ASP.NET (4.6)

ASP.NET is a unified Web development model that includes the services necessary for you to build enterprise-class Web applications with a minimum of coding. ASP.NET is part of the .NET Framework, and when coding ASP.NET applications you have access to classes in the .NET Framework

* <asp:Login>: Provides a standard login capability that allows the users to enter their credentials
* <asp:LoginName>: Allows you to display the name of the logged-in user
* <asp:LoginStatus>: Displays whether the user is authenticated or not
* <asp:LoginView>: Provides various login views depending on the selected template
* <asp:PasswordRecovery>: Provides the web site administrators with the capability to email the users their lost password
* 1- **Persistent** Cookie:- A cookie which store your information in your hard disk, and it will stored till time either you will not delete that or cookie time period does not expire. it is browser independent cookie. We need to set expire time.
* 2- **Non - Persistent** Cookie(Temporary cookie):- A cookie which will be alive till that time until your browser is alive, As you will logout to your browser or close your browser, cookie will get expire and user information would also be lost. So it is a browser dependent cookie,

ASP.NET offers three frameworks for creating web applications: ASP.NET Web Forms, ASP.NET MVC, and ASP.NET Web Pages.

ASP.NET Web forms (aspx)- RAD and WYSIWYG

ASP.NET Web Pages – cshtml and vbhtml

In IIS 6.0, there are two request processing pipelines – one for native-code ISAPI filters and the other for managed applications like ASP.NET. However in IIS 7.0, there is one unified request processing pipeline for all requests.  if IIS 7 is configured to work in Classic mode instead of Integrated mode, then it behaves like IIS 6.

When a request comes to IIS then it will first enter HTTP.Sys part of IIS Kernel where it will find routing for the related Application Pool which serves this request.Then,this request is send to the User Mode of IIS where WAS services pass it to the related Application Pool. In an application pool, a worker process is executing which learns the extension of the request and initiate HttpRequest Processing Pipeline. When the unified pipeline receives the first request for any resource in an application, an instance of the [ApplicationManager](https://msdn.microsoft.com/en-us/library/system.web.hosting.applicationmanager.aspx) class is created, which is the application domain that the request is processed in. After the application domain has been created and the [HostingEnvironment](https://msdn.microsoft.com/en-us/library/system.web.hosting.hostingenvironment.aspx) object has been instantiated, application objects such as [HttpContext](https://msdn.microsoft.com/en-us/library/system.web.httpcontext.aspx), [HttpRequest](https://msdn.microsoft.com/en-us/library/system.web.httprequest.aspx), and [HttpResponse](https://msdn.microsoft.com/en-us/library/system.web.httpresponse.aspx) are created and initialized.

HttpRequest Pipeline consist of series of Http Modules and HttpHandler. Whenever some preprocessing is required on the request it is done by using Http Modules and following list of events are fired.

* **Begin Request** - When the new request gets created, the Begin Request will get fired and it will be the first event which will be always raised during the request processing.
* **Authenticate Request** - This event confirms that the authentication configuration has authenticated the request and subscription to this event ensures that before processing the attached handler and module, it is authenticated.
* **Authorize Request** - This event confirms that the request has been authorized by ASP.NET. To implement the custom authorization, you can use this event.
* **Resolve Request Cache** - Once the authorization event gets completed, this event calls the caching module to serve the request from the cache, bypassing execution of the event handler.
* **Map Request Handler** - This event is used by ASP.NET framework to check the extension of the file and accordingly checks the handler for the request.
* **Acquire Request State** - When this event is raised, ASP.NET acquires the state information (Session State) that is associated with the request. In this case the request must has a valid Session ID.
* **Execute Request Handler** - When the handler generates the output, this event gets raised. This is the only request which is not exposed by the HttpApplication class.(MVC controller action is fired in this event)
* **Release Request State** - This event gets raised after ASP.NET finishes executing all request handlers. It also signals ASP.NET State modules to save the current request state.
* **Update Request Cache** - Once ASP.NET finishes the execution of event handlers, this event gets fired to let the caching modules store responses that will be reused to serve identical requests from the cache.
* **Log Request** - This request gets fired before ASP.NET performs any logging for the current request. This event can also be used to perform the custom logging.
* **End Request** - Occurs as the last event in the HTTP pipeline chain of execution when ASP.NET responds to a request. In this event, you can compress or encrypt the response.

After this httpHandler fires and initates the page life cycle.

Cross Page Post Back – Get value of a control from page to another control. On aspx set PostbackURl property of the control to the url of second page and on second page in code behind file use PreviousPage.FindControl("TextBox1")

ASP.net page life Cycle

* Preinit-

Check the [IsPostBack](https://msdn.microsoft.com/en-us/library/system.web.ui.page.ispostback.aspx) property to determine whether this is the first time the page is being processed. The [IsCallback](https://msdn.microsoft.com/en-us/library/system.web.ui.page.iscallback.aspx) and [IsCrossPagePostBack](https://msdn.microsoft.com/en-us/library/system.web.ui.page.iscrosspagepostback.aspx) properties have also been set at this time.Create or re-create dynamic controls.Set a master page dynamically.Set the [Theme](https://msdn.microsoft.com/en-us/library/system.web.ui.page.theme.aspx) property dynamically.Read or set profile property values.

* Init

Raised after all controls have been initialized and any skin settings have been applied. Use this event to read or initialize control properties.

* InitComplete

Use this event to make changes to view state that you want to make sure are persisted after the next postback.

* PreLoad

Raised after the page loads view state for itself and all controls

* Load

The [Page](https://msdn.microsoft.com/en-us/library/system.web.ui.page.aspx) object calls the [OnLoad](https://msdn.microsoft.com/en-us/library/system.web.ui.control.onload.aspx) method on the [Page](https://msdn.microsoft.com/en-us/library/system.web.ui.page.aspx) object, and then recursively does the same for each child control until the page and all controls are loaded. The [Load](https://msdn.microsoft.com/en-us/library/system.web.ui.control.load.aspx) event of individual controls occurs after the [Load](https://msdn.microsoft.com/en-us/library/system.web.ui.control.load.aspx) event of the page.Use the [OnLoad](https://msdn.microsoft.com/en-us/library/system.web.ui.control.onload.aspx) event method to set properties in controls and to establish database connections.

* Control events – Textbox,button events
* [LoadComplete](https://msdn.microsoft.com/en-us/library/system.web.ui.page.loadcomplete.aspx) - Use this event for tasks that require that all other controls on the page be loaded.
* [PreRender](https://msdn.microsoft.com/en-us/library/system.web.ui.control.prerender.aspx) –
* [PreRender](https://msdn.microsoft.com/en-us/library/system.web.ui.control.prerender.aspx)Complete - Raised after each data bound control whose [DataSourceID](https://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.databoundcontrol.datasourceid.aspx) property is set calls its [DataBind](https://msdn.microsoft.com/en-us/library/system.web.ui.control.databind.aspx) method
* SaveStateComplete - Raised after view state and control state have been saved for the page and for all controls.
* Render - All ASP.NET Web server controls have a [Render](https://msdn.microsoft.com/en-us/library/system.web.ui.control.render.aspx) method that writes out the control's markup to send to the browser.
* Unload - Raised for each control and then for the page.

After Postback

* **Page\_LoadState Event** - During the round trips of client-to-server and server-to-client, ASP.NET maintains the state of the controls and the page, in the form of ViewState. ViewState contains the control ID and the value of the control, as a key-value pair. This ViewState is loaded during the post back of the request. **After InitComplete**
* **Page\_ProcessPostData Event** - This event processes the controls state by updating the control values with the data which is post back by the request.
* **Server Control Events** - Here all the events of the controls will be processed like Button click event, Textbox TextChanged event etc. **After Page\_Load**

**HttpHandler and Modules**

ASP.NET uses Http Handler and Http Modules mechanisms to process incoming ASP.NET requests, generate a response, and return that response to the client.

ASP.NET passes each incoming request through a layer of preprocessing HttpModules in the pipeline. ASP.NET allows multiple modules to exist in the pipeline for each request. After the incoming request has passed through each module, it is passed to the HttpHandler which serves the request. Notice that although a single request may pass through many different modules,it can be processed by one handler only. The handler is generally responsible for creating a response to the incoming HTTP request. After the handler has completed execution and generated a response, the response is passed back through a series of post-processing modules, before it is returned to the client.

**HTTPMODULES**

HttpModules are simple classes that can plug themselves into the request-processing pipeline. They do this by hooking into a handful of events thrown by the application as it processes the HTTP request. To create an HttpModule, you simply create a class that derives from the System.Web.IHttpModule interface.

Whenever a custom Http Module is Loaded it create instance of HttpApplication using which we can get current Request,Response and Session. Various HttpRequest event are fired too like – BeginRequest,AuthenticateRequest,AuthorizeRequest etc

This interface requires you to implement two methods: Init and Dispose

<system.webServer>

<modules>

<add name="AppendMessage" type="AppendMessage, App\_Code" />

</modules>

</system.webServer>

Uses:

*Security*: For authenticating a request before the request is handled.

*Statistics and Logging*: Since modules are called for every request they can be used for gathering statistics and for logging information.

*Custom header*:  Since response can be modified, one can add custom header information to the response. Like File and Image compression mechanism

HttpHandlers(\*.ashx)

Handlers are the last stop for incoming HTTP requests and are ultimately the point in the requestprocessing

pipeline that is responsible for serving up the requested content, be it an ASPX page, .ashx ,HTML,

plaintext, or an image.

It is inherited from IHttpHandler and has ProcessRequest Method and IsReusable property

It can be used to check server health to ie (Ping IIS)

The normal entry point for a handler is the ProcessRequest method. However you may have code in the class constructor which puts together some instance values which are expensive to build.

If you specify Reusable to be true the application can cache the instance and reuse it in another request by simply calling its ProcessRequest method again and again, without having to reconstruct it each time.

**State Management Techniques in ASP.net**

HTTP is a stateless protocol, so it cannot remember things. Many current web applications have to

maintain state, though: They need to remember the logged-in user, the contents of the shopping cart,

and more. ASP.NET’s session management support provides an easy API to store data during a session.

|  |  |  |
| --- | --- | --- |
| **SERVER-SIDE OPTION** | PROS | CONS |
| Application State | Fast. Shared among all users. | State is stored once per server |
| Cache Object  (Application Scope) | Like the Application State but includes  expiration via dependencies (see  Chapter 22 on caching). | State is stored once per server in multiple  server confi gurations. |
| Session State | Three choices: in process, out of  process, and DB-backed. Can be  confi gured as cookieless | Can be abused. You pay a serialization  cost when objects leave the process.  In process requires web server affi nity.  Cookieless confi guration makes hijacking  easier. |
| Custom Database | State can be accessed by any server in  a web farm. | Pay a serialization and persistence cost  when objects leave the process. |

|  |  |  |
| --- | --- | --- |
| **CLIENT-SIDE OPTION** | PROS | CONS |
| Cookie | Simple | Can be rejected by browser. Not appropriate  for large amounts of data. Inappropriate for  sensitive data. Size cost is paid on every HTTP  request and response. |
| Hidden Field | Simple for page-scoped data | Not appropriate for large amounts of data.  Inappropriate for sensitive data. |
| View State | Simple for page-scoped data | Encoding of serialized object as binary Base64-  encoded data adds approximately 30 percent  overhead. Small serialization cost. Has a  negative reputation, particularly with DataGrids. |
| Control State | Simple for page-scoped control specific data | Like ViewState, but used for controls that  require ViewState even if the developer has  turned it off |
| QueryString | Incredibly simple and often  convenient if you want your URLs  to be modifi ed directly by the  end user | Comparatively complex. Can’t hold a lot of  information. Inappropriate for sensitive data.  Easily modifi ed by the end user. |
| HTML5 Web Storage | Simple API to store name-value  Pairs | Data is never automatically sent to the server, so  mostly used for client logic only. |

HttpContext.Current.Session["SomeSessionState"] = "Here is some data";

string myValue = (string)Session["mykey"];

Label1.Text = myValue;

ASP.NET ships with the following three storage providers:

➤ **In-Process Session State Store**: Stores sessions in the ASP.NET in-memory cache

➤ **Out-of-Process Session State Store**: Stores sessions in the ASP.NET state server service aspnet\_

state.exe

➤ **Sql Session State Store**: Stores sessions in Microsoft SQL Server database and is confi gured with

aspnet\_regsql.exe

The format of the web.config fi le’s sessionState element is shown in the following code:

<configuration>

<system.web>

<sessionState mode="Off|InProc|StateServer|SQLServer|Custom" ../>

</system.web>

**In-Process Session State**

When the confi guration is set to InProc, session data is stored in the HttpRuntime’s internal cache in an implementation of ISessionStateItemCollection that implements ICollection. The session state key is a 120-bit value string that indexes this global dictionary of object references. When session state is in process, objects are stored as live references. This mechanism is incredibly fast because no serialization occurs, nor do objects leave the process space. Certainly, your objects are not garbage-collected if they exist in the In-Process Session object because a reference is still being held.

If the worker process or application domain recycles, all session state data is lost. In

addition, the ASP.NET application may restart for a number of reasons, such as the following:

➤ You have changed the web.config or Global.asax fi le or “touched” it by changing its modified date.

➤ You have modifed fi les in the \bin or \App\_Code directory.

➤ The processModel element has been set in the web.config or machine.config fi le indicating when the application should restart. Conditions that could generate a restart might be a memory limit or request-queue limit.

➤ Antivirus software modifies any of the previously mentioned fi les. This is particularly common with antivirus software that *inoculates* fi les.

More details about the page and the Session object can be displayed to the developer if page tracing is enabled. <trace enabled="true" pageOutput="true" />

ASP.net validators

ASP.NET validation controls validate the user input data to ensure that useless, unauthenticated, or contradictory data don't get stored

* RequiredFieldValidator
* RangeValidator
* CompareValidator
* RegularExpressionValidator
* CustomValidator
* ValidationSummary

ASP.NET allows you to declare exactly what your page requires of the Session object via the EnableSessionState @Page attribute. The options are True, False, or ReadOnly:

➤ EnableSessionState=*"*True*"*: The page requires read and write access to the session. The session

with that session ID will be locked during each request.

➤ EnableSessionState=*"*False*"*: The page does not require access to the Session. If the code uses

the Session object anyway, an HttpException is thrown, stopping page execution.

➤ EnableSessionState=*"*ReadOnly*"*: The page requires read-only access to the session. A reader lock is held on the session for each request, but concurrent reads from other pages can occur. The order in which locks are requested is essential. As soon as a writer lock is requested, even before a thread is granted access, all subsequent reader lock requests are blocked, regardless of whether a reader lock is

currently held. Although ASP.NET can obviously handle multiple requests, only one request at a time gets write access to a session. By setting the session state to ReadOnly, multiple requests may access the session at the same time.

<%@ Page Language="C#" **EnableSessionState=**"**ReadOnly**" %>

**Out-of-Process Session State**

Out-of-process session state is held in a process called aspnet\_state.exe that runs as a Windows Service.You can start the ASP.NET state service by using the Services MMC snap-in or by running the following net command from an administrative command line:

net start aspnet\_state

<configuration>

<system.web>

<sessionState mode="StateServer"

stateConnectionString="tcpip=127.0.0.1:42424"/>

</system.web>

</configuration>

ASP.NET 4.*x* includes a capability to compress the objects that are stored in an out-of-process state. This is illustrated in the following snippet of code:

<sessionState

mode="SqlServer"

sqlConnectionString="data source=dbserver;Initial Catalog=aspnetstate"

allowCustomSqlDatabase="true"

**compressionEnabled=**"**true**"

/>

**SQL-Backed Session State**

ASP.NET sessions can also be stored in a SQL Server database. InProc offers speed, state server offers a resilience/speed balance, and storing sessions in SQL Server offers resilience that can serve sessions to a large web farm that persists across IIS restarts, if necessary SQL-backed session state is confi gured with aspnet\_regsql.exe.

<sessionState mode="SQLServer" sqlConnectionString="data source=127.0.0.1;user

id=sa;password=Wrox"/ >

ASPStateTempSessions where data related to sessions are stored in tempdb

The ASP.NET session state ID was stored in a cookie. Some devices do not support

cookies, or a user may have turned off cookie support in her browser. If you include the cookieless=*"*UseUri*"* attribute in the web.config fi le, ASP.NET does not send the

ASP.NET session ID back as a cookie. Instead, it modifi es every URL to include the session ID just before

the requested page:

<sessionState mode="SQLServer" cookieless="UseUri" sqlConnectionString="data

source=127.0.0.1;user id=sa;password=Wrox" />

**THE APPLICATION OBJECT**

The Application object is the equivalent of a bag of global variables for your ASP.NET application. The Application object is not global to the machine; it is global to the HttpApplication. If you are running in the context of a web farm, each ASP.NET application on each web server has its own Application object. Because ASP.NET applications are multithreaded and are receiving requests that are being handled by your code on multiple threads, access to the Application object should be managed using the Application.Lock and Application.Unlock methods. If your code does not call Unlock directly (which it should, shame on you) the lock is removed implicitly at the end of the HttpRequest that called Lock originally.

**QUERYSTRINGS**

The URL, or QueryString, is the ideal place for navigation-specifi c — not user-specifi c — data.

string v = Request.**QueryString**["param"];

if (v != null)

{

Response.Write("param is ");

Response.Write(v);

}

**COOKIES**

Cookies should be used to store only non-sensitive information, or information that can be retrieved from an authoritative source. Cookies should not be trusted, and their

contents should be able to be validated. For example, if a Forms Authentication cookie has been tampered with, the user is logged out and an exception is thrown. If an invalid session ID cookie is passed in for an expired session, a new cookie can be assigned

➤ Cookies are passed back and forth on *every* request. That means you are paying for the size of your cookie during *every* HTTP GET and HTTP POST. If you have ten 1-pixel spacer GIFs on your page used for table layouts, the user’s browser is sending the same cookie *eleven* times — once for the page itself,and once for each spacer GIF, even if the GIF is already cached.

➤ Cookies can be stolen, sniffed, and faked. If your code counts on a cookie’s value, have a plan in your code for the inevitability that the cookie will be corrupted or tampered with.

➤ What is the expected behavior of your application if a cookie does not show? What if it is 4096 bytes? Be prepared. You should design your application around the “principle of least surprise.”

Your application should attempt to heal itself if cookies are found missing or if they are larger than expected.

➤ Think twice before Base64 encoding anything large and placing it in a cookie. If your design depends on this kind of technique, rethink using either the session or another backing-store.

**HIDDEN FIELDS, VIEWSTATE, AND CONTROLSTATE**

Hidden input fields such as <input type=*"*hidden*"* name=*"*myName*"* /> are sent back as name/valuepairs in a form POST exactly like any other control, except that they are not rendered.

ViewState, on the other hand, exposes itself as a collection of key/value pairs like the Session object, but renders itself as a hidden field with the name *"* \_\_VIEWSTATE*"* like this:

<input type="hidden" name="\_\_VIEWSTATE" value="/AAASSDAS ... Y/lOI=" />

Any objects put into the ViewState must be marked Serializable. ViewState serializes the objects with a special binary formatter called the LosFormatter. LOS stands for *limited object serialization*. It serialize any kind of object, but it is optimized to contain strings, arrays, and hashtables.

Control State

In addition to view state, ASP.NET supports control state. The page uses control state to persist control information that must be retained between postbacks, even if view state is disabled for the page or for a control. Like view state, control state is stored in one or more hidden fields.

Set value in view state

  //Value of Textbox1 and TectBox2 is assigin on the ViewState

    ViewState["name"] = TextBox1.Text;

    ViewState["password"] = TextBox2.Text;

Get value from viewstate

 if (ViewState["name"] != null)

    {

        TextBox1.Text = ViewState["name"].ToString();

    }

    if (ViewState["password"] != null)

    {

         TextBox2.Text = ViewState["password"].ToString();

    }

To disable view state: <%Page Language="C#" EnableViewState="false";%>

To secure view state: <%Page Language="C#" EnableViewState="true ViewStateEncryptionMode="Always" %>

ViewStateEncryptionMode :Always ,Auto,Never

<%Page Language="C#" EnableViewState="true"  EnableViewStateMac="true";%>

Set cookies

HttpCookie aCookie = new HttpCookie("lastVisit");

aCookie.Value = DateTime.Now.ToString();

aCookie.Expires = DateTime.Now.AddDays(1);

Response.Cookies.Add(aCookie);

Read Cookies

if(Request.Cookies["userName"] != null)

{

HttpCookie aCookie = Request.Cookies["userName"];

Label1.Text = Server.HtmlEncode(aCookie.Value);

}

View state data is encrypted and Hidden field is not encrypted

Caching

*Caching* is the process of storing frequently used data on the server to fulfll subsequent requests.

**Output Caching**

*Output caching* is a way to keep the dynamically generated page content in the server’s memory or disk for later retrieval. This type of cache saves post-rendered content so it will not have to be regenerated again the next time it is requested. After a page is cached, it can be served up again when any subsequent requests are made to the server. You apply output caching by inserting an OutputCache page directive at the top of an.aspx page, as follows:

<%@ OutputCache Duration="60" VaryByParam="None" %>

The Duration attribute defi nes the number of seconds a page is stored in the cache.

<%@ OutputCache Duration="20" Location="Server" VaryByParam="state"  
VaryByCustom="minorversion" VaryByHeader="Accept-Language"%>

Caching Different versions of page in ASP.Net

* **VaryByParam: -** Caches different version depending on input parameters/querystrings send through HTTP POST/GET.
* **VaryByHeader: -** Caches different version depending on the contents of the page header.(cookie,accept language)
* **VaryByCustom:-** Lets you customize the way the cache handles page variations by declaring the attribute and overriding the GetVaryByCustomString handler.
* **VaryByControl:-** Caches different versions of a user control based on the value of properties of ASP objects in the control.

The Cache object is defined in the System.Web.Caching namespace. When you add an item to the cache, you can define the dependency relationships that can force that item to be removed from the cache under specific activities of dependencies.

* File dependency: Allows you to invalidate a specific cache item when a disk based file or files change.
* Time-based expiration: Allows you to invalidate a specific cache item depending on a predefined time.
* Key dependency: Allows you to invalidate a specific cache item when another cached item changes
* SQL cache dependencies is a new feature in ASP.NET 2.0 which can automatically invalidate a cached data object (such as a DataSet) when the related data is modified in the database
* Page Output Caching

Page output caching adds the response of a page to the cache object. Later when the page is requested, the page is displayed from the cache rather than creating the page object and displaying it. Page output caching is good if the site is fairly static.

* Page Fragment Caching

If parts of the page are changing, you can wrap the static sections as user controls and cache the user controls using page fragment caching.

Cache.Insert("DSN", connectionString,New CacheDependency(Server.MapPath("~/myconfig.xml")))

Object Caching demo:

Var obj={EmployeeID: 1,EmpName:”gagan”};

Cache[‘userobj’] = obj;

Data Caching

In the very first time retrieve the records from the sql server database and stored into the Cache memory. After that records will retrieve from Cache memory (IIS).

if (Cache["Employee"] == null)

{

dtEmployee = new DataTable("Employee");

dtEmployee.Columns.Add("EmpID", typeof(int));

dtEmployee.Columns.Add("EmpName", typeof(string));

DataRow rs = dtEmployee.NewRow();

rs[0] = 10;  
 rs[1] = "Annathurai";  
 dtEmployee.Rows.Add(rs);  
  
 //To assign datatable to cache memory.

Cache["Employee"] = dtEmployee; Cache.Insert("Employee", dtEmployee, null, Cache.NoAbsoluteExpiration, ts); Response.Write("Its processing with Database hit");

}

AJAX in asp.net

The controls provided by ASP.NET for having AJAX functionality are:

1. ScriptManager -This control is required on each page that needs to have AJAX features implemented on it. The main functionality of a ScriptManager control is to push Microsoft AJAX framework code to the client side when the page is being rendered. This control can be thought of as the agent which will write the JavaScript required on the client side to facilitate AJAX functionality.
2. ScriptManagerProxy - ScriptManagerProxy should be used on content pages that have master pages containing a ScriptManagercontrol. It can also be used inside UserControls when the page containing the UserControl already has the ScriptManager control.
3. UpdatePanel - This is a container control that contains other ASP.NET controls. This control defines a region that is capable of making partial page updates. We can add various server controls inside an UpdatePanel and this controls inside the UpdatePanel will communicate to the server irrespective of the page's postback.
4. UpdateProgress - The scenario we just handled gave us the results instantly, but imagine a scenario where the server side processing for the asynchronous event takes some time. If the operation is time consuming then we can provide the user feedback by using the UpdateProgress control inside the UpdatePanel.
5. Timer - There might be some scenarios where we want to update a particular portion of the page after some time duration irrespective of user action. To achieve this, we can use the Timer control.

|  |  |
| --- | --- |
| GridView | DataGrid |
| It was introduced with Asp.Net 2.0. | It was introduced with Asp.Net 1.0. |
| Built-in supports for Paging and Sorting. | For sorting you need to handle SortCommand event and rebind grid required and for paging you need to handle the PageIndexChanged event and rebind grid required. |
| Built-in supports for Update and Delete operations. | Need to write code for implementing Update and Delete operations. |
| Supports auto format or style features. | This features is not supported. |
| Performance is slow as compared to DataGrid | Performance is fast as compared to GridView. |

|  |  |
| --- | --- |
| ListView | GridView |
| It was introduced with Asp.Net 3.5. | It was introduced with Asp.Net 2.0. |
| Template driven. | Rendered as Table. |
| Built-in supports for Data grouping. | Need to write custom code. |
| Built-in supports for Insert operation. | Need to write custom code. |
| Provides flexible layout to your data. | Need to write custom code. |
| Performance is fast is compared to GridView. | Performance is slow as compared to ListView. |

|  |  |
| --- | --- |
| DataList | Repeater |
| Rendered as Table. | Template driven. |
| Automatically generates columns from the data source. | This features is not supported. |
| Selection of row is supported. | Selection of row is not supported. |
| Editing of contents is supported. | Editing of contents is not supported. |
| You can arrange data items horizontally or vertically in DataList by using property RepeatDirection. | This features is not supported. |
| Performance is slow as compared to Repeater | This is very light weight and fast data control among all the data control. |

|  |  |
| --- | --- |
| GridView | Repeater |
| It was introduced with Asp.Net 2.0. | It was introduced with Asp.Net 1.0. |
| Rendered as Table. | Template driven. |
| Automatically generates columns from the data source. | This features is not supported. |
| Selection of row is supported. | Selection of row is not supported. |
| Editing of contents is supported. | Editing of contents is not supported. |
| Built-in Paging and Sorting is provided. | You need to write custom code. |
| Supports auto format or style features. | This has no this features. |
| Performance is very slow as compared to Repeater. | This is very light weight and fast data control among all the data control. |

|  |  |
| --- | --- |
| GridView | DataList |
| It was introduced with Asp.Net 2.0. | It was introduced with Asp.Net 1.0. |
| Built-in Paging and Sorting is provided. | You need to write custom code. |
| Built-in supports for Update and Delete operations. | Need to write code for implementing Update and Delete operations. |
| Supports auto format or style features. | This features is not supported. |
| RepeatDirection property is not supported. | You can arrange data items horizontally or vertically in DataList by using property RepeatDirection. |
| Doesn’t support customizable row separator. | Supports customizable row separator by using SeparatorTemplate. |
| Performance is slow as compared to DataList. | Performance is fast is compared to GridView. |

Security

The simple addition of the <authentication> element to the web.config fi le turns on everything that you

need to start using the membership service provided by ASP.NET 4.5

➤ name: Defi nes the name used for the cookie sent to end users after they have been authenticated. By

default, this cookie is named .ASPXAUTH.

➤ loginUrl: Specifi es the page location to which the HTTP request is redirected for logging in the

user if no valid authentication cookie (.ASPXAUTH or otherwise) is found. By default, it is set to

Login.aspx.

➤ protection: Specifi es the amount of protection that you want to apply to the cookie that is stored

on the end user’s machine after he has been authenticated. The possible settings include All, None,

Encryption, and Validation. You should always attempt to use All.

➤ timeout: Defi nes the amount of time (in minutes) after which the cookie expires. The default value is

30 minutes.

➤ path: Specifi es the path for cookies issued by the application.

➤ requireSSL: Defi nes whether you require that credentials be sent over an encrypted wire (SSL)

instead of clear text.

➤ slidingExpiration: Specifi es whether the timeout of the cookie is on a sliding scale. The default

value is true. This means that the end user’s cookie does not expire until 30 minutes (or the time

specifi ed in the timeout attribute) after the last request to the application has been made. If the value

of the slidingExpiration attribute is set to false, the cookie expires 30 minutes from the fi rst

request.

➤ cookieless: Specifi es how the cookies are handled by ASP.NET. The possible values

include UseDeviceProfile, UseCookies, AutoDetect, and UseUri. The default value is

UseDeviceProfile. This value detects whether to use cookies based on the user agent of the device.

UseCookies requires that all requests have the credentials stored in a cookie. AutoDetect autodetermines

whether the details are stored in a cookie on the client or within the URI (it does this by

sending a test cookie fi rst). Finally, UseUri forces ASP.NET to store the details within the URI on all

instances  
  
<configuration>

<system.web>

**<authentication mode="Forms">**

**<forms name=".ASPXAUTH"**

**loginUrl="Login.aspx"**

**protection="All"**

**timeout="30"**

**path="/"**

**requireSSL="false"**

**slidingExpiration="true"**

**cookieless="UseDeviceProfile" />**

**</authentication>**

</system.web>

</configuration>

Windows Authentication

One good advantage of Windows Authentication is it's security. Your web application will enjoy the same security policy that the Windows OS uses.  
  
And another advantage of Windows Authentication is, the user's credentials will not travel over the internet so there is less security issues from the data transfer.

Impersonation is independent of the authentication **mode** configured using the [authentication](https://msdn.microsoft.com/en-us/library/532aee0e.aspx) configuration element. The authentication element is used to determine the [User](https://msdn.microsoft.com/en-us/library/system.web.httpcontext.user.aspx) property of the current [HttpContext](https://msdn.microsoft.com/en-us/library/system.web.httpcontext.aspx). Impersonation is used to determine the [WindowsIdentity](https://msdn.microsoft.com/en-us/library/system.security.principal.windowsidentity.aspx) of the ASP.NET application.

We have added an <authentication> tag within the <system.web> section. And the authentication mode is “Windows”.  
  
So now the application will check whether or not the user has credentials in the server computer. If the user has credentials in the server then he/she is an authenticated user, otherwise not.

protected void Page\_Load(object sender, EventArgs e)

{

   Response.Write("Username := " + HttpContext.Current.User.Identity.Name);

}

Passport authentication lets you to use Microsoft's passport service to authenticate users of your application. If your users have signed up with passport, and you configure the authentication mode of the application to the passport authentication, all authentication duties are offloaded to the passport servers.

Passport uses an encrypted cookie mechanism to indicate authenticated users. If users have already signed into passport when they visit your site, they'll be considered authenticated by ASP.NET. Otherwise they'll be redirected to the passport servers to log in. When they are successfully log in, they'll be redirected back to your site

Authorization

The first step in working with the role management service is to change any of the role management

provider’s behaviors either in the machine.config or from the web.config fi les.

<roleManager

enabled="false"

cacheRolesInCookie="false"

cookieName=".ASPXROLES"

cookieTimeout="30"

cookiePath="/"

cookieRequireSSL="false"

cookieSlidingExpiration="true"

cookieProtection="All"

defaultProvider="AspNetSqlRoleProvider"

createPersistentCookie="false"

maxCachedResults="25">

<providers>

<clear />

<add connectionStringName="LocalSqlServer" applicationName="/"

name="AspNetSqlRoleProvider" type="System.Web.Security.SqlRoleProvider,

System.Web, Version=4.0.0.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a" />

<add applicationName="/" name="AspNetWindowsTokenRoleProvider"

type="System.Web.Security.WindowsTokenRoleProvider, System.Web,

Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />

</providers>

</roleManager>

<configuration>

<system.web>

<authorization>

<allow roles="Admins"/>

<deny users="\*"/>

</authorization>

</system.web>

</configuration>

<https://www.codeproject.com/Articles/14976/ASP-NET-Caching-Dependencies>

<http://www.aspdotnet-suresh.com/2016/08/fragment-caching-in-aspnet-with-example.html>

<http://codebetter.com/darrellnorton/2004/05/04/asp-net-varybycustom-page-output-caching/>