Unit -3
ASP.NET Validation Controls & State Management

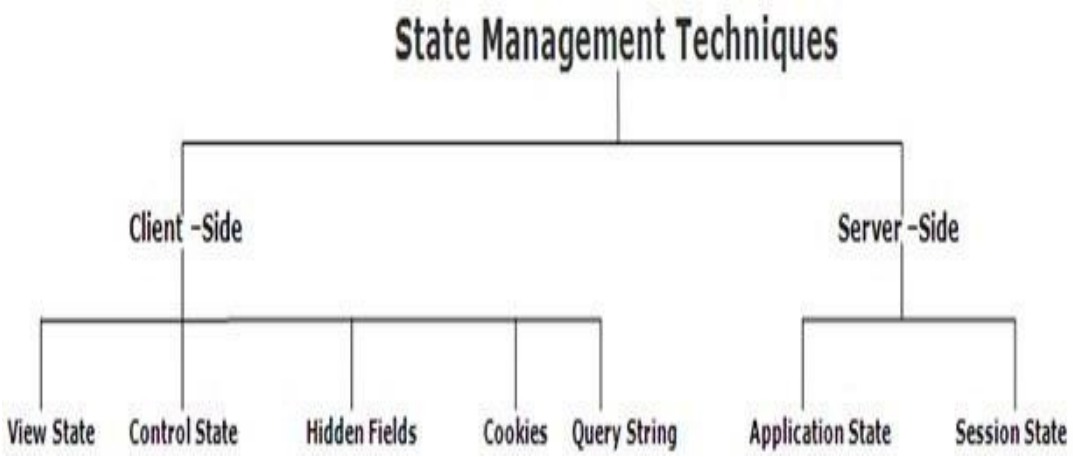
What is validation?

An important aspect of creating ASP.NET Web pages for user input is to be able to check that the information users enter is valid. ASP.NET provides a set of validation controls that provide an easy-to-use but powerful way to check for errors and, if necessary, display messages to the user.

Why do we use validation controls?

Validation is an important part of any web application. User's input must always be validated before sending across different layers of the application.
Validation controls are used to,

- Implement presentation logic.
- To validate user input data.
- Data format, data type and data range is used for validation.



Client side validation is good but we have to be dependent on browser and scripting language support.

Client side validation is considered convenient for users as they get instant feedback. The main advantage is that it prevents a page from being postback to the server until the client validation is executed successfully.

For developer point of view serve side is preferable because it will not fail, it is not dependent on browser and scripting language.

You can use ASP.NET validation, which will ensure client, and server validation. It work on both end; first it will work on client validation and than on server validation. At any cost server validation will work always whether client validation is executed or not. So you have a safety of validation check.

Validation Controls in ASP.NET:

An important aspect of creating ASP.NET Web pages for user input is to be able to check that the information users enter is valid. ASP.NET provides a set of validation controls that provide an easy-to-use but powerful way to check for errors and, if necessary, display messages to the user.

There are six types of validation controls in ASP.NET

1. RequiredFieldValidation Control
2. CompareValidator Control
3. RangeValidator Control
4. RegularExpressionValidator Control
5. CustomValidator Control
6. ValidationSummary

For client script .NET used JavaScript. WebUIValidation.js file is used for client validation by .NET

Important points for validation controls :

- ControlToValidate property is mandatory to all validate controls.
- One validation control will validate only one input control but multiple validate control can be assigned to a input control.

Validation Properties :

- Usually, Validation is invoked in response to user actions like clicking submit button or entering data. Suppose, you wish to perform validation on page when user clicks submit button.
-
- Server validation will only performed when CauseValidation is set to true.
-
- When the value of the CausesValidation property is set to true, you can also use the ValidationGroup property to specify the name of the validation group for which the Button control causes validation.
-
- Page has a Validate() method. If it is true this methods is executed. Validate() executes each validation control.

To make this happen, simply set the CauseValidation property to true for submit button as shown below:

```
<asp:Button ID="Button2" runat="server" Text="Submit" CausesValidation=true />
```

RequiredFieldValidation	Makes an input control a required field
CompareValidator	Compares the value of one input control to the value of another input control or to a fixed value
RangeValidator	Checks that the user enters a value that falls between two values
RegularExpressionValidator	Ensures that the value of an input control matches a specified pattern
CustomValidator	Allows you to write a method to handle the validation of the value entered
ValidationSummary	Displays a report of all validation errors occurred in a Web page

All validation controls are rendered in form as (label are referred as on client by server)

1. RequiredFieldValidation Control :

The RequiredFieldValidator control is simple validation control, which checks to see if the data is entered for the input control. You can have a RequiredFieldValidator control for each form element on which you wish to enforce Mandatory Field rule.

Example:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat="server"
Style="top: 98px; left: 367px; position: absolute; height: 26px; width: 162px"
ErrorMessage="password required" ControlToValidate="TextBox2">
</asp:RequiredFieldValidator>
```

2. CompareValidator Control :

The CompareValidator control allows you to make comparison to compare data entered in an input control with a constant value or a value in a different control.

It can most commonly be used when you need to confirm the password entered by the user at the registration time. The data is always case sensitive.

Example:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server" Style="top: 145px;
    left: 367px; position: absolute; height: 26px; width: 162px" ErrorMessage="password
required"
    ControlToValidate="TextBox3"></asp:RequiredFieldValidator>
```

3. RangeValidator Control :

The RangeValidator Server Control is another validator control, which checks to see if a control value is within a valid range. The attributes that are necessary to this control are: MaximumValue, MinimumValue, and Type.

Example:

```
<asp:RangeValidator ID="RangeValidator1" runat="server"
Style="top: 194px; left: 365px; position: absolute; height: 22px; width: 105px"
ErrorMessage="RangeValidator" ControlToValidate="TextBox4" MaximumValue="100"
MinimumValue="18" Type="Integer"></asp:RangeValidator>
```

4. RegularExpressionValidator Control :

A regular expression is a powerful pattern matching language that can be used to identify simple and complex characters sequence that would otherwise require writing code to perform.

Using RegularExpressionValidator server control, you can check a user's input based on a pattern that you define using a regular expression.

It is used to validate complex expressions. These expressions can be phone number, email address, zip code and many more. Using Regular Expression Validator is very simple. Simply set the ValidationExpression property to any type of expression you want and it will validate it.

If you don't find your desired regular expression, you can create your custom one.

Example:

```
 <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
Style="top: 234px;
      left: 366px; position: absolute; height: 22px; width: 177px"
      ErrorMessage="RegularExpressionValidator" ControlToValidate="TextBox5"

ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"></asp:RegularExpressio
nValidator>
```

5.CustomValidator Control :

You can solve your purpose with ASP.NET validation control. But if you still don't find solution you can create your own custom validator control.

The CustomValidator Control can be used on client side and server side. JavaScript is used to do client validation and you can use any .NET language to do server side validation.

I will explain you CustomValidator using server side. You should rely more on server side validation.

To write CustomValidator on server side you override ServerValidate event.

Example:

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Untitled Page</title>
```

```
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:Label ID="Label1" runat="server" Text="User ID:"></asp:Label>
 <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
 <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
         ControlToValidate="TextBox1" ErrorMessage="User id
required"></asp:RequiredFieldValidator>

      <asp:CustomValidator ID="CustomValidator1" runat="server"
OnServerValidate="UserCustomValidate"
         ControlToValidate="TextBox1"
         ErrorMessage="User ID should have atleast a capital, small and digit and should be
greater than 5 and less
than 26 letters"
         SetFocusOnError="True"></asp:CustomValidator>
      </div>
   <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="Submit" />
   </form>
</body>
</html>
```

6. ValidationSummary :

ASP.NET has provided an additional control that complements the validator controls.

The ValidationSummary control is reporting control, which is used by the other validation controls on a page.

You can use this validation control to consolidate errors reporting for all the validation errors that occur on a page instead of leaving this up to each and every individual validation control.

The validation summary control will collect all the error messages of all the non-valid controls and put them in a tidy list.

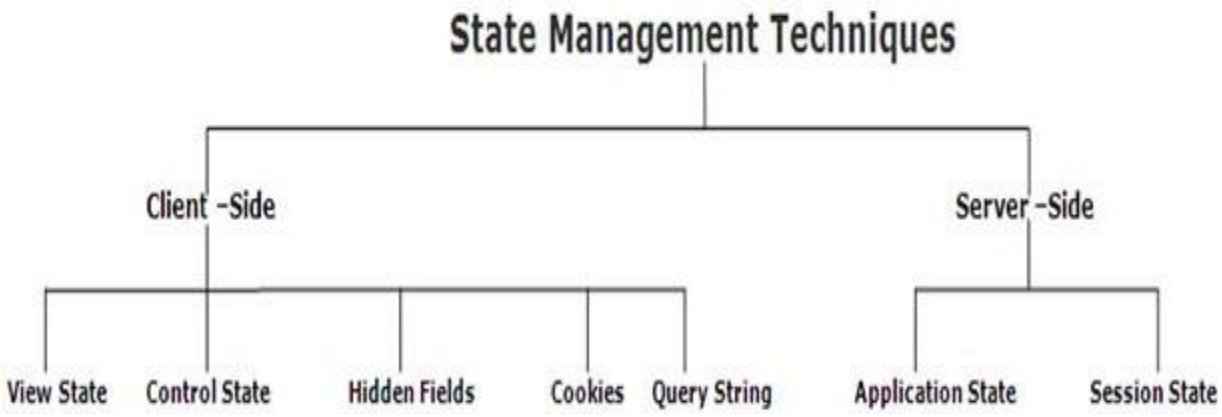
Example :

```
<asp:ValidationSummary ID="ValidationSummary1" runat="server"
      style="top: 390px; left: 44px; position: absolute; height: 38px; width: 625px" />
```

What is State Management ?

HyperText Transfer Protocol (HTTP) is a stateless protocol. When the client disconnects from the server, the ASP.NET engine discards the page objects. This way, each web application can scale up to serve numerous requests simultaneously without running out of server memory.
However, there needs to be some technique to store the information between requests and to retrieve it when required. This information i.e., the current value of all the controls and variables for the current user in the current session is called the State.

There are two types of state management system in ASP.NET.



A .Client side state management system

1.View state :

View state is an inbuilt feature of ASP.NET that retains values between multiple requests for the same page. ASP.NET page contains a hidden form field named __VIEWSTATE. This hidden form field stores the value of the control's property. By default view state is enabled for page and its controls. You can disable view state by setting the property EnableViewState as false. Storing too much data into View State can hamper the performance of web page.
Therefore we should take care while enabling and disabling the property EnableViewState.

Example:

```
//writing information to view state
ViewState.Add("MyInfo", "Welcome");
//read information from view state
if (ViewState["MyInfo"] != null)
{
    string data = (string)ViewState["MyInfo"];
}
```

2.Hidden fields :

Hidden fields in HTML are simply input fields and not visible on the browser during execution. Hidden fields are used to store data at the page level. Hidden fields are simple to implement for a page specific data and stores small amount of data. We should not use hidden fields for sensitive data. It has no built-in compression, encryption technique.

Example :

```
/writing information to Hidden field
HiddenField1.Value = "Welcome";
//read information from Hidden field
string str = HiddenField1.Value;
```

3. Cookies :

A cookie is a small amount of data that server creates on the client. Cookie is small text information. You can store only string values when using a cookie. When a request sent to web server, server creates a cookie, and sent to browser with an additional HTTP header.

Some common uses of cookies are:

- Authentication of user.
- Identification of a user session.
- User's preferences.
- Shopping cart contents.
- Remember users between visits.

Limitation of cookies

- Cookie can store only string value.
- Cookies are browser dependent.
- Cookies are not secure.
- Cookies can store small amount of data.
- Size of cookies is limited to 4096 bytes.

Important properties of HttpCookie :

- Domain: Enables you to get or set the domain of the cookie.
- Expires: It contains the expiration time of the cookie.
- HasKeys: Returns bool value, indicating whether the cookie has subkeys.
- Name: Provides the name of the cookie.
- Path: Enables you to get or set the virtual path to submit with the cookie.
- Secure: It contains true if the cookie is to be passed with SSL.
- Value: It contains the value of the cookie.

Example :

```
Response.Cookies["EmpCookies"]["EmpID"] = txtID.Text;
Response.Cookies["EmpCookies"]["FirstName"] = txtFirstName.Text;
Response.Cookies["EmpCookies"]["LastName"] = txtLastName.Text;
Response.Cookies["EmpCookies"]["Address"] = txtAddress.Text;
Response.Cookies["message"].Expires = DateTime.Now.AddYears(1);

//Reading Cookie.
string info;
  if (Request.Cookies["EmpCookies"] != null)
  {
     info = Request.Cookies["EmpCookies"]["EmpID"] + "</br>";
     info += Request.Cookies["EmpCookies"]["FirstName"] + "</br>";
     info += Request.Cookies["EmpCookies"]["LastName"] + "</br>";
     info += Request.Cookies["EmpCookies"]["Address"] + "</br>";

     Label1.Text = info;
  }
```

4. Query String :

The Query String object is helpful when we want to transfer a value from one page to another. Query String is very easy to use. Query string values are appended to the end of the page URL. It uses a question mark (?), followed by the parameter name followed by an equal sign (=) and its value. You can append multiple query string parameters using the ampersand (&) sign.Always remember, we should not send lots of data through QueryString. Another limitation is that information we send through QueryString is visible on the address bar.

Important points about QueryString:

- It is easy to use.
- Sensitive data should not pass using QueryString.
- Browsers have 2,083-character limits on URLs. Therefore there is limit to pass the data.
- QueryString is a part of URL.
- It uses one or more than one parameter.
- It uses "&" sign while using more than one parameter.
- SPACE is encoded as '+' or '%20'

Points to Remember

Some features of query strings are:

- Used for enabling the View State Property
- Defines a custom view
- View State property declaration
- Can't be modified
- Accessed directly or disabled

Example :

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    if (Request.QueryString["Name"] != null && Request.QueryString["Name"] != string.Empty)
      lblName.Text = Request.QueryString["Name"];

    if (Request.QueryString["DeptName"] != null && Request.QueryString["Name"] !=
string.Empty)
      lblDeptName.Text = Request.QueryString["DeptName"];
  }
}
```

5. Control State :

Control state is based on the custom control option. For expected results from CONTROL STATE we need to enable the property of view state. As I already described you can manually change those settings.

B. Server-Side of State Management in ASP NET

1. Session State :

In ASP.NET session is a state that is used to store and retrieve values of a user.
It helps to identify requests from the same browser during a time period (session). It is used to store value for the particular time session. By default, ASP.NET session state is enabled for all ASP.NET applications.
Each created session is stored in SessionStateItemCollection object. We can get current session value by using Session property of Page object. Let's see an example, how to create an access session in asp.net application.
property of the HttpSessionState class.

Properties:
SessionID: The unique session identifier.
Item(name): The value of the session state item with the specified name. This is the default
Count: The number of items in the session state collection.
TimeOut: Gets and sets the amount of time, in minutes, allowed between requests before the session-state provider terminates the session.

Methods:
Add(name, value): Adds an item to the session state collection.
Clear : Removes all the items from the session state collection.
Remove(name): Removes the specified item from the session state collection.
RemoveAll: Removes all keys and values from the session-state collection.
RemoveAt: Deletes an item at a specified index from the session-state collection.

Example :

```
using System;
using System.Web.UI;
namespace SessionExample
{
   public partial class _Default : Page
   {
      protected void login_Click(object sender, EventArgs e)
      {
         if (password.Text=="qwe123")
         {
            // Storing email to Session variable
            Session["email"] = email.Text;
         }
         // Checking Session variable is not empty
         if (Session["email"] != null)
```

```
        {
            // Displaying stored email
            Label3.Text = "This email is stored to the session.";
            Label4.Text = Session["email"].ToString();
        }
    }
  }
}
```

2 . Application State :

The ASP.NET application is the collection of all web pages, code and other files within a single virtual directory on a web server. When information is stored in application state, it is available to all the users.

To provide for the use of application state, ASP.NET creates an application state object for each application from the HTTPApplicationState class and stores this object in server memory. This object is represented by class file global.asax.

Application State is mostly used to store hit counters and other statistical data, global application data like tax rate, discount rate etc. and to keep the track of users visiting the site.

The HttpApplicationState class has the following properties:

Properties:

Item(name): The value of the application state item with the specified name. This is the default property of the HttpApplicationState class.

Count : The number of items in the application state collection.

Methods:

Add(name, value): Adds an item to the application state collection.

Clear: Removes all the items from the application state collection.

Remove(name) : Removes the specified item from the application state collection.

RemoveAll: Removes all objects from an HttpApplicationState collection.

RemoveAt: Removes an HttpApplicationState object from a collection by index.

Lock() : Locks the application state collection so only the current user can access it.

Unlock(): Unlocks the application state collection so all the users can access it.

Application state data is generally maintained by writing handlers for the events:

- Application_Start
- Application_End
- Application_Error
- Session_Start
- Session_End

Example:

```
Void Application_Start(object sender, EventArgs e)
{
  Application["startMessage"] = "The application has started.";
}

Void Application_End(object sender, EventArgs e)
{
  Application["endtMessage"] = "The application has ended.";
}
```