

ASP.NET STATE MANAGEMENT

State management is the process by which you maintain state and page information over multiple requests for the same or different pages. As is true for any HTTP-based technology, Web Forms pages are stateless, which means that they do not automatically indicate whether the requests in a sequence are all from the same client or even whether a single browser instance is still actively viewing a page or site. Furthermore, pages are destroyed and re-created with each round trip to the server; therefore, page information will not exist beyond the life cycle of a single page

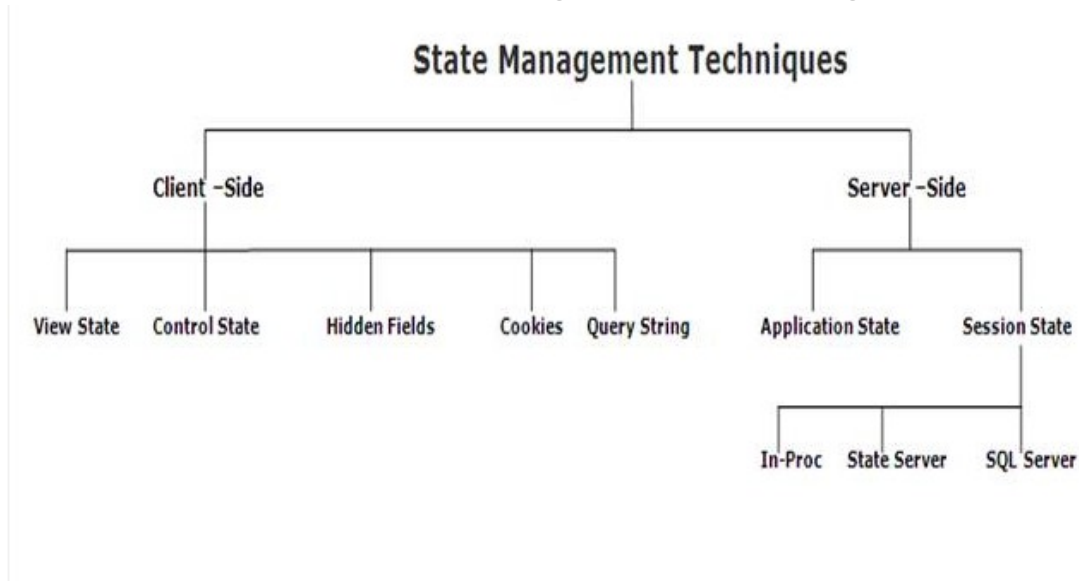
Windows programming is very different from web applications in terms of state management . Now what does it actually means . Let us understand it in deep . As we know that whenever we request for any web page from server , we use HTTP for it .HTTP is stateless .Now what does it mean stateless??

Client and server 's connectivity starts when client request for web page and server respond the respective web page . Server allocates required memory for compiling and processing the web page and it's controls at server side itself . But just after delivering page to client ,server removes all the allocated memory and gets disconnected . This phenomenon is called stateless . Therefore web processing is called stateless .

A new instance of the Web page class is created each time the page is posted to the server. In traditional Web programming, this would typically mean that all information associated with the page and the controls on the page would be lost with each round trip. For example, if a user enters information into a text box, that information would be lost in the round trip from the browser or client device to the server

A professional ASP.NET site might look like a continuously running application, but that's really just a clever illusion. In a typical web request, the client connects to the web server and requests a page. When the page is delivered, the connection is severed, and the web server discards all the page objects from memory. By the time the user receives a page, the web page code has already stopped running, and there's no information left in the web server's memory. This stateless design has one significant advantage. Because clients need to be connected for only a few seconds at most, a web server can handle a huge number of nearly simultaneous requests without a performance hit. However, if you want to retain information for a longer period of time so it can be used over multiple post-backs or on multiple pages, you need to take additional steps.

In ASP.NET, there are 2 state management methodologies. These are:



Client Side State Management

Storing page information using client-side options doesn't use server resources. These options typically have minimal security but fast server performance because the demand on server resources is modest. However, because you must send information to the client for it to be stored, there is a practical limit on how much information you can store this way.

Whenever we use client side state management, the state related information will directly get stored on the client side. That particular information will travel back and communicate with every request generated by the user then afterwards provides responses after server side communication.

Client Side | Techniques

Client side state management techniques are:

- Hidden fields
- View state
- Control State
- Cookies
- Query Strings

Hidden Fields

You can store page-specific information in a hidden field on your page as a way of maintaining the state of your page.

If you use hidden fields, it is best to store only small amounts of frequently changed data on the client.

ASP.NET allows you to store information in a HiddenField control, which renders as a standard HTML hidden field. A hidden field does not render visibly in the browser, but you can set its properties just as you can with a standard control. When a page is submitted to the server, the content of a hidden field is sent in the HTTP form collection along with the values of other controls. A hidden field acts as a repository for any page-specific information that you want to store directly in the page.

A HiddenField control stores a single variable in its Value property and must be explicitly added to the page.

In order for hidden-field values to be available during page processing, you must submit the page using an HTTP POST command. If you use hidden fields and a page is processed in response to a link or an HTTP GET command, the hidden fields will not be available.

Note

If you use hidden fields, you must submit your pages to the server using the HTTP POST method rather than requesting the page via the page URL (the HTTP GET method).

See the below example where I have used Hidden-field :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:HiddenField ID="HiddenField1" runat="server" />
    </div>
  </form>
</body>
</html>
```

Presently Hidden-field does not have any value stored which we can provide either at design time or run time .

Let us see the properties and events associated with it :

Property : Value

Event : ValueChanged

```
protected void Page_Load(object sender, EventArgs e)
{
    HiddenField1.Value = "One";
}
protected void Page_PreRender(object s, EventArgs e)
{
    Response.Write(HiddenField1.Value.ToString());
}
```

On Load Event , we have stored the value to “one”

and on Page_PreRender Event we have fetched the value and displayed it to user .

We can also give value to it at design time by using Value property and can retrieve it at run time .

We can also work on ValueChanged Event for tracking the change in value at run time .

Example 1:

Let us take an example of Hidden-field to be used at client side only and its value will not be taken to server .

See the image below :

A screenshot of a web form. It features a text input field with the label "BackGround Color" to its left. Below the input field are two buttons: "Change Color" on the left and "Display Color" on the right. The buttons have a light blue gradient and a slight shadow.

Here we will take Background color from user and save it to Hidden-field on clicking of ChangeColor Button along with saving values to hidden-field , we will also change the background color .

If user will click on Display Color Button , then will display the current color value stored in Hidden-field in paragraph .

See the apsx page code :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <style type="text/css">
        #Button1
        {
            width: 93px;
        }
    </style>
    <script type="text/javascript">

        function changecolor() {

            form1.HiddenField1.value = form1.TextBox1.value;
            document.getElementById("b1").style.backgroundColor = form1.HiddenField1.value;
        }
        function displaycolor() {
            p1.innerHTML = "<h1> Present Color is  " + form1.HiddenField1.value + "</h1>";
        }
    </script>
</head>
<body>
    <div>
        <input type="text" value="BackGround Color" />
        <input type="button" value="Change Color" />
        <input type="button" value="Display Color" />
    </div>
    <div>
        <p></p>
    </div>
</body>
</html>
```


If we see the page above , We are having textbox for taking background color from user and then apply it to background of web page . The very important thing above is use of hidden-field for storing the color and changing if user submits the new name .

Let us see the Source code of page.aspx page :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Hidden Field Example </title>
    <script type="text/javascript">

        function PageLoad() {

            form1.HiddenField1.value = form1.TextBox1.value;

        }

    </script>
</head>
<body id="b1" runat="server">
    <form id="form1" runat="server">
        <div>
            <asp:HiddenField ID="HiddenField1" runat="server" Value="Yellow"
                onvaluechanged="HiddenField1_ValueChanged" />
        </div>
        <p>
            &nbsp;   Background Color&nbsp;   
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        </p>
        <p>
            <input type="submit" name="SubmitButton" value="Submit" onclick="PageLoad()" />
        </p>
        <p>
            <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        </p>
    </form>
</body>
</html>
```

Explanation for codes:

A. `<asp:HiddenField ID="HiddenField1" runat="server" Value="Yellow" onvaluechanged="HiddenField1_ValueChanged" />`

Here we have used Hidden-field control with it's default value Yellow provided at design time and also has ValueChanged event added.

B. `<input type="submit" name="SubmitButton" value="Submit" onclick="PageLoad()" />`

Here we have used HTML input control with type submit which is calling PageLoad() function on click of it and then submitting the web page .

```
C.function PageLoad() {
    form1.HiddenField1.value = form1.TextBox1.value; }
```

Here we have created PageLoad() function which will be called on click of button , which will change the hidden-field value to the new value taken from textbox.

Means as user will click on button , it will first call PageLoad() function which will change the HiddenField value with the new value from textbox and then page will be submitted to server where Hidden-field ValueChanged Event handler will be invoked accordingly.

See the code-behind page code :

```
public partial class hiddenfieldex : System.Web.UI.Page
{
    string color = "";
    protected void Page_Load(object sender, EventArgs e)
    {
        color = HiddenField1.Value;
        Label1.Text = "Default Color is " + HiddenField1.Value;
    }
    protected void Page_PreRender(object s, EventArgs e)
    {
        b1.Style.Add(HtmlTextWriterStyle.BackgroundColor,color);

    }
    protected void Button1_Click(object sender, EventArgs e)
    {

    }
    protected void HiddenField1_ValueChanged(object sender, EventArgs e)
    {
        Label1.Text = "";
        Label1.Text = "Color changed to " + HiddenField1.Value;
    }
}
```

Advantages of using hidden fields are:

- No server resources are required** : The hidden field is stored and read from the page.
- Widespread support** Almost all browsers and client devices support forms with hidden fields.
- Simple implementation** Hidden fields are standard HTML controls that require no complex programming logic.

Disadvantages of using hidden fields are:

- Potential security risks** : The hidden field can be tampered with. The information in the hidden field can be seen if the page output source is viewed directly, creating a potential security issue. You can manually encrypt and decrypt the contents of a hidden field, but doing so requires extra coding and overhead. If security is a concern, consider using a server-based state mechanism so that no sensitive information is sent to the client.
- Simple storage architecture** : The hidden field does not support rich data types. Hidden fields offer a single string value field in which to place information. To store multiple values, you must implement delimited strings and the code to parse those strings. You can manually serialize and de-serialize rich data types to and from hidden fields, respectively. However, it requires extra code to do so. If you need to store rich data types on the client, consider using view state instead. View state has serialization built-in, and it stores data in hidden fields.
- Performance considerations**: Because hidden fields are stored in the page itself, storing large values can cause the page to slow down when users display it and when they post it.
- Storage limitations**: If the amount of data in a hidden field becomes very large, some proxies and firewalls will prevent access to the page that contains them. Because the maximum amount can vary with different firewall and proxy implementations, large hidden fields can be sporadically problematic. If you need to store many items of data, consider doing one of the following:
 - Put each item in a separate hidden field.
 - Use view state with view-state chunking turned on, which automatically separates data into multiple hidden fields.
 - Instead of storing data on the client, persist the data on the server. The more data you send to the client, the slower the apparent response time of your application will be because the browser will need to download or send more data.

View-state:

A Web application is stateless. A new instance of the Web page class is created each time the page is requested from the server. This would ordinarily mean that all information associated with the page and its controls would be lost with each round trip. For example, if a user enters information into a text box on an HTML Web page, that information is sent to the server, but is not returned to the client. To overcome this inherent limitation of Web programming, the ASP.NET page framework includes several state-management features, one of which is view state, to preserve page and control values between round trips to the Web server.

View state is the method that the ASP.NET page framework uses by default to preserve page and control values between round trips. When the HTML for the page is rendered, the current state of the page and values that need to be retained during postback are serialized into base64-encoded strings and output in the view state hidden field or fields. You can change the default behavior and store view state in another location such as a SQL Server database by implementing a custom PageStatePersister class to store page data. For an example of storing page state on a stream rather than in a hidden page field Web Forms pages provide the ViewState property as a built-in structure for automatically retaining values between multiple requests for the same page. View state is maintained as a hidden field in the page.

The ViewState property provides a dictionary object for retaining values between multiple requests for the same page. This is the default method that the page uses to preserve page and control property values between round trips. You can access view state in your own code using the page's ViewState property to preserve data during round trips to the Web server. The ViewState property is a dictionary containing key/value pairs containing the view state data.

When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field, or multiple hidden fields if the amount of data stored in the ViewState property exceeds the specified value in the MaxPageStateFieldLength property. When the page is posted back to the server, the page parses the view-state string at page initialization and restores property information in the page.

View State provides page level state management i.e., as long as the user is on the current page, state is available and the user redirects to the next page and the current page state is lost. View State can store any type of data because it is object type but it is preferable not to store a complex type of data due to the need for serialization and deserialization on each post back. View state is enabled by default for all server side controls of ASP.NET with a property EnableViewState set to true.

You can use view state to store your own page-specific values across round trips when the page posts back to itself. For example, if your application is maintaining user-specific information — that is, information that is used in the page but is not necessarily part of any control — you can store it in view state.

Data Types You Can Store in View State

You can store objects of the following types in view state:

- Strings
- Integers
- Boolean values
- Array objects
- Array List objects
- Hash tables
- Custom type converters (see the [TypeConverter](#) class for more information)

You can store other types of data as well, but the class must be compiled with the Serializable attribute so that view state can serialize them into XML.

You can enable or disable view-state either at control level ,at page level,at application level.
Let us see some simple examples of view-state :

Example 1:

Please see the web page below :



Here we are having one label without any text property given , with one text box and one button. Presently we have not disabled view-state at page level or not at control level As see the code below :

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default6.aspx.cs" Inherits="Default6"
EnableViewState="True" %>

<!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></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>

            <p>
                <asp:Label ID="Label1" runat="server"></asp:Label>
                &nbsp;
                <asp:TextBox ID="TextBox1" runat="server" Width="142px"></asp:TextBox>

            </p>
        </div>
        <asp:Button ID="Button1" runat="server" onclick="Button1_Click" />
    </form>
</body>
</html>
```

We can very easily see , that I have not provided Text Property in Label,Text box and Button.

Now we will add the text property in load event of page , see below the code-behind file

```

public partial class Default6 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            Label1.Text = "Name";
            TextBox1.Text = "Narendra";
            Button1.Text = "Submit";
        }
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
    }
}

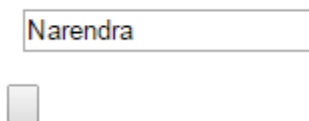
```

On the fresh request of web page , load event will be invoked and will check for post back or not then will change the text property accordingly.

On later submit by buttons , code in load event will not be invoked and the Text Values for multiple controls will be taken from View-state data stored in hidden-fields which is also posted in between multiple round trips.

Because EnableViewState is enabled , it will maintain the state of controls like it's property values.

Now we will change the EnableViewState properties of Label and Button control to False . And Now run the web page again , now you can see that on first request , web page will display the values as provided in Load Event . But as we submit the page again , then the controls like Label and Button will loose it's previous value means values before post back .See the image below of web page after post back .



This happened because view-state of controls (Label and Button) were disabled due to which it's state was not submitted with page and ASP.NET could not load view-state values from the hidden-fields.

This is how ASP.NET maintains the states of controls by providing view-state property and enabled by default for all controls and page individually .

Example 2:

Let us see one more example of View-state . See the image below :



Hello World
Hello World
Hello World
Hello World

Change Message

Empty Postback

As we can see that above we are Two Labels,Two TextBoxes, Two Button . Let us see the Source code below to understand it better :

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default7.aspx.cs" Inherits="Default7"%>

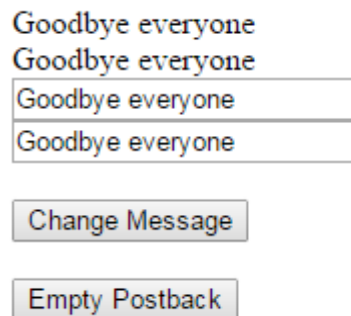
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label runat="server" ID="lblMessage" EnableViewState="true" Text="Hello
World"></asp:Label><br />
        <asp:Label runat="server" ID="lblMessage1" EnableViewState="false" Text="Hello
World"></asp:Label><br />
        <asp:Textbox runat="server" ID="txtMessage" EnableViewState="true" Text="Hello
World"></asp:Textbox><br />
        <asp:Textbox runat="server" ID="txtMessage1" EnableViewState="false" Text="Hello
World"></asp:Textbox><br />
        <br />
        <asp:Button runat="server" Text="Change Message" ID="btnSubmit"
onclick="btnSubmit_Click"></asp:Button><br />
        <br />
        <asp:Button ID="btnEmptyPostBack" runat="server" Text="Empty Postback"></asp:Button>
    </form>
</body>
</html>
```

See the EnableViewState Property of all controls : One Label and One Textbox is having EnableViewState property false and rest two is having true.

Two Buttons , one button with having event handler and rest is used for post back purpose only .

If we run the web page then it will display the web page as shown above . If user click on Button with text “Change Message “ then the output will be as below :



Text of every label and textboxes is changed to “Goodbye everyone”..
Let us see the code-behind page which is doing the above changing .

```
public partial class Default7 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        lblMessage.Text = "Goodbye everyone";
        lblMessage1.Text = "Goodbye everyone";
        txtMessage.Text = "Goodbye everyone";
        txtMessage1.Text = "Goodbye everyone";
    }
}
```

Now this is cleared that , why changes are going on .

Now if we click on second button with text “Empty Postback “ then what changes will be done , let us see it below :

Goodbye everyone
Hello World

Goodbye everyone
Goodbye everyone

Very interesting , The controls with false property of EnableViewState , could not retrieve the previous holding data due to not having it's state data maintained and so restored to default set values. But there was no any affect EnableViewState property on Textboxes that we will discuss in deep below:

Advantages of Setting the EnableViewState to false:

ASP.NET automatically set EnableViewState property to true for all controls and page . But if we change it false then what are the advantages for it ? We need to do so if you want to increase the page processing performance between multiple round-trips. Otherwise it is not required.

Common misconception regarding ViewState and TextBox

There is common myth that most of the asp.net developers believe that ViewState is responsible for maintaining values of controls like textbox. But that's not true. :)

Some days before , I was working with Text Boxes and making its property EnableViewState to false , but I am still getting it's value on submitting the page on different round-trips.

Let us discuss in deep :

Put a text box with EnableViewState= "false" and a button on the page.

```
<asp:textbox id="TextBox1" EnableViewState="False" runat="server"/>
```

```
<asp:button id="Button1" onclick="Button1_Click" runat="server" text="Button"/>
```

Enter some value in the text box and Click the button, a post back happens. Request goes to server and response is sent back to the client. You must be expecting that value that you have entered in Textbox will be lost and it will be empty after post back. Look at the textbox now. Oowwww!! **Text box is still maintaining its value. Ridiculous** .. is not it? Though You have explicitly told that ASP.NET that don't maintain text-box's view state but it does.

Why such thing is happening with it ? It is totally ignoring the view-state property .

Why ViewState does not maintain values of textbox?

Controls like Textbox, is inherited from `IpostBackDataHandler` interface. After `Page_init()` events, `LoadViewState` event is executed which is responsible for load view state for the controls. Page class loads the ViewState from the hidden field `__ViewState`. After `LoadViewState` event, `LoadPostBackData` event is executed, in which values of those controls (which are inherited from `IpostBackDataHandler` interface) is assigned from HTTP Post header. Server simply takes the value from the header and add an attribute to the textbox "value" which is set to the value previously entered by the user.

Let's examine why this happens?

Page LifeCycle and ViewState

In page life cycle, two events are associated with `ViewState`:

- Load View State: This stage follows the initialization stage of page life cycle. During this stage, `ViewState` information saved in the previous post back is loaded into controls. As there is no need to check and load previous data, when the page is loaded for the first time this stage will not happen. On subsequent post back of the page as there may be previous data for the controls, the page will go through this stage.
- Save View State: This stage precedes the render stage of the page. During this stage, current state (`value`) of controls is serialized into 64 bit encoded `string` and persisted in the hidden control (`__ViewState`) in the page.
- Load Post back Data stage: Though this stage has nothing to do with `ViewState`, it causes most of the misconception among developers. This stage only happens when the page has been posted back. ASP.NET controls which implement `IpostBackEventHandler` will update its value (`state`) from the appropriate post back data. **The important things to note about this stage are as follows:**
 - 1.State (`value`) of controls are NOT retrieved from `ViewState` but from posted back form.
 - 2.Page class will hand over the posted back data to only those controls which implement `IPostBackEventHandler`.
 - 3.This stage follows the Load View State stage, in other words state of controls set during the Load View State stage will be overwritten in this stage.

You can also see the details of TextBox in ASP.NET as shown below by choosing Go to Definition or putting cursor over it and then press F12.

```
public class TextBox : WebControl, IPostBackDataHandler, IEditableTextControl, ITextControl
```

We can see TextBox is implementing IPostBackDataHandler .

Storing your own values in View State

Apart of it ,You can use view state to store your own page-specific values across round trips when the page posts back to itself. For example, if your application is maintaining user-specific information — that is, information that is used in the page but is not necessarily part of any control — you can store it in view state.

A server control's view state is the accumulation of all its property values. In order to preserve these values across HTTP requests, ASP.NET server controls use this property, which is an instance of the **StateBag class**, to store the property values. The values are then passed as a variable to an HTML hidden input element when subsequent requests are processed.

If you want to add one variable in View State,

```
ViewState["Var"]=Count;
```

For Retrieving information from View State

```
string Test=ViewState["Var"];
```

Let us see how ViewState is used with the help of the following example. In the example we try to save the number of postbacks on button click.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack)
    {
        if (ViewState["count"] != null)
        {
            int ViewstateVal = Convert.ToInt32(ViewState["count"]) + 1;
            Label1.Text = ViewstateVal.ToString();
            ViewState["count"]=ViewstateVal.ToString();
        }
        else
        {
            ViewState["count"] = "1";
        }
    }
}
```



```

}
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text=ViewState["count"].ToString();
}

```

How to make view state secure?

As I already discuss View state information is stored in a hidden filed in a form of Base64 Encoding String.

See the hidden-field which stores the values of view-state which you can see by source code of running web page by right clicking the running web page and selecting “view page Source “.

```

<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUJODM0Njg4MTI2D2QWAgIDD2QWAgIBDw8WAh4EVGV4dAUQR29vZGJ5ZSBldmVyeW9u
ZWRkZGBCVv7h7x9H+pYdRYZWKtdwrtwm5R2vacXAgSsvG8w" />

```

Many of ASP.NET Programmers assume that this is an **Encrypted format**, but I am saying it again, that this is not a encrypted string. It can be break easily. To make your view state secure, There are two option for that,

- First**, you can make sure that the view state information is tamper-proof by using "**hash code**". You can do this by adding "**EnableViewStateMAC=true**" with your page directive. MAC Stands for "**Message Authentication Code**"

```

<%@ Page Language="C#" EnableViewState="true" EnableViewStateMac="true"

```

A **hash code** , is a cryptographically strong **checksum**, which is calculated by ASP.NET and its added with the view state content and stored in hidden filed. At the time of next post back, the checksum data again verified , if there are some mismatch, Post back will be rejected. we can set this property to web.config file also.

- Second** option is to set **ViewStateEncryptionMode="Always"** with your page directives, which will encrypt the view state data. You can add this in following way

```

<%@ Page Language="C#" EnableViewState="true" ViewStateEncryptionMode="Always"

```

It **ViewStateEncryptionMode** has three different options to set:

- Always**
- Auto**
- Never**

Always, mean encrypt the view state always, **Never** means, Never encrypt the view state data and **Auto** Says , encrypt if any control request specially for encryption. For auto , control must call **Page.RegisterRequiresViewStateEncryption()** method for request encryption.

we can set the Setting for "**EnableViewStateMAC**" and **ViewStateEncryptionMode**" in web.config also.

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

Advantages of using view state are:

- No server resources are required** The view state is contained in a structure within the page code.
- Simple implementation** View state does not require any custom programming to use. It is on by default to maintain state data on controls.
- Enhanced security features** The values in view state are hashed, compressed, and encoded for Unicode implementations, which provides more security than using hidden fields.

Disadvantages of using view state are:

- Performance considerations** Because the view state is stored in the page itself, storing large values can cause the page to slow down when users display it and when they post it. This is especially relevant for mobile devices, where bandwidth is often a limitation.
- Device limitations** Mobile devices might not have the memory capacity to store a large amount of view-state data.
- Potential security risks** The view state is stored in one or more hidden fields on the page. Although view state stores data in a hashed format, it can still be tampered with. The information in the hidden field can be seen if the page output source is viewed directly, creating a potential security issue. For more information,

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.

Cookies provide a means in Web applications to store user-specific information. For example, when a user visits your site, you can use cookies to store user preferences or other information. When the user visits your Web site another time, the application can retrieve the information it stored earlier.

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 are also known by many names, HTTP Cookie, Web Cookie, Browser Cookie, Session Cookie, etc.

Type of Cookies?

1. Persist Cookie - A cookie has not have expired time Which is called as Persist Cookie
2. Non-Persist Cookie - A cookie has expired time Which is called as Non-Persist Cookie

Writing Cookies

The browser is responsible for managing cookies on a user system. Cookies are sent to the browser via the `HttpResponse` object that exposes a collection called `Cookies`. You can access the `HttpResponse` object as the `Response` property of your `Page` class. Any cookies that you want to send to the browser must be added to this collection. When creating a cookie, you specify a `Name` and `Value`. Each cookie must have a unique name so that it can be identified later when reading it from the browser. Because cookies are stored by name, naming two cookies the same will cause one to be overwritten.

You can also set a cookie's date and time expiration. Expired cookies are deleted by the browser when a user visits the site that wrote the cookies. The expiration of a cookie should be set for as long as your application considers the cookie value to be valid. For a cookie to effectively never expire, you can set the expiration date to be 50 years from now.

If you do not set the cookie's expiration, the cookie is created but it is not stored on the user's hard disk. Instead, the cookie is maintained as part of the user's session information. When the user closes the browser, the cookie is discarded. A non-persistent cookie like this is useful for information that needs to be stored for only a short time or that for security reasons should not be written to disk on the client computer. For example, non-persistent cookies are useful if the user is working on a public computer, where you do not want to write the cookie to disk.

There are two ways to write a cookie to a user's computer. You can either directly set cookie properties on the Cookies collection or you can create an instance of the HttpCookie object and add it to the Cookies collection. You must create cookies before the ASP.NET page is rendered to the client. For example, you can write a cookie in a Page_Load event handler but not in a Page_Unload event handler.

Cookies with One Value

There are many ways to create cookies, I am going to outline some of them below:

Non Persistence cookies :

First way :

By using HttpCookie class , see below :

```
HttpCookie cookie = new HttpCookie("info",TextBox1.Text);

//can also give value as below as alternate to above code where it is given with cookie name

// cookie.Value = TextBox1.Text;

Response.Cookies.Add(cookie);
```

Second way :

By using Response class

```
Response.Cookies["info"].Value = TextBox1.Text;
```

Persistence cookie :

First way :

By using HttpCookie class , see below :

```
HttpCookie cookie = new HttpCookie("info",TextBox1.Text);

//can also give value as below as alternate to above code where it is given with cookie name

// cookie.Value = TextBox1.Text;

cookie.Expires = DateTime.Now.AddSeconds(30);
Response.Cookies.Add(cookie);
```

Second way :

By using Response class

```
Response.Cookies["info"].Value = TextBox1.Text;  
Response.Cookies["info"].Expires = DateTime.Now.AddSeconds(30);
```

Cookies with more than one Value

By using HttpCookie class:

```
HttpCookie cookie = new HttpCookie("info");  
cookie["name"] = TextBox1.Text;  
cookie["city"] = TextBox2.Text;  
cookie.Expires = DateTime.Now.AddSeconds(30);  
Response.Cookies.Add(cookie);
```

By using Response Class :

```
Response.Cookies["info"]["name"] = TextBox1.Text;  
Response.Cookies["info"]["city"] = TextBox2.Text;  
Response.Cookies["info"].Expires = DateTime.Now.AddSeconds(30);
```

Reading Cookies

For reading cookie with single value :

```
string name = Request.Cookies["info"].Value;  
Response.Write(name);
```

For Reading Cookie with multiple values :

First way :

```
HttpCookie ck = Request.Cookies["info"];  
Response.Write(ck["name"]);  
Response.Write(ck["city"]);
```

Second way :

```
string name = Request.Cookies["info"]["name"];
string city = Request.Cookies["info"]["city"];
Response.Write(name);
Response.Write(city);
```

Modifying Cookies:

You cannot directly modify a cookie. Instead, changing a cookie consists of creating a new cookie with new values and then sending the cookie to the browser to overwrite the old version on the client. The following code example shows how you can change the value of a cookie that stores a count of the user's visits to the site:

Let us take below example where we are checking first the existence of cookie and then modifying it accordingly, you may write the below code

```
int counter;
if (Request.Cookies["counter"] == null)
    counter = 0;
else
{
    counter = int.Parse(Request.Cookies["counter"].Value);
}
counter++;

Response.Cookies["counter"].Value = counter.ToString();
Response.Cookies["counter"].Expires = DateTime.Now.AddDays(1);
```

Example 1:

Let us see the example below where we are asking user to provide some details like :

Name	<input type="text" value="Deepak Singh"/>
City	<input type="text" value="Noida"/>
Color	<input type="text" value="Yellow"/>
<input type="button" value="Set Cooki"/> <input type="button" value="Redirect"/>	

Provide the above details and then click on Set Cookie button to set the cookies and then redirect to new page displaying the saved details from cookies .

Below is the code-behind page of above page :

```

public partial class Default8 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Button1_Click(object sender, EventArgs e)
    {

        Response.Cookies["info"]["name"] = TextBox1.Text;
        Response.Cookies["info"]["city"] = TextBox2.Text;
        Response.Cookies["info"]["color"] = TextBox3.Text;
        Response.Cookies["info"].Expires = DateTime.Now.AddMinutes(2);

    }
    protected void Button2_Click(object sender, EventArgs e)
    {
        Response.Redirect("~/Default9.aspx");
    }
}

```

As you will redirect o Default9.aspx page , it will fetch the details from cookies and display it accordingly.

See the code below of Page Default9.aspx.cs

```

protected void Page_Load(object sender, EventArgs e)
{
    if (Request.Cookies["info"] != null)
    {
        string name = Request.Cookies["info"]["name"];
        string city = Request.Cookies["info"]["city"];
        string color = Request.Cookies["info"]["color"];
        Label1.Text= "<h1>Welocome , " + name + " you belong to City : " + city + "</h1>";
        div1.Style.Add(HtmlTextWriterStyle.BackgroundColor, color);
        div1.Style.Add(HtmlTextWriterStyle.Height, "400");
        div1.Style.Add(HtmlTextWriterStyle.BorderStyle, "Inset");

    }
    else
    {
        Response.Write("<h1>Sorry Cannot retrive Cookie Values , Cookies Expired or not found </h1>");
        Label1.Text = "";
    }
}

```

On Page Load Event , we have firstly checked for the existence of cookie and then fetching details otherwise will display error.

Let us see , it's output below :

Welocome , Deepak Singh you belong to City : Agra

In case of cookie not found or cookie get expired the below message will be displayed :

Sorry Cannot retrive Cookie Values , Cookies Expired or not found

Example 2:

This example is just a little changes to previous example with adding one more cookie to display the last visited date to user for the web page accepting details like Name , City and Color .

Need to add some codes on code-behind page file on load event as below :

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.Cookies["time"] != null)
    {
        string time = Request.Cookies["time"].Value.ToString();
        DateTime dt = DateTime.Parse(time);
        Response.Write("You Last Visited This Web Site on : <b>" + dt.ToString("dd-MM-yyyy")+"</b>");
    }
    else
    {
        Response.Cookies["time"].Value = DateTime.Now.ToString();
        Response.Cookies["time"].Expires = DateTime.Now.AddMonths(3);
    }
}
```

here we have declared new cookie with name “test” which is holding the first creation time from system and later retrieving the same date from cookie and parsing it to date time . We are also changing the date time format into mentioned format by using toString() methodolgy pattern as “dd-MM-yyyy”.

Deleting Cookies:

You cannot directly delete a cookie on a user's computer. However, you can direct the user's browser to delete the cookie by setting the cookie's expiration date to a past date. The next time a user makes a request to a page within the domain or path that set the cookie, the browser will determine that the cookie has expired and remove it.


To assign a past expiration date on a cookie

1. Determine whether the cookie exists, and if so, create a new cookie with the same name.
2. Set the cookie's expiration date to a time in the past.
3. Add the cookie to the [Cookies](#) collection object.

Let us delete the cookie on the web page with name **Default9.aspx** which is displaying the records from cookie like name,city and changing the color .

Take one button on the same web page and do the code as given below on click event handler:

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (Request.Cookies["info"] != null)
    {
        HttpCookie myCookie = new HttpCookie("info");
        myCookie.Expires = DateTime.Now.AddDays(-1);
        Response.Cookies.Add(myCookie);
    }
}
```



Welocome , Deepak you belong to City : Delhi

Delete Cookie

As you will click on the button as shown above with text Delete Cookie, Cookie will be removed , and after opening the same web page again , you will get the same message of not found cookie .

I hope most of the things regarding cookie is understood by you , rest it is your operation on it .

Advantages of Cookies

Following are the main advantages of using cookies in a web application:

- It's very simple to use and implement.
- Browser takes care of sending the data.
- For multiple sites with cookies, the browser automatically arranges them.

Disadvantages of Cookies

The main disadvantages of cookies are:

- It stores data in simple text format, so it's not secure at all.
- There is a size limit for cookies data (4096 bytes / 4KB).
- The maximum number of cookies allowed is also limited. Most browsers provide limits the number of cookies to 20. If new cookies come, the old ones are discarded. Some browsers support up to 300.
- We need to configure the browser. Cookies will not work on a high security configuration of the browser. [I have explained this in details.]

Query Strings

A query string is information that is appended to the end of a page URL. For more information, see ASP.NET State Management Overview.

You can use a query string to submit data back to your page or to another page through the URL. Query strings provide a simple but limited way of maintaining some state information. For example, query strings are an easy way to pass information from one page to another, such as passing a product number to another page where it will be processed.

Put this code to your submit button event handler.

```
private void btnSubmit_Click(object sender, System.EventArgs e)
{
    Response.Redirect("default2.aspx?Name=" + TextBox1.Text + "&LastName=" + TextBox2.Text);
}
```

Reading data from query string in Default2.aspx page:

```
private void Page_Load(object sender, System.EventArgs e)
{
    string name = Request.QueryString["Name"];
    string lname = Request.QueryString["LastName"];

    Response.Write("Your First Name is : " + name + "<br/>");
    Response.Write("Your Last Name is : " + lname + "<br/>");
}
```

Advantages of using query strings are:



- No server resources are required** : The query string is contained in the HTTP request for a specific URL.
- Widespread support** : Almost all browsers and client devices support using query strings to pass values.
- Simple implementation** : ASP.NET provides full support for the query-string method, including methods of reading query strings using the [Params](#) property of the [HttpRequest](#) object.

Disadvantages of using query strings are:

- Potential security risks** The information in the query string is directly visible to the user via the browser's user interface. A user can bookmark the URL or send the URL to other users, thereby passing the information in the query string along with it. If you are concerned about any sensitive data in the query string, consider using hidden fields in a form that uses POST instead of using query strings. For more information, see [ASP.NET Web Application Security](#) and [Basic Security Practices for Web Applications](#).
- Limited capacity** Some browsers and client devices impose a 2083-character limit on the length of URLs.

Client-Side Method State Management Summary

The following table lists the client-side state management options that are available with ASP.NET, and provides recommendations about when you should use each option.

State management option	Recommended usage
View state	Use when you need to store small amounts of information for a page that will post back to itself. Using the ViewState property provides functionality with basic security.
Control state	Use when you need to store small amounts of state information for a control between round trips to the server.
Hidden fields	Use when you need to store small amounts of information for a page that will post back to itself or to another page, and when security is not an issue. <div> Note You can use a hidden field only on pages that are submitted to the server.</div>
Cookies	Use when you need to store small amounts of information on the client and security is not an issue.
Query string	Use when you are transferring small amounts of information from one page to another and security is not an issue. <div> Note You can use query strings only if you are requesting the same page, or another page via a link.</div>

Server Side State Management

Server side state management is different from client side state management but the operations and working is somewhat same in functionality. In server side state management, all the information is stored in the user memory. Due to this functionality, there are more secure domains at server side in comparison to client side state management.

Server-side options for storing page information typically have higher security than client-side options, but they can use more Web server resources, which can lead to scalability issues when the size of the information store is large. ASP.NET provides several options to implement server-side state management.

Server Side | Technique

Server side state management techniques are:

- Session State
- Application State
- Profile Properties
- Database Support

ASP.NET Session State

ASP.NET session state enables you to store and retrieve values for a user as the user navigates ASP.NET pages in a Web application. HTTP is a stateless protocol. This means that a Web server treats each HTTP request for a page as an independent request. The server retains no knowledge of variable values that were used during previous requests. ASP.NET session state identifies requests from the same browser during a limited time window as a session, and provides a way to persist variable values for the duration of that session. By default, ASP.NET session state is enabled for all ASP.NET applications.

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.

Let us understand the concept of Session , see the below example :

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class Default10 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write("Session Id : " + Session.SessionID + "<br/>");
        Response.Write("Is Session New : " + Session.IsNewSession + "<br/>");
        Response.Write("Is Cookie Less : " + Session.IsCookieless + "<br/>");
        Response.Write("Timeout : " + Session.Timeout + "<br/>");
        Response.Write("No of objects : " + Session.Count + "<br/>");

    }
}
```

Above is the simple example which displays the session details every time user requests for the web page. You may see , every time we get the different session id because browser is not saving the session id on client-Side by using cookies.

Cookies are maintained only if you create the session variables . Let us see how we can create it :

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.

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.

Session Variables Using Cookies:

Session variables are stored in a **SessionStateItemCollection** object that is exposed through the **HttpContext.Session** property. In an ASP.NET page, the current session variables are exposed through the Session property of the Page object.

The collection of session variables is indexed by the name of the variable or by an integer index. Session variables are created by referring to the session variable by name. You do not have to declare a session variable or explicitly add it to the collection.

See the example below :

```
public partial class Default10 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            Session["num"] = 1;

        }

        Response.Write("Session Id : " + Session.SessionID + "<br/>");
        Response.Write("Is Session New : " + Session.IsNewSession + "<br/>");
        Response.Write("Is Cookie Less : " + Session.IsCookieless + "<br/>");
        Response.Write("Timeout : " + Session.Timeout + "<br/>");
        Response.Write("No of objects : " + Session.Count + "<br/>");
        Response.Write("The value of session is : " + Session["num"].ToString());
    }
}
```

Here we have created a session variable with key “num” and initialized with value 1. Further due to creating the session variable, this time server will send the SessionId to browser and will be saved it in a cookie by name “**ASP.NET_SessionId**”. That you can see in cookie information as shown below :

Cookies and site data



Site

Locally stored data

Remove all shown

loca

localhost

1 cookie

ASP.NET_SessionId

Name: ASP.NET_SessionId
Content: gzzwcvcu0ge5sexjmzsy2cmh
Domain: localhost
Path: /
Send for: Any kind of connection
Accessible to script: No (HttpOnly)
Created: Thursday, April 2, 2015 at 6:08:42 PM
Expires: When the browsing session ends

Remove

