

State Management

A web page is stateless, which means that it is not capable of storing any information by itself. This is because whenever a request for a web page comes from the client by post-back of a web page, the same page will not be returned to the client after processing. In fact a new instance of the Page class is created and is given to the client. This means that all the controls are created again and so the information of the objects (page and its controls) has to be stored somewhere to again display them to the client.

By default ASP.NET provides state for all its controls and also for its web forms. But any data that is declared in code window as well as for HTML controls will not be provided any state by ASP.NET. So, ASP.NET provides several options to store data of the pages. This data can be stored at client as well as at server. The options provided for storing the data at client are:

- i. View state
- ii. Control state (introduced in ASP.NET 2.0)
- iii. Hidden fields
- iv. Cookies
- v. Query strings

The options provided for storing data at server are:

- i. Application state
- ii. Session state

How ASP.NET maintains controls' state in a stateless environment

ASP.NET uses internally a hidden field to maintain state for its content. This hidden field will be created by ASP.NET and will be attached to client results. Client carries hidden field just like other fields during request and response. With this hidden field server always contains the new data along the old data in the form of hidden field. ASP.NET uses multiple hidden fields for its form and its control values.

View state:

In order to maintain state for our own data we can use view state object of ASP.NET. View state stores all its data in the same hidden field that it uses to store page-view state.

All data stored in view state is encoded in simple base64 algorithm and stored in the hidden field as string. Though it is encoded, it is not fully secured form of data. We have to externally provide security addition when required. During page initialization the data in view state string is parsed and provided to the page.

View state can be used for storing any type of data because it is of object type. View state provides good performance for application without giving any load to server. However it is preferred to store simple form of data only in View state because the more

complex data, the more serialization and deserialization occurs for every request and response.

Ex: ViewState["a"]=0; --- stores the value 0 in object a.

Demo: View state storing complex data

In this demo we show how the data from a DataSet can be stored inside a View state.

For this let us design a simple form with a button and a GridView. When the button is clicked it should display the data from a jobs table in pubs database of sql data source.

For this we write the code for retrieving data from database in our data access layer. To create a data access layer add a new class and give name, say, DataLayer. We make this class static so that there is no need for creating its object in other layers. In the method GetData we write our code as follows:

Here we have specified the connection string in our web.config file.

```
static string s = ConfigurationSettings.AppSettings["cnstr"];

public static DataSet GetData(string selstr)

{
    SqlConnection cn = new SqlConnection(s);
    DataSet ds = new DataSet();
    SqlDataAdapter da = new SqlDataAdapter(selstr, cn);
    da.Fill(ds);
    return ds;
}
```

When the button is clicked, it should retrieve the data from this layer.

Normally for the data it is enough to call the method GetData and display the result in GridView. But with View state, we can directly read data from it without the need to make a trip to database again. For the first time we get data from database through GetData method and display the result in GridView, at the same time storing the data in View state. When the button is clicked, it first checks whether the data is present in View state, if yes it displays the result directly from it or else calls the GetData method.

We write the Button_Click code as follows:

```
protected void Button1_Click(object sender, EventArgs e)
{
    DataSet myds;
    if (ViewState["dsdata"] == null)
    {
        myds = DataLayer.GetData("select * from jobs");
        ViewState["dsdata"] = myds;
    }
    else
        myds = (DataSet)ViewState["dsdata"];
    GridView1.DataSource = myds.Tables[0];
    GridView1.DataBind();
}
```

}

Control State:

Control state is a new feature introduced in ASP.NET 2.0 to overcome some of the limitations of view state. This works very similar to the view state, but has the advantage of providing persistent data during post-backs. The ControlState property is used to store data that is specific to a control and it cannot be switched off like a view state. That means the data in control state persists even when view state for the page is switched off. This is very much essential when there is some critical data associated with a control which should not be lost during post-backs.

Hidden fields:

Hidden field is a ASP.NET server-side control which can be used to store normal data during postbacks. As it is a hidden field, it is not visible at the browser though it is rendered as a simple html field. It can be used to store any page specific values which are sent to the server in http form during postbacks for processing. As the data in hidden fields can be easily read, it is used only for storing data which doesn't need much security. Hidden fields are available only when the page is submitted using HTTP POST command.

Ex:

In this simple example, we see how hidden field value can be retrieved on postback.

First we create the hidden field control with id "Message" in our code window.

```
<input type="hidden" id="Message" value="Hello..how r u?"  
runat="server" />
```

When the page is rendered, the hidden field control is rendered in html form like this:

```
<input name="ctl00$ContentPlaceholder1$Message" type="hidden"  
id="ctl00_ContentPlaceholder1_Message" value="Hello..how r u?" />
```

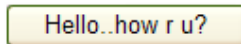
Now to read this value after postback, we take a simple form with a button control on it. Our intention is to show the hidden field value as the button text when it is clicked (when the button performs the postback of the page). Initially the text on the button is "Show Message" as shown below.

Show Message

For this we write the Button_Click event like this:

```
Button1.Text = Message.Value;
```

When the button is clicked, the value in the hidden field is read and assigned to the button text which is displayed when the page is posted back as show below:



Hello..how r u?

As shown here, hidden field can be used to store any simple values which can be read easily.

Cookies:

Till now we have seen state management only at page level. But to maintain state across pages we use cookies. Cookie is a small piece of information stored in the form of a file. Cookie is like a variable and it is maintained at the client side.

Cookies are of two types:

- i. In-memory cookies
- ii. Persistent cookies

A cookie that exists as long as user is working with application is called an in-memory cookie.

A cookie that exists when user is working with application as well as when user leaves the application is called persistent cookie (online & offline). To make cookie available offline, browser stores it in its temporary files location as a file with cookie value or some browser as a file with multiple cookie values.

The main points to remember when using cookies are:

- All storage cookies are based on domains (optionally with the path of domain). There is a limit for the cookies per domain in every browser. The Internet Explorer standard limitation is it can store ten cookies per domain and the size limit is 1 MB. This is the standard limitation, but can be changed as needed.
- Cookies are totally unsecured- since they are maintained by the browser itself, there are chances of manipulations.
- Cookies' information travels in the form of HTTP headers in plain textual format. (However cookies' data can be encrypted using SSL (Secure Socket Layer) but nobody encrypts and uses cookies because encryption will load on performance).
- We cannot guarantee the existence of the cookie. So always check whether cookie exists and then perform the job.

Creating cookies:

In JavaScript:

To set a cookie in JavaScript we should write a function for it. For this we show a simple demo to create and read a cookie using JavaScript.

To create a cookie and read a cookie we write two JavaScript functions as shown below. The SetCookie function creates a cookie that will expire in any specified number of days. If the days are not specified then by default it will expire in 4 days.

The GetCookie function reads the created cookie when the cookie name is passed to it.

```
<script language="javascript">
    function SetCookie(cName, cValue, expDays) {
        var today = new Date();
        var expire = new Date();
        if (expDays == null || expDays == 0)
            expDays = 4;
        expire.setTime(today.getTime() + 3600000 * 24 * expDays);
        document.cookie = cName + "=" + escape(cValue)
            + ";expires=" + expire.toGMTString();
    }
</script>

    function GetCookie(cName) {
        if (document.cookie.length > 0) {
            startIndex = document.cookie.indexOf(cName + "=");
            if (startIndex != -1) {
                startIndex = startIndex + cName.length + 1;
                endIndex = document.cookie.indexOf(";", startIndex);
                if (endIndex == -1) endIndex = document.cookie.length;
                return unescape(document.cookie.substring(startIndex,
                    endIndex));
            }
        }
        return "";
    }
}
```

To read the value of this cookie we take a simple html button. When the button is clicked, the cookie value is read and assigned to the button title.

```
<body>
    <form id="form1" runat="server">
        <div>
            <script language="javascript">
                SetCookie("mycookie", "hello", 3);
            </script>
            <button id="btn1" onclick="f1()"></button>
        </div>
    </form>
</body>
</html>
```

To read the cookie value and assign it to button we use this simple script:

```
function f1() {  
    form1.btn1.value = GetCookie("mycookie");  
}
```

This is how we can create and read cookies in JavaScript.

Cookies with ASP.NET:

For creating cookies using ASP.NET we have two options:

- i. Using Response.Cookies collection
- ii. Using HttpCookie class of .NET

Similarly for reading cookies we use Request.Cookies collection.

Demo: Using HttpCookie class for creating and reading in-memory cookies:

In this we create a cookie in one page and read it from another page. For this we create a form with a text box, a button and a label for the text box. In another page we place a label to display the cookie value on page load. To navigate to the second page we place a link button on the first page.

To create a cookie we write the following code in Button_Click event of first form:

```
HttpCookie x = new HttpCookie("user");  
x.Value = TextBox1.Text;  
Response.Cookies.Add(x);  
Response.Write("Information stored");
```

Similarly to read the cookie and display the value in label, we write the following code in Page_Load event of the second page:

```
if (Request.Cookies["user"] != null)  
    Label1.Text = Request.Cookies["user"].Value;  
else  
    Label1.Text = "Guest";
```

This will display the user name if a cookie had already been created for that user, otherwise it will display the user as Guest.

Creating a persistent cookie with ASP.NET

Persistent cookie is based on date and time. This type of cookie is accessible across pages even when the application is closed and reopened, as long as it exists (before its expiry time). For persistent cookie the method of cookie creation is same like in the previous case, but here we specify the expiry time for the cookie.

```
HttpCookie x = new HttpCookie("user");  
x.Value = TextBox1.Text;  
x.Expires = DateTime.Now.AddDays(2); //expires in 2 days  
Response.Cookies.Add(x);
```

To make the cookies exist at the client without getting expired, 'Remember Me' option is used. This uses the Forms authentication ticket which contains Forms authentication cookie. The standard expiry date is 50 years from the creation of cookie in this option provided that this persistent cookie is not deleted explicitly by the client. The default authentication ticket name is .ASPXAUTH, to which we assign a value when the user signs in. But it is always preferable to give a custom name for this ticket as sometimes conflicts may arise with other cookies.

To remove this cookie from the browser we have an option called 'Forget Me'. When this option is selected, the persistent cookie that exists at the browser is deleted and the user cannot login automatically the next time. To make this option work we use the Signout method of FormsAuthentication class. This removes the authentication ticket from the browser.

QueryString:

It is a format that is used to transfer values from one page to other page. Whenever form is submitted the values are posted to server by browser using query string format only. The query string format is

<program name>?<name>=<value>&<name>=<value>....

Ex: [Demo.aspx?a=10&b=20&c=30](#)

The example shows that we are calling the page Demo.aspx by passing three values to the server. We can prepare query string for our programmatic requirements and pass values between stateless pages.

Ex1 :

[Response.write\(Request\["TextBox1"\]\);](#) This is to get a text box value in the same form

Ex2:

In the first form(calling form)

[Response.Redirect\("view.aspx?a="+TextBox1.Text\);](#)

In second form(called form)

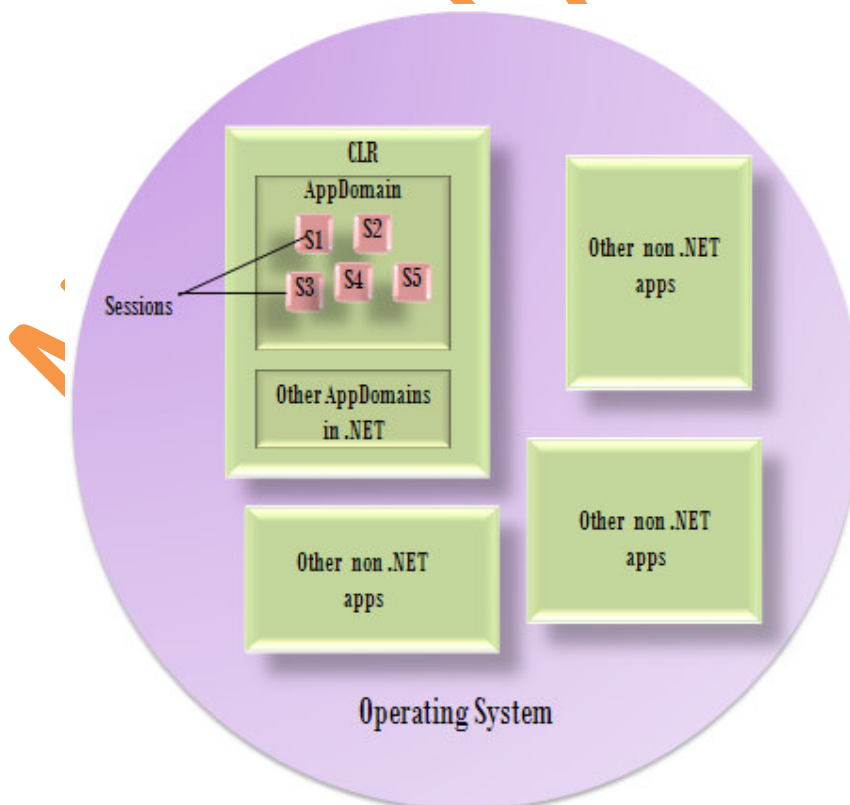
[ListBox1.Text=Request\["a"\];](#)

Here Response.Redirect is calling view.aspx along with one query string parameter (a). We need to do this because values from one page are not accessible to new page especially in case of Response.Redirect. The value of (a) is passed to view.aspx through the query string and that value can be read using Request property.

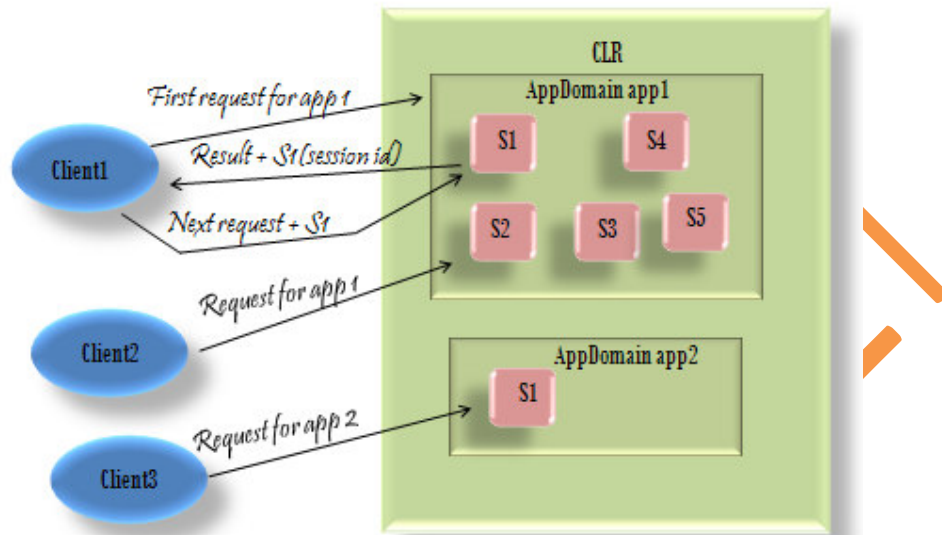
Application and Session Objects of ASP.NET

Application and session objects are also used for state management. These objects of ASP.NET represent both ASP.NET application and sessions which are created based on user requests and responses.

- In .NET every application is called as AppDomain and it is created by CLR. In case of ASP.NET application AppDomain is created when first user makes a request for resource and then onwards AppDomain will exist as long as users are active and all server side objects exist.
- Unlike application a session is created for every user who makes a request for a resource. Sessions are created to identify user and exist as long as user is active or based on their time. All sessions by default run under application process only. ASP.NET even provides sessions to be separated from application which means that application and session run as two different processes.
- Application and session processes run for every request and response and ASP.NET provides two objects called Application and Session, which are used to refer these processes. We can use these objects to create data with their scope and also in the form of events.



For the first time when a client requests an app, server creates AppDomain and creates a session for the client. If next client requests the same app, it only creates another session and this gives session ID to that client.



If user does not respond within timeout period, the session will be destroyed by the server. By default in ASP.NET the session timeout is 20 mins which can be increased or decreased.

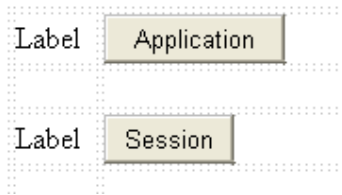
Variables in application and sessions

```
Ex: Application["x"]=10;  
Application["dsdata"]=ds;  
Session["y"]=1; (gets created only in current session)
```

Application and Session variables are of type object and as we know object can store any form of data, these objects can store any data. Sessions and their content are stored at server only and session Id is the only value that will travel from client to server and server to client.

Demo: In this simple demo we'll see how Session and Application variables are incremented each time the button is clicked.

For this, design a form with two labels and two buttons.



The first label and button correspond to Application variable and the second label and button correspond to Session variable.

```
protected void btnApplication_Click(object sender, EventArgs e)
{
    if (Application["a"] == null)
        Application["a"] = 0;
    Application["a"] = Convert.ToInt32(Application["a"]) + 1;
    Label1.Text = Application["a"].ToString();
}

protected void btnSession_Click(object sender, EventArgs e)
{
    if (Session["b"] == null)
        Session["b"] = 0;
    Session["b"] = Convert.ToInt32(Session["b"]) + 1;
    Label2.Text = Session["b"].ToString();
}
```

As Application variables are common to all clients, the changes made by one client to that variable get reflected to other clients as well. But Session variable is different for different clients, so it does not get affected by other clients.

If we run the above program from different clients, the session value will be different for different clients, but all have the same Application variable value.

Difference between Sessions and Cookies

Sessions and cookies store data at user level i.e. they are accessible to all pages of the user.

Differences between Sessions and Cookies:

- Sessions are created at server side
Cookies are created at client side
- Sessions are unlimited
Cookies are limited
- Default time-out for sessions
For cookies time-out should be explicitly specified
- No persistence of sessions
Cookies can be persistent
- Sessions are secured
Cookies are not at all secure
- Sessions can store any type of data

Cookies cannot store complex data

Ex: Sessions are used for:

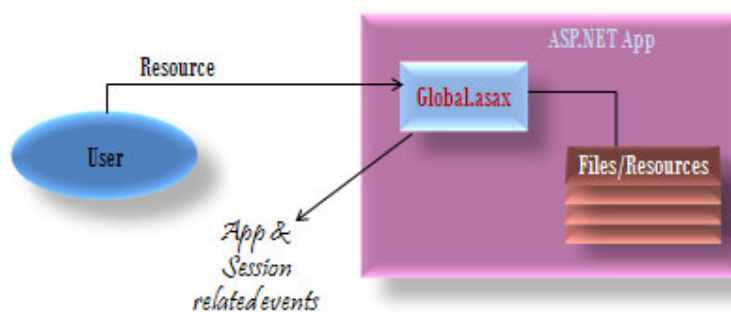
- Credit card numbers
- For passwords
- Identities
- Any important data

Cookies are used for

- Usernames
- Shopping items
- Preferences

Application and Session events:

Application and session variables can be declared in any part of the project. But application and Session events must be written in a special file called Active Server Application file. Global.asax is called as Active Server Application file. This file contains a class which extends HttpApplication class and for user it is a global application class.



Global.asax must be placed in the application root directory and by default VS will not create any Global.asax file as part of ASP.NET project. Using Add → New Item we have to explicitly create it. Global.asax is configured by self to reject any direct requests, which means that every user will execute Global.asax but when request is made to some resource of application, not directly. The contents of Global.asax are programs that are written in the form of events. All applications and session related events will be present in this file.

Some important events:

1. Application_Start:
Fires when the first user makes a request or when application starts. Normally we use this event to retrieve and initialize some start-up content from entire application.
2. Application_End:

Fires when ASP.NET application ends because of restart or when all sessions and objects are closed to end application

3. Application_Error:

When all below levels i.e. try-catch, page error and custom errors ignore or fail to handle error then Application_Error event fires.

4. Session_Start:

Whenever a new session is created Session_Start event fires

5. Session_End:

Fires when session is closed because of time-out and because of application end. We can end sessions programmatically also using Session_Abandon() method.

Demo:

In this demo we'll see how to find the number of users visiting a web page.

For this we design a simple form with just two labels to display the total number of users and also total number of online users.

Users count is taken as Application variables with name TotalUsers and OnlineUsers. Here TotalUsers represents the number of users who have visited the page and OnlineUsers represent the number of users online at that time. An Application variable can be changed from every session. So if we increment the variables value in Session_Start, it will increment the number of users by 1 whenever a new client runs the application. Similarly when the client quits, the OnlineUsers variable is decremented by one since we have written the code for that in Session_End event. We write the code for these events in our Global.asax file.

```
void Application_Start(object sender, EventArgs e)
{
    // Code that runs on application startup
    Application["TotalUsers"] = 0;
    Application["OnlineUsers"] = 0;
}

void Session_Start(object sender, EventArgs e)
{
    // Code that runs when a new session is started
    Application["TotalUsers"] =
        Convert.ToInt32(Application["TotalUsers"]) + 1;
    Application["OnlineUsers"] =
        Convert.ToInt32(Application["OnlineUsers"]) + 1;
}

void Session_End(object sender, EventArgs e)
{
    // Code that runs when a session ends.
    Application["OnlineUsers"] =
        Convert.ToInt32(Application["OnlineUsers"]) - 1;
}
```

Now in Page_Load event of our web page we write the following code:

```
Session.Timeout = 1; //Session expires in 1 minute
Label1.Text = Application["TotalUsers"].ToString();
Label2.Text = Application["OnlineUsers"].ToString();
```

Here we have specified the session time-out in Page_Load event. If we write the Session.Timeout in Page_Load event, it works only for the current page. To make the session timeout work even when navigating to other pages also, then we should specify the time-out in web.config file.

We can specify the time-out in our web.config file in <system.web> tag:

```
<system.web>
  <sessionState timeout="5"></sessionState>
  ....
</system.web>
```

Note: To kill sessions we should use Session.Abandon() method

Sessions outside application process:

Till now we have seen sessions that run under application process. By default, all sessions run under application process only. ASP.NET provides two more choices to create sessions outside the application so that application restarts don't affect the sessions and performance-wise we can do better.

- i. ASP.NET State Service
- ii. Using SQL Server (this is rarely used)

Out-of-process: Here sessions are maintained outside the application for better performance. These sessions exist even if the application goes down.

- i. To use ASP.NET state service for sessions, we have to follow the below steps:
 - a. Go to Services (Control Panel → Administrative Tools → Services)
 - b. Select ASP.NET State Service and click on Start
 - c. Go to web.config and specify Out-of-process session state using <sessionState> element.

Ex:

```
<sessionState mode="StateServer"
stateConnectionString="104.323.203.2" ..... >
</sessionState>
```

- ii. We can even store the ASP.NET sessions in SQL Server database. To create database to store these sessions, we need to run the utility aspnet_regsql.exe, which is located in the .NET framework directory. In web.config we can specify the session state as given below:

```
<sessionState mode="SQLServer" stateConnectionString="<connection
string>" ... >
</sessionState>
```

Drawbacks of Out-of-process sessions:

1. Maintaining another server is an overhead
2. Data will be serialized and deserialized between two systems for sharing. So preferred is to use simple types only in session which don't need serialization.

In this article, we have seen different ways for managing the state in web pages. Based upon our requirement we should choose the right options optimising performance and maintaining security.

Nagaraj ASP.NET