancels the current session immediately and releases all the memory it occupied. This is a useful technique in a logoff page to ensure that server memory is reclaimed as quickly as possible.
Clear()	Removes all the session items but doesn't change the current session identifier.

## Session State Configuration

You configure session state through the web.config file for your current application. The configuration file allows you to set advanced options such as the timeout and the session state mode. If you're creating your web application in Visual Studio, your project will include an automatically generated web.config file.

The following listing shows the most important options that you can set for the `<sessionState>` element. Keep in mind that you won't use all of these details at the same time.

Some settings only apply to certain session state *modes*, as you'll see shortly.

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

```

<configuration>
  <system.web>
    <!-- Other settings omitted. -->
    <sessionState
      cookieless="UseCookies" cookieName="ASP.NET_SessionId"
      regenerateExpiredSessionId="false"
      timeout="20"
      mode="InProc"
      stateConnectionString="tcpip=127.0.0.1:42424"
      stateNetworkTimeout="10"
      sqlConnectionString="data source=127.0.0.1;Integrated Security=SSPI"
      sqlCommandTimeout="30"
      allowCustomSqlDatabase="false"
      customProvider=""
    />
  </system.web>
</configuration>

```

The following sections describe the preceding session state settings.

## Cookieless

You can set the cookieless setting to one of the values defined by the `HttpCookieMode` enumeration, as described in Table.

**Table :** *HttpCookieMode Values*

Value	Description
UseCookies	Cookies are always used, even if the browser or device doesn't support cookies or they are disabled. This is the default. If the device does not support cookies, session information will be lost over subsequent requests, because each request will get a new ID.
UseUri	Cookies are never used, regardless of the capabilities of the browser or device. Instead, the session ID is stored in the URL.
UseDeviceProfile	ASP.NET chooses whether to use cookieless sessions by examining the <code>BrowserCapabilities</code> object. The drawback is that this object indicates what the device should support—it doesn't take into account that the user may have disabled cookies in a browser that supports them.
AutoDetect	ASP.NET attempts to determine whether the browser supports cookies by attempting to set and retrieve a cookie (a technique commonly used on the Web). This technique can correctly determine whether a browser supports cookies but has them disabled, in which case cookieless mode is used instead.

Here's an example that forces cookieless mode (which is useful for testing):

```
<sessionState cookieless="UseUri" ... />
```

In cookieless mode, the session ID will automatically be inserted into the URL. When ASP.NET receives a request, it will remove the ID, retrieve the session collection, and forward the request to the appropriate directory. Figure 7-10 shows a munged URL.



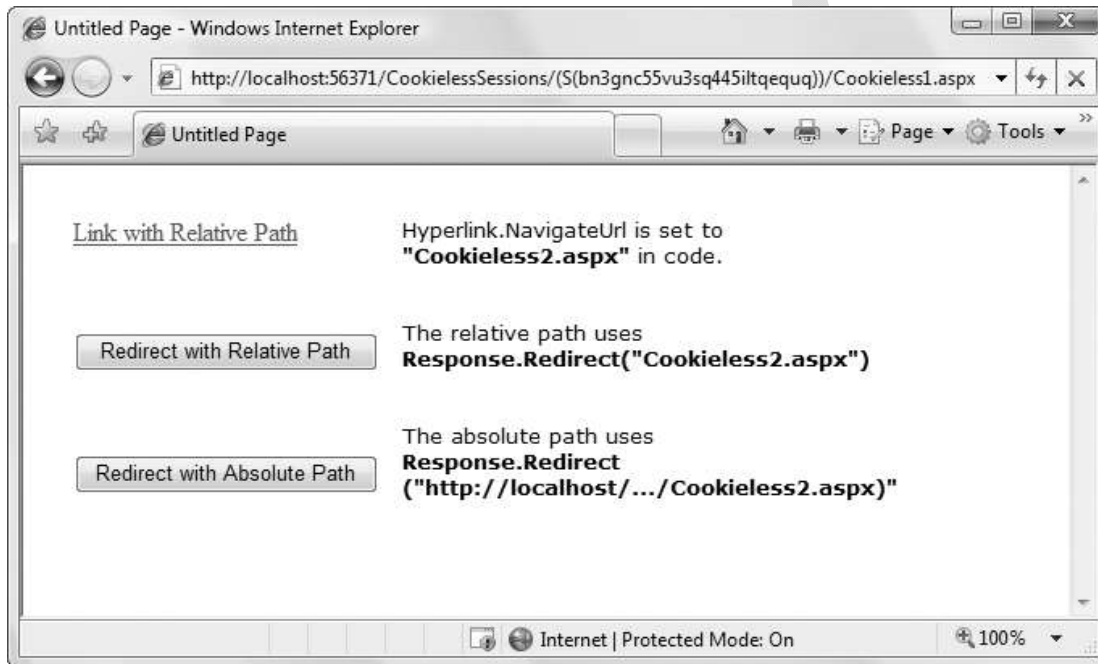
**Figure :** *A munged URL with the session ID*

Because the session ID is inserted in the current URL, relative links also automatically gain the session ID. In other words, if the user is currently stationed on Page1.aspx and clicks a relative link to Page2.aspx, the relative link includes the current session ID as part of the URL. The same is true if you call Response.Redirect() with a relative URL, as shown here:

```
Response.Redirect("Page2.aspx")
```

Figure shows a sample website that tests cookieless sessions. It contains two pages and uses cookieless mode. The first page (Cookieless1.aspx) contains a HyperLink control and two buttons, all of which take you to a second page (Cookieless2.aspx). The trick is that these controls have different ways of performing their navigation. Only two of them work with cookieless session—the third loses the current session.

The HyperLink control navigates to the page specified in its NavigateUrl property, which is set to the relative path Cookieless2.aspx. If you click this link, the session ID is retained in the URL, and the new page can retrieve the session information. This demonstrates that cookieless sessions work with relative links.



**Figure :** *Three tests of cookieless sessions*

The two buttons on this page use programmatic redirection by calling the Response.Redirect() method. The first button uses the relative path Cookieless2.aspx, much like the HyperLink control. This approach works with cookieless session state, and preserves the munged URL with no extra steps required.

```
Protected Sub cmdLink_Click(ByVal sender As Object, ByVal As EventArgs) Handles cmdLink.Click
    Response.Redirect("Cookieless2.aspx")
End Sub
```

The only real limitation of cookieless state is that you cannot use absolute links (links that include the full URL, starting with http://). The second button uses an absolute link to demonstrate this problem. Because ASP.NET cannot insert the session ID into the URL, the session is lost.

```
protected void cmdLinkAbsolute_Click(object sender, EventArgs e)
{
```



```
String url = "http://localhost:56371/CookielessSessions/Cookieless2.aspx";
Response.Redirect(url);
}
```

Now the target page (Following Figure) checks for the session information, but can't find it. Writing the code to demonstrate this problem in a test environment is a bit tricky. The problem is that Visual Studio's integrated web server chooses a different port for your website every time you start it. As a result, you'll need to edit the code every time you open Visual Studio so that your URL uses the right port number (such as 56371 in the previous example).



**Figure :** *A lost session*

There's another workaround. You can use some crafty code that gets the current URL from the page and just modifies the last part of it (changing the page name from Cookieless1.aspx to Cookieless2.aspx). Here's how:

' Create a new URL based on the current URL (but ending with the page Cookieless2.aspx instead of Cookieless1.aspx.

```
String url = "http://" & Request.Url.Authority & Request.Url.Segments(0) & Request.Url.Segments(1) & _
    "Cookieless2.aspx";
Response.Redirect(url);
```

Of course, if you deploy your website to a real virtual directory that's hosted by IIS, you won't use a randomly chosen port number anymore, and you won't experience this quirk. Chapter 9 has more about virtual directories and website deployment.

## Application State

Application state allows you to store global objects that can be accessed by any client. Application state is based on the `System.Web.HttpApplicationState` class, which is provided in all web pages through the built-in `Application` object.

Application state is similar to session state. It supports the same type of objects, retains information on the server, and uses the same dictionary-based syntax. A common example with application state is a global counter that tracks how many times an operation has been performed by all the web application's clients.

For example, you could create a `Global.asax` event handler that tracks how many sessions have been created or how many requests have been received into the application. Or you can use similar logic in the `Page.Load` event handler to track how many times a given page has been requested by various clients. Here's an example of the latter:

```
protected void Page_Load(object sender, EventArgs e)
{
    int Count = System.Convert.ToInt32(Application["HitCounterForOrderPage"]);
    Count += 1;
    Application["HitCounterForOrderPage"] = Count;
    lblCounter.Text = Count.ToString();
}
```

Once again, application state items are stored as objects, so you need to cast them when you retrieve them from the collection. Items in application state never time out. They last until the application or server is restarted, or the application domain refreshes itself (because of automatic process recycling settings or an update to one of the pages or components in the application).

Application state isn't often used, because it's generally inefficient. In the previous example, the counter would probably not keep an accurate count, particularly in times of heavy traffic. For example, if two clients requested the page at the same time, you could have a sequence of events like this:

1. User A retrieves the current count (432).
2. User B retrieves the current count (432).
3. User A sets the current count to 433.
4. User B sets the current count to 433.

In other words, one request isn't counted because two clients access the counter at the same time. To prevent this problem, you need to use the `Lock()` and `Unlock()` methods, which explicitly allow only one client to access the Application state collection at a time.

```
protected void Page_Load(object sender, EventArgs e)
{
    // Acquire exclusive access.
    Application.Lock();
    int Count = System.Convert.ToInt32(Application["HitCounterForOrderPage"]);
    Count += 1;
    Application["HitCounter"] = Count;
    // Release exclusive access.
    Application.Unlock();
    lblCounter.Text = Count.ToString();
}
```

Unfortunately, all other clients requesting the page will be stalled until the Application collection is released. This can drastically reduce performance. Generally, frequently modified values are poor candidates for application state. In fact, application state is rarely used in the .NET world because its two most common uses have been replaced by easier, more efficient methods:

- In the past, application state was used to store application-wide constants, such as a database connection string. This type of constant can be stored in the `web.config` file.
- Application state can also be used to store frequently used information that is timeconsuming to create, such as a full product catalog that requires a database lookup.

## The Global.asax File

The `Global.asax` file allows you to write code that responds to global application events. These events fire at various points during the lifetime of a web application, including when the application domain is first created (when the first request is received for a page in your website folder).

To add a `Global.asax` file to an application in Visual Studio, choose `Website > Add New Item`, and select the `Global Application Class` file type. Then, click `OK`.

The `Global.asax` file looks similar to a normal `.aspx` file, except that it can't contain any HTML or ASP.NET tags. Instead, it contains event handlers. For example, the following `Global.asax` file reacts to the `Application.EndRequest` event, which happens just before the page is sent to the user:

```
<%@ Application Language="VB" %>
```

```

<script runat="server">

public void Application_EndRequest(object sender, EventArgs e)
{
    // Code that runs at the end of every request.
    Response.Write("<hr>This page was served at " + DateTime.Now.ToString());
}
</script>

```

This event handler uses the Write() method of the built-in Response object to write a footer at the bottom of the page with the date and time that the page was created (see Figure 5-10).

Each ASP.NET application can have one Global.asax file. Once you place it in the appropriate website directory, ASP.NET recognizes it and uses it automatically. For example, if you add the Global.asax file shown previously to a web application, every web page in that application will include a footer.

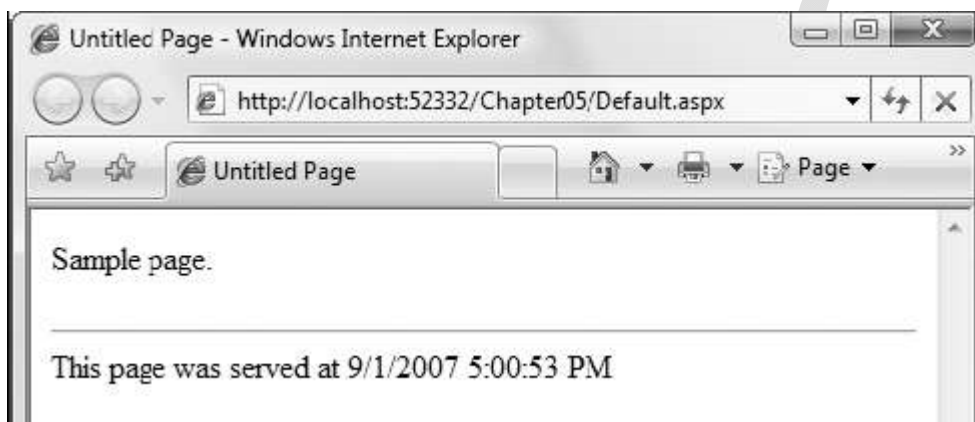


Figure: HelloWorld.aspx with an automatic footer

### Additional Application Events

Event-Handling Method	Description
Application_Start()	Occurs when the application starts, which is the first time it receives a request from any user.
Application_End()	Occurs when the application is shutting down, generally because the web server is being restarted. You can insert cleanup code here.
Application_BeginRequest()	Occurs with each request the application receives, just before the page code is executed.
Application_EndRequest()	Occurs with each request the application receives, just after the page code is executed.
Session_Start()	Occurs whenever a new user request is received and a session is started.
Session_End()	Occurs when a session times out or is programmatically ended.
Application_Error()	Occurs in response to an unhandled error.

## Login Control

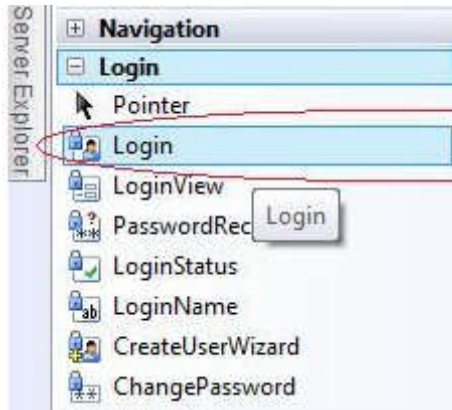
The Login Control provides a user interface that facilitate in authenticating users of the website on the basis of username and password.

The Login control displays a user interface for user authentication. The Login control contains text boxes for the user name and password and a check box that allows users to indicate that the server want to store username and password using ASP.NET membership.

The Login control has properties for customized display, for customized messages, and for links to other pages where users can change their password or recover a forgotten password. The Login control can be used as a standalone control on a main or home page, or you can use it on a dedicated login page.

If you use the Login control with ASP.NET membership, you do not need to write code to perform authentication. However, if you want to create your own authentication logic, you can handle the Login control's Authenticate event and add custom authentication code.

Drag and drop Login control on page from ToolBox.



**Log In**

User Name:

Password:

Remember me next time.

[Register User](#)

[Forgot Password?](#)

The Login class contains various methods, properties and events used to work with the Login control. These members enable you to customize Login control, such as displaying messages and creating links to change password and recovering a forgotten password. The Login control contains built-in authentication logic.

Following are some important properties that are very useful.

### Properties of the Login Control

<i>TitleText</i>	Indicates the text to be displayed in the heading of the control.
<i>InstructionText</i>	Indicates the text that appears below the heading of the control.

<i>UserNameLabelText</i>	Indicates the label text of the username text box.
<i>PasswordLabelText</i>	Indicates the label text of the password text box.
<i>FailureText</i>	Indicates the text that is displayed after failure of login attempt.
<i>UserName</i>	Indicates the initial value in the username text box.
<i>LoginButtonText</i>	Indicates the text of the Login button.
<i>LoginButtonType</i>	Button/Link/Image. Indicates the type of login button.
<i>DestinationPageUrl</i>	Indicates the URL to be sent after login attempt successful.
<i>DisplayRememberMe</i>	true/false. Indicates whether to show Remember Me checkbox or not.
<i>VisibleWhenLoggedIn</i>	true/false. If false, the control is not displayed on the page when the user is logged in.
<i>CreateUserUrl</i>	Indicates the url of the create user page.
<i>CreateUserText</i>	Indicates the text of the create user link.
<i>PasswordRecoveryUrl</i>	Indicates the url of the password recovery page.
<i>PasswordRecoveryText</i>	Indicates the text of the password recovery link.
<b>Style of the Login Control</b>	
<i>CheckBoxStyle</i>	Indicates the style property of the Remember Me checkbox.
<i>FailureStyle</i>	Indicates the style property of the failure text.
<i>TitleTextStyle</i>	Indicates the style property of the title text.
<i>LoginButtonStyle</i>	Indicates the style property of the Login button.
<i>TextBoxStyle</i>	Indicates the style property of the TextBox.
<i>LabelStyle</i>	Indicates the style property of the labels of text box.
<i>HyperLinkStyle</i>	Indicates the style property of the hyperlink in the control.
<i>InstructionTextStyle</i>	Indicates the style property of the Instruction text that appears below the heading of the control.
<b>Events of the Login Control</b>	
<i>LoggingIn</i>	Fires before user is going to authenticate.
<i>LoggedIn</i>	Fires after user is authenticated.
<i>LoginError</i>	Fires after failure of login attempt.
<i>Authenticate</i>	Fires to authenticate the user. This is the function where you need to write your own code to validate the user.

### **LoginView control**

The LoginView control allows you to display different information to anonymous and logged-in users.

The control displays one of two templates:

1. AnonymousTemplate – Display when the user is not logged in
2. LoggedInTemplate. – Display when the user is logged in

In the templates, you can add markup and controls that display information appropriate for anonymous users and authenticated users, respectively.

The LoginView control also includes events for ViewChanging and ViewChanged, which allow you to write handlers for when the user logs in and changes status.

LoginView control is very simple yet very powerful and customizable. It allows user to customize its view for both anonymous user and logged in user. Internally, When it is rendered on the page, it is implemented through <table></table> HTML tag.

#### DEMO : LoginView

Welcome, Guest [Login](#)

<b>Properties of the LoginView</b>	
<i>AnonymousTemplate</i>	Gets or sets the template to display to users who are not logged in to the Web site.
<i>Controls</i>	Gets the ControlCollection object that contains the child controls for the LoginView control.
<i>EnableTheming</i>	Gets or sets a value indicating whether themes can be applied to the LoginView control.
<i>LoggedInTemplate</i>	Gets or sets the template to display to Web site users who are logged in to the Web site but are not members of one of the role groups specified in the RoleGroups property.
<i>RoleGroup</i>	Gets a collection of role groups that associate content templates with particular roles.
<i>SkinID</i>	Gets or sets the skin to apply to the LoginView control.
<b>Methods of the LoginView</b>	
<i>DataBind</i>	Binds a data source to the LoginView control and all its child controls.
<i>Focus</i>	Sets input focus to a control.
<i>OnViewChanged</i>	Raises the ViewChanged event after the LoginView control switches views.
<i>OnViewChanging</i>	Raises the ViewChanging event before the LoginView control switches views.
<b>Events of the LoginView</b>	
<i>ViewChanged</i>	Occurs after the view is changed.
<i>ViewChanging</i>	Occurs before the view is changed.

## **LoginStatus control**

The LoginStatus control specifies whether or not a particular user has logged on to the website. The text displayed on this control changes according to the login status of the user. When the user is not logged in, it displays the Login text as a hyperlink that facilitates the user in navigating to the login page.

The LoginStatus control displays the Logout text as hyperlink when user is logged on to the website.

The LoginStatus control provides the following two views:

1. Logged Out : Displayed when the user is not logged in.
2. Logged In : Displayed when the user is logged in.

[Login](#) [UserName]

The LoginStatus class contains various methods, properties and events used to work with the LoginStatus control.

<b>Properties of the LoginStatus</b>	
<i>LoginImageUrl</i>	Gets or sets the URL of the image used for the login link.
<i>LoginText</i>	Gets or sets the text used for the login link.
<i>LogoutAction</i>	Gets or sets a value that determines the action taken when a user logs out of a Web site with the LoginStatus control.
<i>LogoutImageUrl</i>	Gets or sets the URL of the image used for the logout button.
<i>LogoutPageUrl</i>	Gets or sets the URL of the logout page.
<i>LogoutText</i>	Gets or sets the text used for the logout link.
<b>Methods of the LoginStatus</b>	
<i>CreateChildControls</i>	Creates the child controls that make up the LoginStatus control.
<i>OnLoggedOut</i>	Raises the LoggedOut event after the user clicks the logout link and logout processing is complete.
<i>OnLoggingOut</i>	Raises the LoggingOut event when a user clicks the logout link on the LoginStatus control.
<b>Events of the LoginStatus</b>	
<i>LoggingOut</i>	Raised when the user clicks the logout button.
<i>LoggedOut</i>	Raised after the user clicks the logout link and the logout process is complete.

### **LoginName control**

The LoginName control is responsible for displaying current logged in user name.

[Login](#) [UserName]

### **PasswordRecovery control**

PasswordRecovery control is used to recover or reset the forgotten password of a user. This control does not display the password on the browser, but sends the password as an email message to the e-mail address specified at the time of registration.

The PasswordRecovery control has the following three views.

1. **UserName** : In this view, the user is asked to enter its username for its password to be recovered.
2. **Question** : In this view, the user is asked to enter the answer for the security question.
3. **Success** : In this view, a message is displayed to the user, which specifies that the retrieved password has been sent to the user.

To retrieve or reset passwords, you must set some properties of the ASP.NET Membership service.

The PasswordRecovery class contains various methods, properties and events used to work with the PasswordRecovery Control.

Forgot Your Password?

Enter your User Name to receive your password.

User Name:

### CreateUserWizard control

The CreateUserWizard control is a ready to use control that is used to create a new user to the website. In other words we can say that it a replacement of Registration form of a website. It has few steps to complete the registration process.

The CreateUserWizard control gathers the following user information:

- User name
- Password
- Confirmation of password
- E-mail address
- Security question
- Security answer

The CreateUserWizard control uses Membership service to create user's details.

Following are some important events that are very useful.

<b>Events of CreateUserWizard Control</b>	
<i>ContinueButtonClick</i>	Fires when user clicks Continue button in the last wizard step.
<i>CreatingUser</i>	Fires before creating a new user.
<i>CreatedUser</i>	Fires after new user created.
<i>CreateUserError</i>	Fires when creation of user is not successful.



**Sign Up for Your New Account**

User Name:  \*

Password:  \*

Confirm Password:  \*

E-mail:  \*

Security Question:  \*

Security Answer:  \*

The Password and Confirmation Password must match.

Create User

## ChangePassword control

The ChangePassword control allows users to change their password. The user must first supply the original password and then create and confirm the new password. If the original password is correct, the user password is changed to the new password. The control also includes support for sending an e-mail message about the new password.

The ChangePassword control includes two templated views that are displayed to the user.

- The first is the ChangePasswordTemplate, which displays the user interface used to gather the data required to change the user password.
- The second template is the SuccessTemplate, which defines the user interface that is displayed after a user password has been successfully changed.

The ChangePassword control works with authenticated and non-authenticated users. If a user has not been authenticated, the control prompts the user for a login name. If the user is authenticated, the control populates the text box with the user's login name.

**Change Your Password**

Password:  \*

New Password:  \*

Confirm New Password:  \*

The Confirm New Password must match the New Password entry.

Change Password Cancel

## SITE NAVIGATION AND SITE MAPS

You've already learned simple ways to send a website visitor from one page to another. For example, you can add HTML links (or HyperLink controls) to your page to let users surf through your site. If you want to perform page navigation in response to another action, you can call the `Response.Redirect()` method or the `Server.Transfer()` method in your code. But in professional web applications, the navigation requirements are more intensive. These applications need a system that allows users to surf through a hierarchy of pages, without forcing you to write the same tedious navigation code in every page.

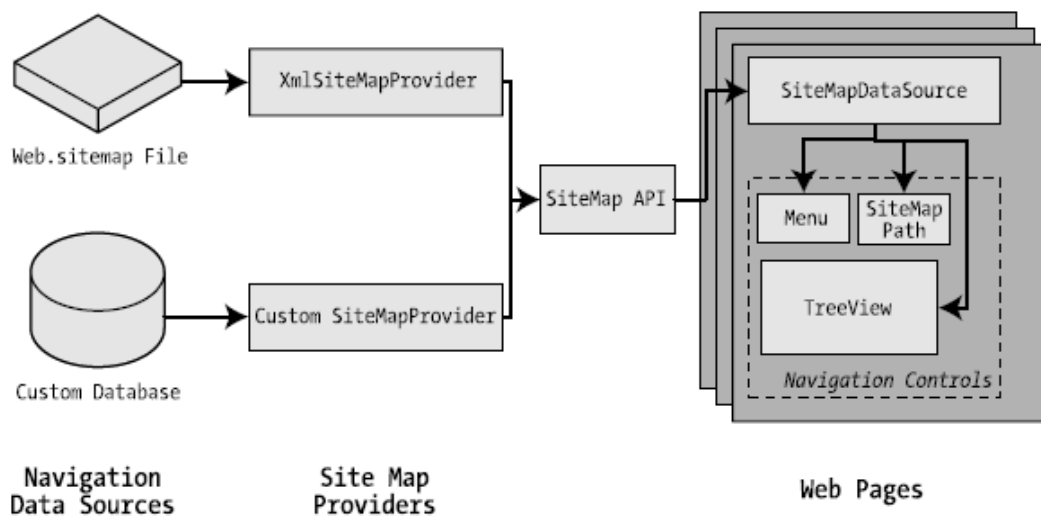
## Site Maps

ASP.NET navigation is flexible, configurable, and pluggable. It consists of three components:

- A way to define the navigational structure of your website. This part is the XML site map, which is (by default) stored in a file.
- A convenient way to read the information in the site map file and convert it to an object model. The `SiteMapDataSource` control and the `XmlSiteMapProvider` perform this part.
- A way to use the site map information to display the user's current position and give the user the ability to easily move from one place to another. This part takes place through the navigation controls you bind to the `SiteMapDataSource` control, which can include breadcrumb links, lists, menus, and trees.

You can customize or extend each of these ingredients separately. For example, if you want to change the appearance of your navigation controls, you simply need to bind different controls to the `SiteMapDataSource`. On the other hand, if you want to read site map information from a different type of file or from a different location, you need to change your site map provider.

Figure shows how these pieces fit together.



## Defining a Site Map

You can create it in Visual Studio by right on **Website** ➤ **Add New Item** and then choosing the **Site Map** option.

### Rules for creating Site Maps:

#### Rule 1: Site Maps Begin with the <siteMap> Element

Every Web.sitemap file begins by declaring the <siteMap> element and ends by closing that element. You place the actual site map information between the start and end tags

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
.....
.....
.....
</siteMap>
```

#### Rule 2: Each Page Is Represented by a <siteMapNode> Element

To insert a page into the site map,

Add the <siteMapNode> element with some basic information.

- Namely, you need to supply the title of the page (which appears in the navigation controls),
- a description (which you may or may not choose to use),
- the URL (the link for the page).

You add these three pieces of information using three attributes. The attributes are named title, description, and url, as shown here:

```
<siteMapNode description="CLASS" title="CLASS LIST" url="~/CLASS.aspx">
```

Here's a complete, valid site map file that uses this page to define a website with exactly one page:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode description="CLASS" title="CLASS LIST" url="~/CLASS.aspx">
    <siteMapNode description="FYBCA" title="FYBCA" url="~/FYBCA.aspx"/>
  </siteMapNode>
</siteMap>
```

#### Rule 3: A <siteMapNode> Element Can Contain Other <siteMapNode> Elements

Site maps don't consist of simple lists of pages. Instead, they divide pages into groups. To represent this in a site map file, you place one <siteMapNode> inside another. Instead of using the empty element syntax shown previously, you'll need to split your <siteMapNode> element into a start tag and an end tag:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
```

```

<siteMapNode description="CLASS" title="CLASS LIST" url="~/CLASS.aspx">
  <siteMapNode description="SEM-I" title="SEM-I" url="~/SEM1.aspx" />
  <siteMapNode description="SEM-II" title="SEM-II" url="~/SEM2.aspx" />
  <siteMapNode description="SEM-III" title="SEM-III" url="~/SEM3.aspx" />
</siteMapNode>
</siteMap>

```

When you show this part of the site map in a web page, the Products node will appear as ordinary text, not a clickable link.

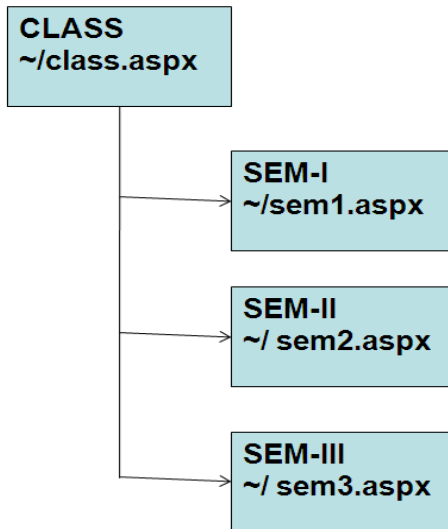


Figure : Three node in a site map

**Rule 4: Every Site Map Begins with a Single <siteMapNode>**

**Rule 5: Duplicate URLs Are Not Allowed**

### Binding an Ordinary Page to a Site Map

To bind the site Map to the page , add the SiteMapDataSource control to your page. You can drag and drop it from the Data tab of the Toolbox. It creates a tag like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
```

SiteMapDataSource - SiteMapDataSource1

The SiteMapDataSource control appears as a gray box on your page in Visual Studio, but it's invisible when you run the page.

The last step is to add controls that are linked to the SiteMapDataSource

These are the three navigation controls:

**TreeView:** The TreeView displays a “tree” of grouped links that shows your whole site map at a look.

**Menu:** The Menu displays a multilevel menu. By default, you'll see only the first level, but other levels pop up when you move the mouse over the subheadings.

**SiteMapPath:** The SiteMapPath is the simplest navigation control—it displays the full path you need to take through the site map to get to the current page. For example, it might show

CLASS > SEM I > US01CBCA03

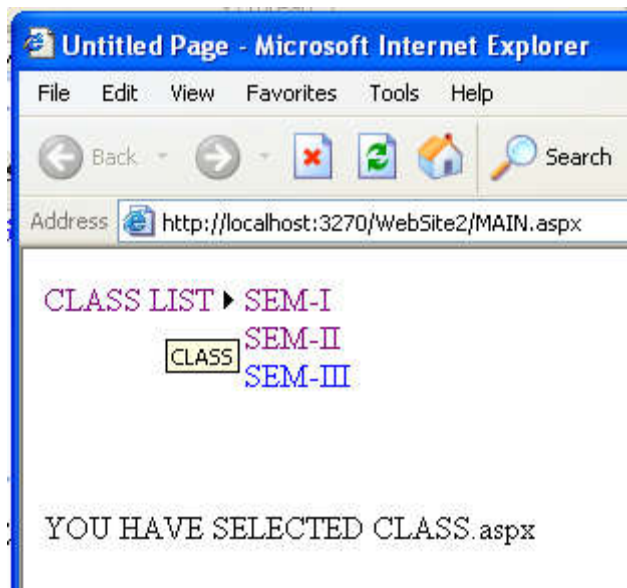
To connect a control to the SiteMapDataSource, you simply need to set its DataSourceIDproperty to match the name of the SiteMapDataSource.

For example, if you added a TreeView, you should tweak the tag so it looks like this:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
</asp:TreeView>
```

For Menu

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1">
</asp:Menu>
```



### Binding a Master Page to a Site Map

Website navigation works best when combined with another ASP.NET feature—master pages. That’s because you’ll usually want to show the same navigation controls on every page. The easiest way to do this is to create a master page that includes the SiteMapDataSource and the navigation controls. You can then reuse this template for every other page on your site.

Here’s how you might define a basic structure in your master page that puts navigation controls on the left:

```
<%@ Master Language="VB" CodeFile="MasterPage.master.vb" Inherits="MasterPage" %>
<html>
  <head runat="server">
    <title>Navigation Test</title>
  </head>
  <body>
    <form id="form1" runat="server">
      <table>
        <tr>
```

```

<td style="width: 226px;vertical-align: top;">
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1" />
</td>

<td style="vertical-align: top;">
<asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server" />
</td>
</tr>
</table>
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</form>
</body>
</html>

```

Then, create a child with some simple static content:

### **Using Different Site Maps in the Same File**

Imagine you want to have a dealer section and an employee section on your website. You might split this into two structures and define them both under different branches in the same file, like this:

```

<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
<siteMapNode title="Root" description="Root" url="~/default.aspx">
<siteMapNode title="Dealer Home" description="Dealer Home"
url="~/default_dealer.aspx">
...
</siteMapNode>
<siteMapNode title="Employee Home" description="Employee Home"
url="~/default_employee.aspx">
...
</siteMapNode>
</siteMapNode>
</siteMap>

```

To bind the SiteMapDataSource to the dealer view (which starts at the Dealer Home page), you simply set the StartingNodeUrl property to “~/default\_dealer.aspx”.

You can even make your life easier by breaking a single site map into separate files using the siteMapFile attribute, like this:

```

<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
<siteMapNode title="Root" description="Root" url="~/default.aspx">
<siteMapNode siteMapFile="Dealers.sitemap" />
<siteMapNode siteMapFile="Employees.sitemap" />
</siteMapNode>
</siteMap>

```

### **SiteMapNode Navigational Properties**

#### **Property Description**

ParentNode	Returns the node one level up in the navigation hierarchy, which contains the current node. On the root node, this returns a null reference.
------------	--

ChildNodes	Provides a collection of all the child nodes. You can check the HasChildNodes property to determine whether child nodes exist.
PreviousSibling	Returns the previous node that's at the same level (or a null reference if no such node exists).
NextSibling	Returns the next node that's at the same level (or a null reference if no such node exists).

To see this in action, consider the following code, which configures two labels on a page to show the heading and description information retrieved from the current node:

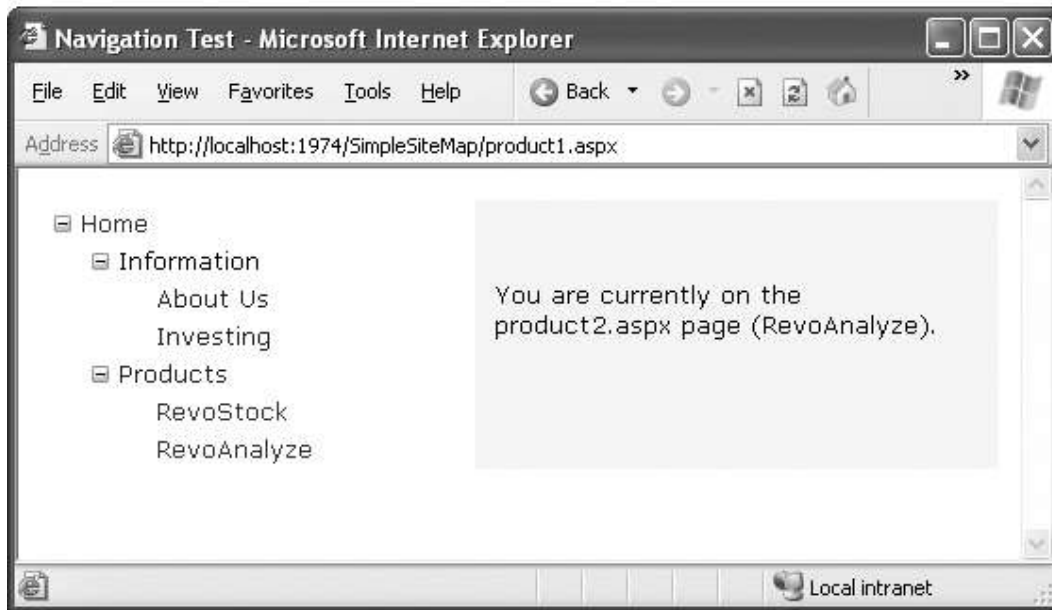
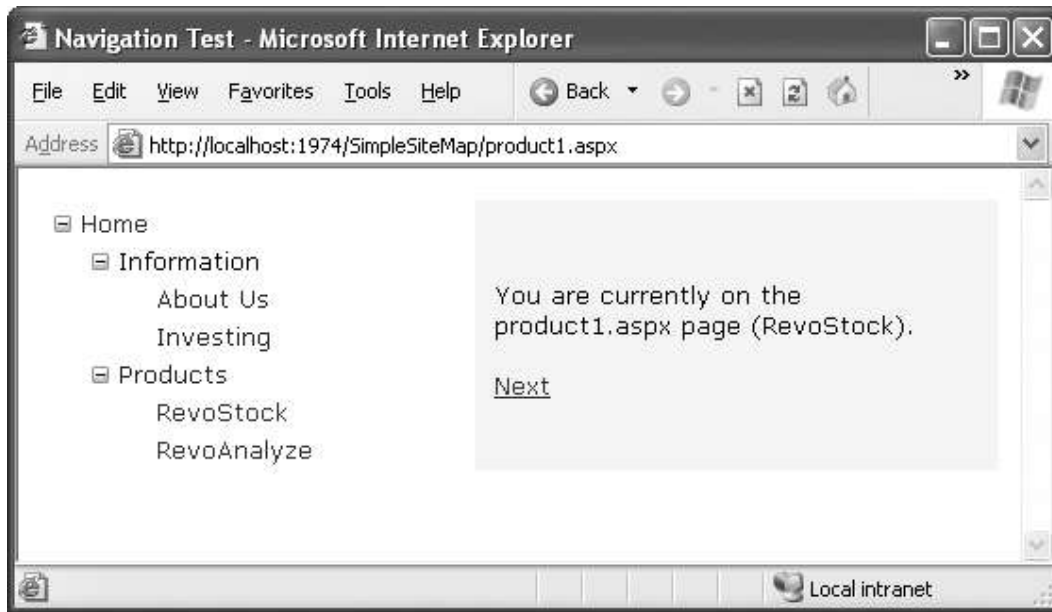
```
protected void Page_Load(object sender, EventArgs e)
{
    lblHead.Text = SiteMap.CurrentNode.Title;
    lblDescription.Text = SiteMap.CurrentNode.Description;
}
```

If you're using master pages, you could place this code in the code-behind for your master page, so that every page is assigned its title from the site map.

The next example is a little more ambitious. It implements a Previous/Next set of links, allowing the user to traverse an entire set of subnodes. The code checks for the existence of sibling nodes, and if there aren't any in the required position, it simply hides the links:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (SiteMap.CurrentNode.NextSibling != null)
    {
        lnkNext.NavigateUrl = SiteMap.CurrentNode.NextSibling.Url;
        lnkNext.Visible = true;
    }
    else
        lnkNext.Visible = false;
}
```

The first picture shows the Next link on the product1.aspx page. The second picture shows how this link disappears when you navigate to product2.aspx (either by clicking the Next link or the RevoAnalyze link in the TreeView).



## Mapping URLs

You define URL mapping in the `<urlMappings>` section of the `web.config` file. You supply two pieces of information—the request URL (as the attribute `url`) and the new destination URL (`mappedUrl`). Here's an example:

```
<configuration>
<system.web>
  <urlMappings enabled="true">
    <add url="~/category.aspx" mappedUrl="~/default.aspx?category=default" />
    <add url="~/software.aspx" mappedUrl="~/default.aspx?category=software" />
  </urlMappings>
</system.web>
</configuration>
```



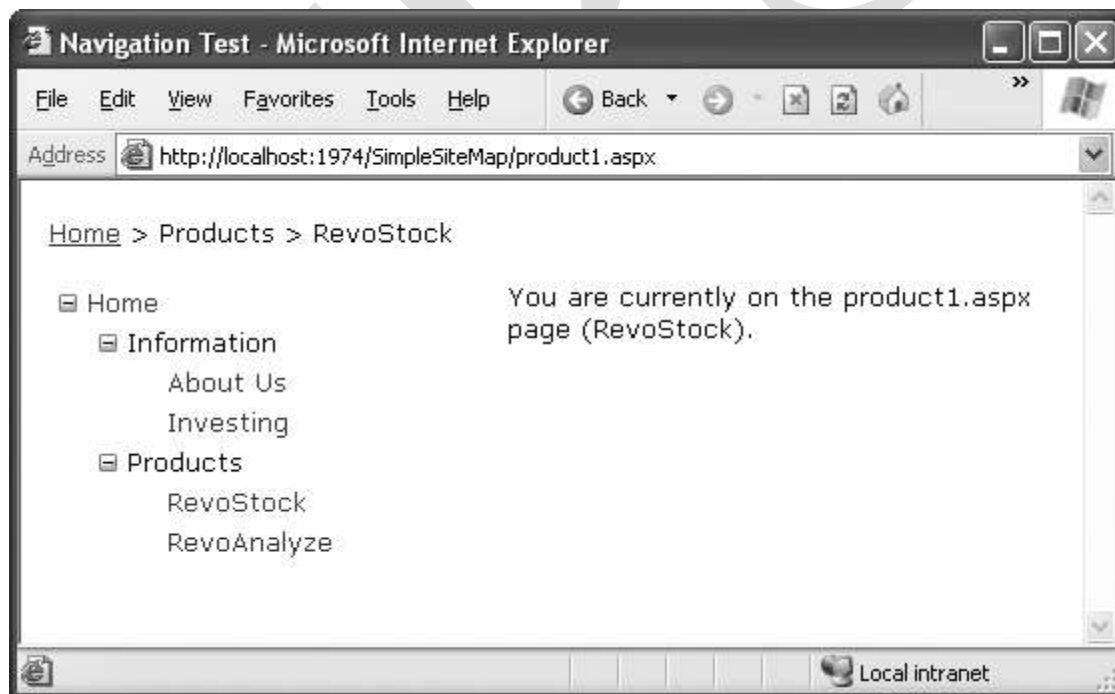
```
</urlMappings>
...
.....
.....
</system.web>
</configuration>
```

## The SiteMapPath Control

The TreeView shows the available pages, but it doesn't indicate where you're currently positioned. To solve this problem, it's common to use the TreeView in conjunction with the SiteMapPath control. Because the SiteMapPath is always used for displaying navigational information (unlike the TreeView, which can also show other types of data), you don't even need to explicitly link it to the SiteMapDataSource:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" />
```

The SiteMapPath provides ***breadcrumb navigation***, which means it shows the user's current location and allows the user to navigate up the hierarchy to a higher level using links. Following Figure shows an example with a SiteMapPath control when the user is on the product1.aspx page. Using the SiteMapPath control, the user can return to the default.aspx page.



*Breadcrumb navigation with SiteMapPath*

**SiteMapPath Appearance-Related Properties****Property Description**

ShowToolTips	Set this to False if you don't want the description text to appear when the user hovers over a part of the site map path.
ParentLevelsDisplayed	This sets the maximum number of levels above the current page that will be shown at once. By default, this setting is -1, which means all levels will be shown
RenderCurrentNodeAsLink	If True, the portion of the page that indicates the current page is turned into a clickable link. By default, this is False because the user is already at the current page.
PathDirection	You have two choices: RootToCurrent (the default) and CurrentToRoot (which reverses the order of levels in the path).
PathSeparator	This indicates the characters that will be placed between each level in the path. The default is the greater-than symbol (>). Another common path separator is the colon (:).

**Using SiteMapPath Styles and Templates**

Style	Template	Applies To
NodeStyle	NodeTemplate	All parts of the path except the root and current node
CurrentNodeStyle	CurrentNodeTemplate	The node representing the current page.
RootNodeStyle	RootNodeTemplate	The node representing the root. If the root node is the same as the current node, the current node template or styles are used.
PathSeparatorStyle	PathSeparatorTemplate	The separator in between each node

Imagine you want to change how the current node is displayed so that it's shown in italics. To get the name of the current node, you need to write a data-binding expression that retrieves the title. This data-binding expression is bracketed between <%# and %> characters and uses a method named Eval() to retrieve information from a SiteMapNode object that represents a page.

Here's what the template looks like:

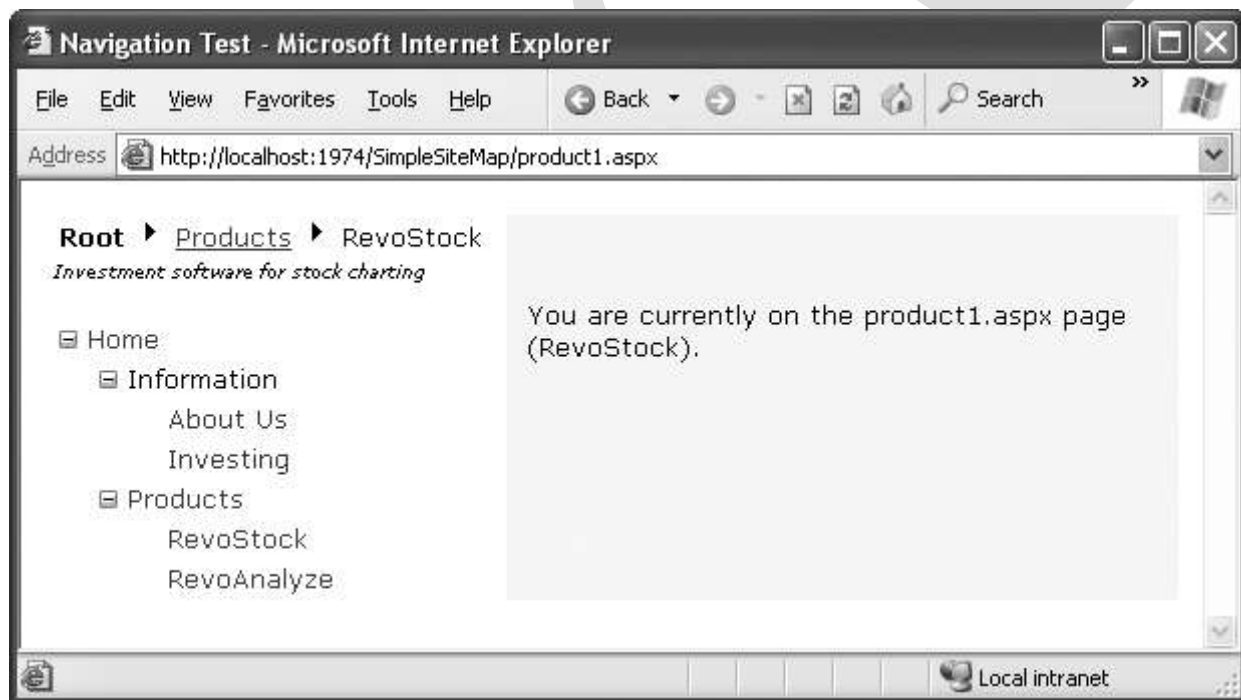
```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
<CurrentNodeTemplate>
    <i><%# Eval("Title") %></i>
</CurrentNodeTemplate>
</asp:SiteMapPath>
```

Data binding also gives you the ability to retrieve other information from the site map node, such as the description. Consider the following example:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
<PathSeparatorTemplate>
    <asp:Image ID="Image1" ImageUrl="~/arrowright.gif" runat="server" />
</PathSeparatorTemplate>

<RootNodeTemplate>
    <b>Root</b>
</RootNodeTemplate>

<CurrentNodeTemplate>
    <%# Eval("Title") %> <br />
    <small><i><%# Eval("Description") %></i></small>
</CurrentNodeTemplate>
</asp:SiteMapPath>
```



*A SiteMapPath with templates*

## The TreeView Control

the TreeView can show a portion of the full site map or the entire site map. Each node becomes a link that, when clicked, takes the user to the new page. If you hover over a link, you'll see the corresponding description information appear in a tooltip.

## TreeView Properties

### Useful TreeView Properties

Property	Description
MaxDataBindDepth	Determines how many levels the TreeView will show. By default, MaxDataBindDepth is -1, and you'll see the entire tree. However, if you use a value such as 2, you'll see only two levels under the starting node. This can help you pare down the display of long, multileveled site maps.
ExpandDepth	Lets you specify how many levels of nodes will be visible at first. If you use 0, the TreeView begins completely closed. If you use 1, only the first level is expanded, and so on. By default, ExpandDepth is set to the constant FullyExpand (-1), which means the tree is fully expanded and all the nodes are visible on the page.
NodeIndent	Sets the number of pixels between each level of nodes in the TreeView. Set this to 0 to create a nonindented TreeView, which saves space. A nonindented TreeView allows you to emulate an in-place menu (see, for example, Figure 14-12).
ImageSet	Lets you use a predefined collection of node images for collapsed, expanded, and nonexpandable nodes. You specify one of the values in the TreeViewImageSet enumeration. You can override any node images you want to change by setting the CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl properties.
CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl	Sets the pictures that are shown next to nodes for collapsed nodes (CollapseImageUrl) and expanded nodes (ExpandImageUrl). The NoExpandImageUrl is used if the node doesn't have any children. If you don't want to create your own custom node images, you can use the ImageSet property instead to use one of several built-in image collections.
NodeWrap	Lets a node text wrap over more than one line when set to True.
ShowExpandCollapse	Hides the expand/collapse boxes when set to False. This isn't recommended, because the user won't have a way to expand or collapse a level without clicking it (which causes the browser to navigate to the page).
ShowLines	Adds lines that connect every node when set to True.
ShowCheckBoxes	Shows a check box next to every node when set to True. This isn't terribly useful for site maps, but it is useful with other types of trees.

### TreeView Styles

Styles are represented by the TreeNodeStyle class, which derives from the more conventional Style class.

**Table : TreeNodeStyle-Added Properties**

Property	Description
ImageUrl	The URL for the image shown next to the node.
NodeSpacing	The space (in pixels) between the current node and the node above and below.

VerticalPadding	The space (in pixels) between the top and bottom of the node text and border around the text.
HorizontalPadding	The space (in pixels) between the left and right of the node text and border around the text.
ChildNodesPadding	The space (in pixels) between the last child node of an expanded parent node and the following node (for example, between the Investing and Products nodes in above (sitemap template) Figure ).

## Applying Styles to Node Types

The TreeView allows you to individually control the styles for types of nodes—for example, root nodes, nodes that contain other nodes, selected nodes, and so on. Table lists different TreeView styles and explains what nodes they affect.

**Table: TreeView Style Properties**

Property	Description
NodeStyle	Applies to all nodes. The other styles may override some or all of the details that are specified in the NodeStyle.
RootNodeStyle	Applies only to the first-level (root) node.
ParentNodeStyle	Applies to any node that contains other nodes, except root nodes.
LeafNodeStyle	Applies to any node that doesn't contain child nodes and isn't a root node.
SelectedNodeStyle	Applies to the currently selected node.
HoverNodeStyle	Applies to the node the user is hovering over with the mouse. These settings are applied only in up-level clients that support the necessary dynamic script.

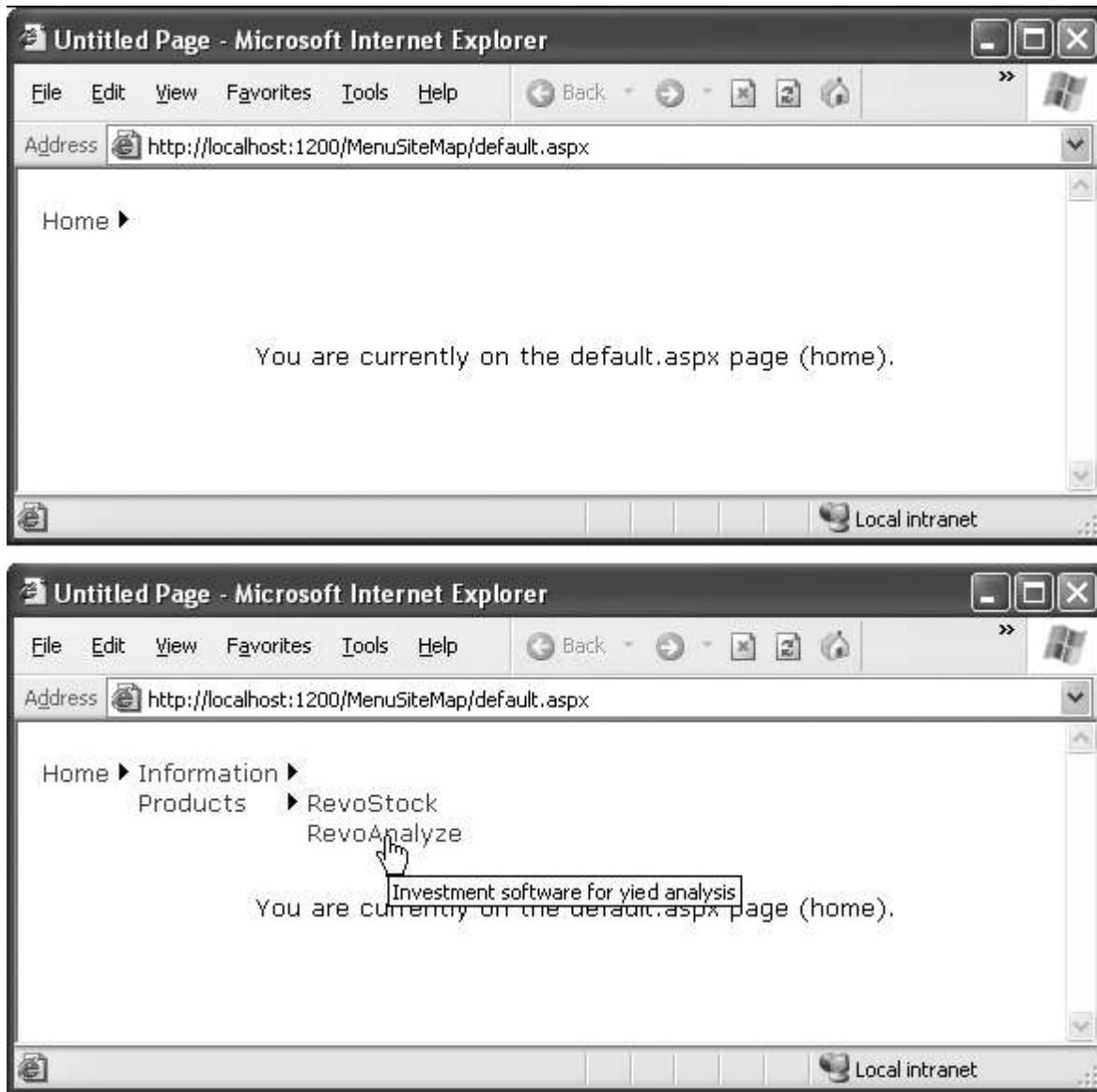
### **For example**

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
  <NodeStyle Font-Names="Tahoma" Font-Size="10pt" ForeColor="Blue"
    HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
  <ParentNodeStyle Font-Bold="False" />
  <HoverNodeStyle Font-Underline="True" ForeColor="#5555DD" />
  <SelectedNodeStyle Font-Underline="True" ForeColor="#5555DD" />
</asp:TreeView>
```

## The Menu Control

The Menu control is another rich control that supports hierarchical data. Like the TreeView, you can bind the Menu control to a data source, or you can fill it by hand using MenuItem objects. To try the Menu control, remove the TreeView from your master page, and add the following Menu control tag:

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1" />
```



*Navigating through the menu*

#### **Difference between tree view and Menu as follow:**

- The Menu displays a single submenu. The TreeView can expand an arbitrary number of node branches at a time.
- The Menu displays a root level of links in the page. All other items are displayed using fly-out menus that appear over any other content on the page. The TreeView shows all its items inline in the page.
- The Menu supports templates. The TreeView does not. (Menu templates are discussed later in this section.)
- The TreeView supports check boxes for any node. The Menu does not.
- The Menu supports horizontal and vertical layouts, depending on the Orientation property. The TreeView supports only vertical layout.

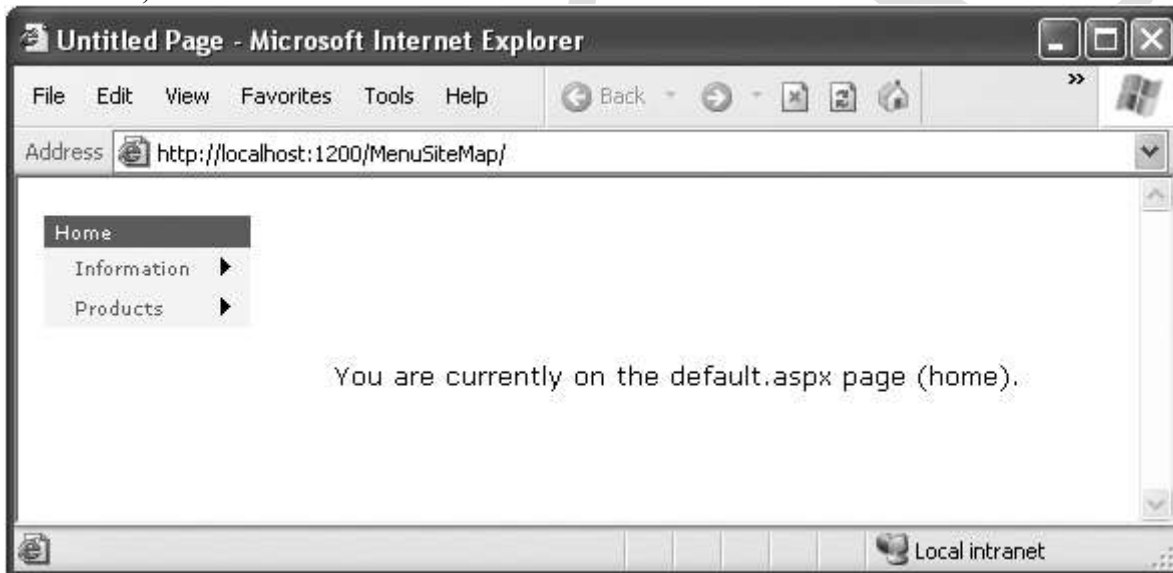
#### **Menu Styles**

The Menu supports defining different menu styles for different menu levels.

**Table Menu Styles**

Static Style	Dynamic Style	Description
StaticMenuStyle	DynamicMenuStyle	Sets the appearance of the overall “box” in which all the menu items appear. In the case of StaticMenuStyle, this box appears on the page, and with DynamicMenuStyle it appears as a pop-up.
StaticMenuItemStyle	DynamicMenuItemStyle	Sets the appearance of individual menu items
StaticSelectedStyle	DynamicSelectedStyle	Sets the appearance of the selected item. Note that the selected item isn't the item that's currently being hovered over; it's the item that was previously clicked (and that triggered the last postback)
StaticHoverStyle	DynamicHoverStyle	Sets the appearance of the item that the user is hovering over with the mouse.

Following Figure shows the menu with StaticDisplayLevels set to 2 (and some styles applied through the Auto Format link).



*A menu with two static levels*

## **Validation**

It verifies control values entered correctly and block page processing until the control values are valid. This is very critical to application security. **Validation** can be done in server side or client side.

There are two types of Validation:

- 1) Client side validation
- 2) Server side validation

**1) Client side validation:**

- When validation is done using a script (usually in the form of JavaScript) in the page that is posted to the end user's browser to perform validations on the data entered in the form before the form is posted back to the originating server. Then , client-side validation has occurred.
- Client-side validation is quick and responsive for the end user. client-side validation is the more insecure form of validation.
- When a page is generated in an end user's browser, this end user can look at the code of the page quite easily (simply by right-clicking his mouse in the browser and selecting View Code).

**2) Server side validation:**

- When validation occurs on server, where application resides it is called server side validation.
- The more secure form of validation is server-side validation.
- It is more secure because these checks cannot be easily bypassed.

**The best approach is always to perform client-side validation first and then, after the form passes and is posted to the server, to perform the validation checks again using server-side validation.**

### **Client side validation vs. server side validation**

#### **Server-Side Validation**

You can use the validator controls to verify a page automatically when the user submits it or manually in your code. The first approach is the most common. When using automatic validation, the user receives a normal page and begins to fill in the input controls. When finished, the user clicks a button to submit the page. Every button has a

CausesValidation property, which can be set to true or false. What happens when the user clicks the button depends on the value of the CausesValidation property:

- If CausesValidation is false, ASP.NET will ignore the validation controls, the page will be

posted back, and your event-handling code will run normally.

- If CausesValidation is true (the default), ASP.NET will automatically validate the page when the user clicks the button. It does this by performing the validation for each control on the page. If any control fails to validate, ASP.NET will return the page with some error information, depending on your settings. Your click event-handling code may or may not be executed—meaning you'll have to specifically check in the event handler whether the page is valid.

#### **Client-Side Validation**

ASP.NET automatically adds JavaScript code for client-side validation. In this case, when the user clicks a CausesValidation button, the same error messages will



appear without the page needing to be submitted and returned from the server. This increases the responsiveness of your web page.

However, even if the page validates successfully on the client side, ASP.NET still revalidates it when it's received at the server. This is because it's easy for an experienced user to circumvent client-side validation. For example, a malicious user might delete the block of

JavaScript validation code and continue working with the page. By performing the validation at both ends, ASP.NET makes sure your application can be as responsive as possible while also remaining secure.

## Overview of the Validation controls

Validation server controls are used to validate user-input. A Validation server control is used to validate the data of an input control. If the data does not pass validation, it will display an error message to the user.

The syntax for creating a Validation server control is:

```
<asp:control_name id="some_id" runat="server" />
```

There are different types of validation controls:

- 1) RequiredFieldValidator control
- 2) RangeValidator control
- 3) CompareValidator control
- 4) RegularExpressionValidator control
- 5) CustomValidator control
- 6) ValidationSummary control

<u>Validation Control</u>	<u>Description</u>
<b>RequiredFieldValidator</b>	It makes sure the user enters data in the associated Data-entry control.
<b>RangeValidator</b>	It makes sure that the user-entered data passes validation criteria that you set yourself.
<b>CompareValidator</b>	It uses comparison operates to compare user-entered data to a constant value or the value in another Data-entry.
<b>RegularExpressionValidator</b>	It makes sure that the user-entered data matches a regular expression.
<b>CustomValidator</b>	It makes sure that the user-entered data passes validations criteria that you set yourself.
<b>ValidationSummary</b>	It displays the list of all the validation errors on the web page.

**Common Properties of validation controls:**

Property	Description
----------	-------------

<b>ControlToValidate</b>	<b>Gets or sets the input control to validate.</b>
Display	Gets or sets the display behavior of the error message in a validation control.
EnableClientScript	Gets or sets a value indicating whether client-side validation is enabled.
Enabled	Gets or sets a value that indicates whether the validation control is enabled.
<b>ErrorMessage</b>	<b>Gets or sets the text for the error message displayed in a <u>ValidationSummary</u> control when validation fails.</b>
ForeColor	Gets or sets the color of the message displayed when validation fails.
IsValid	Gets or sets a value that indicates whether the associated input control passes validation.
SetFocusOnError	Gets or sets a value that indicates whether focus is set to the control specified by the <u>ControlToValidate</u> property when validation fails.
<b>Text</b>	<b>Gets or sets the text displayed in the validation control when validation fails.</b>
<b>ValidationGroup</b>	<b>Gets or sets the name of the validation group to which this validation control belongs.</b>

### Common Methods of validation controls:

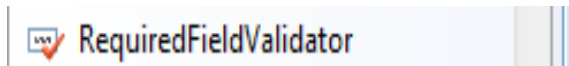
Property	Description
Validate	It performs validation on the associated input control and updates the <i>IsValid</i> property.

To use these controls, you set the *ErrorMessage* property to the error message you want to display, and the *ControlToValidate* property to the control you want to check.

**1) RequiredFieldValidator Control:** This is the simplest validation control that makes sure that the users have entered data into a Data-entry control. Suppose that the users are entering data for buying shoes in Data-entry control. In that case you may want to make sure that the users enter the number of shoes they want to buy. If they omit to enter a value, this validation control will display its error message.

This control has a *Initial value* property, which is set to an empty string (“”) by default. If the data has not changed from that value when validation occurs, the control displays its error message.

## Figure of the RequiredFieldValidator



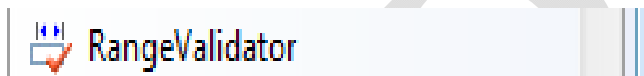
### →Properties:

Property	Description
InitialValue	Specifies the starting value of the input control. Default value is ""

## 2) RangeValidator Control:

A Range validator tests if the value of a Data-Entry control is inside a specified range of values. You use three main properties- *ControlToValidate*, *MinimumValue* and *maximumValue*. The *ControlToValidate* property contains the Data-Entry control to validate, *MinimumValue* and *MaximumValue* properties hold the minimum and maximum values of the valid range. If you set one of the *MinimumValue* and *MaximumValue* properties, you also must set the other. Also set the *Type* property to the data type of the value to compare, the possible values are the same as for comparison validators.

## Figure of the RangeValidator



### →Properties:

Property	Description
MaximumValue	Specifies the maximum value of the input control
MinimumValue	Specifies the minimum value of the input control
Type	Specifies the data type of the value to check. The types are: <ul style="list-style-type: none"> <li>• Currency</li> <li>• Date</li> <li>• Double</li> <li>• Integer</li> <li>• String</li> </ul>

## 3) CompareValidator Control:

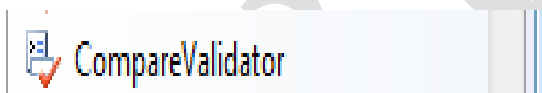
A CompareValidator compares the value entered by the user into a Data-Entry control with the value entered into another Data-Entry control or with a constant value. As usual for validation controls, you indicate the Data-Entry control to validate by setting the ControlToValidate property. If you want to compare a specific Data-Entry control to another, set the *ControlToCompare* property to specify the control to compare with.

You can also compare Date-Entry value to the constant value, for that you have to set the *ValueToCompare* property.

Use the Type property to specify the type of comparison to perform. Here are the possibilities:

<b>Operator</b>	<b>Discription</b>
Equal	Checks if the compared value are equal.
Not Equal	Checks if the compared value are not equal.
GreaterThan	Checks for the greater than relationship.
GreaterThanEqual	Checks for the greater than or equal relationship.
LessThan	Checks for the less than relationship.
LessThanEqual	Checks for the less than or equal relationship.
DataTypeCheck	Compares data types between the value enters into the Data-Entry control being validated and the data type specified by the Type property

### Figure of the CompareValidator



#### →Properties:

<b>Property</b>	<b>Description</b>
ControlToCompare	The name of the control to compare with
Operator	The type of comparison to perform. The operators are: <ul style="list-style-type: none"> <li>• Equal</li> <li>• GreaterThan</li> <li>• GreaterThanEqual</li> <li>• LessThan</li> <li>• LessThanEqual</li> <li>• NotEqual</li> <li>• DataTypeCheck</li> </ul>

ValueToCompare	A specified value to compare with
----------------	-----------------------------------

#### 4) RegularExpressionValidator control:

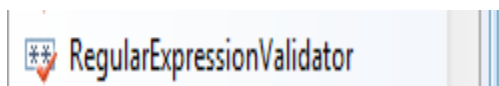
A RegularExpresson validator control is used to check if the value in Data-Entry control matches a pattern defined by a regular expression. You use regular expression to see if the text matches a certain pattern, which is a great way to check if the user has entered text in the way you want.

In general regular expressions are made up of text with embedded codes that start with a back slash (\) as well as other control codes. For Example:

```
\b[A-Za-z]+\b
```

The code for a word boundary is \b and a 'character class' is a set of characters surrounded with '[' and ']' that lets you specify what characters you want to accept. So this regular expression will match a word made up of uppercase and/or lowercase letters here.

#### Figure of the RegularExpressionValidator



#### →Properties:


Property	Description
ValidationExpression	Specifies the expression used to validate input control. The expression validation syntax is different on the client than on the server. JScript is used on the client. On the server, the language you have specified is used

#### 5) CustomValidator control:

With a custom validator, you set the ClientValidationfunction property to the names of a script function, such as Javascript or VBscript function. This function will pass two arguments- sources and arguments, source, gives the source control to validate, and arguments, hold data to validate as asguments.Value. if you validate the data, you set arguments.IsValid to TRUE else to FALSE.

Using custom validator is perhaps the most powerful way to use validators. Existing beyond the simple range checking and field checking validators, custom validators also let you write your own customization code.

#### Figure of the CustomValidator

 CustomValidator

### →Properties:

Property	Description
ClientValidationFunction	<p>Specifies the name of the client-side validation script function to be executed.</p> <p><b>Note:</b> The script must be in a language that the browser supports, such as VBScript or JScript</p> <p>With VBScript, the function must be in the form:</p> <p>Sub FunctionName (source, arguments)</p> <p>With JScript, the function must be in the form:</p> <p>Function FunctionName (source, arguments)</p>
ValidateEmptyText	Sets a Boolean value indicating whether empty text should be validated.

### →Events:

Property	Description
ServerValidate	It occurs when validation takes place on the server.

## 6) ValidationSummary control:

The Validationsummary control, which summarize the error messages from all validators on a web page in one location. The summary can be displayed as a list, as a bulleted list, or as a single paragraph, based on the DisplayMode property. You can also specify if the summary should be displayed in the web page and in a message box by setting the ShowSummary and showMessagebox properties, respectively.

### Figure of the ValidationSummary

 ValidationSummary

### →Properties:

Property	Description
DisplayMode	How to display the summary. Legal values are:

	<ul style="list-style-type: none"><li>• BulletList</li><li>• List</li><li>• SingleParagraph</li></ul>
EnableClientScript	A Boolean value that specifies whether client-side validation is enabled or not
Enabled	A Boolean value that specifies whether the validation control is enabled or not
ForeColor	The fore color of the control
HeaderText	A header in the ValidationSummary control
ShowMessageBox	A Boolean value that specifies whether the summary should be displayed in a message box or not
ShowSummary	A Boolean value that specifies whether the ValidationSummary control should be displayed or hidden
ValidationGroup	Sets the group of controls for which the validationSummary object displays validation messages.

VP