**Introduction to ASP.NET State Management**





[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Introduction 3](#_Toc223531477)

[Application State 3](#_Toc223531478)

[Session State 5](#_Toc223531479)

[Cookies 9](#_Toc223531480)

[ViewState Overview 22](#_Toc223531481)

[Caching 27](#_Toc223531482)

## Introduction

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

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

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

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

###   Application State

Application state is a data repository available to all classes in an ASP.NET application. Application state is stored in memory on the server and is faster than storing and retrieving information in a database. Unlike session state, which is specific to a single user session, application state applies to all users and sessions. Therefore, application state is a useful place to store small amounts of often-used data that does not change from one user to another. For information on saving data on a per-user basis see [ASP.NET Session State Overview](http://msdn.microsoft.com/en-us/library/ms178581.aspx) and [ASP.NET Profile Properties Overview](http://msdn.microsoft.com/en-us/library/2y3fs9xs.aspx).

 Using Application State

Application state is stored in an instance of the [HttpApplicationState](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.aspx) class. This class exposes a key-value dictionary of objects.

The [HttpApplicationState](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.aspx) instance is created the first time a user accesses any URL resource in an application. The [HttpApplicationState](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.aspx) class is most often accessed through the [Application](http://msdn.microsoft.com/en-us/library/system.web.httpcontext.application.aspx) property of the [HttpContext](http://msdn.microsoft.com/en-us/library/system.web.httpcontext.aspx) class.

You can use application state in two ways. You can add, access, or remove values from the [Contents](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.contents.aspx) collection directly through code. The [HttpApplicationState](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.aspx) class can be accessed at any time during the life of an application. However, it is often useful to load application state data when the application starts. To do so, you can put code to load application state into the Application\_Start method in the Global.asax file. For more information see [ASP.NET Application Life Cycle Overview for IIS 5.0 and 6.0](http://msdn.microsoft.com/en-us/library/ms178473.aspx).

Alternatively, you can add objects to the [StaticObjects](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.staticobjects.aspx) collection via an <object runat="server"> declaration in your Web application's Global.asax file. Application state defined in this way can then be accessed from code anywhere in your application. The following example shows an object declaration for an application state value:

Copy Code

<object runat="server" scope="application" ID="MyInfo"

 PROGID="MSWC.MYINFO">

</object>

You can add objects to the [StaticObjects](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.staticobjects.aspx) collection only in the Global.asax file. The collection throws a [NotSupportedException](http://msdn.microsoft.com/en-us/library/system.notsupportedexception.aspx) if you attempt to add objects directly through code.

You can access members of objects stored in application state without having to reference the [Application](http://msdn.microsoft.com/en-us/library/system.web.httpcontext.application.aspx) collection. The following code example shows how to reference a member of an object defined in the [StaticObjects](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.staticobjects.aspx) collection of application state:

C#

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

 Label1.Text = MyInfo.Title;

End Sub

 Application State Considerations

When using application state, you must be aware of the following important considerations:

* Resources   Because it is stored in memory, application state is very fast compared to saving data to disk or a database. However, storing large blocks of data in application state can fill up server memory, causing the server to page memory to disk. As an alternative to using application state, you can use the ASP.NET cache mechanism for storing large amounts of application data. The ASP.NET cache also stores data in memory and is therefore very fast; however, ASP.NET actively manages the cache and will remove items when memory becomes scarce. For more information see [ASP.NET Caching Overview](http://msdn.microsoft.com/en-us/library/ms178597.aspx).
* Volatility   Because application state is stored in server memory, it is lost whenever the application is stopped or restarted. For example, if the Web.config file is changed, the application is restarted and all application state is lost unless application state values have been written to a non-volatile storage medium such as a database.
* Scalability   Application state is not shared among multiple servers serving the same application, as in a Web farm, or among multiple worker processes serving the same application on the same server, as in a Web garden. Your application therefore cannot rely on application state containing the same data for application state across different servers or processes. If your application will run in multi-processor or multi-server environments, consider using a more scalable option, such as a database, for data that must preserve fidelity across the application.
* Concurrency   Application state is free-threaded, which means that application state data can be accessed simultaneously by many threads. Therefore, it is important to ensure that when you update application state data, you do so in a thread-safe manner by including built-in synchronization support. You can use the [Lock](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.lock.aspx) and [UnLock](http://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate.unlock.aspx) methods to ensure data integrity by locking the data for writing by only one source at a time. You can also reduce the likelihood of concurrency problems by initializing application state values in the Application\_Start method in the Global.asax file.

###  Session State

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

Alternatives to session state include the following:

* Application state, which stores variables that can be accessed by all users of an ASP.NET application.
* Profile properties, which persists user values in a data store without expiring them.
* ASP.NET caching, which stores values in memory that is available to all ASP.NET applications.
* View state, which persists values in a page.
* Cookies.
* The query string and fields on an HTML form that are available from an HTTP request.

##### Session Variables

Session variables are stored in a [SessionStateItemCollection](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstateitemcollection.aspx) object that is exposed through the [HttpContext..::.Session](http://msdn.microsoft.com/en-us/library/system.web.httpcontext.session.aspx) property. In an ASP.NET page, the current session variables are exposed through the Session property of the Page object.

The collection of session variables is indexed by the name of the variable or by an integer index. Session variables are created by referring to the session variable by name. You do not have to declare a session variable or explicitly add it to the collection. The following example shows how to create session variables in an ASP.NET page for the first and last name of a user, and set them to values retrieved from [TextBox](http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.textbox.aspx) controls.

C#

Session["FirstName"] = FirstNameTextBox.Text;

Session["LastName"] = LastNameTextBox.Text;

Session variables can be any valid .NET Framework type. The following example stores an [ArrayList](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) object in a session variable named StockPicks. The value returned by the StockPicks session variable must be cast to the appropriate type when you retrieve it from the [SessionStateItemCollection](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstateitemcollection.aspx).

C#

// When retrieving an object from session state, cast it to

// the appropriate type.

ArrayList stockPicks = (ArrayList)Session["StockPicks"];

// Write the modified stock picks list back to session state.

Session["StockPicks"] = stockPicks;

##### Session Identifiers

Sessions are identified by a unique identifier that can be read by using the [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) property. When session state is enabled for an ASP.NET application, each request for a page in the application is examined for a [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value sent from the browser. If no [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value is supplied, ASP.NET starts a new session and the [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value for that session is sent to the browser with the response.

By default, [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) values are stored in a cookie. However, you can also configure the application to store [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) values in the URL for a "cookieless" session.

A session is considered active as long as requests continue to be made with the same [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value. If the time between requests for a particular session exceeds the specified time-out value in minutes, the session is considered expired. Requests made with an expired [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value result in a new session.

##### Cookieless SessionIDs

By default, the [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value is stored in a non-expiring session cookie in the browser. However, you can specify that session identifiers should not be stored in a cookie by setting the cookieless attribute to true in the [sessionState](http://msdn.microsoft.com/en-us/library/h6bb9cz9.aspx) section of the Web.config file.

The following example shows a Web.config file that configures an ASP.NET application to use cookieless session identifiers.

<configuration>

 <system.web>

 <sessionState cookieless="true"

 regenerateExpiredSessionId="true" />

 </system.web>

</configuration>

ASP.NET maintains cookieless session state by automatically inserting a unique session ID into the page's URL. For example, the following URL has been modified by ASP.NET to include the unique session ID lit3py55t21z5v55vlm25s55:

http://www.example.com/(S(lit3py55t21z5v55vlm25s55))/orderform.aspx

When ASP.NET sends a page to the browser, it modifies any links in the page that use an application-relative path by embedding a session ID value in the links. (Links with absolute paths are not modified.) Session state is maintained as long as the user clicks links that have been modified in this manner. However, if the client rewrites a URL that is supplied by the application, ASP.NET may not be able to resolve the session ID and associate the request with an existing session. In that case, a new session is started for the request.

The session ID is embedded in the URL after the slash that follows the application name and before any remaining file or virtual directory identifier. This enables ASP.NET to resolve the application name before involving the [SessionStateModule](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstatemodule.aspx) in the request.

##### Regenerating Expired Session Identifiers

By default, the session ID values that are used in cookieless sessions are recycled. That is, if a request is made with a session ID that has expired, a new session is started by using the [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value that is supplied with the request. This can result in a session unintentionally being shared when a link that contains a cookieless [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value is used by multiple browsers. (This can occur if the link is passed through a search engine, through an e-mail message, or through another program.) You can reduce the chance of session data being shared by configuring the application not to recycle session identifiers. To do this, set the regenerateExpiredSessionId attribute of the [sessionState](http://msdn.microsoft.com/en-us/library/h6bb9cz9.aspx) configuration element to true. This generates a new session ID when a cookieless session request is made with an expired session ID.

##### Custom Session Identifiers

You can implement a custom class to supply and validate [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) values. To do so, create a class that inherits the [SessionIDManager](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.aspx) class and override the [CreateSessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.createsessionid.aspx) and [Validate](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.validate.aspx) methods with your own implementations. For an example, see the example provided for the [CreateSessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.createsessionid.aspx) method.

You can replace the [SessionIDManager](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.aspx) class by creating a class that implements the [ISessionIDManager](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.isessionidmanager.aspx) interface. For example, you might have a Web application that associates a unique identifier with non-ASP.NET pages (such as HTML pages or images) by using an ISAPI filter. You can implement a custom [SessionIDManager](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionidmanager.aspx) class to use this unique identifier with ASP.NET session state. If your custom class supports cookieless session identifiers, you must implement a solution for sending and retrieving session identifiers in the URL.

##### Session Modes

ASP.NET session state supports several storage options for session variables. Each option is identified as a session-state [Mode](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.mode.aspx) type. The default behavior is to store session variables in the memory space of the ASP.NET worker process. However, you can also specify that session state should be stored in a separate process, in a SQL Server database, or in a custom data source. If you do not want session state enabled for your application, you can set the session mode to [Off](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstatemode.off.aspx).

##### Session Events

ASP.NET provides two events that help you manage user sessions. The **Session\_OnStart** event is raised when a new session starts, and the **Session\_OnEnd** event is raised when a session is abandoned or expires. Session events are specified in the Global.asax file for an ASP.NET application.

The **Session\_OnEnd** event is not supported if the session [Mode](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.mode.aspx) property is set to a value other than [InProc](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstatemode.inproc.aspx), which is the default mode.

##### Configuring Session State

Session state is configured by using the [sessionState](http://msdn.microsoft.com/en-us/library/h6bb9cz9.aspx) element of the system.web configuration section. You can also configure session state by using the [EnableSessionState](http://msdn.microsoft.com/en-us/library/system.web.configuration.pagessection.enablesessionstate.aspx) value in the @ Page directive.

The sessionState element enables you to specify the following options:

* The mode in which the session will store data.
* The way in which session identifier values are sent between the client and the server.
* The session [Timeout](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.timeout.aspx) value.
* Supporting values that are based on the session [Mode](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.mode.aspx) setting.

The following example shows a sessionState element that configures an application for [SQLServer](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstatemode.sqlserver.aspx) session mode. It sets the [Timeout](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.timeout.aspx) value to 30 minutes, and specifies that session identifiers are stored in the URL.

<sessionState mode="SQLServer"

 cookieless="true "

 regenerateExpiredSessionId="true "

 timeout="30"

 sqlConnectionString="Data Source=MySqlServer;Integrated Security=SSPI;"

 stateNetworkTimeout="30"/>

You can disable session state for an application by setting the session-state mode to [Off](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.sessionstatemode.off.aspx). If you want to disable session state for only a particular page of an application, you can set the [EnableSessionState](http://msdn.microsoft.com/en-us/library/system.web.configuration.pagessection.enablesessionstate.aspx) value in the @ Page directive to false. The [EnableSessionState](http://msdn.microsoft.com/en-us/library/system.web.configuration.pagessection.enablesessionstate.aspx) value can also be set to ReadOnly to provide read-only access to session variables.

##### Concurrent Requests and Session State

Access to ASP.NET session state is exclusive per session, which means that if two different users make concurrent requests, access to each separate session is granted concurrently. However, if two concurrent requests are made for the same session (by using the same [SessionID](http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx) value), the first request gets exclusive access to the session information. The second request executes only after the first request is finished. (The second session can also get access if the exclusive lock on the information is freed because the first request exceeds the lock time-out.) If the [EnableSessionState](http://msdn.microsoft.com/en-us/library/system.web.configuration.pagessection.enablesessionstate.aspx) value in the @ Page directive is set to ReadOnly, a request for the read-only session information does not result in an exclusive lock on the session data. However, read-only requests for session data might still have to wait for a lock set by a read-write request for session data to clear.

### Cookies

A cookie is a small bit of text that accompanies requests and pages as they go between the Web server and browser. The cookie contains information the Web application can read whenever the user visits the site.

For example, if a user requests a page from your site and your application sends not just a page, but also a cookie containing the date and time, when the user's browser gets the page, the browser also gets the cookie, which it stores in a folder on the user's hard disk.

Later, if user requests a page from your site again, when the user enters the URL the browser looks on the local hard disk for a cookie associated with the URL. If the cookie exists, the browser sends the cookie to your site along with the page request. Your application can then determine the date and time that the user last visited the site. You might use the information to display a message to the user or check an expiration date.

Cookies are associated with a Web site, not with a specific page, so the browser and server will exchange cookie information no matter what page the user requests from your site. As the user visits different sites, each site might send a cookie to the user's browser as well; the browser stores all the cookies separately.

Cookies help Web sites store information about visitors. More generally, cookies are one way of maintaining continuity in a Web application—that is, of performing state management. Except for the brief time when they are actually exchanging information, the browser and Web server are disconnected. Each request a user makes to a Web server is treated independently of any other request. Many times, however, it's useful for the Web server to recognize users when they request a page. For example, the Web server on a shopping site keeps track of individual shoppers so the site can manage shopping carts and other user-specific information. A cookie therefore acts as a kind of calling card, presenting pertinent identification that helps an application know how to proceed.

Cookies are used for many purposes, all relating to helping the Web site remember users. For example, a site conducting a poll might use a cookie simply as a Boolean value to indicate whether a user's browser has already participated in voting so that the user cannot vote twice. A site that asks a user to log on might use a cookie to record that the user already logged on so that the user does not have to keep entering credentials.

##### Cookie Limitations

Most browsers support cookies of up to 4096 bytes. Because of this small limit, cookies are best used to store small amounts of data, or better yet, an identifier such as a user ID. The user ID can then be used to identify the user and read user information from a database or other data store. (See the section "Cookies and Security" below for information about security implications of storing user information.)

Browsers also impose limitations on how many cookies your site can store on the user's computer. Most browsers allow only 20 cookies per site; if you try to store more, the oldest cookies are discarded. Some browsers also put an absolute limit, usually 300, on the number of cookies they will accept from all sites combined.

A cookie limitation that you might encounter is that users can set their browser to refuse cookies. If you define a P3P privacy policy and place it in the root of your Web site, more browsers will accept cookies from your site. However, you might have to avoid cookies altogether and use a different mechanism to store user-specific information. A common method for storing user information is session state, but session state depends on cookies, as explained later in the section "Cookies and Session State."

Although cookies can be very useful in your application, the application should not depend on being able to store cookies. Do not use cookies to support critical features. If your application must rely on cookies, you can test to see whether the browser will accept cookies. See the "Checking Whether a Browser Accepts Cookies" section later in this topic.

##### Writing Cookies

The browser is responsible for managing cookies on a user system. Cookies are sent to the browser via the [HttpResponse](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.aspx) object that exposes a collection called [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx). You can access the [HttpResponse](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.aspx) object as the [Response](http://msdn.microsoft.com/en-us/library/system.web.ui.page.response.aspx) property of your [Page](http://msdn.microsoft.com/en-us/library/system.web.ui.page.aspx) class. Any cookies that you want to send to the browser must be added to this collection. When creating a cookie, you specify a [Name](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.name.aspx) and [Value](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.value.aspx). Each cookie must have a unique name so that it can be identified later when reading it from the browser. Because cookies are stored by name, naming two cookies the same will cause one to be overwritten.

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

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

You can add cookies to the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) collection in a number of ways. The following example shows two methods to write cookies:

C#

Response.Cookies["userName"].Value = "patrick";

Response.Cookies["userName"].Expires = DateTime.Now.AddDays(1);

HttpCookie aCookie = new HttpCookie("lastVisit");

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

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

Response.Cookies.Add(aCookie);

The example adds two cookies to the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) collection, one named userName and the other named lastVisit. For the first cookie, the values of the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) collection are set directly. You can add values to the collection this way because [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) derives from a specialized collection of type [NameObjectCollectionBase](http://msdn.microsoft.com/en-us/library/system.collections.specialized.nameobjectcollectionbase.aspx).

For the second cookie, the code creates an instance of an object of type [HttpCookie](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.aspx), sets its properties, and then adds it to the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) collection via the [Add](http://msdn.microsoft.com/en-us/library/system.web.httpcookiecollection.add.aspx) method. When you instantiate an [HttpCookie](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.aspx) object, you must pass the cookie name as part of the constructor.

Both examples accomplish the same task, writing a cookie to the browser. In both methods, the expiration value must be of type [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx). However, the lastVisited value is also a date-time value. Because all cookie values are stored as strings, the date-time value has to be converted to a [String](http://msdn.microsoft.com/en-us/library/system.string.aspx) .

##### Cookies with More Than One Value

You can store one value in a cookie, such as user name and last visit. You can also store multiple name-value pairs in a single cookie. The name-value pairs are referred to as subkeys. (Subkeys are laid out much like a query string in a URL.) For example, instead of creating two separate cookies named userName and lastVisit, you can create a single cookie named userInfo that has the subkeys userName and lastVisit.

You might use subkeys for several reasons. First, it is convenient to put related or similar information into a single cookie. In addition, because all the information is in a single cookie, cookie attributes such as expiration apply to all the information. (Conversely, if you want to assign different expiration dates to different types of information, you should store the information in separate cookies.)

A cookie with subkeys also helps you limit the size of cookie files. As noted earlier in the "Cookie Limitations" section, cookies are usually limited to 4096 bytes and you can't store more than 20 cookies per site. By using a single cookie with subkeys, you use fewer of those 20 cookies that your site is allotted. In addition, a single cookie takes up about 50 characters for overhead (expiration information, and so on), plus the length of the value that you store in it, all of which counts toward the 4096-byte limit. If you store five subkeys instead of five separate cookies, you save the overhead of the separate cookies and can save around 200 bytes.

To create a cookie with subkeys, you can use a variation of the syntax for writing a single cookie. The following example shows two ways to write the same cookie, each with two subkeys:

C#

Response.Cookies["userInfo"]["userName"] = "patrick";

Response.Cookies["userInfo"]["lastVisit"] = DateTime.Now.ToString();

Response.Cookies["userInfo"].Expires = DateTime.Now.AddDays(1);

HttpCookie aCookie = new HttpCookie("userInfo");

aCookie.Values["userName"] = "patrick";

aCookie.Values["lastVisit"] = DateTime.Now.ToString();

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

Response.Cookies.Add(aCookie);

##### Controlling Cookie Scope

By default, all cookies for a site are stored together on the client, and all cookies are sent to the server with any request to that site. In other words, every page in a site gets all of the cookies for that site. However, you can set the scope of cookies in two ways:

* Limit the scope of cookies to a folder on the server, which allows you to limit cookies to an application on the site.
* Set scope to a domain, which allows you to specify which subdomains in a domain can access a cookie.

##### Limiting Cookies to a Folder or Application

To limit cookies to a folder on the server, set the cookie's [Path](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.path.aspx) property, as in the following example:

C#

HttpCookie appCookie = new HttpCookie("AppCookie");

appCookie.Value = "written " + DateTime.Now.ToString();

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

appCookie.Path = "/Application1";

Response.Cookies.Add(appCookie);

The path can either be a physical path under the site root or a virtual root. The effect will be that the cookie is available only to pages in the Application1 folder or virtual root. For example, if your site is called www.contoso.com, the cookie created in the previous example will be available to pages with the path http://www.contoso.com/Application1/ and to any pages beneath that folder. However, the cookie will not be available to pages in other applications such as http://www.contoso.com/Application2/ or just http://www.contoso.com/.

##### Limiting Cookie Domain Scope

By default, cookies are associated with a specific domain. For example, if your site is www.contoso.com, the cookies you write are sent to the server when users request any page from that site. (This might not include cookies with a specific path value.) If your site has subdomains—for example, contoso.com, sales.contoso.com, and support.contoso.com—then you can associate cookies with a specific subdomain. To do so, set the cookie's [Domain](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.domain.aspx) property, as in this example:

C#

Response.Cookies["domain"].Value = DateTime.Now.ToString();

Response.Cookies["domain"].Expires = DateTime.Now.AddDays(1);

Response.Cookies["domain"].Domain = "support.contoso.com";

When the domain is set in this way, the cookie will be available only to pages in the specified subdomain. You can also use the [Domain](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.domain.aspx) property to create a cookie that can be shared among multiple subdomains, as shown in the following example:

C#

Response.Cookies["domain"].Value = DateTime.Now.ToString();

Response.Cookies["domain"].Expires = DateTime.Now.AddDays(1);

Response.Cookies["domain"].Domain = "contoso.com";

The cookie will then be available to the primary domain as well as to sales.contoso.com and support.contoso.com domains.

##### Reading Cookies

When a browser makes a request to the server, it sends the cookies for that server along with the request. In your ASP.NET applications, you can read the cookies using the [HttpRequest](http://msdn.microsoft.com/en-us/library/system.web.httprequest.aspx) object, which is available as the [Request](http://msdn.microsoft.com/en-us/library/system.web.ui.page.request.aspx) property of your [Page](http://msdn.microsoft.com/en-us/library/system.web.ui.page.aspx) class. The structure of the [HttpRequest](http://msdn.microsoft.com/en-us/library/system.web.httprequest.aspx) object is essentially the same as that of the [HttpResponse](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.aspx) object, so you can read cookies out of the [HttpRequest](http://msdn.microsoft.com/en-us/library/system.web.httprequest.aspx) object much the same way you wrote cookies into the [HttpResponse](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.aspx) object. The following code example shows two ways to get the value of a cookie named username and display its value in a [Label](http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.label.aspx) control:

C#

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

 Label1.Text = Server.HtmlEncode(Request.Cookies["userName"].Value);

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

{

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

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

}

Before trying to get the value of a cookie, you should make sure that the cookie exists; if the cookie does not exist, you will get a [NullReferenceException](http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx) exception. Notice also that the [HtmlEncode](http://msdn.microsoft.com/en-us/library/system.web.httpserverutility.htmlencode.aspx) method was called to encode the contents of a cookie before displaying it in the page. This makes certain that a malicious user has not added executable script into the cookie. For more about cookie security, see the "Cookies and Security" section.

Reading the value of a subkey in a cookie is likewise similar to setting it. The following code example shows one way to get the value of a subkey:

C#

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

{

 Label1.Text =

 Server.HtmlEncode(Request.Cookies["userInfo"]["userName"]);

 Label2.Text =

 Server.HtmlEncode(Request.Cookies["userInfo"]["lastVisit"]);

}

In the preceding example, the code reads the value of the subkey lastVisit, which was set earlier to the string representation of a DateTime value. Cookies store values as strings, so if you want to use the lastVisit value as a date, you have to convert it to the appropriate type, as in this example:

C#

DateTime dt;

dt = DateTime.Parse(Request.Cookies["userInfo"]["lastVisit"]);

The subkeys in a cookie are typed as a collection of type [NameValueCollection](http://msdn.microsoft.com/en-us/library/system.collections.specialized.namevaluecollection.aspx). Therefore, another way to get an individual subkey is to get the subkeys collection and then extract the subkey value by name, as shown in the following example:

C#

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

{

 System.Collections.Specialized.NameValueCollection

 UserInfoCookieCollection;

 UserInfoCookieCollection = Request.Cookies["userInfo"].Values;

 Label1.Text =

 Server.HtmlEncode(UserInfoCookieCollection["userName"]);

 Label2.Text =

 Server.HtmlEncode(UserInfoCookieCollection["lastVisit"]);

}

##### Changing a Cookie's Expiration Date

The browser is responsible for managing cookies, and the cookie's expiration time and date help the browser manage its store of cookies. Therefore, although you can read the name and value of a cookie, you cannot read the cookie's expiration date and time. When the browser sends cookie information to the server, the browser does not include the expiration information. (The cookie's [Expires](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.expires.aspx) property always returns a date-time value of zero.) If you are concerned about the expiration date of a cookie, you must reset it, which is covered in the "Modifying and Deleting Cookies" section.

##### Reading Cookie Collections

You might occasionally need to read through all the cookies available to the page. To read the names and values of all the cookies available to the page, you can loop through the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httprequest.cookies.aspx) collection using code such as the following.

C#

System.Text.StringBuilder output = new System.Text.StringBuilder();

HttpCookie aCookie;

for(int i=0; i<Request.Cookies.Count; i++)

{

 aCookie = Request.Cookies[i];

 output.Append("Cookie name = " + Server.HtmlEncode(aCookie.Name)

 + "<br />");

 output.Append("Cookie value = " + Server.HtmlEncode(aCookie.Value)

 + "<br /><br />");

}

Label1.Text = output.ToString();

A limitation of the preceding example is that if the cookie has subkeys, the display shows the subkeys as a single name/value string. You can read a cookie's [HasKeys](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.haskeys.aspx) property to determine whether the cookie has subkeys. If so, you can read the subkey collection to get individual subkey names and values. You can read subkey values from the Values collection directly by index value. The corresponding subkey names are available in the [AllKeys](http://msdn.microsoft.com/en-us/library/system.collections.specialized.namevaluecollection.allkeys.aspx) member of the [Values](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.values.aspx) collection, which returns an array of strings. You can also use the [Keys](http://msdn.microsoft.com/en-us/library/system.collections.specialized.nameobjectcollectionbase.keys.aspx) member of the [Values](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.values.aspx) collection. However, the [AllKeys](http://msdn.microsoft.com/en-us/library/system.web.httpcookiecollection.allkeys.aspx) property is cached the first time it is accessed. In contrast, the [Keys](http://msdn.microsoft.com/en-us/library/system.collections.specialized.nameobjectcollectionbase.keys.aspx) property builds an array each time it is accessed. For this reason, the [AllKeys](http://msdn.microsoft.com/en-us/library/system.web.httpcookiecollection.allkeys.aspx) property is much faster on subsequent accesses within the context of the same page request.

The following example shows a modification of the preceding example. It uses the [HasKeys](http://msdn.microsoft.com/en-us/library/system.collections.specialized.namevaluecollection.haskeys.aspx) property to test for subkeys, and if subkeys are detected, the example gets subkeys from the Values collection:

C#

for(int i=0; i<Request.Cookies.Count; i++)

{

 aCookie = Request.Cookies[i];

 output.Append("Name = " + aCookie.Name + "<br />");

 if(aCookie.HasKeys)

 {

 for(int j=0; j<aCookie.Values.Count; j++)

 {

 subkeyName = Server.HtmlEncode(aCookie.Values.AllKeys[j]);

 subkeyValue = Server.HtmlEncode(aCookie.Values[j]);

 output.Append("Subkey name = " + subkeyName + "<br />");

 output.Append("Subkey value = " + subkeyValue +

 "<br /><br />");

 }

 }

 else

 {

 output.Append("Value = " + Server.HtmlEncode(aCookie.Value) +

 "<br /><br />");

 }

}

Label1.Text = output.ToString();

Alternatively, you can extract the subkeys as a [NameValueCollection](http://msdn.microsoft.com/en-us/library/system.collections.specialized.namevaluecollection.aspx) object as shown in the following example:

C#

System.Text.StringBuilder output = new System.Text.StringBuilder();

HttpCookie aCookie;

string subkeyName;

string subkeyValue;

for (int i = 0; i < Request.Cookies.Count; i++)

{

 aCookie = Request.Cookies[i];

 output.Append("Name = " + aCookie.Name + "<br />");

 if (aCookie.HasKeys)

 {

 System.Collections.Specialized.NameValueCollection CookieValues =

 aCookie.Values;

 string[] CookieValueNames = CookieValues.AllKeys;

 for (int j = 0; j < CookieValues.Count; j++)

 {

 subkeyName = Server.HtmlEncode(CookieValueNames[j]);

 subkeyValue = Server.HtmlEncode(CookieValues[j]);

 output.Append("Subkey name = " + subkeyName + "<br />");

 output.Append("Subkey value = " + subkeyValue +

 "<br /><br />");

 }

 }

 else

 {

 output.Append("Value = " + Server.HtmlEncode(aCookie.Value) +

 "<br /><br />");

 }

}

Label1.Text = output.ToString();

##### Modifying and Deleting Cookies

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

C#

int counter;

if (Request.Cookies["counter"] == null)

 counter = 0;

else

{

 counter = int.Parse(Request.Cookies["counter"].Value);

}

counter++;

Response.Cookies["counter"].Value = counter.ToString();

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

##### Deleting Cookies

Deleting a cookie—physically removing it from the user's hard disk—is a variation on modifying it. You cannot directly remove a cookie because the cookie is on the user's computer. However, you can have the browser delete the cookie for you. The technique is to create a new cookie with the same name as the cookie to be deleted, but to set the cookie's expiration to a date earlier than today. When the browser checks the cookie's expiration, the browser will discard the now-outdated cookie. The following code example shows one way to delete all the cookies available to the application:

C#

HttpCookie aCookie;

string cookieName;

int limit = Request.Cookies.Count;

for (int i=0; i<limit; i++)

{

 cookieName = Request.Cookies[i].Name;

 aCookie = new HttpCookie(cookieName);

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

 Response.Cookies.Add(aCookie);

}

##### Modifying or Deleting Subkeys

Modifying an individual subkey is the same as creating it, as shown in the following example:

C#

Response.Cookies["userInfo"]["lastVisit"] = DateTime.Now.ToString();

Response.Cookies["userInfo"].Expires = DateTime.Now.AddDays(1);

To delete an individual subkey, you manipulate the cookie's [Values](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.values.aspx) collection, which holds the subkeys. You first recreate the cookie by getting it from the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httprequest.cookies.aspx) object. You can then call the [Remove](http://msdn.microsoft.com/en-us/library/system.web.httpcookiecollection.remove.aspx) method of the [Values](http://msdn.microsoft.com/en-us/library/system.web.httpcookie.values.aspx) collection, passing to the Remove method the name of the subkey to delete. You then add the cookie to the [Cookies](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cookies.aspx) collection so it will be sent in its modified form back to the browser. The following code example shows how to delete a subkey. In the sample, the name of the subkey to remove is specified in a variable.

C#

string subkeyName;

subkeyName = "userName";

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

aCookie.Values.Remove(subkeyName);

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

Response.Cookies.Add(aCookie);

##### Cookies and Security

The security issues with cookies are similar to those of getting data from the client. In your application, cookies are another form of user input and are therefore subject to examining and spoofing. A user can as a minimum see the data that you store in a cookie, since the cookie is available on the user's own computer. The user can also change the cookie before the browser sends it to you.

You should never store sensitive data in a cookie, such as user names, passwords, credit card numbers, and so on. Do not put anything in a cookie that should not be in the hands of a user or of someone who might somehow steal the cookie.

Similarly, be suspicious of information you get out of a cookie. Do not assume that the data is the same as when you wrote it out; use the same safeguards in working with cookie values that you would with data that a user has typed into a Web page. The examples earlier in this topic showed HTML-encoding the contents of a cookie before displaying the value in a page, as you would before displaying any information you get from users.

Cookies are sent between browser and server as plain text, and anyone who can intercept your Web traffic can read the cookie. You can set a cookie property that causes the cookie to be transmitted only if the connection uses the Secure Sockets Layer (SSL). SSL does not protect the cookie from being read or manipulated while it is on the user's computer, but it does prevent the cookie from being read while in transit because the cookie is encrypted.

##### Determining Whether a Browser Accepts Cookies

Users can set their browser to refuse cookies. No error is raised if a cookie cannot be written. The browser likewise does not send any information to the server about its current cookie settings.

One way to determine whether cookies are accepted is by trying to write a cookie and then trying to read it back again. If you cannot read the cookie you wrote, you assume that cookies are turned off in the browser.

The following code example shows how you might test whether cookies are accepted. The sample consists of two pages. The first page writes out a cookie, and then redirects the browser to the second page. The second page tries to read the cookie. It in turn redirects the browser back to the first page, adding to the URL a query string variable with the results of the test.

The code for the first page looks like this:

C#

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

{

 if (!Page.IsPostBack)

 {

 if (Request.QueryString["AcceptsCookies"] == null)

 {

 Response.Cookies["TestCookie"].Value = "ok";

 Response.Cookies["TestCookie"].Expires =

 DateTime.Now.AddMinutes(1);

 Response.Redirect("TestForCookies.aspx?redirect=" +

 Server.UrlEncode(Request.Url.ToString()));

 }

 else

 {

 Label1.Text = "Accept cookies = " +

 Server.UrlEncode(

 Request.QueryString["AcceptsCookies"]);

 }

 }

}

The page first tests to see if this is a postback, and if not, the page looks for the query string variable name AcceptsCookies that contains the test results. If there is no query string variable, the test has not been completed, so the code writes out a cookie named TestCookie. After writing out the cookie, the sample calls [Redirect](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.redirect.aspx) to transfer to the test page TestForCookies.aspx. Appended to the URL of the test page is a query string variable named redirect containing the URL of the current page; this will allow you to redirect back to this page after performing the test.

The test page can consist entirely of code; it does not need to contain controls. The following code example illustrates the test page.

C#

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

{

 string redirect = Request.QueryString["redirect"];

 string acceptsCookies;

 if(Request.Cookies["TestCookie"] ==null)

 acceptsCookies = "no";

 else

 {

 acceptsCookies = "yes";

 // Delete test cookie.

 Response.Cookies["TestCookie"].Expires =

 DateTime.Now.AddDays(-1);

 }

 Response.Redirect(redirect + "?AcceptsCookies=" + acceptsCookies,

 true);

}

After reading the redirect query string variable, the code tries to read the cookie. For housekeeping purposes, if the cookie exists, it is immediately deleted. When the test is finished, the code constructs a new URL from the URL passed to it in the redirect query string variable. The new URL also includes a query string variable containing test results. The final step is to use the new URL to redirect the browser to the original page.

An improvement in the example would be to keep the cookie test results in a persistent store such as a database so that the test does not have to be repeated each time the user views the original page. (Storing the test results in session state by default requires cookies.)

##### Cookies and Session State

When a user navigates to your site, the server establishes a unique session for that user that lasts for the duration of the user's visit. For each session, ASP.NET maintains session state information where applications can store user-specific information.ASP.NET must track a session ID for each user so that it can map the user to session state information on the server. By default, ASP.NET uses a non-persistent cookie to store the session state. However, if a user has disabled cookies on the browser, session state information cannot be stored in a cookie.

ASP.NET offers an alternative in the form of cookieless sessions. You can configure your application to store session IDs not in a cookie, but in the URLs of pages in your site. If your application relies on session state, you might consider configuring it to use cookieless sessions. However, under some limited circumstances, if the user shares the URL with someone else—perhaps to send the URL to a colleague while the user's session is still active—then both users can end up sharing the same session, with unpredictable results.

###   ViewState Overview

A Web application is stateless. A new instance of the Web page class is created every time that the page is requested from the server. This would ordinarily mean that all information in the page and in its controls would be lost with each round trip. For example, by default if a user enters information into a text box on an HTML Web page, that information is sent to the server. However, it is not returned to the browser in the response.

To overcome this intrinsic limitation of Web programming, the ASP.NET page framework includes several state-management features to preserve page and control values between round trips to the Web server. One of these features is view state[ASP.NET State Management Overview](http://msdn.microsoft.com/en-us/library/75x4ha6s.aspx).

By default, the ASP.NET page framework uses view state to preserve page and control values between round trips. When the HTML for the page is rendered, the current state of the page and values that must be retained during postback are serialized into base64-encoded strings. They are then put into a hidden field or fields in the page.

You can access view state in your code by using the page's [ViewState](http://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate.aspx) property. The [ViewState](http://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate.aspx) property is a dictionary that contains key/value pairs that contain the view-state data.

You can change the default behavior and store view state in another location such as a SQL Server database by implementing a custom [PageStatePersister](http://msdn.microsoft.com/en-us/library/system.web.ui.pagestatepersister.aspx) class to store page data. For an example of how to store page state in a stream instead of in a hidden field, see the example for the [PageStatePersister](http://msdn.microsoft.com/en-us/library/system.web.ui.pagestatepersister.aspx) class.

##### Considerations for Using View State

View state provides state information for a specific ASP.NET page. If you need to use information on more than one page, or if you need the information to persist across visits to the Web site, you must use another method for maintaining state. You can use application state, session state, or profile properties.

View state information is serialized into XML and then encoded by using base-64 encoding, which can generate large amounts of data. When the page is posted to the server, the contents of view state are sent as part of the page postback information. If view state contains a large amount of information, it can affect performance of the page. Test the performance of your pages by using typical data for your application to determine whether the size of view state is causing performance problems. For alternatives to using view state, see [ASP.NET State Management Recommendations](http://msdn.microsoft.com/en-us/library/z1hkazw7.aspx).

If you do not have to store control information for individual controls, you can disable view state for a control. If a control on a page is refreshed from the data store on each postback, you can turn view state off for that control in order to reduce the size of view state. For example, you might turn view state off for a control such as the [GridView](http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.gridview.aspx) control.

Another consideration is that if the amount of data in a hidden field becomes large, some proxies and firewalls will prevent access to the page that contains them. Because the maximum allowed amount can vary with different firewall and proxy implementations, large hidden fields can cause intermittent problems. If the amount of data that is stored in the [ViewState](http://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate.aspx) property exceeds the value specified in the page's [MaxPageStateFieldLength](http://msdn.microsoft.com/en-us/library/system.web.ui.page.maxpagestatefieldlength.aspx) property, the page splits view state into multiple hidden fields. This reduces the size of individual hidden fields below the size that firewalls reject.

Some mobile devices do not allow hidden fields at all. Therefore, view state will not work for those devices. For more information and alternatives, see [ASP.NET Mobile Web Development Overview](http://msdn.microsoft.com/en-us/library/ms178619.aspx).

##### 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.

##### Saving Values in View State

You can access view-state information by using the page's [ViewState](http://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate.aspx) property, which exposes a dictionary object. You can use this dictionary to store custom values. A typical use is to store the value of custom properties that you define in the page.

Because view state is sent as a hidden field, you can make changes to view state until the page's [PreRenderComplete](http://msdn.microsoft.com/en-us/library/system.web.ui.page.prerendercomplete.aspx) event. After the page is rendered to the browser, changes to view state will not be saved.

The information in the hidden view-state field can be seen by users if they view the source of the Web page and can decode base-64 encoded strings. This creates a potential security issue. For more information about security issues with view state, see [Securing View State](http://msdn.microsoft.com/en-us/library/bb386448.aspx#SecuringViewState) later in this topic.

To save a value to view state, create a new item that contains the value to save and add the item to the view-state dictionary. The following example shows an ASP.NET Web page with code that saves a string and an integer value in view state.

<%@ Page Language="C#" %>

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

<script runat="server">

 // Sample ArrayList for the page.

 ArrayList PageArrayList;

 ArrayList CreateArray()

 {

 // Create a sample ArrayList.

 ArrayList result = new ArrayList(4);

 result.Add("item 1");

 result.Add("item 2");

 result.Add("item 3");

 result.Add("item 4");

 return result;

 }

 void Page\_Load(object sender, EventArgs e)

 {

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

 {

 PageArrayList = (ArrayList)ViewState["arrayListInViewState"];

 }

 else

 {

 // ArrayList isn't in view state, so it must be created and populated.

 PageArrayList = CreateArray();

 }

 // Code that uses PageArrayList.

 }

 void Page\_PreRender(object sender, EventArgs e)

 {

 // Save PageArrayList before the page is rendered.

 ViewState.Add("arrayListInViewState", PageArrayList);

 }

</script>

<html >

<head runat="server">

 <title>View state sample</title>

</head>

<body>

 <form id="form1" runat="server">

 <div>

 </div>

 </form>

</body>

</html>

##### Data Types You Can Store in View State

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

* Strings
* Integers
* Boolean values
* Array objects
* ArrayList objects
* Hash tables
* Custom type converters (see the [TypeConverter](http://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.aspx) class for more information)

You can store other types of data also, but the class must be compiled with the [Serializable](http://msdn.microsoft.com/en-us/library/system.reflection.typeattributes.serializable.aspx) attribute so that its values can be serialized for view state.

##### Reading Values from View State

To read a value from view state, you get the [ViewState](http://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate.aspx) property of page and then read the value from the view-state dictionary.

The following example shows how you can get an [ArrayList](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) object named arrayListInViewState from view state and then bind a [GridView](http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.gridview.aspx) control to the object as a data source.

C#

if (ViewState["color"] == null)

 // No such value in view state, take appropriate action.

If you try to use a nonexistent view-state entry in some other way (for example, to examine its type), a [NullReferenceException](http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx) exception is thrown.

[Back to top](http://msdn.microsoft.com/en-us/library/bb386448.aspx#Introduction)

##### Securing View State

By default, view-state data is stored in the page in a hidden field and is encoded using base64 encoding. In addition, a hash of the view-state data is created from the data by using a machine authentication code (MAC) key. The hash value is added to the encoded view-state data and the resulting string is stored in the page. When the page is posted back to the server, the ASP.NET page framework re-computes the hash value and compares it with the value stored in view state. If the hash values do not match, an exception is raised that indicates that view-state data might be invalid.

By creating a hash value, the ASP.NET page framework can test whether the view-state data has been corrupted or tampered with. However, even if it is not tampered with, view-state data can still be intercepted and read by malicious users.

##### Usign the MAC for Computing the View-State Hash Value

The MAC key that is used to compute the view-state hash value is either auto-generated or specified in the Machine.config file. If the key is auto-generated, it is created based on the MAC address of the computer, which is the unique GUID value of the network adapter in that computer.

It can be difficult for malicious users to reverse-engineer the MAC key based on the hash value in view state. Thus, MAC encoding is a reasonably reliable way to determine whether view-state data has been changed.

In general, the larger the MAC key that is used to generate the hash, the less likely it is that the hash value for different strings will be the same. When the key is auto-generated, ASP.NET uses SHA-1 encoding to create a large key. However, in a Web-farm environment, the key must be the same across all the servers. If the key is not the same, and the page is posted back to a different server than the one that created the page, the ASP.NET page framework will raise an exception. Therefore, in a Web farm environment, you should specify a key in the Machine.config file instead of letting ASP.NET auto-generate one. In that case, make sure that you create a key that is long enough to offer sufficient security for the hashed value. However, he longer the key is, the more time it takes to create a hash. Therefore, you must weigh security needs versus performance needs.

##### Encrypting View State

Although MAC encoding helps prevent tampering with view-state data, it does not prevent users from viewing the data. You can prevent people from viewing this data in two ways: by transmitting the page over SSL, and by encrypting the view-state data. Requiring the page to be sent over SSL can help prevent data-packet sniffing and unauthorized data access by people who are not the intended recipients of the page.

However, the user who requested the page can still view the view-state data because SSL decrypts the page to display it in the browser. This is fine if you are not concerned about authorized users having access to view-state data. However, in some cases, controls might use view state to store information that no users should have access to. For example, the page might contain a data-bound control that stores item identifiers (data keys) in view state. If those identifiers contain sensitive data, such as customer IDs, you should encrypt the view-state data in addition to or instead of sending the page over SSL.

To encrypt the data, set the page's [ViewStateEncryptionMode](http://msdn.microsoft.com/en-us/library/system.web.ui.page.viewstateencryptionmode.aspx) property to true. If you store information in view state, you can use regular read and write techniques; the page handles all encryption and decryption for you. Encrypting view-state data can affect the performance of your application. Therefore, do not use encryption unless you need it.

##### Control State Encryption

Controls that use control state can require that view state be encrypted by calling the [RegisterRequiresViewStateEncryption](http://msdn.microsoft.com/en-us/library/system.web.ui.page.registerrequiresviewstateencryption.aspx) method. If any control in the page requires that view state be encrypted, all view state in the page will be encrypted.

##### Per-user View-state Encoding

If a Web site authenticates users, you can set the [ViewStateUserKey](http://msdn.microsoft.com/en-us/library/system.web.ui.page.viewstateuserkey.aspx) property in the Page\_Init event handler to associate the page's view state with a specific user. This helps prevent one-click attacks, in which a malicious user creates a valid, pre-filled Web page with view state from a previously created page. The attacker then lures a victim into clicking a link that sends the page to the server by using the victim's identity.

When the [ViewStateUserKey](http://msdn.microsoft.com/en-us/library/system.web.ui.page.viewstateuserkey.aspx) property is set, the attacker's identity is used to create the hash of the view state of the original page. When the victim is lured into resending the page, the hash values will be different because the user keys are different. The page will fail verification and an exception will be thrown.

You must associate the [ViewStateUserKey](http://msdn.microsoft.com/en-us/library/system.web.ui.page.viewstateuserkey.aspx) property with a unique value for each user, such as the user name or identifier.

##### Securing Configuration in Shared Hosting Environment

In a shared hosting environment, malicious users can potentially modify state-management properties that might affect other applications on the computer. This can be done through direct modification of the Machine.config file, modification by way of the configuration classes, and other administration and configuration tools. You can help prevent modification to your application configuration by encrypting sections of configuration files.

###     Caching

The ASP.NET cache is a general-purpose cache facility for Web applications. It provides both a simple interface for caching and a more advanced interface that exposes expiration and change dependency services.

Caching is an extremely important technique for building high-performance, scalable server applications. Some items that are expensive to construct can be built once and then used for some amount of time before they are considered invalid. These items are stored in memory where they can be efficiently retrieved and used without incurring the cost of reconstructing them.

One of the most important factors in building high-performance, scalable Web applications is the ability to store items, whether data objects, pages, or parts of a page, in memory the initial time they are requested. You can store these items on the Web server or other software in the request stream, such as the proxy server or browser. This allows you to avoid recreating information that satisfied a previous request, particularly information that demands significant processor time or other resources. Known as caching, it allows you to use a number of techniques to store page output or application data across HTTP requests and reuse it. Thus, the server does not have to recreate information, saving time and resources.

ASP.NET provides two types of caching that you can use to create high-performance Web applications. The first is called output caching, which allows you to store dynamic page and user control responses on any HTTP 1.1 cache-capable device in the output stream, from the originating server to the requesting browser. On subsequent requests, the page or user control code is not executed; the cached output is used to satisfy the request. The second type of caching is traditional application data caching, which you can use to programmatically store arbitrary objects, such as data sets, to server memory so that your application can save the time and resources it takes to recreate them.

##### OutputCache

You have your choice of a high-level declarative API or a low-level programmatic API when manipulating the output cache for a page. You can use the former by including the [@ OutputCache](http://msdn.microsoft.com/en-us/library/hdxfb6cy%28VS.71%29.aspx) directive in the .aspx file for the page. The **@ OutputCache** directive can meet nearly all the common needs you may have when you want to cache a page's output. The following directive, when included in an .aspx file, sets an expiration of 60 seconds for the cached output of a dynamically generated page.

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

**CAUTION**   When you use the **@ OutputCache** directive, the **Duration** and **VaryByParam** attributes are required. If you do not include them, a parser error occurs when the page is first requested. If you do not want to use the functionality that the **VaryByParam** attribute provides, you must set its value to **None**. For more information about using the **VaryByParam** attribute, see [Caching Multiple Versions of a Page](http://msdn.microsoft.com/en-us/library/xadzbzd6%28VS.71%29.aspx).

ASP.NET also includes a set of APIs that control output-cache expirations and policies for a page programmatically through the [HttpCachePolicy](http://msdn.microsoft.com/en-us/library/system.web.httpcachepolicy%28VS.71%29.aspx) class. This class, its methods, and its properties are available through the [HttpResponse.Cache](http://msdn.microsoft.com/en-us/library/system.web.httpresponse.cache%28VS.71%29.aspx) property. In turn, you can access this property from the **Page** object through the [Page.Response](http://msdn.microsoft.com/en-us/library/system.web.ui.page.response%28VS.71%29.aspx) property.

For example, the following code, when included in a page's code-declaration block or its code-behind class, sets an expiration of 60 seconds using the [HttpCachePolicy.SetExpires](http://msdn.microsoft.com/en-us/library/system.web.httpcachepolicy.setexpires%28VS.71%29.aspx) method for the dynamically generated page.

[C#]

Response.Cache.SetExpires(DateTime.Now.AddSeconds(60));

Once you have enabled output caching, the initial HTTP GET request for the page places its dynamic content in the output cache for the amount of time you specify. The output cache satisfies subsequent GET, HEAD, or POST requests for that page until the amount of time you specify expires.

You can enable or disable page output caching for cache-capable devices in the request stream either declaratively or programmatically as well. In the **@ OutputCache** directive for a page you can use the **Location** attribute to specify whether the page's output can be cached on proxy servers, browser clients, the originating Web server, or all or none of these. You can do the same programmatically using the [HttpCachePolicy.SetCacheability](http://msdn.microsoft.com/en-us/library/system.web.httpcachepolicy.setcacheability%28VS.71%29.aspx) method to specify the appropriate [HttpCacheability](http://msdn.microsoft.com/en-us/library/system.web.httpcacheability%28VS.71%29.aspx) enumeration value for your page. For more information, see [Setting the Cacheability of a Page](http://msdn.microsoft.com/en-us/library/w9s3a17d%28VS.71%29.aspx).

Responses generated by GET requests with query string parameters or form POST requests with parameters can also be cached, but caching for the passed parameters must be explicitly enabled using the **@ OutputCache** directive's **VaryByParam** attribute. For more information, see [Caching Multiple Versions of a Page](http://msdn.microsoft.com/en-us/library/xadzbzd6%28VS.71%29.aspx).

Remember that any manipulations that you want to make programmatically to the output cache must be made in the code-declaration block of an .aspx file, or in a code-behind class associated with the .aspx file.

##### Cache object and Dependencies

ASP.NET has a powerful, easy-to-use caching mechanism that allows you to store objects that require a large amount of server resources to create in memory. It is implemented by the [Cache](http://msdn.microsoft.com/en-us/library/system.web.caching.cache%28VS.71%29.aspx) class, with instances private to each application, and its lifetime is tied to that of the application. When the application is restarted, the **Cache** object is recreated.

The **Cache** class has been designed for ease of use. By using keys paired with values, you can place items in the **Cache** and later retrieve them. For examples of how to do this, see [Adding Items to the Cache](http://msdn.microsoft.com/en-us/library/18c1wd61%28VS.71%29.aspx) and [Retrieving Values of Cached Items](http://msdn.microsoft.com/en-us/library/xhy3h9f9%28VS.71%29.aspx).

While the **Cache** class offers a simple interface for you to customize cache settings, it also offers powerful features that allow you to customize how items are cached and how long they are cached. For example, when system memory becomes scarce, the cache automatically removes seldom used or unimportant items to allow memory to be used to process a high volume of requests. This technique is called scavenging. It is one of the ways that the cache ensures that data that is not current does not consume valuable server resources.

You can instruct the **Cache** to give certain items priority over other items when it performs scavenging. To indicate that a specific item is of greater or lesser importance than another, specify one of the [CacheItemPriority](http://msdn.microsoft.com/en-us/library/system.web.caching.cacheitempriority%28VS.71%29.aspx) enumeration values when you add an item using the [Cache.Add](http://msdn.microsoft.com/en-us/library/system.web.caching.cache.add%28VS.71%29.aspx) method or [Cache.Insert](http://msdn.microsoft.com/en-us/library/system.web.caching.cache.insert%28VS.71%29.aspx) method.

You can also establish an expiration policy for an item when you add it to the **Cache** using the **Add** method or **Insert** method. You can define the lifetime for an item by using the *absoluteExpiration* parameter, which is of the type [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime%28VS.71%29.aspx) and allows you to specify the exact time the item will expire. You can also use the *slidingExpiration* parameter, which is of the type [TimeSpan](http://msdn.microsoft.com/en-us/library/system.timespan%28VS.71%29.aspx). It allows you to specify the elapsed time before the item expires based on the time it is accessed. Once the item expires, it is removed from the cache. Attempts to retrieve its value will return **null** unless the item is added to the **Cache** again.

For volatile items that are stored in the **Cache**, such as those that have regular data refreshes, or those that are valid for only a set amount of time, set an expiration policy that keeps those items in the **Cache** as long as their data remains current. For example, if you are writing an application that tracks sports scores by obtaining the data from a frequently updated Web site, you can cache the scores for a game as long as those scores do not change on the source Web site. In this case, you can set an expiration policy that is based on how often the Web site updates the scores. You can write code that determines if an up-to-date score is in the **Cache**. If the score is not up to date, the code can update the score from the source Web site.

Finally, ASP.NET allows you to define the validity of a cached item, based on an external file, a directory, or another cached item. These are called file dependencies and key dependencies. If a dependency changes, the cached item is invalidated and removed from the **Cache**. You can use this technique to remove items from the **Cache** when their data source changes. For example, if you write an application that processes financial data from an XML file and renders it in a graph, you can insert the data from the file in the **Cache** and maintain a dependency on that XML file. When the file is updated, the item is removed from the cache, your application rereads the file, and a new version of the item is inserted.