[Unit 4/5: Server side scripting with ASP.NET]

Web Technology (CSC-353)

Jagdish Bhatta

Central Department of Computer Science & Information Technology

Tribhuvan University

.NET an Overview:

ASP.net and VB.net

ASP.NET stands for Active Server Pages .NET, and VB.NET stands for Visual Basic.NET. VB.NET, put simply, is a programming language, and ASP.NET is a technology used to render dynamic web content. An ASP.NET web site is typically made up of code written in either VB.NET or C# (C Sharp). When creating a web site with VB.NET, you are actually creating an ASP.NET application *using* VB.NET. This is different from a traditional Active Server Page (ASP) page, in that an ASP.NET application is written using fully-featured programming languages with full functionality, like VB.NET, instead of scripting languages like Visual Basic Script (VBScript).

Microsoft .net

Microsoft .NET is a package of software that consists of clients, servers, and development tools. This package includes the Microsoft .NET Framework, development tools such as Visual Studio 2008, a set of server applications such as Microsoft Windows Server 2003 and Microsoft SQL Server, and client-side applications such as Windows XP and Microsoft Office. Microsoft .NET Framework includes many other subcomponents that allow software that has been written in different languages to work together by establishing rules for language independence. Using the Microsoft .NET Framework as a base, software development toolmakers can create development tools for different languages such as COBOL or C++. Microsoft itself used the .NET Framework to create VS, which is a development tool used to create software using the VB or C# programming languages.

The Microsoft .NET Framework also provides many common functions that previously needed to be built by the developer. This includes access to the file system, access to the registry, and easier development when using the Windows Application Programming Interfaces (API) to access operating system–level functionality. This allows the developer to concentrate more on business problems, instead of worrying how to access low-level windows functionality.

The Common Language Runtime

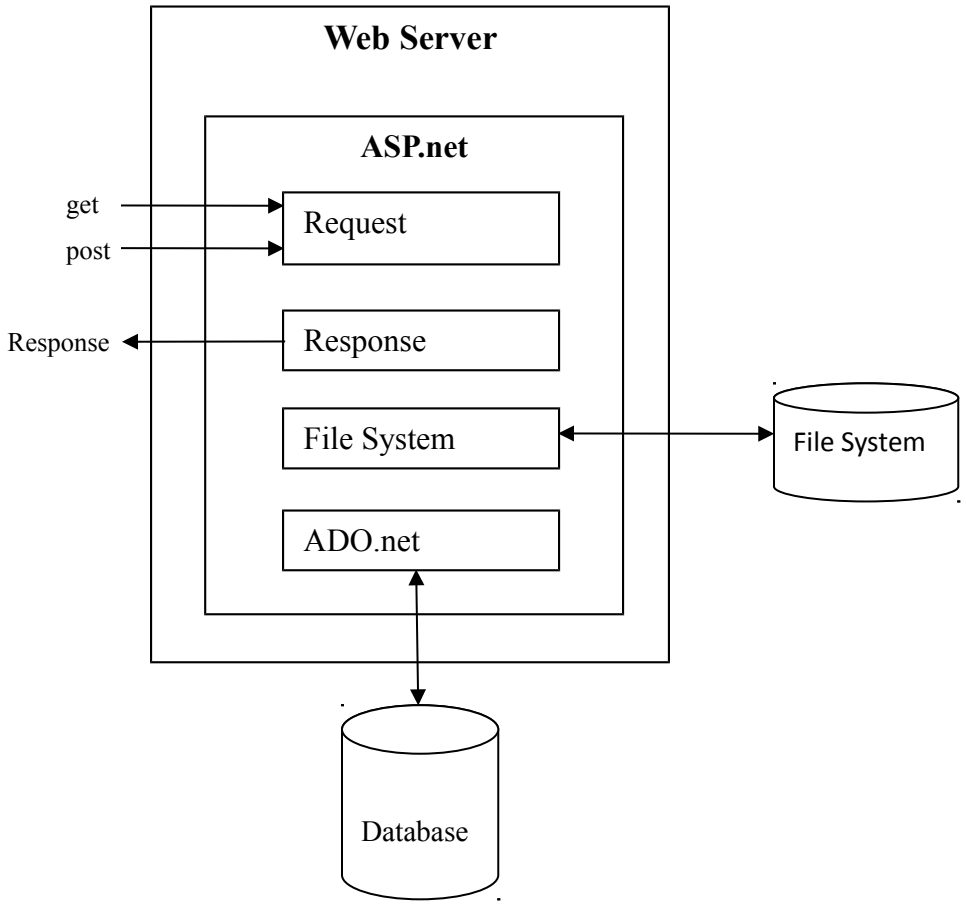
The Microsoft **Common Language Runtime** (CLR) is one of the components within the .NET Framework. The **CLR provides runtime services, including loading and execution of code**. The CLR essentially takes the language-specific code that was written and translates it **Microsoft Intermediate Language** (MSIL) code. The resulting code is the same, no matter what language the original code was written in. This is what allows code written with VB to work with code written in C#. This is also the most important aspect of the .NET Framework for a software development company, because one developer can write code in VB and another developer can write code with C#, but the application will still work without a problem, allowing companies to use their existing skill sets. Without this framework and the MSIL, an entire application would need to be built using the same language. This would require a software development company to have a full staff of developers that know a specific development language, such as VB. A single program, written in multiple languages, works mainly because the framework contains a set of common data types that must be used by all languages building applications with the .NET Framework. This set of data types is the **Common Type System (CTS)**, which defines how types are declared, used, and managed. To accommodate the CLR, some of the data types within languages such as VB needed to be changed so they could work better with data types from other languages such as C++.

Assemblies

An assembly is the main component of a .NET Framework application and is a collection of all of the functionality for the particular application. The assembly is created as either a .dll file for web sites or an .exe file for Windows applications, and it contains all of the MSIL code to be used by the framework. Without the assembly there is no application. The creation of an assembly is automatically performed by VS. It is possible to create applications for the .NET Framework without VS—however, you need to use the various tools that come with the .NET Framework Software Development Kit (SDK) to create the assemblies and perform other tasks that are automatically done by VS.

How Web Servers Execute ASP Files

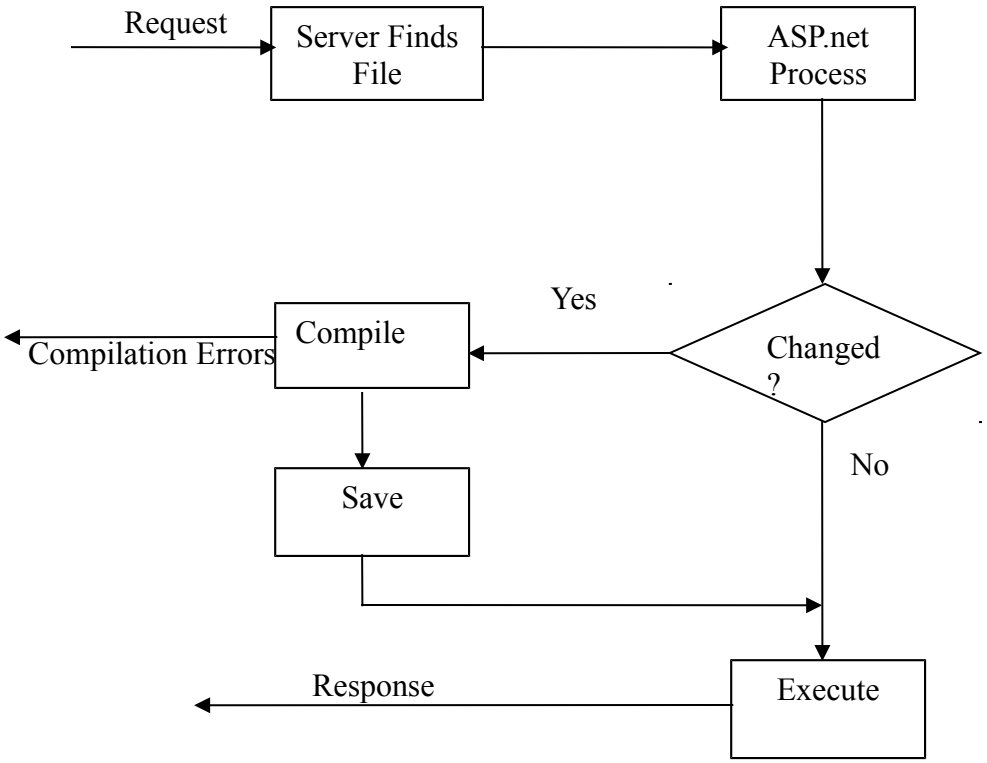
When a site visitor requests a Web page address, the browser contacts the Web server specified in the address URL and makes a request for the page by formulating a HTTP request, which is sent to the Web server. The Web server on receiving the request determines the file type requested and passes processing to the appropriate handler. ASP.NET files are compiled, if necessary, into .NET Page classes and then executed, with the results sent to the client's browser. Compilation means that on first load ASP.NET applications take longer to display than previous versions of ASP, but once compiled they are noticeably faster. The browser can request information from and send information to the server using two HTTP methods, GET and POST.



When the server receives this request, it will find the page that was requested using the path information specified, and the relevant system will process the page. When the response is complete, it is flushed back out to the user's browser, usually as HTML but not necessarily, and

the browser renders this page as it arrives as the page on screen. The process of compiling and delivering ASP.NET pages goes through the following stages:

1. IIS matches the URL in the request against a file on the physical file system (hard disk) by translating the virtual path (for example, /site/ index.aspx) into a path relative to the site's Web root (for example, d:\domains\thisSite\wwwroot\site\index.aspx).
2. Once the file is found, the file extension (.aspx) is matched against a list of known file types for either sending on to the visitor or for processing.
3. If this is first visit to the page since the file was last changed, the ASP code is compiled into an assembly using the Common Language Runtime compiler, into MSIL, and then into machine-specific binary code for execution.
4. The binary code is a .NET class .dll and is stored in a temporary location.
5. Next time the page is requested the server will check to see if the code has changed. If the code is the same, then the compilation step is skipped and the previously compiled class code is executed; otherwise, the class is deleted and recompiled from the new source.
6. The compiled code is executed and the request values are interpreted, such as form input fields or URL parameters.
7. If the developer has used Web forms, then the server can detect what software the visitor is using and render pages that are tailored to the visitors' requirements, for example, returning Netscape specific code, or Wireless Markup Language (WML) code for mobiles.
8. Any results are delivered back to the visitor's Web browser.
9. Form elements are converted into client side markup and script, HTML and JavaScript for Web browsers, and WML and WMLScript for mobiles, for example.



What is Classic ASP and ASP.net?

Microsoft's previous server side scripting technology ASP (Active Server Pages) is now often called classic ASP. ASP 3.0 was the last version of classic ASP. ASP.NET is the next generation ASP, but it's not an upgraded version of ASP. ASP.NET is an entirely new technology for server-side scripting. It was written from the ground up and is not backward compatible with classic ASP. ASP.NET is a server side scripting technology that enables scripts (embedded in web pages) to be executed by an Internet server.

- ASP.NET is a Microsoft Technology
- ASP stands for Active Server Pages
- ASP.NET is a program that runs inside IIS
- IIS (Internet Information Services) is Microsoft's Internet server
- IIS comes as a free component with Windows servers
- IIS is also a part of Windows 2000 and XP Professional

The .NET Framework consists of 3 main parts:

Programming languages:

- C# (Pronounced C sharp)
- Visual Basic (VB .NET)
- J# (Pronounced J sharp)

Server technologies and client technologies:

- ASP .NET (Active Server Pages)
- Windows Forms (Windows desktop solutions)
- Compact Framework (PDA / Mobile solutions)

Development environments:

- Visual Studio .NET (VS .NET)
- Visual Web Developer

Features of ASP.net:

- **ASP.NET Controls:** ASP.NET contains a large set of HTML controls. Almost all HTML elements on a page can be defined as ASP.NET control objects that can be controlled by scripts. ASP.NET also contains a new set of object-oriented input controls, like programmable list-boxes and validation controls. A new data grid control supports sorting, data paging, and everything you can expect from a dataset control.
- **Event Aware Controls:** All ASP.NET objects on a Web page can expose events that can be processed by ASP.NET code. Load, Click and Change events handled by code makes coding much simpler and much better organized.
- **ASP.NET Components:** ASP.NET components are heavily based on XML.
- **User Authentication:** ASP.NET supports form-based user authentication, cookie management, and automatic redirecting of unauthorized logins.
- **User Accounts and Roles:** ASP.NET allows user accounts and roles, to give each user (with a given role) access to different server code and executables.
- **High Scalability:** Much has been done with ASP.NET to provide greater scalability. Server-to-server communication has been greatly enhanced, making it possible to scale an application over several servers. One example of this is the ability to run XML parsers, XSL transformations and even resource hungry session objects on other servers.

- **Compiled Code:** The first request for an ASP.NET page on the server will compile the ASP.NET code and keep a cached copy in memory. The result of this is greatly increased performance.
- **Easy Configuration:** Configuration of ASP.NET is done with plain text files. Configuration files can be uploaded or changed while the application is running. No need to restart the server. No more metabase or registry puzzle.
- **Easy Deployment:** No more server-restart to deploy or replace compiled code. ASP.NET simply redirects all new requests to the new code.
- **Compatibility:** ASP.NET is not fully compatible with earlier versions of ASP, so most of the old ASP code will need some changes to run under ASP.NET. To overcome this problem, ASP.NET uses a new file extension ".aspx". This will make ASP.NET applications able to run side by side with standard ASP applications on the same server.

Creating ASP.NET Application

Simple HTML page that will display "Hello W3Schools" in an Internet browser can be written like this:

```
<html>
        <body bgcolor="yellow">
        <center>
        <h2>Hello W3Schools!</h2>
        </center>
        </body>
</html>
```

The simplest way to convert an HTML page into an ASP.NET page is to copy the HTML file to a new file with an **.aspx** extension

```
<html>
        <body bgcolor="yellow">
        <center>
        <h2>Hello W3Schools!</h2>
        </center>
        </body>
```

```
</html>
```

Fundamentally an ASP.NET page is just the same as an HTML page. An HTML page has the extension .htm. If a browser requests an HTML page from the server, the server sends the page to the browser without any modifications. An ASP.NET page has the extension .aspx. If a browser requests an ASP.NET page, the server processes any executable code in the page, before the result is sent back to the browser. The ASP.NET page above does not contain any executable code, so nothing is executed. In the next examples we will add some executable code to the page to demonstrate the difference between static HTML pages and dynamic ASP pages.

Dynamic Page in Classic ASP and ASP.net

To demonstrate how ASP can display pages with dynamic content, we have added some executable code (in red) to the previous example:

```
<html>
    <body bgcolor="yellow">
    <center>
    <h2>Hello W3Schools!</h2>
    <p><%Response.Write(now())%></p>
    </center>
    </body>
</html>
```

The code inside the <% --%> tags is executed on the server. Response.Write is ASP code for writing something to the HTML output stream. Now() is a function returning the servers current date and time. This same code can also be used as ASP.NET page. The code above illustrates a limitation in Classic ASP: The code block has to be placed where you want the output to appear. With Classic ASP it is impossible to separate executable code from the HTML itself. This makes the page difficult to read, and difficult to maintain.

ASP.NET - Server Controls

ASP.NET has solved the "spaghetti-code" problem described above with server controls. Server controls are tags that are understood by the server. There are three kinds of server controls:

- HTML Server Controls - Traditional HTML tags
- Web Server Controls - New ASP.NET tags
- Validation Server Controls - For input validation

ASP.NET - HTML Server Controls

HTML server controls are HTML tags understood by the server. HTML elements in ASP.NET files are, by default, treated as text. To make these elements programmable, add a runat="server" attribute to the HTML element. This attribute indicates that the element should be treated as a server control. The id attribute is added to identify the server control. The id reference can be used to manipulate the server control at run time. All HTML server controls must be within a <form> tag with the runat="server" attribute. The runat="server" attribute indicates that the form should be processed on the server. It also indicates that the enclosed controls can be accessed by server scripts.

In the following example we declare an HtmlAnchor server control in an .aspx file. Then we manipulate the HRef attribute of the HtmlAnchor control in an event handler (an event handler is a subroutine that executes code for a given event). The Page_Load event is one of many events that ASP.NET understands.

```
<script runat="server">
    Sub Page_Load
    link1.HRef="http://www.w3schools.com"
    End Sub
    </script>
    <html>
    <body>
    <form runat="server">
    <a id="link1" runat="server">Visit W3Schools!</a>
    </form>
    </body>
</html>
```

The executable code itself has been moved outside the HTML.

ASP.NET - Web Server Controls

Web server controls are special ASP.NET tags understood by the server. Like HTML server controls, Web server controls are also created on the server and they require a runat="server" attribute to work. However, Web server controls do not necessarily map to any existing HTML elements and they may represent more complex elements. The syntax for creating a Web server control is:

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

In the following example we declare a Button server control in an .aspx file. Then we create an event handler for the Click event which changes the text on the button:

```
<script runat="server">
        Sub submit(Source As Object, e As EventArgs)
        button1.Text="You clicked me!"
End Sub
</script>
<html>
        <body>
        <form runat="server">
        <asp:Button id="button1" Text="Click me!" runat="server" OnClick="submit"/>
        </form>
        </body>
</html>
```

ASP.NET - Validation Server Controls

Validation server controls are used to validate user-input. If the user-input does not pass validation, it will display an error message to the user. Each validation control performs a specific type of validation (like validating against a specific value or a range of values). By default, page validation is performed when a Button, ImageButton, or LinkButton control is clicked. You can prevent validation when a button control is clicked by setting the CausesValidation property to false. The syntax for creating a Validation server control is:

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

In the following example we declare one TextBox control, one Button control, and one RangeValidator control in an .aspx file. If validation fails, the text "The value must be from 1 to 100!" will be displayed in the RangeValidator control:

```
<html>
<body>
<form runat="server">
<p>Enter a number from 1 to 100:
<asp:TextBox id="tbox1" runat="server" />
<br /><br />
<asp:Button Text="Submit" runat="server" />
</p>
<p>
<asp:RangeValidator  ControlToValidate="tbox1" MinimumValue="1"
MaximumValue="100" Type="Integer" Text="The value must be from 1 to 100!"
runat="server" />
</p>
</form>
</body>
</html>
```

ASP.NET Web Forms

All server controls must appear within a <form> tag, and the <form> tag must contain the runat="server" attribute. The runat="server" attribute indicates that the form should be processed on the server. It also indicates that the enclosed controls can be accessed by server scripts:

```
<form runat="server">
...HTML + server controls
</form>
```

The form is always submitted to the page itself. If you specify an action attribute, it is ignored. If you omit the method attribute, it will be set to method="post" by default. Also, if you do not specify the name and id attributes, they are automatically assigned by ASP.NET. An .aspx page

can only contain ONE <form runat="server"> control!. If you select view source in an .aspx page containing a form with no name, method, action, or id attribute specified, you will see that ASP.NET has added these attributes to the form. It looks something like this:

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
...some code
</form>
```

Submitting a Form

A form is most often submitted by clicking on a button. The Button server control in ASP.NET has the following format:

```
<asp:Button id="id" text="label" OnClick="sub" runat="server" />
```

The id attribute defines a unique name for the button and the text attribute assigns a label to the button. The onClick event handler specifies a named subroutine to execute. In the following example we declare a Button control in an .aspx file. A button click runs a subroutine which changes the text on the button:

Maintaining the ViewState

When a form is submitted in classic ASP, all form values are cleared. Suppose you have submitted a form with a lot of information and the server comes back with an error. You will have to go back to the form and correct the information. You click the back button, and what happens.....ALL form values are CLEARED, and you will have to start all over again! The site did not maintain your ViewState. When a form is submitted in ASP .NET, the form reappears in the browser window together with all form values. How come? This is because ASP .NET maintains your ViewState. The ViewState indicates the status of the page when submitted to the server. The status is defined through a hidden field placed on each page with a <form runat="server"> control. The source could look something like this:

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNl4Q=" />
.....some code
</form>
```

Maintaining the ViewState is the default setting for ASP.NET Web Forms. If you want to NOT maintain the ViewState, include the directive <%@ Page EnableViewState="false" %> at the top of an .aspx page or add the attribute EnableViewState="false" to any control. Look at the following .aspx file. It demonstrates the "old" way to do it. When you click on the submit button, the form value will disappear:

```
<html>
<body>
<form action="demo_classicasp.aspx" method="post">
Your name: <input type="text" name="fname" size="20">
<input type="submit" value="Submit">
</form>
<%
dim fname
fname=Request.Form("fname")
If fname<>"" Then
Response.Write("Hello " & fname & "!")
End If
%>
</body>
</html>
```

Here is the new ASP .NET way. When you click on the submit button, the form value will NOT disappear:

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Hello " & txt1.Text & "!"
End Sub
</script>
<html>
<body>
<form runat="server">
Your name: <asp:TextBox id="txt1" runat="server" />
```

```
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The TextBox Control

The TextBox control is used to create a text box where the user can input text. The TextBox control's attributes and properties are listed in our web controls reference page. The example below demonstrates some of the attributes you may use with the TextBox control:

```
<html>
<body>
<form runat="server">
A basic TextBox:
<asp:TextBox id="tb1" runat="server" />
<br /><br />
A password TextBox:
<asp:TextBox id="tb2" TextMode="password" runat="server" />
<br /><br />
A TextBox with text:
<asp:TextBox id="tb4" Text="Hello World!" runat="server" />
<br /><br />
A multiline TextBox:
<asp:TextBox id="tb3" TextMode="multiline" runat="server" />
<br /><br />
A TextBox with height:
<asp:TextBox id="tb6" rows="5" TextMode="multiline"
runat="server" />
<br /><br />
A TextBox with width:
<asp:TextBox id="tb5" columns="30" runat="server" />
```

```
</form>
</body>
</html>
```

Add a Script

The contents and settings of a TextBox control may be changed by server scripts when a form is submitted. A form can be submitted by clicking on a button or when a user changes the value in the TextBox control. In the following example we declare one TextBox control, one Button control, and one Label control in an .aspx file. When the submit button is triggered, the submit subroutine is executed. The submit subroutine writes a text to the Label control:

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>
<html>
<body>
<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

In the following example we declare one TextBox control and one Label control in an .aspx file. When you change the value in the TextBox and either click outside the TextBox or press the Tab key, the change subroutine is executed. The submit subroutine writes a text to the Label control:

```
<script runat="server">
Sub change(sender As Object, e As EventArgs)
lbl1.Text="You changed text to " & txt1.Text
```

```
End Sub
</script>
<html>
<body>
<form runat="server">
Enter your name:
<asp:TextBox id="txt1" runat="server"
text="Hello World!"
ontextchanged="change" autopostback="true"/>
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The Button Control

The Button control is used to display a push button. The push button may be a submit button or a command button. By default, this control is a submit button. A submit button does not have a command name and it posts the page back to the server when it is clicked. It is possible to write an event handler to control the actions performed when the submit button is clicked. A command button has a command name and allows you to create multiple Button controls on a page. It is possible to write an event handler to control the actions performed when the command button is clicked. The example below demonstrates a simple Button control:

```
<html>
<body>
<form runat="server">
<asp:Button id="b1" Text="Submit" runat="server" />
</form>
</body>
</html>
```

Data Binding

The following controls are list controls which support data binding:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

The selectable items in each of the above controls are usually defined by one or more asp:ListItem controls, like this:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="countrylist" runat="server">
<asp:ListItem value="N" text="Norway" />
<asp:ListItem value="S" text="Sweden" />
<asp:ListItem value="F" text="France" />
<asp:ListItem value="I" text="Italy" />
</asp:RadioButtonList>
</form>
</body>
</html>
```

However, with data binding we may use a separate source, like a database, an XML file, or a script to fill the list with selectable items. By using an imported source, the data is separated from the HTML, and any changes to the items are made in the separate data source.

The ArrayList object is a collection of items containing a single data value. Items are added to the ArrayList with the Add() method. The following code creates a new ArrayList object named mycountries and four items are added:

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
```

```
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
end if
end sub
</script>
```

By default, an ArrayList object contains 16 entries. An ArrayList can be sized to its final size with the TrimToSize() method:

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
  mycountries.TrimToSize()
end if
end sub
</script>
```

An ArrayList can also be sorted alphabetically or numerically with the Sort() method:

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
  mycountries.TrimToSize()
  mycountries.Sort()
```

```
end if
end sub
</script>
```

To sort in reverse order, apply the Reverse() method after the Sort() method:

```
script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
 dim mycountries=New ArrayList
 mycountries.Add("Norway")
 mycountries.Add("Sweden")
 mycountries.Add("France")
 mycountries.Add("Italy")
 mycountries.TrimToSize()
 mycountries.Sort()
 mycountries.Reverse()
end if
end sub
</script>
```

An ArrayList object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
 dim mycountries=New ArrayList
 mycountries.Add("Norway")
 mycountries.Add("Sweden")
```

```
  mycountries.Add("France")
  mycountries.Add("Italy")
  mycountries.TrimToSize()
  mycountries.Sort()
  rb.DataSource=mycountries
  rb.DataBind()
end if
end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>
</body>
</html>
```

The DataSource property of the RadioButtonList control is set to the ArrayList and it defines the data source of the RadioButtonList control. The DataBind() method of the RadioButtonList control binds the data source with the RadioButtonList control. The data values are used as both the Text and Value properties for the control.

Creating a HashTable

The Hashtable object contains items in key/value pairs. The keys are used as indexes, and very quick searches can be made for values by searching through their keys. Items are added to the Hashtable with the Add() method. The following code creates a Hashtable named mycountries and four elements are added:

```
<script runat="server">
Sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N","Norway")
```

```
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
end if
end sub
</script>
```

A Hashtable object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the list:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N", "Norway")
  mycountries.Add("S", "Sweden")
  mycountries.Add("F", "France")
  mycountries.Add("I", "Italy")
```

```
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value"
  rb.DataBind()
end if
end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N", "Norway")
  mycountries.Add("S", "Sweden")
  mycountries.Add("F", "France")
  mycountries.Add("I", "Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value"
  rb.DataBind()
end if
end sub
sub displayMessage(s as Object,e As EventArgs)
```

```
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True"
onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

You cannot choose the sort order of the items added to the Hashtable. To sort items alphabetically or numerically, use the SortedList object.

The SortedList Object

The SortedList object contains items in key/value pairs. A SortedList object automatically sort the items in alphabetic or numeric order. Items are added to the SortedList with the Add() method. A SortedList can be sized to its final size with the TrimToSize() method. The following code creates a SortedList named mycountries and four elements are added:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
end if
end sub
```

```
</script>
```

A SortedList object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the list:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value"
  rb.DataBind()
end if
end sub
```

```
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value"
  rb.DataBind()
end if
end sub
sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>
<html>
<body>
<form runat="server">
```

```
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

ASP .NET - XML Files

Here is an XML file named "countries.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<countries>
<country>
  <text>Norway</text>
  <value>N</value>
</country>
<country>
  <text>Sweden</text>
  <value>S</value>
</country>
<country>
  <text>France</text>
  <value>F</value>
</country>
<country>
  <text>Italy</text>
  <value>I</value>
</country>
</countries>
```

Bind a DataSet to a List Control

First, import the "System.Data" namespace. We need this namespace to work with DataSet objects. Include the following directive at the top of an .aspx page:

```
<%@ Import Namespace="System.Data" %>
```

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
 dim mycountries=New DataSet
 mycountries.ReadXml(MapPath("countries.xml"))
end if
end sub
```

To bind the DataSet to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the XML DataSet:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
 dim mycountries=New DataSet
 mycountries.ReadXml(MapPath("countries.xml"))
 rb.DataSource=mycountries
 rb.DataValueField="value"
 rb.DataTextField="text"
 rb.DataBind()
end if
```

```
end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New DataSet
  mycountries.ReadXml(MapPath("countries.xml"))
  rb.DataSource=mycountries
  rb.DataValueField="value"
  rb.DataTextField="text"
  rb.DataBind()
end if
end sub
sub displayMessage(s as Object,e As EventArgs)
lbl1.text="Your favorite country is: " & rb.SelectedItem.Text
end sub
</script>
<html>
<body>
<form runat="server">
```

```
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The Repeater Control

The Repeater control is used to display a repeated list of items that are bound to the control. The Repeater control may be bound to a database table, an XML file, or another list of items. Here we will show how to bind an XML file to a Repeater control.

We will use the following XML file in our examples ("cdcatalog.xml"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
<cd>
  <title>Empire Burlesque</title>
  <artist>Bob Dylan</artist>
  <country>USA</country>
  <company>Columbia</company>
  <price>10.90</price>
  <year>1985</year>
</cd>
<cd>
  <title>Hide your heart</title>
  <artist>Bonnie Tyler</artist>
  <country>UK</country>
  <company>CBS Records</company>
  <price>9.90</price>
  <year>1988</year>
</cd>
<cd>
```

```
  <title>Greatest Hits</title>
  <artist>Dolly Parton</artist>
  <country>USA</country>
  <company>RCA</company>
  <price>9.90</price>
  <year>1982</year>
</cd>
<cd>
  <title>Still got the blues</title>
  <artist>Gary Moore</artist>
  <country>UK</country>
  <company>Virgin records</company>
  <price>10.20</price>
  <year>1990</year>
</cd>
<cd>
  <title>Eros</title>
  <artist>Eros Ramazzotti</artist>
  <country>EU</country>
  <company>BMG</company>
  <price>9.90</price>
  <year>1997</year>
</cd>
</catalog>
```

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
```

```
end if
end sub
```

Then we create a Repeater control in an .aspx page. The contents of the <HeaderTemplate> element are rendered first and only once within the output, then the contents of the <ItemTemplate> element are repeated for each "record" in the DataSet, and last, the contents of the <FooterTemplate> element are rendered once within the output:

```
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
...
</HeaderTemplate>
<ItemTemplate>
...
</ItemTemplate>
<FooterTemplate>
...
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Then we add the script that creates the DataSet and binds the mycdcatalog DataSet to the Repeater control. We also fill the Repeater control with HTML tags and bind the data items to the cells in the<ItemTemplate> section with the <%#Container.DataItem("fieldname")%> method:

Example

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
```

```
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
  cdcatalog.DataBind()
end if
end sub
</script>
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
```

```
</tr>
</ItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

Using the <AlternatingItemTemplate>

You can add an <AlternatingItemTemplate> element after the <ItemTemplate> element to describe the appearance of alternating rows of output. In the following example each other row in the table will be displayed in a light grey color:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
  cdcatalog.DataBind()
end if
end sub
</script>
```

```
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>
<AlternatingItemTemplate>
<tr bgcolor="#e8e8e8">
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
```

```
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</AlternatingItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

Using the <SeparatorTemplate>

The <SeparatorTemplate> element can be used to describe a separator between each record. The following example inserts a horizontal line between each table row:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
  cdcatalog.DataBind()
end if
end sub
```

```
</script>
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="0" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>
<SeparatorTemplate>
<tr>
<td colspan="6"><hr /></td>
</tr>
</SeparatorTemplate>
```

```
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body> </html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

ASP.NET - The DataList Control

The DataList control is, like the Repeater control, used to display a repeated list of items that are bound to the control. However, the DataList control adds a table around the data items by default. Bind a DataSet to a DataList Control The DataList control is, like the Repeater control, used to display a repeated list of items that are bound to the control. However, the DataList control adds a table around the data items by default. The DataList control may be bound to a database table, an XML file, or another list of items. Here we will show how to bind an XML file to a DataList control.

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
```

```
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
end if
end sub
```

Then we create a DataList in an .aspx page. The contents of the <HeaderTemplate> element are rendered first and only once within the output, then the contents of the <ItemTemplate> element are repeated for each "record" in the DataSet, and last, the contents of the <FooterTemplate> element are rendered once within the output:

Then we add the script that creates the DataSet and binds the mycdcatalog DataSet to the DataList control. We also fill the DataList control with a <HeaderTemplate> that contains the header of the table, an <ItemTemplate> that contains the data items to display, and a <FooterTemplate> that contains a text. Note that the gridlines attribute of the DataList is set to "both" to display table borders:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
  cdcatalog.DataBind()
end if
end sub
</script>
<html>
<body>
<form runat="server">
<asp:DataList id="cdcatalog" gridlines="both" runat="server">
<HeaderTemplate>
My CD Catalog
</HeaderTemplate>
<ItemTemplate>
```

```
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>
<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>
</asp:DataList>
</form>
</body>
</html>
```

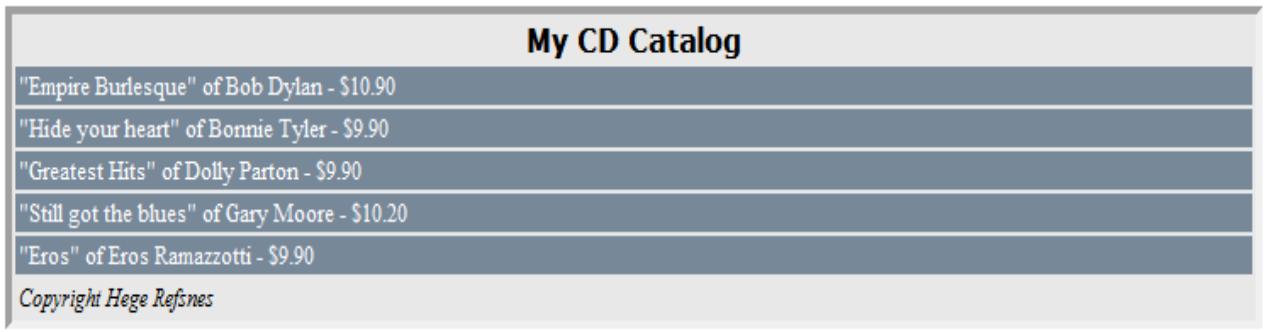
My CD Catalog
"Empire Burlesque" of Bob Dylan - $10.90
"Hide your heart" of Bonnie Tyler - $9.90
"Greatest Hits" of Dolly Parton - $9.90
"Still got the blues" of Gary Moore - $10.20
"Eros" of Eros Ramazzotti - $9.90
Copyright Hege Refsnes

You can also add styles to the DataList control to make the output more fancy:

Example

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
  cdcatalog.DataBind()
end if
end sub
</script>
<html>
<body>
```

```
<form runat="server">
<asp:DataList id="cdcatalog" runat="server" cellpadding="2" cellspacing="2"
borderstyle="inset" backcolor="#e8e8e8" width="100%" headerstyle-font-name="Verdana"
headerstyle-font-size="12pt" headerstyle-horizontalalign="center" headerstyle-font-bold="true"
itemstyle-backcolor="#778899" itemstyle-forecolor="#ffffff" footerstyle-font-size="9pt"
footerstyle-font-italic="true">
<HeaderTemplate>
My CD Catalog
</HeaderTemplate>
<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>
<FooterTemplate>
Copyright Hege Refsnes
</FooterTemplate>
</asp:DataList>
</form>
</body>
</html>
```



Using SQL Server with ASP.NET

Microsoft SQL Server is based on the client/server architecture, in which data is stored on a centralized computer called a server. Other computers, called clients, can access the data stored on the server through a network. The client/server architecture prevents data inconsistency. You can access data stored on a SQL server through Web Forms. To do so, you can create Web applications that have data access controls. These data access Web controls present the data in a consistent manner irrespective of the actual source, such as Microsoft SQL Server or MS Access. Therefore, while creating a Web application, you do not need to worry about the format of the data. However, before you can access or manipulate data from a SQL server, you need to perform the following steps in the specified sequence:

1. Establish a connection with the SQL Server.
2. Write the actual command to access or manipulate data.
3. Create a result set of the data from the data source with which the application can work. This result set is called the data set and is disconnected from the actual source. The application accesses and updates data in the data set, which is later reconciled with the actual data source.

Selecting Data from Table

To achieve this functionality, you first need to import two namespaces, System.Data and System.Data.SqlClient, into your Web Forms page. The syntax is given as follows:

```
<%@ Import Namespace="System.Data.OleDb" %>
```

We need this namespace to work with Microsoft Access and other OLE DB database providers. We will create the connection to the database in the Page_Load subroutine. We create a dbconn variable as a new OleDbConnection class with a connection string which identifies the OLE DB provider and the location of the database. Then we open the database connection:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
dim dbconn
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;//;;;;data source="
& server.mappath("northwind.mdb"))
dbconn.Open()
```

```
end sub
</script>
```

The connection string must be a continuous string without a line break. To specify the records to retrieve from the database, we will create a dbcomm variable as a new OleDbCommand class. The OleDbCommand class is for issuing SQL queries against database tables:

```
<%@ Import Namespace="System.Data.OleDb" %>
    <script runat="server">
    sub Page_Load
    dim dbconn,sql,dbcomm
    dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
    data source=" & server.mappath("northwind.mdb"))
    dbconn.Open()
    sql="SELECT * FROM customers"
    dbcomm=New OleDbCommand(sql,dbconn)
    end sub
</script>
```

The OleDbDataReader class is used to read a stream of records from a data source. A DataReader is created by calling the ExecuteReader method of the OleDbCommand object:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("Bank.mdb"))
dbconn.Open()
sql="SELECT * FROM customers"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
end sub
</script>
```

Then we bind the DataReader to a Repeater control:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
        dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data
source=" & Server.MapPath("Bank.mdb"))
dbconn.Open()
    sql = "SELECT * From customer where Address='Mnr'"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
customers.DataSource=dbread
customers.DataBind()
dbread.Close()
dbconn.Close()
end sub
</script>
<html>
<body>
<form id="Form1" runat="server">
<asp:Repeater id="customers" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Customer ID</th>
<th>Customer Name</th>
<th>Address</th>
<th>Age</th>
<th>Mobile</th>
<th>Email</th>
</tr>
```

```
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("Cid")%></td>
<td><%#Container.DataItem("CName")%></td>
<td><%#Container.DataItem("Address")%></td>
<td><%#Container.DataItem("Age")%></td>
<td><%#Container.DataItem("Mobile")%></td>
<td><%#Container.DataItem("Email")%></td>
</tr>
</ItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Creating Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
    dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data
source=" & Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "Create Table Products (ProductID VarChar (4) Primary Key, ProductName
VarChar (20), UnitPrice Money,QtyAvailable Integer)"
    dbcomm = New OleDbCommand(sql, dbconn)
    dbcomm.ExecuteNonQuery()
```

```
dbconn.Close()
end sub
</script>
```

Inserting Data into Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
    dbconn = New
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;datasource=" &
Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "Insert into customer values(3,'Aaryan','Pkr',34,'9803456789','ar@gmail.com')"
    dbcomm = New OleDbCommand(sql, dbconn)
    dbcomm.ExecuteNonQuery()
   dbconn.Close()
end sub
</script>
```

Deleting Data from Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
        dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data
source=" & Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "Delete from customer where Cid=1"
    dbcomm = New OleDbCommand(sql, dbconn)
    dbcomm.ExecuteNonQuery()
```

```
dbconn.Close()
end sub
</script>
```

Updating Data in the Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
        dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data
source=" & Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "update customer set Address='Ktm' where Cid=2"
    dbcomm = New OleDbCommand(sql, dbconn)
    dbcomm.ExecuteNonQuery()
dbconn.Close()
end sub
</script>
```

A Sample GUI Based Form with Database Connectivity

```
<%@ Page Language="VB"%>
<%@ Import Namespace="System.Data.OleDb"%>
<html>
<script language="VB" runat=server>
Sub Insert_Click(Src As Object, E As EventArgs)
' Connect to Database
    Dim cnAccess As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data
source=" & Server.MapPath("Bank.mdb"))
cnAccess.Open()
dim sID, sFName, sLName, sAge, sInsertSQL as string
sID = eID.Text
```

```
sFName = FName.Text
sLName = LName.Text
sAge = Age.Text
'Make the insert statement
      sInsertSQL = "insert into employees values(" & sID & ",'" & sFName & "','" &
sLName & "'," & sAge & ")"
'Make the OleDbCommand object
dim cmdInsert as New OleDbCommand(sInsertSQL,cnAccess)
' This not a query so we do not expect any return data so use
' the ExecuteNonQuery method
cmdInsert.ExecuteNonQuery()
response.write ("Data recorded!")
End Sub
</script>
<body>
<form id="Form1" runat=server>
<h3><font face="Verdana">Enter Employee Details</font></h3>
<table>
<tr>
<td>ID:</td>
<td><asp:textbox id="eID" runat="server"/></td>
</tr>
<tr>
<td>First Name:</td>
<td><asp:textbox id="FName" runat="server"/></td>
</tr>
<tr>
<td>Last Name:</td>
<td><asp:textbox id="LName" runat="server"/></td>
</tr>
<tr>
```

```
<td>Age:</td>
<td><asp:textbox id="Age" runat="server"/></td>
</tr>
</table>
<asp:button ID="Button1" text="Insert" OnClick="Insert_Click" runat=server/>
<p>
<asp:Label id="Msg" ForeColor="red" Font-Name="Verdana" Font-Size=
"10" runat=server />
</form>
</body>
</html>
```

Handling Session and Cookie in ASP.net

A session is defined as the period of time that a unique user interacts with a Web application. Active Server Pages (ASP) developers who wish to retain data for unique user sessions can use an intrinsic feature known as session state. Programmatically, session state is nothing more than memory in the shape of a dictionary or hash table, e.g. key-value pairs, which can be set and read for the duration of a user's session. For example, a user selects stocks to track and the Web application can store these values in the user's ASP session instance:

```
Session("Stocks") = "MSFT; VRSN; GE"
```

On subsequent pages these values are read and the Web application has access to these values without the user re-entering them:

```
Dim StockString
StockString = Session("Stocks")
```

ASP maintains session state by providing the client with a unique key assigned to the user when the session begins. This key is stored in an HTTP cookie that the client sends to the server on each request. The server can then read the key from the cookie and re-inflate the server session state.

Problems with ASP Session State

ASP developers know session state as a great feature, but one that is somewhat limited. These limitations include:

- **Process dependent.** ASP session state exists in the process that hosts ASP; thus the actions that affect the process also affect session state. When the process is recycled or fails, session state is lost.
- **Server farm limitations.** As users move from server to server in a Web server farm, their session state does not follow them. ASP session state is machine specific. Each ASP server provides its own session state, and unless the user returns to the same server, the session state is inaccessible. While network IP level routing solutions can solve such problems, by ensuring that client IPs are routed to the originating server, some ISPs choose to use a proxy load-balancing solution for their clients. Most infamous of these is AOL. Solutions such as AOL's prevent network level routing of requests to servers because the IP addresses for the requestor cannot be guaranteed to be unique.
- **Cookie dependent.** Clients that don't accept HTTP cookies can't take advantage of session state. Some clients believe that cookies compromise security and/or privacy and thus disable them, which disables session state on the server.

These are several of the problem sets that were taken into consideration in the design of ASP.NET session state.

ASP.NET Session State

ASP.NET session state solves all of the above problems associated with classic ASP session state:

- **Process independent.** ASP.NET session state is able to run in a separate process from the ASP.NET host process. If session state is in a separate process, the ASP.NET process can come and go while the session state process remains available. Of course, you can still use session state in process similar to classic ASP, too.
- **Support for server farm configurations.** By moving to an out-of-process model, ASP.NET also solves the server farm problem. The new out-of-process model allows all servers in the farm to share a session state process. You can implement this by changing the ASP.NET configuration to point to a common server.

- **Cookie independent.** Although solutions to the problem of cookieless state management do exist for classic ASP, they're not trivial to implement. ASP.NET, on the other hand, reduces the complexities of cookieless session state to a simple configuration setting.

Using ASP.NET Session State

Before we use session state, we need an application to test it with. Below is the code for a simple Visual Basic application that writes to and reads from session state, **SessionState.aspx**:

```
<Script runat=server>
  Sub Session_Add(sender As Object, e As EventArgs)
     Session("MySession") = text1.Value
        span1.InnerHtml = "Session data updated! <P> Your session contains: <font
color=red>" + Session("MySession").ToString() + "</font>"
  End Sub
  Sub CheckSession(sender As Object, e As EventArgs)
        If (Session("MySession") = "") Then
           span1.InnerHtml = "NOTHING, SESSION DATA LOST!"
        Else
           span1.InnerHtml = "Your session contains: <font color=red>" +
Session("MySession").ToString() + "</font>"
        End If
  End Sub
</Script>
<form id="Form1" runat=server>
  <input id=text1 type=text runat=server>
  <input id="Submit1" type=submit runat=server OnServerClick="Session_Add"
Value="Add to Session State">
  <input id="Submit2" type=submit runat=server OnServerClick="CheckSession"
Value="View Session State">
</form>
<hr size=1>
<font size=6><span id=span1 runat=server/></font>
```

This simple page wires up two server-side events for the **Add** and **View** buttons, and simply sets the session state to the value in the text box. There are four general configuration settings we can look at in more detail: in-process mode, out-of-process mode, SQL Server mode, and Cookieless.

Handling Cookies

A cookie is a small bit of text that accompanies requests and pages as they go between the Web server and browser. The cookie contains information the Web application can read whenever the user visits the site. For example, if a user requests a page from your site and your application sends not just a page, but also a cookie containing the date and time, when the user's browser gets the page, the browser also gets the cookie, which it stores in a folder on the user's hard disk.
Later, if user requests a page from your site again, when the user enters the URL the browser looks on the local hard disk for a cookie associated with the URL. If the cookie exists, the browser sends the cookie to your site along with the page request. Your application can then determine the date and time that the user last visited the site. You might use the information to display a message to the user or check an expiration date.
Cookies are associated with a Web site, not with a specific page, so the browser and server will exchange cookie information no matter what page the user requests from your site. As the user visits different sites, each site might send a cookie to the user's browser as well; the browser stores all the cookies separately. Cookies help Web sites store information about visitors. More generally, cookies are one way of maintaining continuity in a Web application—that is, of performing state management. Except for the brief time when they are actually exchanging information, the browser and Web server are disconnected. Each request a user makes to a Web server is treated independently of any other request. Many times, however, it's useful for the Web server to recognize users when they request a page. For example, the Web server on a shopping site keeps track of individual shoppers so the site can manage shopping carts and other user-specific information. A cookie therefore acts as a kind of calling card, presenting pertinent identification that helps an application know how to proceed.
Cookies are used for many purposes, all relating to helping the Web site remember users. For example, a site conducting a poll might use a cookie simply as a Boolean value to indicate whether a user's browser has already participated in voting so that the user cannot vote twice. A

site that asks a user to log on might use a cookie to record that the user already logged on so that the user does not have to keep entering credentials.

Page Directives

Page directives are used to set various attributes about a page. The ASP Engine and the compiler follow these directives to prepare a page. There are many kinds of directives. The most frequently ones are the following: @ Page,@ Import,@ Implements, @ Register,@ OutputCache and @ Assembly directives. These directives can be placed anywhere in a page, however, these are typically placed at the top.

1. @ Page: We may use this directive to declare many page-related attributes about a particular page. For example, we use this directive to declare the language to be used in a page, such as *<%@ Page Language="VB" Debug="true" %>* page.
2. @ Import: We use this directive to import a namespace in the page class file. For example, in the following directive, we are importing the *System.Data.OleDb* namespace in our page: *<%@ Import Namespace="System.Data.OleDb" %>*.
3. @ OutputCache: We can use this directive to specify how to cache the page. In the following example, we are setting the duration that a page or user control is output cached:*<%@ OutputCache Duration="10" /%>*.
4. @ Register: This directive is used to register a custom control in a page. In the following example, we are registering one of our user custom controls in page: *<%@ Register tagprefix ="utoledo" tagname="Time" Src="TimeUserControl.ascx"%>*.
5. @ Assembly We use this directive to link to an assembly to the current page or user control. The following example shows how to link to an assembly-named payroll: *<%@ Assembly Name="Payroll" %>*.
6. @ Implements This directive enables us to implement an interface in our page. In the following example, we are implementing the *IpostBackEventHandler* interface in one of our user controls:
   <%@ ImplementsInterface="System.Web.UI.IPostBackEventHandler" %>.

Tag Libraries

In a Web application, a common design goal is to separate the display code from business logic. Java tag libraries are one solution to this problem. Tag libraries allow you to isolate business logic from the display code by creating a **Tag** class (which performs the business logic) and including an HTML-like tag in your JSP page. When the Web server encounters the tag within your JSP page, the Web server will call methods within the corresponding Java **Tag** class to produce the required HTML content.

Microsoft® ASP.NET uses Web form controls to serve the same purpose as Java tag libraries. Similar to JSP tags, Web form controls are added to an ASP.NET Web page using an HTML-like syntax. Unlike JSP tags however, a Web form control is actually an object that is contained within your ASP.NET page. This allows you to access information from your Web form control both before and after the page is loaded. **The Microsoft® .NET Framework contains many ready-to-use Web form controls, including a Calendar Web form control and a Crystal Reports Viewer Web control.** If you require different functionality than is provided by these Web form controls, you can either extend the existing Web form controls or create your own Web form controls by implementing various interfaces.

Tag libraries were designed so that Java code could be executed within a JSP page without using Java script blocks, which clutter up the HTML and break the design goal of separating display code from business logic. Instead of script blocks, tag libraries allow you to create custom HTML-like tags that map to a Java class that performs the business logic. Groups of these HTML-like tags are called tag libraries. Creating and using a custom tag library involves three things:

- One or more classes that implement the **javax.servlet.jsp.tagext.Tag** interface. The **Tag** interface defines six methods that allow your JSP page to use the class to create the desired HTML output. There are also classes/interfaces that implement/extend the **Tag** interface, such as **TagSupport** and **BodyTagSupport**, to make it easier for you to develop your custom tag.
- An XML document that describes your tag library. Tag library description files must conform to the JSP tag library description DTD, and generally have an extension of "tld".
- Importing the tag library to the JSP page using the **taglib** directive.

Once the three requirements are met, you can use the tags in your tag library anywhere within your JSP page.

For detail explore: http://msdn.microsoft.com/en-us/library/aa478990.aspx