Unit 8

9.0 State Management

(Text Book Chapter 08)

9.0 State Management

9.0 Introduction

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

A traditional Windows program, users interact with a continuously running application. A portion of memory on the desktop computer is allocated to store the current set of working information. In a traditional web application, the story is quite a bit different. 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 postbacks or on multiple pages, you need to take additional steps. So in other words, traditional web applications do have a problem.

9.1 Solution

To overcome this inherent limitation of traditional Web programming, ASP.NET includes several options that help you preserve data on both a per-page basis and an application-wide basis. These features are as follows:

View state	Control state	Hidden fields
Cookies	Query strings	Application state
Session state	Profile Properties	

9.0 State Management

View state, **control state**, **hidden fields**, **cookies**, and **query strings** all involve **storing data on the client in various ways**. However, **application state, session state, and profile properties a**ll store data in memory on the server. Each option has distinct advantages and disadvantages, depending on the scenario.

9.2 ViewState

View Sate, is the technique used by an ASP.NET Web page to persist changes to the state of a Web Form across **postbacks** The view state of a page is, by default, placed in a hidden form field named **__VIEWSTATE**. This hidden form field can easily get very large, on the order of tens of kilobytes. Not only does the **__VIEWSTATE** form field cause slower downloads, but, whenever the user posts back the Web page, the contents of this hidden form field must be posted back in the HTTP request, thereby lengthening the request time, as well.

9.2.1 How does ViewState work?

All server controls have a property called ViewState. If this is enabled, the ViewState for the control is also enabled. Where and how is ViewState stored? When the page is first created all controls are serialized to the ViewState, which is rendered as a hidden form field named __ViewState. This hidden field corresponds to the server side object known as the ViewState. ViewState for a page is stored as key-value pairs using the System.Web.UI.StateBag object. When a post back occurs, the page de-serializes the ViewState and recreates all controls. The ViewState for the controls in a page is stored as base 64 encoded strings in name - value pairs. When a page is reloaded two methods pertaining to ViewState are called, namely the LoadViewState method and SaveViewState method. The following is the content of the __ViewState hidden field as generated for a pageIn a particular system.

```
<input type="hidden" name="__VIEWSTATE"
value="dNrATo45Tm5QzQ7Oz8AblWpxPjE9MMl0Aq765QnCmP2TQ==" />
```

9.2.2 The ViewState Collection

The ViewState property of the page provides the current view state information. This property provides an instance of the StateBag collection class. The StateBag is a dictionary collection, which means every item is stored in a separate "slot" using a unique string name. For example, consider this code:

```
// The this keyword refers to the current Page object. It's optional.
this.ViewState["Counter"] = 1;
```

This places the value 1 (or rather, an integer that contains the value 1) into the **ViewState** collection and gives it the descriptive name Counter. If currently no item has the name Counter, a new item will be added automatically. If an item is already stored under the name Counter, it will be replaced. When retrieving a value, you use the key name. You also need to cast the retrieved value to the appropriate data type using the casting. This extra step is required because the **ViewState** collection stores all item s as basic objects, which allows it to handle many different data types. Here's the code that retrieves the counter from view state and converts it to an integer:

```
int counter;
counter = (int)this.ViewState["Counter"];
```

Maintaining the ViewState is the default setting for ASP.NET Web Forms. If you want to NOT maintain the ViewState, include the directive **<%@ Page EnableViewState="false"** %> at the top of an .aspx page or Add the attribute **EnableViewState="false"** to any control.

See example following page.

9.0 State Management

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="ViewStateExample-2.aspx.cs"
Inherits="ViewStateExample_2._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:Button ID="button1" runat="server" Text="Click to
Count" OnClick="Count_Clicks" /> <br/>
   <asp:Label ID="label1" runat="server" Text="" />
   </div>
   </form>
</body>
</html>
```

Example 9.2-1a

```
using System;

using System.Xml.Linq;

namespace ViewStateExample_2
{
   public partial class _Default : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
       label1.Text = "Clicks : " +
          (ViewState["count_clicks"] == null ? 0 :
                (int) ViewState["count_clicks"]);
      }

      protected void Count_Clicks(object sender, EventArgse)
      {
         if (ViewState["count_clicks"] == null)
         {
            ViewState["count_clicks"] = 1;
            label1.Text = "Clicks : " +
                  (int)ViewState["count_clicks"];
         }
         else
         {
            ViewState["count_clicks"] =
               (int)ViewState["count_clicks"] + 1;
            label1.Text = "Clicks : " +
                   (int)ViewState["count_clicks"];
         }
      }
   }}
```

Example 9.2-1a

Start the first browser; initially the ViewState Count is 0

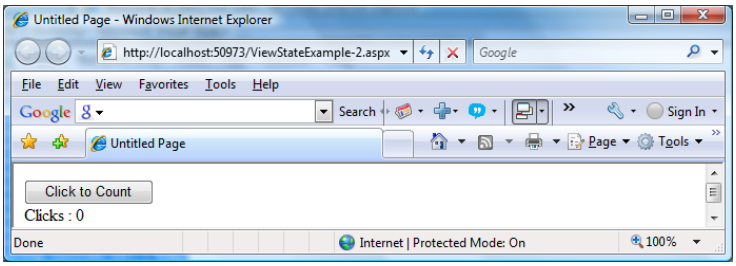


Figure 9.2-1a

Counter updates after Clicking the button several times

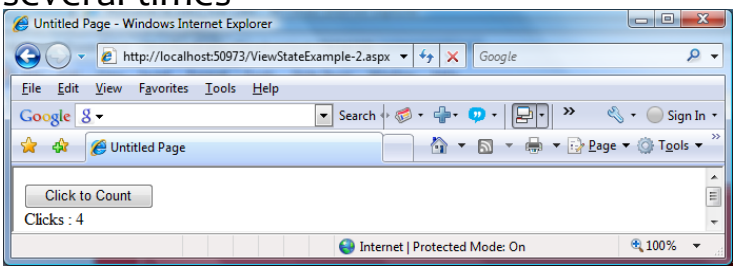


Figure 9.2-1b

Start a second browser while the first is still alive. The Count start at 0 for the new browser

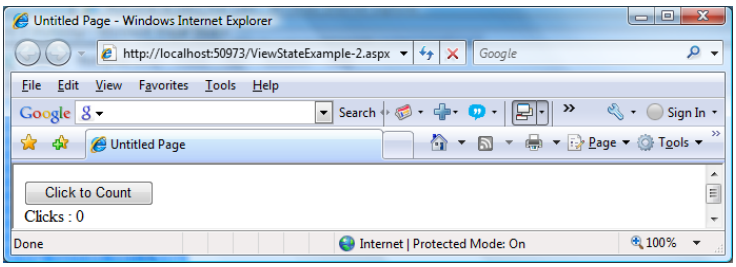


Figure 9.2-2a

Counter updates after Clicking the button several times

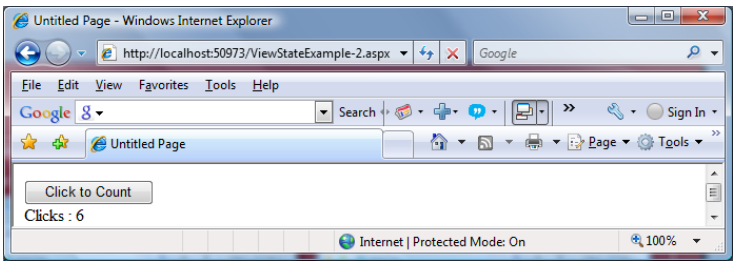


Figure 9.2-2b

9.0 State Management

If a counter need to be shared between the number of clients, then a counter need to be defined as static. Remember static objects are shared by all instances of a class.

```
using System;
.
using System.Xml.Linq;

namespace ViewStateExample_2
{
 public partial class _Default : System.Web.UI.Page
 {
  private static int count = 0;
  protected void Page_Load(object sender, EventArgs e)
  {
   label1.Text = "Clicks : " + count;
  }

  protected void Count_Clicks(object sender, EventArgs e)
  {
   if (ViewState["count_clicks"] == null)
   {
    count = 1;
    ViewState["count_clicks"] = count;
          label1.Text = "Clicks : " +
    (int)ViewState["count_clicks"];
   }
   else
   {
    count = count + 1;
     ViewState["count_clicks"] = count;
        label1.Text = "Clicks : " +
     (int)ViewState["count_clicks"];
   }
 }}}
```

Example 9.2-3

Start a client and click the button several times.

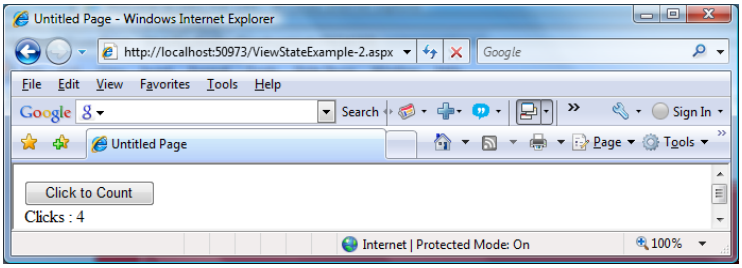


Figure 9.2-2a

Start a second client. The start count is where the last count updated by the first client.

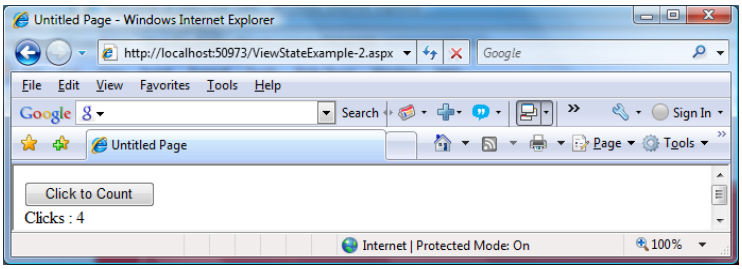


Figure 9.2-2b

9.2.3 Saving and Restoring Values to and from the ViewState

ViewState works with the following types.

Primitive types
Arrays of primitive types
ArrayList and Hashtable
Any other serializable object

To add an ArrayList object to the ViewState use the following statements.

```
ayList obj = new ArrayList(); //Some code ViewState["ViewStateObject"] = obj;
```

To retrieve the ArryList object use the following statement.

```
obj = ViewState["ViewStateObject"];
```

9.2.4 Making View State Secure

Previously we discussed about the format of the view state information string format that usually look like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="dDw3NDg2NTI5MDg7Oz4=" />
```

As you add more information to view state, this value can become much longer. Because this value isn't formatted as clear text, many ASP.NET programmers assume that their view state data is encrypted. It isn't. Instead, the view state information is simply patched together in memory and converted to a Base64 string (which is a special type of string that's always acceptable in an HTML document because it doesn't include any extended characters). A clever hacker could reverse-engineer this string and examine your view state data in a matter of seconds. Tamper-Proof View State If you want to make view state more secure, you have two choices.

First, you can make sure the view state information is tamperproof by instructing ASP.NET to use a hash code. **A hash code is sometimes described as a cryptographically strong checksum**. The idea is that ASP.NET examines all the data in view state, just before it renders the final page. It runs this data through a hashing algorithm (with the help of a secret key value). The hashing algorithm creates a short segment of data, which is the Hash code. This code is then added at the end of the view state data, in the final HTML that's sent to the browser. When the page is posted back, ASP.NET examines the view state data and recalculates the hash code using the same process. It then checks whether the checksum it calculated matches the hash code that is stored in the view state for the page. If a malicious user changes part of the view state data, ASP.NET will end up with a new hash code that doesn't match. At this point, it will reject the postback completely. (You might think a really clever user could get around this by generating fake view state information and a matching hash code.

However, malicious users can't generate the right hash code, because they don't have the same cryptographic key as ASP.NET. This means the hash

hash codes they create won't match.)Hash codes are actually enabled by default, so if you want this functionality, you don't need to take any extra steps.

9.2.5 Private View State

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

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

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

Or you can set the same attribute in a configuration file to configure view state encryption for all the pages in your website:

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

Either way, this enforces encryption. You have three choices for your view state encryption setting—always encrypt (Always), never encrypt (Never), or encrypt only if a control specifically requests it (Auto). The default is Auto, which means that the page won't encrypt its view state unless a control on that page specifically requests it. (Technically, a control makes this request by calling the **Page.RegisterRequiresViewStateEncryption()** method.) If no control calls this method to indicate it has sensitive information, the view state is not encrypted, thereby saving the encryption overhead. On the other hand, a control doesn't have absolute power—if it calls **Page.RegisterRequiresViewStateEncryption()** and the encryption mode is Never, the view state won't be encrypted.

9.0 State Management

9.3 Transferring Information Between Pages (Cross-Page Posting)

9.3.1 Using the `PostBackUrl` property

Normally we use the term **postback** when an ASP.NET page submits it's content back to that page itself. But there can be situation when a page needs to submit it's content to a different target page. This is known as cross page **postback**. In this post we will discuss about how to handle and implement cross page **postback** scenario. Now to make a page **postback** to another page we have set the **PostBackUrl** property as shown below:

```
<asp:Button ID="Button2" runat="server"
        Text="CrossPagePostback"
                PostBackUrl="~/TargetForm.aspx" />
```

After the content is submitted to the target url we need to retrieve the control values and properties of the page. to do this we have use the PreviousPage property of the Page class as shown below. The PreviousPage maintains a reference to the instance of the source page.

```
Label1.Text = "Crosspage Postback with value :" +
    (Page.PreviousPage.FindControl("TextBox1") as
    TextBox).Text;
```

Suppose we have a public property defined in the source page as shown below:

```
public string ScreenID { get; set; }
```

If we want to access this property in the target page then we need to specify the type of source page in the @PreviousPageType directive of the target page:

```
<%@ PreviousPageType
    VirtualPath="~/TargetForm.aspx"  %>
```

Then we can access the property in target page as follows:

```
Label1.Text = PreviousPage.ScreenID;
```

9.0 State Management

Using the *PostBackUrl* Property Example: the code for the start page

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting1.aspx.cs"
Inherits="CrossPagePosting1._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>
    First Name  :
    <asp:TextBox ID="txtFirstName"
          runat="server"></asp:TextBox> <br />
    Last name :
    <asp:TextBox ID="txtLastName“
       runat="server"></asp:TextBox> <br />
    <br />
    <asp:Button ID="btnPostCommand"
        Text="Click to Cross Page“
     PostBackUrl="CrossPagePosting2.aspx" runat="server" />
   </div>
   </form>
</body>
</html>
```

Example 9.3.-1a

```
using System;
.
using System.Xml.Linq;

namespace CrossPagePosting1
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
  }

  public string FirstName
  {
   get { return txtFirstName.Text; }
  }

  public string LastName
  {
   get { return txtLastName.Text; }
  }
 }
}
```

Example 9.3-1b

Cross page posting can be done in two ways. Both are explained below.

First Method : Here is the code for the Cross-Page using the directive **@PreviousPageType**

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting2.aspx.cs"
Inherits="CrossPagePosting2._Default" %>
<%@ PreviousPageType VirtualPath="~/CrossPagePosting1.aspx" %>

<!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="lblCapture" runat="server"></asp:Label> <br />
 </div>
 </form>
</body>
</html>
```

Example 9.3-1c

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
namespace CrossPagePosting2
{
 public partial class _Default : System.Web.UI.Page
 { protected void Page_Load(object sender, EventArgs e)
  {
   this.lblCapture.Text = "You typed : " +
          PreviousPage.FirstName + " as first name and "+
          PreviousPage.LastName + " as the last name";
    }
  }
}
```

Example 9.3-1d

9.0 State Management

Notice that in this method we have set up the `<%@ PreviousPageType VirtualPath= "~/CrossPagePosting1.aspx" %>` Property.

Second Method: is to use a statement like :
`CrossPagePosting1._Default previousPage = PreviousPage as CrossPagePosting1._Default;` In the Page_Load() method in code behind of the cross-page.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="CrossPagePosting2.aspx.cs"
Inherits="CrossPagePosting2._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="lblCapture" runat="server"></asp:Label> <br />
  </div>
  </form>
</body>
</html>
```

Example 9.3.-1e

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

namespace CrossPagePosting2
{
  public partial class _Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      CrossPagePosting1._Default previousPage = PreviousPage as
                                          CrossPagePosting1._Default;

      this.lblCapture.Text = "You typed : " + previousPage.FirstName +
                                          " as first name and "+
        previousPage.LastName + " as the last name";
    }
  }
}
```

Example 9.3-1f

In both cases you get the same result. You enter a first named and a last name in the first page and then click the "Click to Cross Page" button. The data fields of the first page can be then accessed in the second (Cross Page)

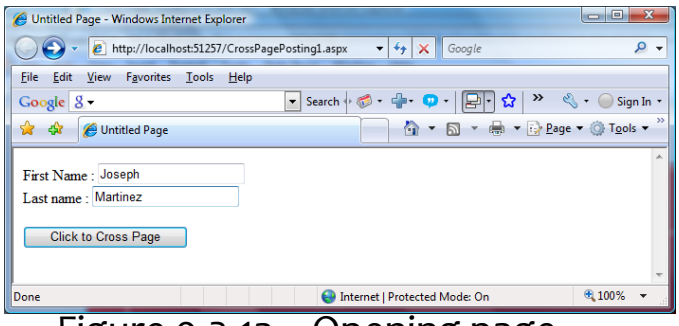


Figure 9.3-1a – Opening page

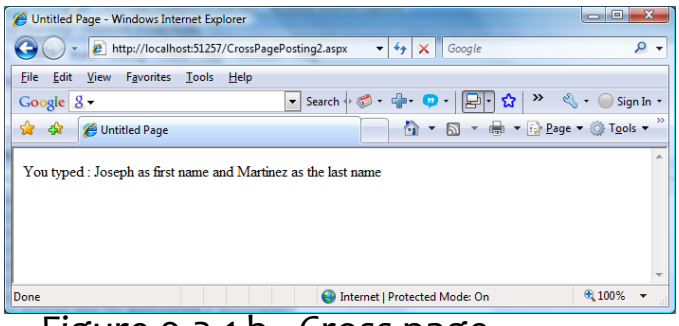


Figure 9.3-1 b– Cross page

9.3.2 Query String Approach (Using QueryString property)

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

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

The query string is the portion of the URL after the question mark. In this case, it defines a single variable named q, which contains the string organic+gardening. The advantage of the query string is that it's lightweight and doesn't exert any kind of burden on the server. However, it also has several limitations:

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

For that reason, you can't place a large amount of information in the query String and still be assured of compatibility with most browsers.

Adding information to the query string is still a useful technique. It's particularly well suited in database applications where you present the user with a list of items that correspond to records in a database, such as products. The user can then select an item and be forwarded to another page with detailed information about the selected item. One easy way to implement this design is to have the first page send the item ID to the second page. The second page then looks that item up in the database and displays the detailed information. You'll notice this technique in e-ommerce sites such as Amazon.

➢9.0 State Management

To store information in the query string, you need to place it there yourself. Unfortunately, you have no collection-based way to do this. Instead, you'll need to insert it into the URL yourself. Here's an example that uses this approach with the Response.Redirect() method:

```
// Go to newpage.aspx. Submit a single query string argument
// named recordID, and set to 10.

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

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

```
// Go to newpage.aspx. Submit two query string arguments:
// recordID (10) and mode (full).

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

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

```
string ID = Request.QueryString["recordID"];
```

Note that information is always retrieved as a string, which can then be converted to another simple data type. Values in the QueryString collection are indexed by the variable name. If you attempt to retrieve a value that isn't present in the query string, you'll get a null reference.

In the example in the next page presents a list of entries. When the user chooses an item by clicking the appropriate item in the list, the user is forwarded to a new page. This page displays the received ID number. This provides a quick and simple query string test with two pages. In a sophisticated application, you would want to combine some of the data control features that are described later.

9.0 State Management

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="QueryStringSender.aspx.cs"
Inherits="QueryStringDemo2.QueryStringSender" %>

<!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>
   <asp:ListBox ID="aListBox" runat="server" Width="175"
Height="100"></asp:ListBox><br/><br/>
   <asp:CheckBox ID="aCheckBox" runat="server"
Text="Show full details" /> <br/><br/>
   <asp:Button ID="aButton" runat="server" Text="View
Information" OnClick="cmdGo_Click" />
   <asp:Label ID="lblError" runat="server"></asp:Label>
   </div>
   </form>
</body>
</html>
```

Example 9.3.-2a

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="QueryStringRecipient.aspx.cs"
Inherits="QueryStringDemo2.QueryStringRecipient" %>

<!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>
   <asp:Panel ID="aPanel" runat="server" BorderColor=""
Width="100%" Height="80px" BackColor="Yellow">
      <asp:Label ID="aLabelR" runat="server" Font-Size="XX-
Large" ForeColor="#CC0000"></asp:Label>
   </asp:Panel>
   </div>
   </form>
</body>
</html>
```

Example 9.3.-2c

```
namespace QueryStringDemo2
{
   public partial class QueryStringSender :
System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e) {
         if (!this.IsPostBack)          {
            // Add sample values.
            aListBox.Items.Add("Econo Sofa");
            aListBox.Items.Add("Supreme Leather Drapery");
            aListBox.Items.Add("Threadbare Carpet");
            aListBox.Items.Add("Antique Lamp");
            aListBox.Items.Add("Retro-Finish Jacuzzi");
         }
      }

      protected void cmdGo_Click(object sender, EventArgs e)
      {
         if (aListBox.SelectedIndex == -1)          {
            lblError.Text = "You must select an item.";
         }
         else  {
            // Forward the user to the information page,
            // with the query string data.
            string url = "QueryStringRecipient.aspx?";
            url += "Item=" + aListBox.SelectedItem.Text + "&";
            url += "Mode=" + aCheckBox.Checked.ToString();
            Response.Redirect(url);
         }
      }
   }
}
```

Example 9.3.-2b

```
namespace QueryStringDemo2
{
   public partial class QueryStringRecipient :
System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         aLabelR.Text = "Item: " +
Request.QueryString["Item"];
         aLabelR.Text += "<br />Show Full Record: ";
         aLabelR.Text += Request.QueryString["Mode"];
      }
   }
}
```

Example 9.3.-2d

9.0 State Management

The following is the output produced by the example in the previous page.

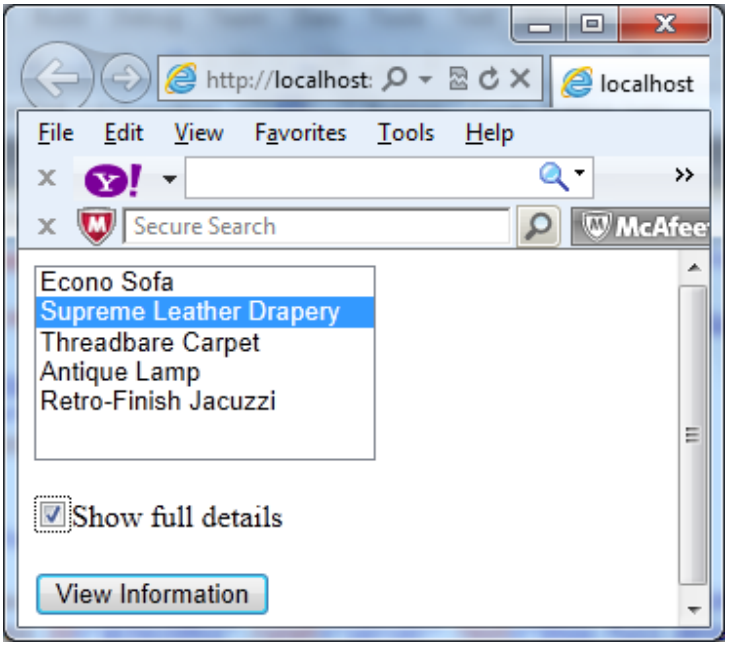


Figure 9.3-2a

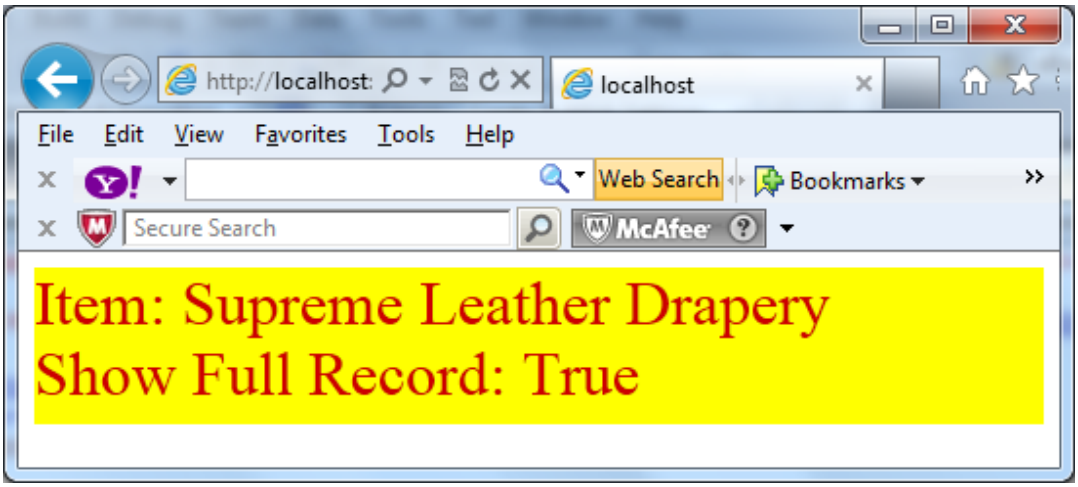


Figure 9.3-2b

One interesting aspect of this example is that it places information in the query string that isn't valid—namely, the space that appears in the item name. When you run the application, you'll notice that ASP.NET encodes the string for you automatically, converting spaces to the valid %20 equivalent escape sequence. The recipient page reads the original values from the QueryString collection without any trouble. This automatic encoding isn't always sufficient. To deal with special characters, you should use the URL encoding technique described in the next section.

9.3.2.1 Potential Problems in Query String Approach

URL Encoding:

One potential problem with the query string is that some characters aren't 1 allowed in a URL. In fact, the list of characters that are allowed in a URL is much shorter than the list of allowed characters in an HTML document. All characters must be alphanumeric or one of a small set of special characters (including $-_.+!*'(),). Some browsers tolerate certain additional special characters (Internet Explorer is notoriously lax), but many do not. Furthermore, some characters have special meaning. For example, the ampersand (&) is used to separate multiple query string parameters, the plus sign (+) is an alternate way to represent a space, and the number sign (#) is used to point to a specific bookmark in a web page. If you try to send query string values that include any of these characters, you'll lose some of your data. You can test this with the previous example by adding items with special characters in the list box.

To avoid potential problems, it's a good idea to perform URL encoding on text values before you place them in the query string. With URL encoding, special characters are replaced by escaped character sequences starting with the percent sign (%), followed by a two-digit hexadecimal representation. For example, the & character becomes %26. The only exception is the space character, which can be represented as the character sequence %20 or the + sign.

To perform URL encoding, you use the **UrlEncode()** and Url**Decode()** methods of the **HttpServerUtility** class. As you learned in Chapter 5, an **HttpServerUtility** object is made available to your code in every web form through the **Page.Server** property. The following code uses the **UrlEncode()** method to rewrite the previous example, so it works with product names that contain special characters:

```
String url = "QueryStringRecipient.aspx?"; url += "Item=“ +
Server.UrlEncode(lstItems.SelectedItem.Text) + "&";
url += "Mode=" + chkDetails.Checked.ToString();
Response.Redirect(url);
```

Notice that it's important not to encode everything. In this example, you can't encode the & character that joins the two query string values, because it truly is a special character.

You can use the **UrlDecode()** method to return a URL-encoded string to its initial value. However, you don't need to take this step with the query string. That's because ASP.NET automatically decodes your values when you access them through the **Request.QueryString** collection. (Many people still make the mistake of decoding the query string values a second time. Usually, decoding already decoded data won't cause a problem. The only exception is if you have a value that includes the + sign. In this case, using **UrlDecode()** will convert the + sign to a space, which isn't what you want.)

9.3.3 Using Cookies

A cookie is often used to identify a user. A cookie is a small file that the server embeds on the user's computer. Each time the same computer requests a page with a browser, it will send the cookie too. With ASP, you can both create and retrieve cookie values. Lets take a look at an example.

```
What is a Cookie?
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Cookie-Page1.aspx.cs" Inherits="Cookie_Page1._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">
<asp:TextBox ID=cookname runat="server" />Cookie Name<br/>
<asp:TextBox ID=cookvalue runat="server" />Cookie value<br/>
<asp:TextBox ID=cookexpiry runat="server" />Days to Expiration<br/>
<asp:Button  ID="submitbutton1" text="Set cookie" OnClick="SubmitButton_Click1" runat="server"/>
<asp:Button  ID="submitbutton2" text="Get cookie" OnClick="SubmitButton_Click2" runat="server"/>
</form>
<asp:Label ID="msg" runat="server" /><br/>
<asp:Label ID="stat" runat="server" />
</body>
</html>
```

Example 9.3-3-a

The Code Behind

```
using System;
using System.Xml.Linq;
namespace Cookie_Page1
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {}
  protected void SubmitButton_Click1(object sender, EventArgs e)
  {  msg.Text = SetCookie(cookname.Text, cookvalue.Text,
    Convert.ToInt32(cookexpiry.Text)).ToString();
    msg.Text += ". Expires: " +
      Response.Cookies[cookname.Text].Expires.ToString();
  }

  protected void SubmitButton_Click2(object sender, EventArgs e)
  { msg.Text = GetCookie(cookname.Text); }

  public bool SetCookie(string cookiename,
          string cookievalue, int iDaysToExpire)
  { try {
        HttpCookie objCookie = new HttpCookie(cookiename);
        Response.Cookies.Clear();
        Response.Cookies.Add(objCookie);
        objCookie.Values.Add(cookiename, cookievalue);
        DateTime dtExpiry = DateTime.Now.AddDays(iDaysToExpire);
        Response.Cookies[cookiename].Expires = dtExpiry;
       }
    catch (Exception e) { return false; }
        return true;
  }}
```

```
public string GetCookie(string cookiename)
{
 string cookyval = "";
 try
 {
  cookyval =
     Request.Cookies[cookiename].Value;
 }
 catch (Exception e)
 {
   cookyval = "";
 }
 return cookyval;
 }
}
```

Example 9.3-3b

9.0 State Management

When you run this example, after you enter the values and click the button, it will take a while to set the cookies. Then close the browser, start the application again, you will see when you start typing on the text boxes, the previous value you entered will pop out in a drop. That's the saved cookie. Here is the result.

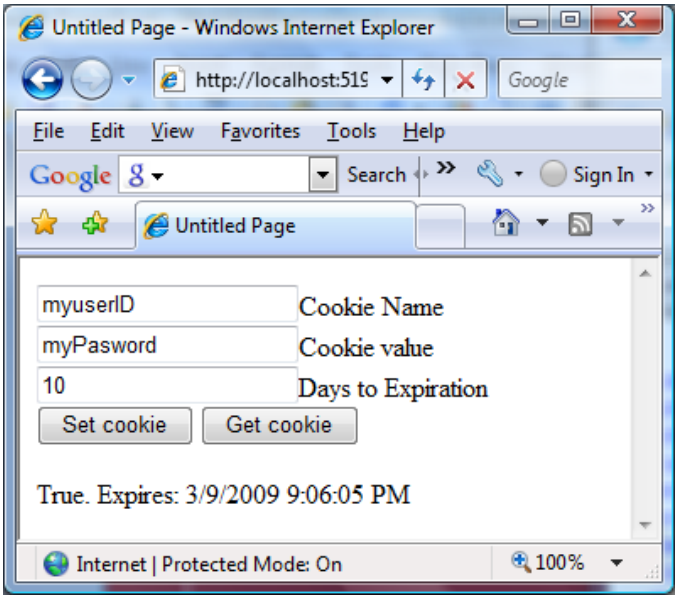


Figure 9.3-3a - First Run. Setting the cookies

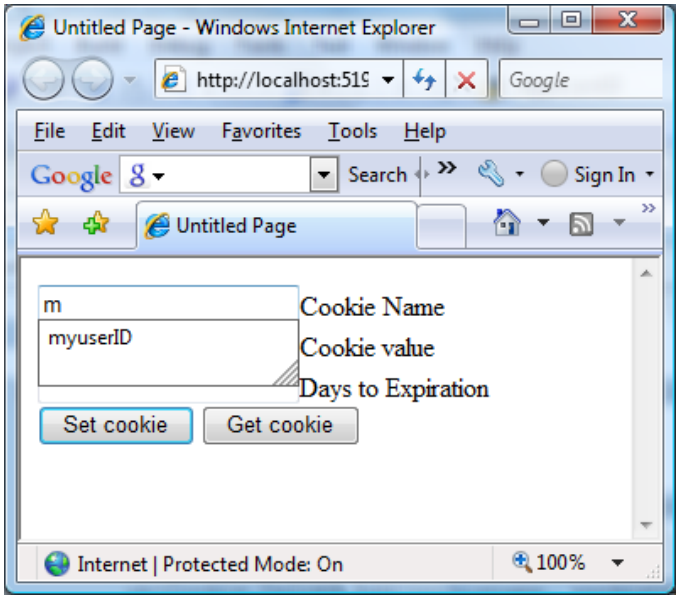


Figure 9.3-3b - Second Run. Notice the cookies

Note : Max number of cookies allowed per website is 20.

9.3.4 Session State

There comes a point in the life of most applications when they begin to have more sophisticated storage requirements. An application might need to store and access complex information such as custom data objects, which can't be easily persisted to a cookie or sent through a query string. Or the application might have stringent security requirements that prevent it from storing information about a client in view state or in a custom cookie. In these situations, **you can use ASP.NET's built-in session state facility**. Session state management is one of ASP.NET's premiere features. It allows you to store any type of data in memory on the server. The information is protected, because it is never transmitted to the client, and it's uniquely bound to a specific session. Every client that accesses the application has a different session and a distinct collection of information. Session state is ideal for storing information such as the items in the current user's shopping basket when the user browses from one page to another.

9.3.4.1 Session Tracking

ASP.NET tracks each session using a unique 120-bit identifier. ASP.NET uses a proprietary algorithm to generate this value, thereby guaranteeing (statistically speaking) that the number is unique and it's random enough that a malicious user can't reverse-engineer or "guess" what session ID a given client will be using. This ID is the only piece of session-related information that is transmitted between the web server and the client. When the client presents the session ID, ASP.NET looks up the corresponding session, retrieves the objects you stored previously, and places them into a special collection so they can be accessed in your code. This process takes place automatically.

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

Using cookies: In this case, the session ID is transmitted in a special cookie (named ASP.NET_SessionId), which ASP.NET creates automatically when the session collection is used. This is the default.

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. In other words, you must think through any use of session state. A careless use of session state is one of the most common reasons that a web application can't scale to serve a large number of clients. Sometimes a better approach is to use caching – which we do not discuss in detail In this course.

9.3.4.2 Using Session State

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

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

You can then retrieve it with an appropriate conversion operation:

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

Of course, before you attempt to use the dsInfo object, you'll need to check that it actually exists—in other words, that it isn't a null reference. If the dsInfo is null, it's up to you to regenerate it. (For example, you might decide to query a database to get the latest data.)

It is important to remember that session variables are now objects. Thus, to avoid a run-time error, you should check whether the variable is set before you try to access it.

9.0 State Management

Session variables are **automatically discarded after they are not used for the time-out setting that is specified in the web.config** file. On each request, the time out is reset. The variables are lost when the session is explicitly abandoned in the code.

When a session is initiated on first request, the server issues a unique session ID to the user. To persist the session ID, store it in an in-memory cookie (which is the default), or embed it within the request URL after the application name. To switch between cookie and **cookieless** session state, set the value of the **cookieless** parameter in the **web.config** file to true or false.

In **cookieless** mode, the server automatically inserts the session ID in the relative URLs only. An absolute URL is not modified, even if it points to the same ASP.NET application, which can cause the loss of session variables.

ASP.NET supports three modes of session state:
InProc: In-Proc mode stores values in the memory of the ASP.NET worker process. Thus, this mode offers the fastest access to these values. However, when the ASP.NET worker process recycles, the state data is lost.

StateServer: Alternately, StateServer mode uses a stand-alone Microsoft Windows service to store session variables. Because this service is independent of Microsoft Internet Information Server (IIS), it can run on a separate server. **You can use this mode for a load-balancing solution** because multiple Web servers can share session variables. Although session variables are not lost if you restart IIS, performance is impacted when you cross process boundaries.

SqlServer: If you are greatly concerned about the persistence of session information, you can use SqlServer mode to leverage Microsoft SQL Server to ensure the highest level of reliability. SqlServer mode is similar to out-of-Process mode, except that **the session data is maintained in a SQL Server**. SqlServer mode also enables you to utilize a state store that is located out of the IIS process and that can be located on the local computer or a remote server.

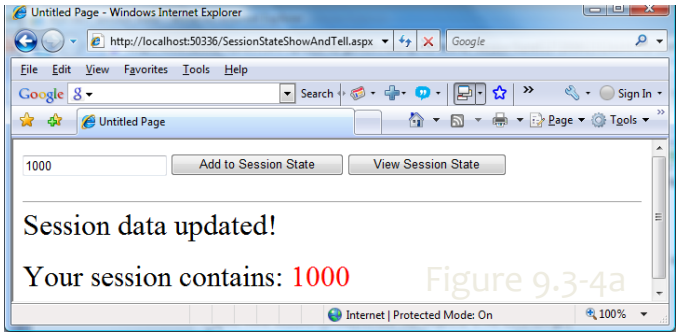
9.0 State Management

In the example below during the session of a client , s session variable is set by the client. Then it is interrogated.

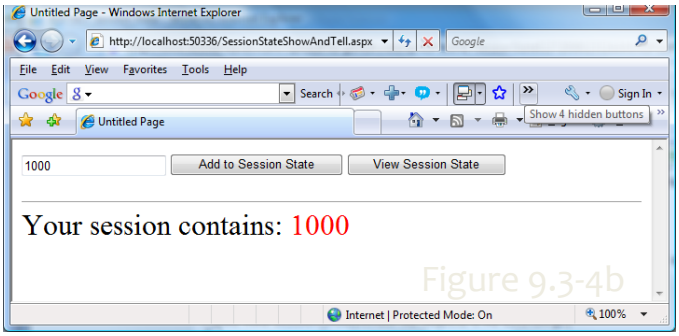
```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="SessionStateShowAndTell.aspx.cs"
Inherits="SessionStateExample._Default" %>=
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
   <asp:TextBox ID="textBox1Text" Text="just Text“
              runat="server" ></asp:TextBox>
   <asp:Button ID="buttonAddState" runat="server“
    Text="Add to Session State" OnClick="SessionAdd" />
   <asp:Button ID="buttonSeeState" runat="server“
    Text="View Session State" OnClick="CheckSession" />
   <asp:Label ID="label1" runat="server" Text="" />
   </div>
   </form>
   <hr size=1>
   <font size=6><span id=span1 runat=server/></font>
</body>
</html>
```

```
using System;
using System.Xml.Linq;

namespace SessionStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  { }
  protected void SessionAdd(object sender, EventArgs e)
  {
   Session["MySession"] = textBox1Text.Text;
   span1.InnerHtml = "Session data updated! <P> “ +
         ”Your session contains: <font color=red>" +
          Session["MySession"].ToString() + "</font>";
  }

  protected void CheckSession(object sender, EventArgs e)
  { if (Session["MySession"] == null)
   {span1.InnerHtml = "NOTHING, SESSION DATA LOST!"; }
   else
   {
    span1.InnerHtml = "Your session contains: “ +
    ”<font color=red>" + Session["MySession"].ToString() +
    "</font>";
   } } }}
```

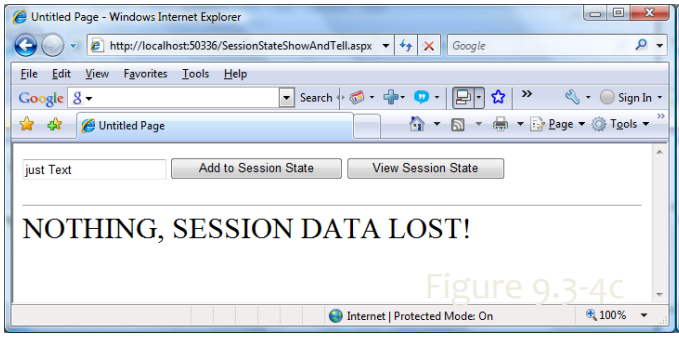
Example 9.3-4a

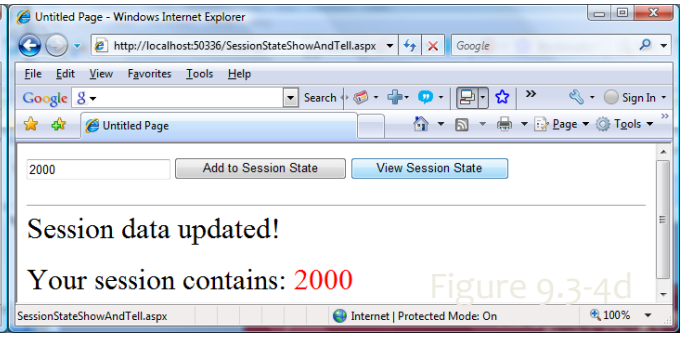


Example 9.3-4b



Then another client is opened. The second client does not see the first clients session data. The second client has it’s own session data.





9.3.4.3 When Does The Session Ends

Session's state is global to the entire application per client. But the client can lose the session data in several ways.

1. If the user closes the browser.
2. If the user accesses the same page through a different browser, although the session will still exist if a web page is accessed through the original browser window. Browsers differ on how they handle this situation.
3. If the session times out due to in activity.
4. If the web page code ends the session by calling the **Session.Abandon()** method.

9.3.4.4 Session Configurations

Sessions can be managed by setting the application specific session configurations. ASP.NET provides three modes of session state storage controlled by mode attribute of **<sessionState>** tag in your web application's **web.config** file. Below is a sample of this tag:

```
<sessionState
   mode="InProc"
   stateConnectionString="tcpip=127.0.0.1:42424"
   sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
   cookieless="false"
   timeout="20"
/>
```

Example 9.3-5

9.3.4.5 Session Timeout

The Session Object : When you are working with an application, you open it, do some changes and then you close it. This is much like a Session. The computer knows who you are. It knows when you start the application and when you end. But on the internet there is one problem: the web server does not know who you are and what you do because the HTTP address doesn't maintain state.

ASP.NET solves this problem by creating a unique cookie for each user. The cookie is sent to the client and it contains information that identifies the user. This interface is called the Session object.

9.0 State Management

The Session object is used to store information about, or change settings for a user session. Variables stored in the Session object hold information about one single user, and are available to all pages in one application. Common information stored in session variables are name, id, and references. The server creates a new Session object for each new user, and destroys the Session object when the session expires.

The Session object's has properties. One is the Timeout property you could use to time out the session when the use do not use the client.

Setting the session time out value:
By default, sessions timeout in 20 minutes. Add the following line to your application's **web.config** to change this value where x is the minutes to wait before a session times out. (Put this line immediately after the <system.web> line.)

```
<sessionState mode="InProc" timeout="x" />
```

Or you can use the **Session.Timeout** in the C# code behind.

In the example below, the **Session.Timeout** is set to one minute. After the application is started, the session will expire in 1 minute. Notice the message "NOTHING, SESSION DATA LOST! when clicked the "View Session State" after one minute.

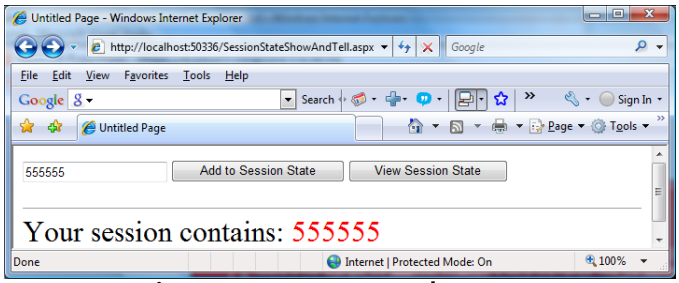


Figure 9.3-5a – At the start

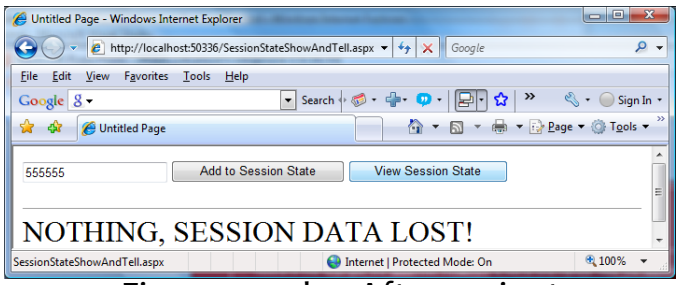


Figure 9.3-5b – After 1 minute

9.3.4.6 What is the difference between ViewState and SessionState?
ViewState persist the values of controls of particular page in the client (browser) when post back operation done. When user requests another page previous page data no longer available.

SessionState persist the data of particular user in the server. This data available till user close the browser or session time completes.

9.3.5 Application State

Application state is something that should be used with care, and in most cases, avoided altogether. Although it is a convenient repository for global data in a Web application, its use can severely limit the scalability of an application, especially if it is used to store shared, updateable state. It is also an unreliable place to store data, because it is replicated with each application instance and is not saved if the application is recycled. With this warning in mind, let's explore how it works.

Application state is accessed through the Application property of the HttpApplication class, which returns an instance of class HttpApplicationState. This class is a named object collection, which means that it can hold data of any type as part of a key/value pair. Example 9.3-6 (a & b) shows a typical use of application state. As soon as the application is started, it loads the data from the database. Subsequent data accesses will not need to go to the database but will instead access the application state object's cached version. Data that is pre-fetched in this way must be static, because it will not be unloaded from the application until the application is recycled or otherwise stopped and restarted.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="ApplicationStateStart.aspx.cs"
Inherits="ApplicationStateExample._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="lblOne" Text="" runat="server" ></asp:Label>
   </div>
   </form>
</body>
</html>
```

Example 9.3-6a

```
using System;
using System.Xml.Linq;
namespace ApplicationStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
   int count = 0;
   if (Application["HitCount"] != null)
   { count = (int)Application["HitCount"]; }

    //Increase the count
    count++;

    //Store it.
    Application["HitCount"] = count;
    lblOne.Text = "Current Hits : " + count;
  }}}
```

Example 9.3-6b

9.0 State Management

In the example 9.3.6 (a & b), the first client (Figure 9.3-6a starts at hits 1. Then hit the refresh button several times that rings up the hits to 9. Then open up another client (Figure 9.3-6b, you will see the it count at 10.

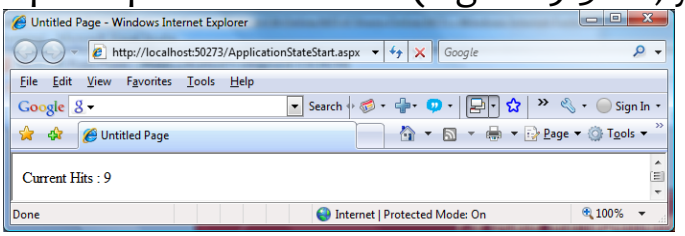


Figure 9.3-6a – At the start

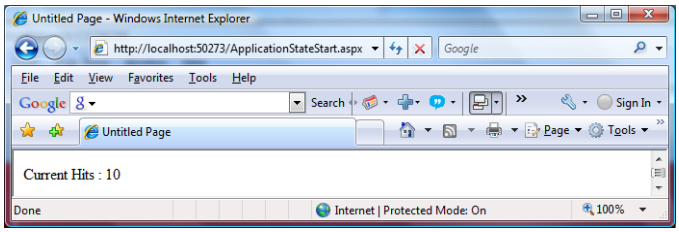


Figure 9.3-6a – At the start

9.3.5.1 Locking Application State

Application State Synchronization:

Multiple threads within an application can simultaneously access values stored in application state. Consequently, when you create something that needs to access application-state values, you must always ensure that the application-state object is free-threaded and performs its own internal synchronization or else performs manual synchronization steps to protect against race conditions, deadlocks, and access violations.

The **HttpApplicationState** class provides two methods, Lock and Unlock, that allow only one thread at a time to access application-state variables.

Calling Lock on the Application object causes ASP.NET to block attempts by code running on other worker threads to access anything in application state. These threads are unblocked only when the thread that called Lock calls the corresponding Unlock method on the Application object.

Application State can be locked for a particular client by calling `Application.Lock()` method. Then when ready it can unlock the state back by using `Application.UnLock();`

In the previous example, we can lock the usage access to count and `Application["HitCount"]` by using the Lock Unlock mechanism as shown below.

The following code Example 9.3-7 demonstrates the use of locking to guard against race conditions.

9.0 State Management

In the example below, when one client is accessing the Application Data, other clients who are trying access the same data are put on hold. So they will feel a delay in processing.

```
using System;
using System.Xml.Linq;

namespace ApplicationStateExample
{
 public partial class _Default : System.Web.UI.Page
 {
  protected void Page_Load(object sender, EventArgs e)
  {
    Application.Lock();
    int count = 0;
    if (Application["HitCount"] != null)
    { count = (int)Application["HitCount"]; }

     //Increase the count
     count++;

     //Store it.
     Application["HitCount"] = count;
     lblOne.Text = "Current Hits : " + count;
     Application.UnLock();
  }
 }
}
```

Example 9.3-7