Lecture 6

HtttpRequest

Objectives

In this lecture you will learn the following

- About HTML Forms
- Learning HttpRequest Properties
- Learning about Cookies
- How the Query String is used?

Coverage Plan

Lecture 6

6.1 Snap Shot

6.2 HTML Form

6.3 HjttpRequest Properties

6.4 Cookies

6.5 Short Summary

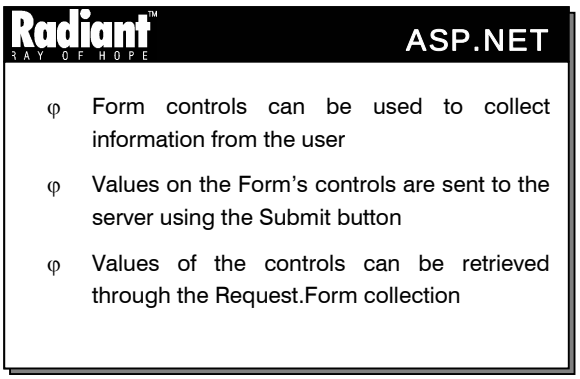
6.6 Brain Storm

6.1 Snap Shot

This session aims at introducing ways to retrieve data from the HTML form and then focusses on the various properties and methods of the HTTPRequest class.

The Request object is responsible for retrieving information from the Web browser. The Request object is filled with various types of collection, properties, and methods that provide many ways to retrieve information from the user. The HttpRequest class is used to gain access to the HTTP request data elements that are supplied by a client.

6.2 HTML Form



To interact with the user, HTML recognizes a few special tags that insert controls on a Form, besides the universal hyperlinks. A control can be used to collect information from the user for registration purpose, take orders over the Internet, or let the user specify selection criteria for record retrieval from database. Before placing any controls on a page, a Form must be created, with the Form tag. All controls must appear within a pair of Form tags:

```
<FORM NAME = "myFrom">
  Controls go here
</FORM>
```

The NAME attribute is optional, but it is advisable to have Form names. The FORM tag also accepts two more attributes namely **METHOD** and **ACTION** that determine how the data will be submitted to the server and processed. The **METHOD** attribute can have the value **POST** or **GET**. The **ACTION** attribute specifies the script that will process the data on the server.

The values entered by the user on the Form's controls are sent to the server using the Submit button. Every Form has a **Submit** button that extracts the ACTION attribute from the <FORM> tag. The values of the controls create a new URL and send it to the server.

The values of the various controls can be retrieved through the Request. Form collection. The Form collection has one member for each control on the Form and individual control's value can be accessed by name. For example, if the Form contains a control named "UserValue", its value can be accessed with the expression **Request.Form ("UserValue")**.

Retrieving Form Data

Whenever a user submits an HTML form, all the form fields and their values are placed in the Form collection of the Request object. This session explains how to retrieve the values of the form fields.

Note : The Form collection contains the values of form elements when an HTML form is submitted with the POST method. When an HTML Form is submitted with the GET method, the values of the form elements are placed in the **QueryString** collection.

Practice 6.1

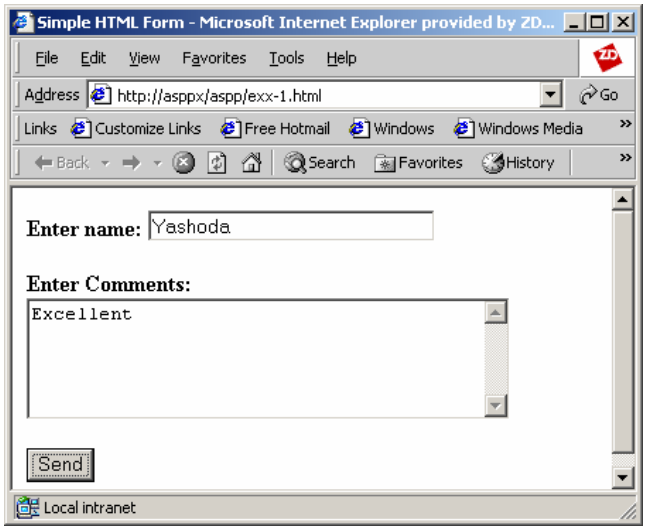
The following code accepts details from the user and submits it to the .aspx file.

html file

```
<html>
<head><title>Simple HTML Form</title></head>
<body>
<form method="post" action="result.aspx">
  <b>Enter name: </b>
     <input name="username" type="text" size=30>
  <p>
  <b>Enter Comments: </b>
  <br><textarea name= "comments" cols=40 rows=5></textarea>
     <p>
  <input type="submit" value="Send">
</form>
</body>
</html>
```

result.aspx

```
<%@Page Language="C#"%>
<%
  string username="";
  string comments="";
  username = Request.Form ["username"];
  comments= Request.Form ["comments"];
%>
<head><title>Result</title></head>
<body>
  User Name : <%=username%>
  <p>
  Comments : <%=comments%>
</body>
```



In the above program the user is prompted to enter a user name and comments. When the user clicks the button labeled **Send**, the contents of the form are submitted to result.aspx. The **Form** collection can be accessed using the **Request** object inside the result.aspx file is then passed to the browser.

When the Send button is clicked after entering the name (Yashoda) and comments (Excellent), the result.aspx file is called which displays the output as shown below:

```
User Name : Yashoda
Comments : Excellent
```

In certain situations it is necessary to display a second form where the user can approve the information entered into the first form. In such a case the user is provided with another opportunity to change the information before the information is actually entered into a database. The following program (confirm.aspx) redisplays the information posted in a previous HTML form with a new form.

Practice 6.2

The following program displays a confirmation form after the submitted details are processed.

html file

```
<html>
<head><title>Simple HTML Form</title></head>
<body>
<form method=”post” action=”confirm.aspx”>
  <b>Enter name: </b>
     <input name=”username” type=”text” size=30>

  <p>
  <b>Enter Comments: </b>
  <br><textarea name= “comments” cols=40 rows=5></textarea>
     <p>

  <input type=”submit” value=”Send”>
</form>
</body>
</html>
```

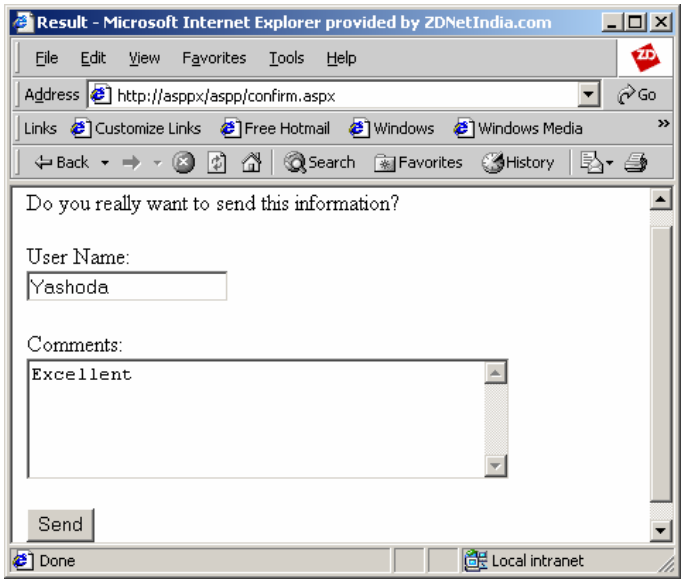
confirm.aspx

```
<%@Page Language="C#"%>
<%
  string username="";
  string comments="";
  username = Request.Form ["username"];
  comments = Request.Form ["comments"];
%>

<head><title>Result</title></head>
<body>

<form method= "post" action= "result.aspx")
  <p>Do you really want to send this information?</p>
  <p>User Name:
  <br><input name= "username"
     type= "text" value= "<%=(username)%>">

  <p>Comments:
  <br><textarea name= "comments" cols=40 rows=5>
  <%=(comments)%></textarea>
  <p>
  <input type="submit" value="Send">
</form>
</body>
```



When the Send button is clicked after entering the details, the **confirm.aspx** file is invoked and the confirmation form is displayed. The user can make further changes and submit the form by clicking the Send button.

6.3 HttpRequest Properties

Radiant RAY OF HOPE ASP.NET

Some of the properties of the HttpRequest class are

- Application
- Browser
- ClientCertificate
- ContentType
- FilePath
- RawUrl
- Url
- Cookies

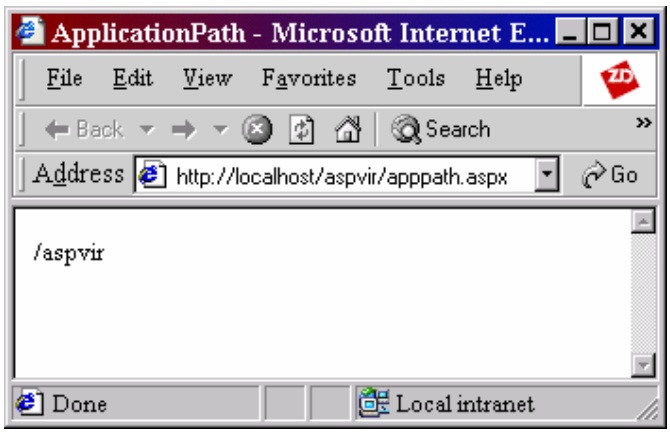
Application Path

This property gets the virtual path to the currently executing server application.

Practice 6.3

The following example illustrates the usage of the ApplicationPath property.

```
<head>
<title> ApplicationPath </title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
 String apath;
 apath=Request.ApplicationPath;
 Info.InnerHtml=apath;
}
</script>
</head>
<form runat="server">
 <span id="Info" runat="server"/>
</form>
```



In the above example, it can be seen that the property **ApplicationPath** is used to get the virtual path. As in the case of the above example the virtual directory is **aspvir** .

Browser

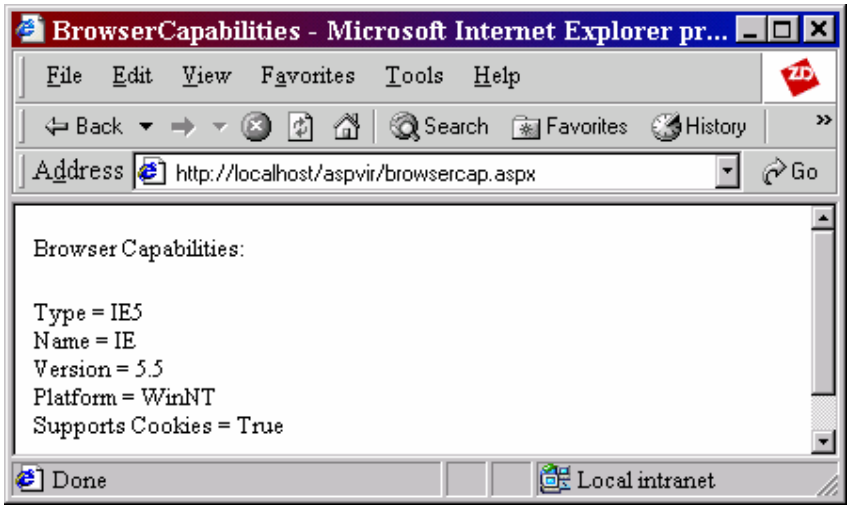
The Browser property gets the virtual path of the currently executing server application.

Practice 6.4

The following example illustrates the usage of the Browser property.

```
<head>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
     HttpBrowserCapabilities bcap = Request.Browser;

     Response.Write("<p>Browser Capabilities:</p>");
     Response.Write("Type = " + bcap.Type + "<br>");
     Response.Write("Name = " + bcap.Browser + "<br>");
     Response.Write("Version = " + bcap.Version + "<br>");
     Response.Write("Platform = " + bcap.Platform + "<br>");
     Response.Write("Supports Cookies = " + bcap.Cookies + "<br>");
}
</script>
/head>
```



This Browser property is used here to get the browser details namely, the browser type , name, version and the platform on which the browser is running and whether the browser supports cookies.

ClientCertificate

This property gets information on the current request's client security certificate.

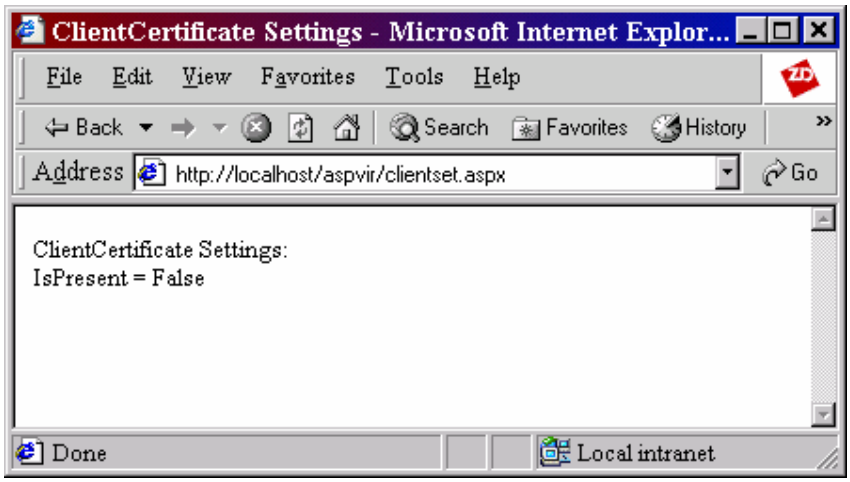
Practice 6.5

The following example illustrates the usage of the ClientCertificate property.

```
<head>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
     HttpClientCertificate cert = Request.ClientCertificate;

     Response.Write("ClientCertificate Settings:<br>");
     Response.Write("IsPresent = " + cert.IsPresent + "<br>");
}
</script>
</head>
<form runat="server">

</form>
</head>
```



As it can be seen from the above example, the information about the client certificate settings details can be found using the ClientCertificate property.

ContentType

This property indicates the Multipurpose Internet Mail Extensions (**MIME**) content type of the incoming request. This property is read-only.

```
contype = Request.ContentType;
Response.Write (contype);
```

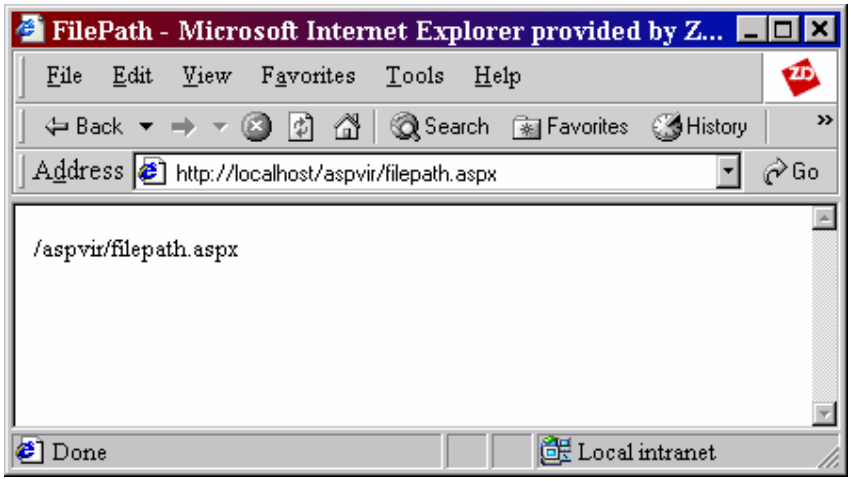
FilePath

This property indicates the virtual path of the current request. It is read-only.

Practice 6.6

The following example illustrates the usage of the Filepath property.

```
<head>
<title> FilePath</title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
 String fiPath;
    fiPath = Request.FilePath;
 Message.InnerHtml=fiPath;
}
</script>
</head>
<form runat="server">
 <span id="Message" runat="server"/>
</form>
```



The above example shows that aspvir is the virtual path.

RawUrl

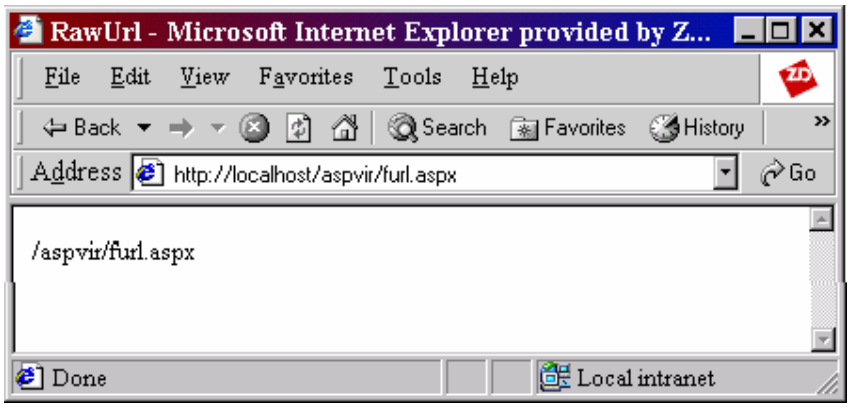
This property indicates the raw URL of the current request.

Practice 6.7

The following program illustrates the usage of the RawUrl property.

```
<head>
<title> RawUrl </title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
```

```
{
  String Rurl;
      Rurl = Request.RawUrl;
  Info.InnerHtml=Rurl;
}
</script>
</head>
<form runat="server">
  <span id="Info" runat="server"/>
</form>
```



The Raw URL of the above request is aspvir/furl.aspx as shown in the above output. It is obvious that **aspvir** is the virtual directory and **furl** is the file name.

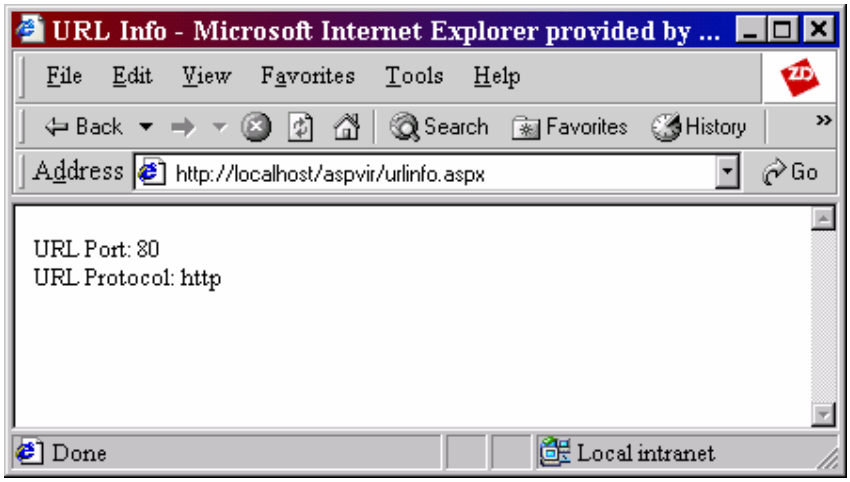
Url

This property gets information regarding the URL of current request.

Practice 6.8

The following example gives information about the URL of the current request.

```
<head>
<title> URL Info </title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
  HttpUrl objUrl = Request.Url;
      Response.Write("URL Port: " + objUrl.Port + "<br>");
      Response.Write("URL Protocol: " + objUrl.Protocol + "<br>");
}
</script>
</head>
<form runat="server">
</form>
```



The above example shows that by using the Url property, the port and protocol information can be obtained.

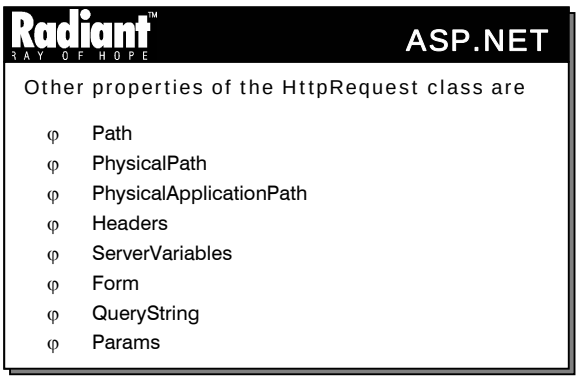
6.4 Cookies

Cookies are small pieces of information, which can be sent to a browser by a server program and stored by the web browser. The web browser will then pass the cookie back to the server every time it makes a request from that server. This facility is particularly useful for allowing authentication.

For example, when a user logs into a password restricted system, a cookie containing that users login/password details can be set with those details, so that the user does not have to re-type their password for every new page they wish to download.

It is important to know that the client's web browser could be configured to refuse cookies and that some browsers manage cookies incorrectly. When a cookie was sent, the server can retrieve it in a successive client's connection. This strategy makes it is possible to track the history of client's connections.

The cookie property gets a collection of the client's cookie variables. Cookies can be seen elaborately in the following session.



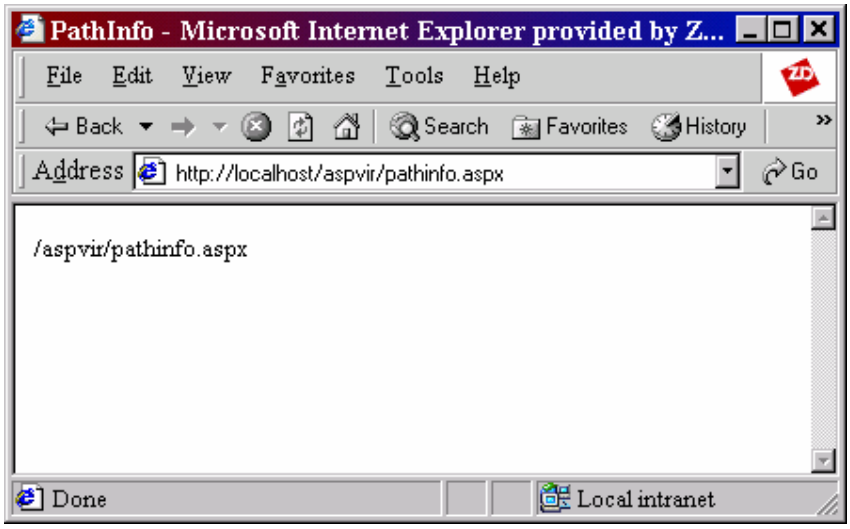
Path

This property indicates the virtual path of the current request and it is read-only.

Practice 6.9

The following example illustrates the usage of the Path property.

```
<head>
<title> PathInfo </title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
 String vpath;
 vpath = Request.Path;
 Info.InnerHtml=vpath;
}
</script>
</head>
<form runat="server">
 <span id="Info" runat="server"/>
</form>
```



As shown from the above example, the Path property is used to get the virtual path of the current request.

ServerVariables

This property gets a collection of Web server variables.

Practice 6.10

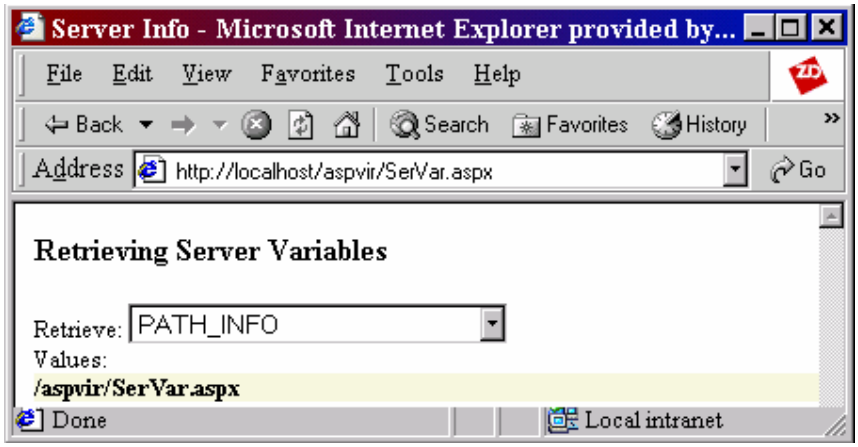
The example shows how the ServerVariables are used.

```
<script runat="server" language="c#">
```

```
<title> server Info </title>
void Page_Load(Object Sender, EventArgs E){
  NameValueCollection NVCSrvElements = Request.ServerVariables;
  if (! IsPostBack){
     Names.DataSource=NVCSrvElements;
     Names.DataBind();
  }else{
     ValuesHead.Text = “Values: “;
if (NVCSrvElements.Get(Names.SelectedItem.Text).ToString() == null) {
        Values.Text = “null”;
     } else {
        Values.Text = NVCSrvElements.Get(
                      Names.SelectedItem.Text).ToString() ;
        }

     }
}
</script>

<html>
<body>
<form runat=”server”>
  <h3>Retrieving Server Variables</h3>
  Retrieve: <asp:DropDownList id=”Names” runat=”server” AutoPostBack=”true”/>
  <br>
  <asp:Label id=”ValuesHead” runat=”server” />
  <br>
  <b>
  <asp:Label id=”Values” runat=”server” BackColor=”Beige” Width=”500" />
  </b>
</form>
</body>
</html>
```



As it can be seen from the output the various server variables information can be obtained by selecting the required property in the list box.

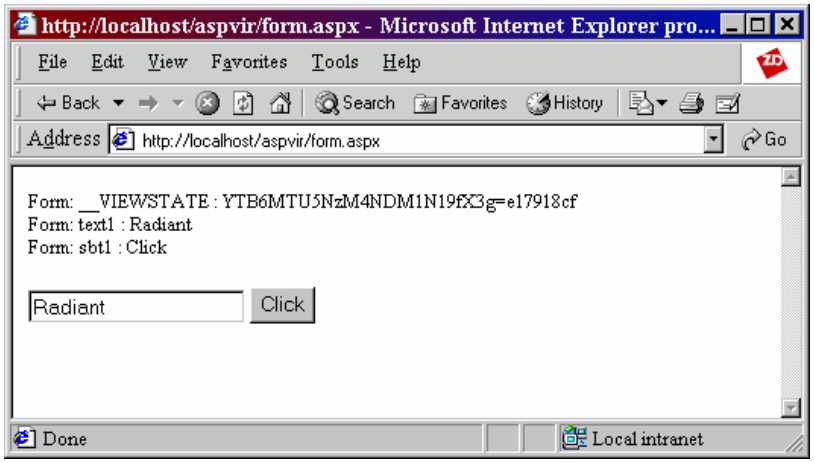
Form

This property gets a collection of the Form variables.

Practice 6.11

The example illustrates the use of Form property to retrieve Form variables.

```
<html>
<title> Form Info </title>
<head>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
  if(IsPostBack){
     int ctr1;
           NameValueCollection nvc;
     nvc=Request.Form; // Load Form variables into NameValueCollection
                       variable.
        String[] arr1 = nvc.AllKeys; // Get names of all forms into a string
                                  array.
     for (ctr1 = 0; ctr1 < arr1.Length; ctr1++) {
        Response.Write("Form: " + arr1[ctr1] + "   :  " +
                       nvc.Get(arr1[ctr1])+ "<br>");
           }
     }
}
</script>
</head>
<body>
<form runat="server">
  <input type="text" id="text1" runat="server">
  <input type="submit" id="sbt1" value="Click" runat="server">
</form>
</body>
</html>
```



The above code first loads all the form variables into the NameValueCollection variable. Then using a for loop the contents of the collection are displayed in the browser.

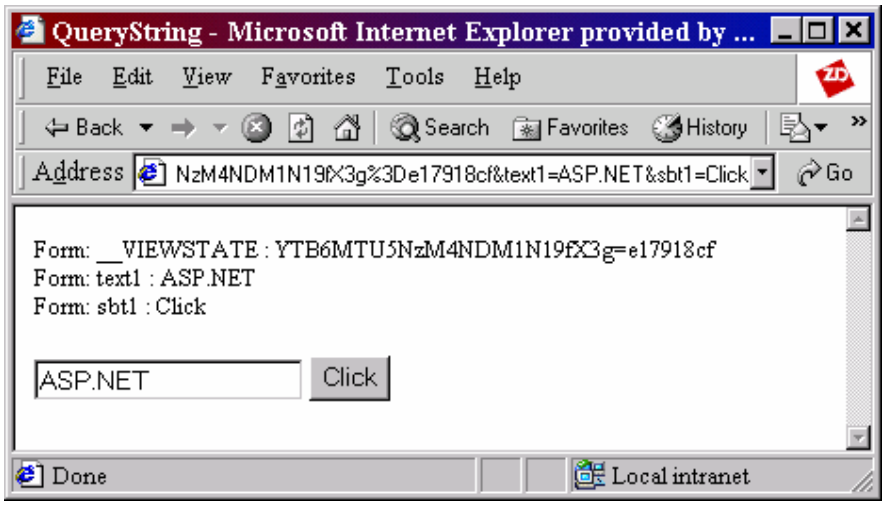
QueryString

This property gets the collection of QueryString variables.

Practice 6.12

The example uses QueryString to retrieve form variables.

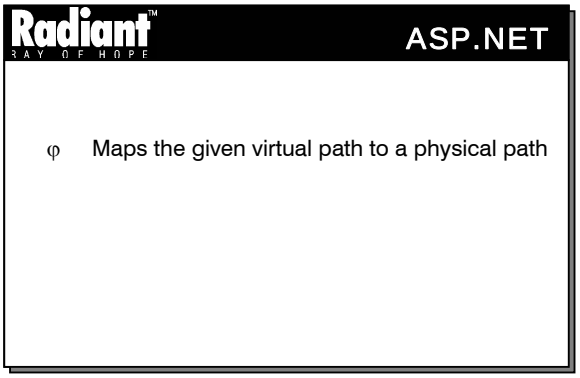
```
<html>
<title> QueryString </title>
<head>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
     if(IsPostBack){
     int ctr1;
           NameValueCollection nvc;
           nvc=Request.QueryString; // Load Form variables into
                                       NameValueCollection variable.
String[] ary1 = nvc.AllKeys; // Get names of all forms into a string array.
     for (ctr1 = 0; ctr1 < ary1.Length; ctr1++) {
        Response.Write("Form: " + ary1[ctr1] +"   :  " +
                    nvc.Get(ary1[ctr1])+ "<br>");
           }
     }
}
</script>
</head>
<body>
<form runat="server" method="Get">
  <input type="text" id="text1" runat="server">
  <input type="submit" id="sbt1" value="Click" runat="server">
</form>
</body>
```



In the above code the variables of the form are retrieved using the Request.QueryString property. The values are then displayed in the browser.

HttpRequest Method

MapPath Method



Syntax

```
public string MapPath(
 string virtualPath
);
```

- **virtualPath** indicates the virtual path for the current request

6.5 Short Summary

- HttpRequest class is used to gain access to the HTTP request data elements supplied by a client

- The Browser property gets the virtual path of the currently executing server application
- ClientCertificate property gets information on the current request's client security certificate
- FilePath property indicates the virtual path of the current request. This property is read-only
- RawUrl property indicates the raw URL of the current request
- Form property gets a collection of Form variables
- QueryString property gets the collection of QueryString variables
- Params property gets a combined collection of QueryString + Form + ServerVariable + Cookies

6.6 Brain Storm

1. What is the use of HttpRequest?
2. What is the difference between the properties RawUrl and Url?
3. What are cookies?
4. What are the properties that are used to read the Form data?

Lecture 7

HttpResponse

Objectives

In this lecture you will learn the following

+ Knowing about the HttpResponse Properties
+ Knowing about the Redirecting

Coverage Plan

Lecture 7

7.1 Snap Shot

7.2 HttpResponse Properties

7.3 HttpResponse Methods

7.4 Short Summary

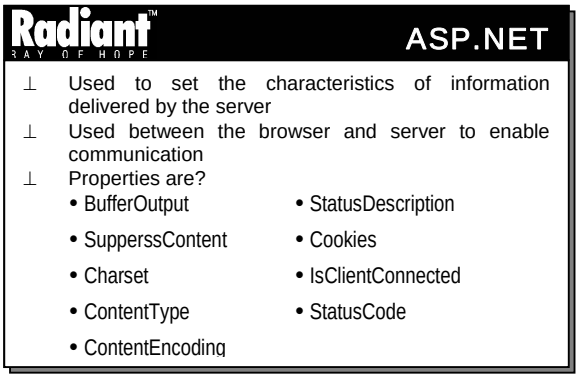
7.5 Brain Storm

7.1 Snap Shot

This session explains the important and commonly used properties and methods of the HttpResponse.

The Response object provides access to all the responses sent to the user. The Response object is responsible for sending the output from the server to the client. The methods and properties of this object control the way in which information is sent from the Web server to a Web browser.

7.2 HttpResponse Properties



HttpResponse has different properties that are used to set the various characteristics of the information delivered by the server. The characteristics are used between the browser and server to provide a communication mechanism between the information provider and the information consumer. The properties include:

BufferOutput

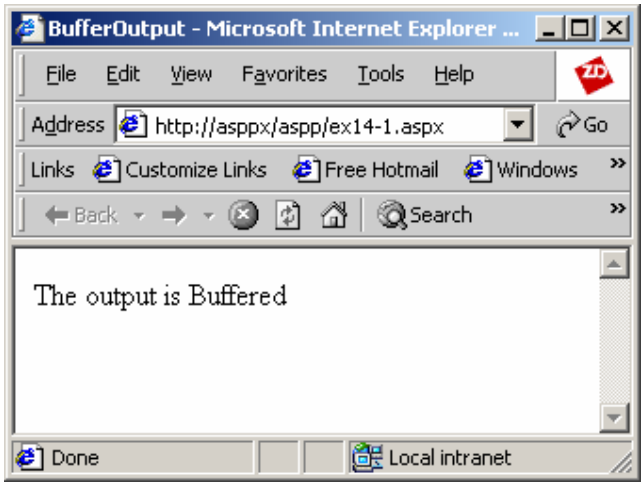
The BufferOutput gets or sets a value indicating whether the HTTP output is buffered or not. This property takes the value true if the output to client is buffered and false otherwise. The default value is true.

Practice 7.1

The following program checks if the output is buffered.

```
<head>
<title>BufferOutput</title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
  if (Response.BufferOutput == true)
     Message.InnerHtml="The output is Buffered";
}
</script>
</head>

<form runat="server">
  <span id="Message" runat="server"/>
</form>
```



The BufferOutput property in the above program checks if the output is buffered and displays a message in the browser.

SuppressContent

The SuppressContent property gets or sets a value indicating that the HTTP content will not be sent to the client. The property returns true if the output is to be suppressed and false otherwise.

The following code removes the suppression of the content.

```
Response.SuppressContent = false;
```

Charset

The Charset property gets or sets the HTTP charset of the output.

Cookies

Cookies are used for session tracking, and user preferences. Cookies may also be used to determine how long the users view a Web page, content (e.g., advertising), the link, and other services. The cookies property gets the **HttpCookie** collection that is sent by the current request.

The following code takes the current cookie collection and fills a string array with the names of cookies.

```
HttpCookieCollection oReqCookies = Response.Cookies;
String[] sReqCookies = oReqCookies.AllKeys;
```

IsClientConnected

The IsClientConnected property gets a value that indicates whether the client is still connected to the server or not. The value of the property will be **true** if the client is currently connected and **false** otherwise.

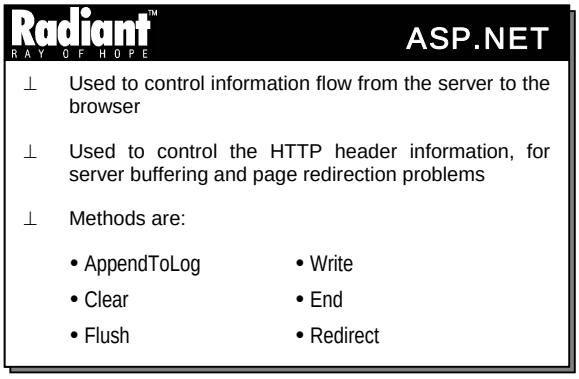
StatusCode

The StatusCode property gets or sets the HTTP status code of the output returned to the client. The return value is an integer code that represents the status of the HTTP output that is returned to the client. The default value is 200. Some StatusCode values and their corresponding meaning are given in Table 7.1

Code	Meaning
200	OK. The operation has been finished successfully
302	Redirection
401	Unauthorized access
403	Access forbidden
404	Requested resource is not found

Table 7.1

7.3 HttpResponse Methods



The methods are used to control the HTTP header information, non-textual content such as binary images, and Web server log files. In addition, the methods control the buffering mechanism happening in the server and page redirection.

AppendToLog

The AppendToLog method adds custom log information to the IIS log file.

Syntax

```
public void AppendToLog( string param );
```

where,. **param** is a string that contains the text to be added to the log file.

The following code appends the text **Page delivered** to the IIS log file.

```
Response.AppendToLog(“Page delivered”);
```

Write

The Write method writes the values to an HTTP output content stream.

Syntax

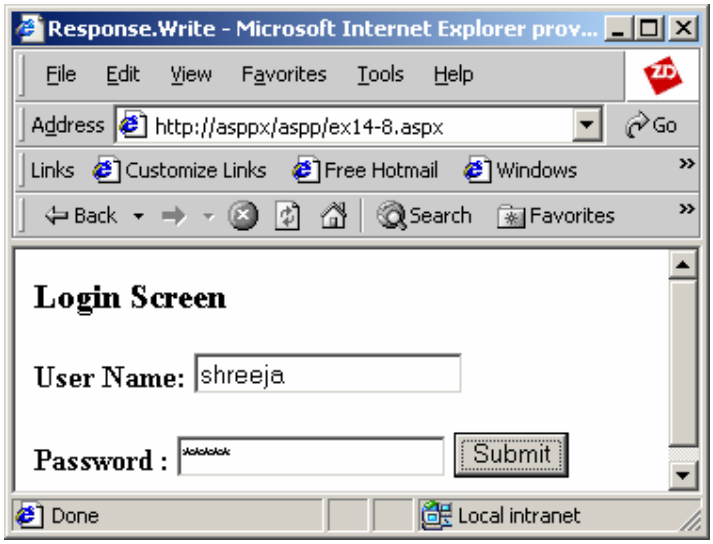
```
public void Write( string s);
```

Here, **s** is the string that has to be written to the HTTP output.

Practice 7.2

The following program adds two input textboxes and a submit button.

```
<head>
<title>Response.Write</title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
  Response.Write("<H3>Login Screen</H3>");
  Response.Write("<b>User Name: </b>");
  Response.Write("<input type=text id=user /> "+"<br><br>");
  Response.Write("<b>Password : </b>");
  Response.Write("<input type=password id=user /> ");
  Response.Write("<input type=submit value=Submit />");
}
</script>
</head>
```



The above program uses the Response.Write method and displays two textboxes – to enter the user name and password. A Submit button is also displayed.

Clear

The Clear method clears all headers and content output from the buffer stream.

Syntax

```
public void Clear();
```

The following code clears the content output from the buffer stream.

```
Response.Clear();
```

End

The End property sends all the currently buffered output to the client and then closes the socket connection.

Syntax

```
public void End();
```

The following code closes the socket after sending the buffered output to the client.

```
Response.End();
```

Flush

The Flush method sends the currently buffered output to the client. The method can be called multiple times during request processing.

Syntax

```
public void Flush();
```

The following code sends the buffered output to the client.

```
Response.Flush();
```

Redirect

The Redirect method redirects a client to a new URL.

Syntax

```
public void Redirect( string url );
```

where, **url** is the new target location.

Practice 7.3

The following program forces an unconditional redirection to another Web site.

```
<head>
<title>Response.Redirect</title>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
  Response.Redirect("http://www.radisoft.com");
}
</script>
</head>
```

As it can be seen from the above output, the program redirects the client to another Web Site called Radiant Software.

7.4 Short Summary

- HttpResponse provides access to the responses that are sent to the user
- The BufferOutput property gets or sets a value that indicates if the HTTP output is buffered
- The HTTP charset is set using the Charset property
- The HTTP MIME type is set using the ContentType property
- Cookies are used for tracking the session. The cookies property is used to retrieve the HttpCookie collection that is sent by the current request
- The IsClientConnected property is used to check if the client is connected to the server
- The Write method writes values to an HTTP output content stream
- The End property closes the socket connection after sending the currently buffered output to the client

7.5 Brain Storm

1. What is the use of HttpResponse?
2. What is the use of the IIS log file?
3. How are session information maintained?

Lecture 8

Application, Session State Management and Cookies

Objectives

In this lecture you will learn the following

- Knowing about HttpApplication
- Learning ab9ut Session
- How to create a Cookie?
- Learning about ServerObject

Coverage Plan

Lecture 8

8.1 Snap Shot

8.2 HttpApplication

8.3 Session

8.4 Cookies

8.5 Server Object

8.6 Short Summary

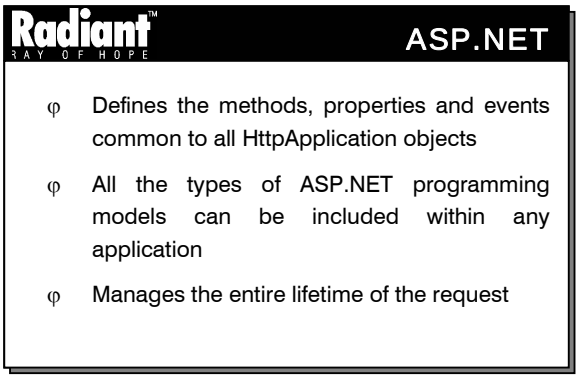
8.7 Brain Storm

8.1 Snap Shot

This session deals with HttpApplication, global.asax file, and the methods to create an Application variable in Global.asax and accessing it in different applications. The session also discusses the properties and methods of an ApplicationState. Topics like Session, SessionState and cookies are also dealt with.

Active Server Pages can be thought in terms of traditional programming. A single Active Server Page is similar to a procedure or subroutine. When related Active Server Pages are grouped together they form an application. One of the greatest challenges faced in constructing a full-featured web application is keeping track of user-specific information when a user navigates the site. Sessions are used to store visitor's preferences and to keep track of the habits and interests of the visitors. This information can be used for advertising purposes, to improve the design of the Web site.

8.2 HttpApplication



The HttpApplication class defines the methods, properties and events common to all HttpApplication objects within the ASP.NET Framework.

An ASP.NET application is defined as a collection of files, pages, handlers, modules and executable code that can be invoked in the scope of a given virtual directory and its subdirectories on a Web Application Server. All the types of ASP.NET programming models can be included within any application (Web Forms, and Web Services). The only condition is that they must coexist in a single virtual directory structure.

Every ASP.NET application on the Web Server is executed within a unique .NET runtime Application Domain. ASP.NET maintains a collection of **HttpApplication** instances of a Web application's lifetime and automatically assigns an instance to process the incoming HTTP request. An **HttpApplication** instance manages the entire lifetime of the request, and is re-used only after completing the request.

Creating an Application

An ASP.NET application is created by placing a simple .aspx page in the virtual directory and executing it in the browser. Then an appropriate code can be added to use the Application object, for example to store objects with application scope. Various event handlers can be defined by creating a **Global.asax** file (discussed later).

Life Time of an Application

An ASP.NET application is created when a request is made for the first time to the server. On receiving the first request, a collection of **HttpApplication** instances are created, and the **Application_Start** event is

fired. An HttpApplication instance processes the request until the last instance is present. When there are no more instances the Application_End event is fired.

Managing Applications

The **Application** object is a built-in object in ASP.NET. The HttpApplication is used to control and manage all the items that are available to all the users of an Active Server application. The **Application** items can be variables that are necessary for the application, or they can be instantiated objects providing special server-side functionality.

An application variable contains data that is used in all the web pages and by the users of an application. Application variables can contain different types of data, including arrays and objects. The common uses of application variables are:

- To display transient information on every Web page. For example, Tip of the day or a daily news update on every Web page
- To record the number of times a banner advertisement on the Web site has been clicked
- To store data that is retrieved from a database, which can then be displayed easily on multiple pages without frequently accessing the database
- To count the number of visitors to a particular Web site

An application variable helps in communicating with the users of the Web site. For example, application variables can be used to create multi-user games or multi-user chat rooms.

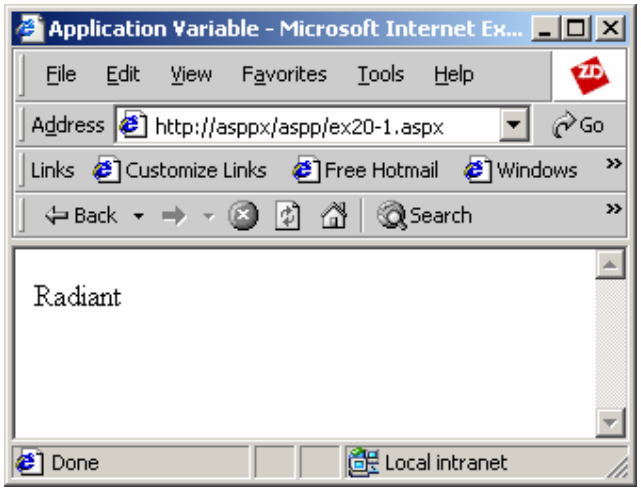
Accessing Application Variable

To create a new application variable, the name of the new variable is passed to the **HttpApplication**.

Practice 8.1

The following program illustrates the use of the application variable.

```
<HTML>
<HEAD><TITLE>Application Variable</TITLE></HEAD>
<BODY>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
Application ["var"] = "Radiant";
Response.Write (Application["var"].ToString());
}
</script>
</BODY>
</HTML>
```



The above example creates a new application variable named **var** and assigns the value **Radiant**. Then, the value of the variable **var** is displayed in the browser. After assigning a value to an application variable, it can be displayed on all the pages in an application.

Global.asax

The Global.asax file (ASP.NET application file) is a file that contains the syntax for coding the server-side application object. The Global.asax file is stored in the root directory of an ASP.NET application. The file handles the application events like Application_Start, Application_End, Session_Start, Session_End, etc. The Global.asax is parsed and compiled by ASP.NET into a .NET Framework class when a request is made for the first time within its application namespace. The Global.asax file itself is configured so that direct URL request is automatically rejected. i.e. users are not allowed to view or download the code.

The ASP.NET Global.asax file is fully compatible with the ASP Global.asa file. A Global.asax file can be created either in a WYSIWIG HTML designer, Notepad, or as a compiled class that is deployed in the application's \bin directory as an assembly.

Note: The Global.asax file is not mandatory; if not defined, the ASP.NET framework assumes that there is no definition for application or session event handlers.

Application Events in Global.asax

The Global.asax provides more than 15 events. The following table describes the different events in the Global.asax.

Event name	Description
Application_Start	Fired when the application is first started. This event is useful for applying application wide setting that are initialized only once.
Session_Start	Fired when a session is first started.
Application_End	Fired when the application ends (times out or reset). The event is fired only once and is used to clean the application code or global references created in the Application_Start event.
Session_End	Fired when a session ends (times out or reset). Fired only once.

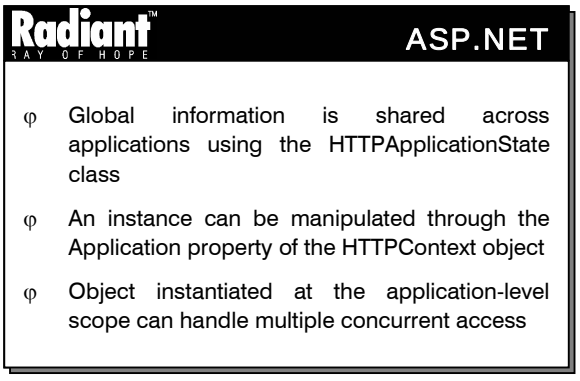
Application_Error	Fired when an unhandled error occurs in the application. Errors are handled using error-trapping code (like Try/Catch blocks).
Application_BeginRequest	Fired whenever a new request is received.
Application_Authenticate	Signals that the request is ready to be authenticated. The event is used by the security module.
Application_Authorize	Signals that the request is ready to be authorized. The event is used by the security module. Used when code has to be run before the user is authorized for a resource.
Application_ResolveRequestCache	Used to short-circuit the processing of requests that are cached. The event is used by the output cache module.
Application_AquireRequestState	Signals that per-request state (session or user) must be obtained.
Application_PreRequestHandlerExecute	Signals that the request handler is about to execute. The request handler will mostly be an ASP.NET page or a Web service.
Application_PostRequestHandlerExecute	Signals the first event available to the user after the handler has completed its work.
Application_ ReleaseRequestState	Called when the request state must he stored, since the application is finished execution.
Application_ UpdateRequestCache	Signals that the code processing is complete and the file is ready to be added to the ASP.NET cache.
Application_ EndRequest	Signals the last event called when the application ends.
Application_PreRequestHeadersSent	Provides an opportunity to add, remove, or update headers and the response body.

Table 8.1

Events in the Global.asax can have different event method prototypes. The following is an example of how the BeginRequest event signature can vary:

```
public void Application_OnBeginRequest()
public void Application_BeginRequest(Object sender, EventArgs e)
```

ApplicationState



Earlier versions of ASP had a single instance of the Application object for each application running in the machine. The values and object references are valid as long as the application is running. However, the

Application object is destroyed when changes are made to the Global.asax file or if the last client Session object is destroyed.

Global information can be shared across applications using the **HTTPApplicationState** class that exposes a key-value dictionary of objects that can be used to store both .NET Framework object instances and scalar values across multiple web requests from multiple clients.

An instance of the **HTTPApplicationState** class is created when the client requests a URL resource from within a particular ASP.NET Application virtual directory namespace for the first time. This is the case for each application that is stored in the computer. This instance can be manipulated through the Application property of the **HTTPContext** object that is provided to all **IHTTPModules** and **IHTTPHandlers** during a given Web request.

The important thing to be noted is that objects instantiated at the application-level scope can handle multiple concurrent access. The limitation while using an HttpApplication is that it is not maintained across a **Web-farm** or in a **Web-garden**. A Web-farm is where more than a single server handles the requests for the same application and a Web-garden is where the same application runs in multiple processes within a single, multi-processor machine.

Application State supports provided by ASP.NET are:

- An easy to use state facility that is compatible with ASP that works with all .NET languages and is consistent with other .NET Framework APIs.
- The application state dictionary is available to all request handlers invoked within an application whereas in ASP it has restricted access only to pages.
- A simple synchronization mechanism that enables coordination of concurrent access to global variables stored in application-state easier.
- Application state values are accessible only from code running within the context of the originating application and not from other applications running on the system.

Using Application State

Application state variables are also global variables for a given ASP.NET application. The impact of creating global variables must be considered before creating them. The points to be considered when using application state variables are:

- The memory occupied by application state variables will not be released until the value is either removed or replaced. For instance, storing 10Mb recordsets that is never used in application state permanently is not the best use of system resources
- The concurrency and synchronization implications of storing – and later correctly accessing – a global variable within a multi-threaded server environment. Multiple running threads within an application can access application state variables simultaneously. Care must be taken to ensure that either the application-scoped object is free-threaded and contains built-in synchronization support, or an application-scoped object is not free-threaded and explicit synchronization methods are coded to avoid deadlocks. Hence, explicit use the "Lock" and "Unlock" methods provided on the **HTTPApplicationState** class must be used to avoid problems
- Since locks that protect global resource are themselves global, the code running on multiple threads that accesses global resources will ultimately battle on these locks. This causes the OS to block the worker threads until the lock becomes available. In a server environment with high load this can

cause severe "thread thrashing" on the system. In multi-processor systems it can lead to processor under-utilization (since all the threads are waiting for a shared lock) and significant drops in overall scalability

- Global data stored in application-state is not stable, since it will be lost when the host containing it is destroyed. To avoid such failures, the state has to be stored either in a database or on some other durable store

- Since the application-state is not shared across either a Web-farm or a Web-garden, variables stored in application-state in these scenarios are global only to the particular process that the application is running

Note: For performance reasons all the built-in .NET collections do not contain built-in synchronization support.

Properties

The properties of the application state are as follows:

All

The All property returns all application state objects as an array of objects.

AllKeys

The AllKeys property retrieves all application state object names in a collection.

Contents

The Contents property returns "this". This property is included for compatibility with ASP.

Count

The count property gets the number of item objects in the application state collection.

Item

The item property is used to add, remove, or update an application state object. This property is the indexer for the **HttpApplicationState** class.

- **Item Property (String) -** Enables a user to add/remove/update a single application state object. This property is the indexer for the **HttpApplicationState** class.

- **Item Property (Int32) -** Enables user to retrieve an application state object by index. This property is the indexer for the **HttpApplicationState** class.

StaticObject

The StaticObject property exposes all objects declared through the <object runat=server></object> tag within the ASP.NET application file.

Collection

A **collection** represents a logical storage unit that helps to manage information.

Keys

The Keys collection represents a collection of the **System.String** keys of a collection. This class is used exclusively to implement the **NameObjectCollectionBase.Keys** property, which allows different methods

for accessing each key in the **NameObjectCollectionBase.Keys** property by using an index, an array or a **Get** method.

Methods

The methods of the application state are discussed below:

Clear

The Clear method removes all objects from the application state collection.

Remove

The Remove method removes an object from the application state collection by specifying a name.

Syntax

```
public void Remove(string name);
```

- name is the object name

RemoveAll

The RemoveAll method removes all objects from the application state collection.

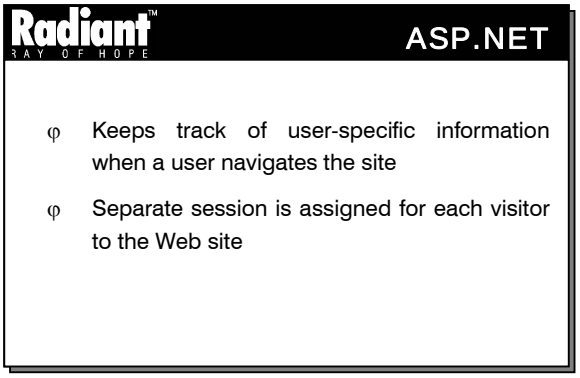
Lock

The Lock method locks access to all application state variables and hence facilitates access synchronization.

UnLock

The Unock method unlocks access to all application state variables. This also facilitates access synchronization.

8.3 Session



A session keeps track of user-specific information when a user navigates the site without asking the users identity for every request from the server. The different information to be maintained are a user's identification and security. In certain advanced applications, customization to the web site must also be maintained. A separate session is assigned for each visitor to the Web site.

Usage of Session

A Session can store the user's preferences like the preferred background color of the Web page, whether to use frames etc. Sessions can also be used to create a virtual shopping basket. Whenever a user selects an item to buy, the item is automatically added to a shopping basket. When the user is ready to quit, he/she can purchase the items in the shopping basket at once. All the item information in the shopping basket can be stored in a session. Sessions can also be used to keep track of the habits and interests of the visitors. This information can be used for advertising purposes, to improve the design of the Web site, or to satisfy the Webmasters' curiosity.

Life Time of an Session

A session starts when a user requests a page from a Web site. A session is closed when the user leaves the Web site.

Managing Sessions

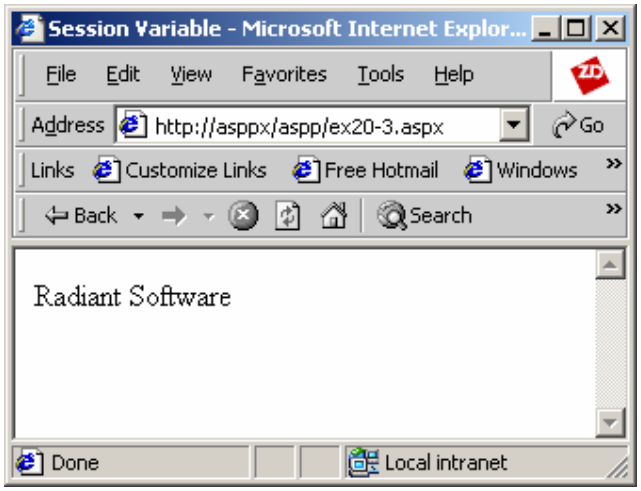
Session variables are similar to **application variables** and can be used in multiple ASP.NET pages. However, unlike an application variable, separate copies of a **session variable** are created for each visitor to the Web site.

In order to store data that will persist throughout a user session, the data has to be stored in a collection of the **Session object**. This is illustrated in the following example:

Practice 8.2

The following program uses a session variable.

```
<HTML>
<HEAD><TITLE>Session Variable</TITLE></HEAD>
<BODY>
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
 Session ["svar"] = "Radiant Software";
 Response.Write (Session["svar"].ToString());
}
</script>
</BODY>
</HTML>
```



The first line in this script assigns the text **Radiant Software** to a session variable named **svar**. The next line in the script displays the text on the screen.

If the same user requests another page the same text is displayed again. For example, consider the following code:

```
<script language="C#" runat="server">
void Page_Load(Object Source, EventArgs e)
{
Response.Write (Session["svar"].ToString());
}
</script>
```

In the above code the session variable was not assigned a value in this code. The session variable **svar** has retained the value it was assigned on the previous page. This cannot be accomplished using a normal script variable. The lifetime of a normal variable extends only throughout a single page. A session variable, on the other hand, persists until the user leaves the Web site.

Note: Session variables exist only in relation to a particular user. The values assigned to a session variable in one user session does not affect the values of the session variables in another user session; i.e. the data stored in session variables are not shared among different users.

Session events in Global.asax

The session handles two events:

- **Session_Start** - Triggered when a session beings
- **Session_End** - Triggered when a session ends

To create a script that executes whenever a new session begins, the script has to be added to the **Session_Start** section of the Global.asax file, as in the example given below:

```
<script language="C#" runat="server">
void Session_Start()
{
  Session["UserName"]= "Unknown";
  Session["UserPassword"]= "Unknown";
}
```

```
</script>
```

The script assigns the value "Unknown" to two session variables named **Username** and **UserPassword**. The **Session_Start** script can be used for other purposes like redirecting the visitors to a new page.

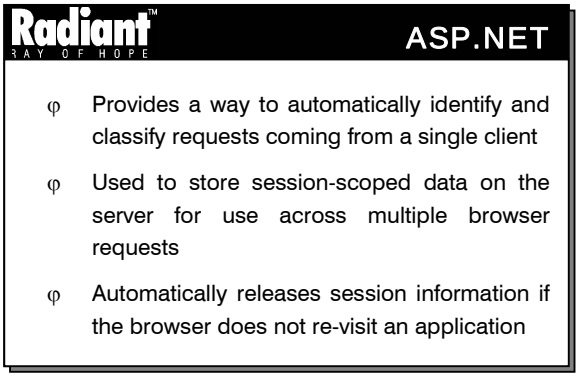
The following code shows the **Session_End** event:

```
<script language="C#" runat="server">
void Session_End()
{
  Response.AppendToLog(Session.SessionID + "ending");
}
</script>
```

The **Session_End** script stores data when the user leaves a session. This information can be used to determine the pages that are more often used to enter and exit the Web site.

When an instance of an object is created with session scope, a new instance of the object is created for each user. In general, the objects created with session scope demand more memory than objects created with application scope. However, there will be less access contention over the object because only one user can access each instance of the object.

SessionState



HTTP is a stateless protocol, which means that it provides no method to automatically recognize that a series of requests are all from the same client. It does not even determine whether a single browser instance is still actively viewing a page or site. As a result, building web applications that need to maintain some cross-request state information (shopping carts, data scrolling, etc) can be extremely challenging without additional infrastructure help.

Session State provides an easy, robust and scalable way to automatically identify and classify requests coming from a single browser client into a logical application "session" on the server. It is also used to store session-scoped data on the server for use across multiple browser requests and to raise the appropriate session lifetime management events (**Session_Start**, **Session_End**, etc.) that can be handled in application code. A Session state will automatically release session information if the browser does not re-visit an application after a specified timeout period.

Session supports provided by ASP.NET are:

- Consistent with other .NET Frameworks APIs
- Reliable session state facility that can survive IIS crashes and worker process restarts and ensures that session data is not lost when these failures occur
- Can be used in both Web-farm and Web-garden scenarios. Enable administrators to allocate more processors/ machines to a web application, and hence increase scalability
- Works with browsers that do not support the use of HTTP cookies for privacy or legal reasons

Properties

The properties of the session state are given below:

Timeout

The timeout property specifies the period in minutes a session should be valid. The default value is 20 minutes.

Cookieless

The Cookieless property indicates whether a cookie should be used as an identity key or not. The default setting is **false**. When the property is set to **true**, ASP.NET automatically encodes cookie data in the URL and passes it along with the request.

Count

The count attribute determines the number of items in the Cookies collection.

Collections

The various collections of the session state are as follows:

Contents

The Contents collection exposes all variable items that have been added to the session-state collection directly through code.

```
Session[“Message”] = “Bar”;
```

StaticObject

The StaticObject collection exposes all variable items that have been added to the Session-state collection through the **<object runat=”server”> </object>** tags with the “Session” scope within the global.asax file.

```
<OBJECT RUNAT=”SERVER” SCOPE=”SESSION” ID=”MyInfo”
  PROGID=”Scripting.Dictionary”>
</OBJECT>
```

The **StaticObjects** collection cannot have objects that are added to this collection in any other place within a .NET Application. The .NET Page Compiler automatically inserts member references to all objects stored within the **StaticObjects** collection at page compilation time. This enables direct access to **Session** objects at the time of page request without having to go through the **Application** collection.

```
<%@Page Language=”C#” %>
<%
Response.Write(MyInfo.ToString());
%>
```

Method

The following are the methods of the Session object.

Abandon

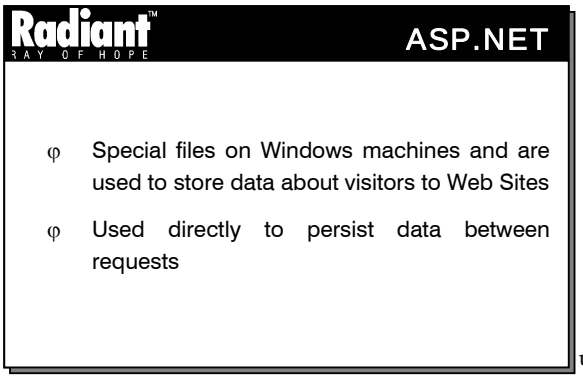
The Abandon method terminates a user's session and releases the memory allocated to maintain user information. If the user leaves the web site, the information is maintained until the session times out. If the user returns before the session times out, the web server does not create another new Session object; it uses the existing one. This is illustrated below:

```
Response.Write (Session.SessionID);
Session.Abandon();
```

In the above code, the session ID of the user is displayed on the screen and then the **Session.Abandon** method is called. When the user requests a new page, the information in the **Session** object is lost and a new ID is assigned to the session. The server treats the user as a new user after the **Abandon** method is called.

The **Session.Abandon** method has certain special scope characteristics. When the **Session.Abandon** method is called, the Web server continues to process the remaining ASP code on the page.

8.4 Cookies



used to store data about visitors to Web sites. A Web server can insert information into these cookie files. Storing cookies on the client is one of the methods used by the ASP.NET's session state in order to associate requests with sessions. Cookies can also be used directly to persist data between requests. However, the data is then stored on the client and sent to the server with every request. The size of the cookie is limited to 4096 bytes by the browser.

Note : Cookies work on the majority of browsers, but fail completely when used with browsers that do not support them, and hence the sessions will also fail.

How to create a Cookie

A cookie is created using HttpCookie class. The Page_Load method in the following code creates a new cookie, initializes it and stores the cookies on the client.

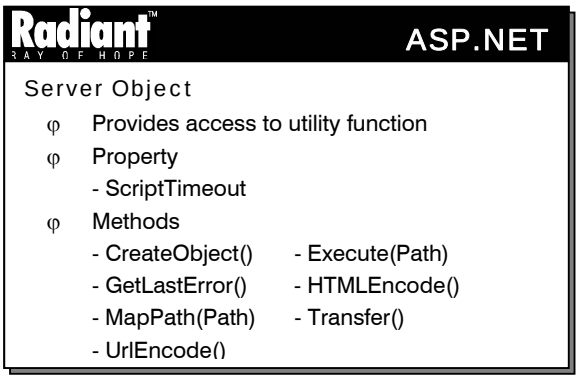
```
protected void Page_Load(Object sender, EventArgs E)
{
        HttpCookie cookie = new HttpCookie("p1");
        cookie.Values.Add("ForeColor", "brown");
        Response.AppendCookie(cookie);
}
```

The HttpCookie class provides a type-safe way to access multiple HTTP cookies. If expiry date is not specified, then the cookie will expire when the user leaves the Web site. The preceding script is used to create a cookie. The other properties of a cookie are:

<%

```
  cookie.Expires = DateTime.MaxValue;
  cookie.Domain =".forindia.com";
  cookie.Path = "/examples";
  cookie.Secure = "True".ToBoolean();
%>
```

8.5 Server Object



The Server object provides access to the utility functions of the server.

Property of Server object

ScriptTimeout

The **ScriptTimeout** property specifies the amount of runtime in seconds for a script before it terminates. The default value is 90 seconds.

The following code causes a script to timeout if the script takes more than 150 seconds to complete.

```
<% Server.ScriptTimeout = 150 %>
```

The value of the **ScriptTimeout** property can be retrieved as well.

```
<% timeout = Server.ScriptTimeout %>
```

Methods of Server object

CreateObject( )

The **CreateObject** method is probably the most widely used and the most important method available through the Built-in Asp.Net Objects. It allows to instantiate the components of a script, or in different terms, create an instance of other objects. As a direct consequence, it is possible to use and access any collections, events, methods, and properties associated with these objects.

There is one mandatory argument **ObjectID** that specifies the type of object to be created.

Execute(Path)

The **Execute** method allows to call another ASPX page from inside an ASPX page. When the called ASPX page completes its tasks, control is then returned to the calling ASPX page. The overall effect is very

similar to a function or subroutine call. Any text or output from the called ASPX page will be displayed on the calling ASPX page. The **Execute** method is a more useful alternative to using server-side includes.

In contrast, the **Transfer** method allows to transfer from one ASPX page to another without returning to the calling ASPX page.

There is one mandatory argument **Path** that is a string specifying either the absolute or relative path of the ASPX page being called. The file name must be included in the path. The entire **Path** must be enclosed inside a pair of quotes. The **Path** argument cannot include a query string, however, any query string that was available to the calling ASPX page will be available to the called ASPX page.

The following is an example that uses the above method:

```
// CallingAsp.aspx
<HTML>
<BODY>
Hello <%Server.Execute("CalledAsp.aspx")%> - Welcome to Radiant
</BODY>
</HTML>
```

```
// CalledAsp.aspx
<%
Response.Write "Anitha"
%>
```

GetLastError( )

The **GetLastError** method returns the last recorded Exception.

HTMLEncode( )

The **HTMLEncode** method applies HTML encoding syntax to a specified string of ASCII characters. For example, this allows to display a HTML tag on a web page and not have it treated as an actual tag.

There is one mandatory argument **String** that is the string to be encoded.

Following is an example of the above method:

```
<% Response.Write Server.HTMLEncode("The tag for a table is: <Table>") %>
```

On executing the above line, the following will be returned to the server:

```
The tag for a table is: &lt;Table&gt;
```

On processing the above line will be displayed in the browser as follows:

```
The tag for a table is: <Table>
```

MapPath(Path)

The **MapPath** method maps a relative or virtual path to a physical path. This method does not check for the validity or the existence of the physical path. If the path starts with a forward or backward slash, the method returns the path as if the path is a full virtual path. If the path does not start with a slash, then the method returns the path relative to the directory of the ASPX file being processed.

There is one mandatory argument path that is the path to be mapped.

Practice 8.3

There is one mandatory argument **Path** that is the path to be mapped.

The following example illustrates the MapPath method.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
The path of this file is <% Response.Write Server.MapPath("test.aspx")
%>
The path of file1 is <% Response.Write Server.MapPath("test\test.aspx")
%>
The path of file2 is <% Response.Write Server.MapPath("\")
%>
</BODY>
</HTML>

The path of this file is D:\Inetpub\wwwroot\test.aspx
The path of file1 is D:\Inetpub\wwwroot\test\test.aspx
The path of file2 is D:\Inetpub\wwwroot
```

In the above code, the first **MapPath** method considers **test.aspx** as a page under the root directory. The second method considers **test.aspx** as a page residing in the **test** sub-directory of the root directory. The third method returns the path of the root directory.

Transfer()

The Transfer method allows to transfer from inside one ASPX page to another ASPX page. All of the state information that has been created for the first (calling) ASPX page will be transferred to the second (called) ASPX page. This transfered information includes all objects and variables that have been given a value in an Application or Session scope, and all items in the Request collections. For example, the second ASPX page will have the same SessionID as the first ASPX page.

When the second (called) ASPX page completes its tasks, control does not return to the first (calling) ASPX page.

In contrast, the Execute method that allows to call another ASPX page, and when the called page has completed its tasks, control returns to the calling ASPX page.

It has one mandatory argument namely Path, that is a string specifying either the absolute or relative path of the ASPX page being called. The file name must be included in the path. The entire Path must be enclosed inside a pair of quotes.

URLEncode()

The **URLEncode** method takes a string and converts it into a URL-encoded format. For example, it is possible to use **URLEncode** to ensure that hyperlinks in Active Server Pages are in the correct format.

This contains a mandatory argument **String** that is the string to be encoded.

The following line of code illustrates the usage of the **URLEncode** method:

```
<% Response.Write Server.URLEncode("http://www.issi.net") %>
```

When the above line is executed in a page, the output is similar to the following:

http%3A%2F%2Fwww%2Eissi%2Enet

8.6 Short Summary

- An ASP.NET application is a collection of files, pages, handlers, modules and executable code that can be invoked in the scope of a given virtual directory and its subdirectories on a Web Application Server
- An ASP.NET application is created when a request is made for the first time to the server
- An application variable contains data that is used in all the web pages and by all the users of an application
- The global.asax file contains syntax for coding the server-side application object
- A collection represents a logical storage unit that helps to manage information
- A session keeps track of user-specific information when a user navigates the site without asking the users identity for every request from the server
- A session begins when a user requests a page from a Web site. A session is closed when the user leaves the Web site
- Session variables like application variables can be used in multiple ASP.NET pages
- Session State provides an easy, robust and scalable way to automatically identify and classify requests coming from a single browser client into a logical application "session" on the server
- Cookies are special files on Windows machines that are used to store data about visitors to Web sites

8.7 Brain Storm

1. What are the uses of Application state?
2. What are the uses of Session state?
3. Explain the importance of global.asax file?
4. What are Cookies? What is their use?
5. How is synchronization of variables achieved?