Nine Options for Managing Persistent User State in Your ASP.NET Application

**SUMMARY**

ASP.NET provides many different ways to persist data between user requests. You can use the Application object, cookies, hidden fields, the Session or Cache objects, and lots of other methods. Deciding when to use each of these can sometimes be difficult. This article will introduce the aforementioned techniques and present some guidelines on when to use them. Although many of these techniques existed in classic ASP, best practices for when to use them have changed with the introduction of the .NET Framework. To persist data in ASP.NET, you'll have to adjust what you learned previously about handling state in ASP.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif

**Contents**

[Application](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S1)  
[Cookies](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S2)  
[Form Post / Hidden Form Field](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S3)  
[QueryString](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S4)  
[Session](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S5)  
[New State Containers in ASP.NET](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S6)  
[Cache](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S7)  
[Context](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S8)  
[ViewState](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S9)  
[Web.config and Machine.config Files](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S10)  
[Conclusion](http://msdn.microsoft.com/en-us/magazine/cc300437.aspx#S11)

Since the dawn of the Web, managing state in the stateless world of HTTP has been a problem for Web developers. More recently, various techniques for storing and retrieving data have emerged. In this article I will describe how ASP.NET developers can maintain or pass state across page requests.

In ASP.NET, there are several different ways to persist data between user requests—so many, in fact, that the novice developer is often confused about which object to use in a particular situation. In order to answer this question, there are three criteria that should be considered:

* Who needs the data?
* How long does the data need to be persisted?
* How large is the set of data?

By answering these questions, you can determine which object provides the best solution for persisting data between requests in an ASP.NET application. **Figure 1** lists the different state management objects and describes when they should be used. In ASP.NET, four new objects have been added: Cache, Context, ViewState, and the Web.Config file. Classic ASP objects that are also available in ASP.NET include Application, Cookie, Form Post with a hidden form field, QueryString, and Session. Note that the proper use of these five data containers has changed, so experienced ASP programmers may have some "unlearning" to do when considering these familiar objects.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 1 Data Container Objects in ASP.NET

|  |  |  |  |
| --- | --- | --- | --- |
| **Persistence Method** | **Who Needs the Data?** | **For How Long?** | **How Much Data?** |
| Application | All users | Until the next application restart | Can be almost any size—it will only be stored once |
| Cookie | One user | As short as desired, or for months or even years if the user doesn't delete their cookies | Minimal, simple data |
| Form Post | One user | For the next request (can be reused across many requests) | Virtually any size—the data is sent back and forth with every page |
| QueryString | One user or one group of users | For the next request (can be reused across many requests) | Minimal, simple data |
| Session | One user | As long as the user is active, plus a timeout period (typically 20 minutes) | Can be almost any size, but should be minimized since every user has their own separate session store |
| Cache | All users or a subset of users | As long or as short as needed | Can be used for large or small, simple or complex data |
| Context | One user | This request only | Can hold large objects, but typically does not since it is often used for every request |  |
| ViewState | One user | One Web form | Minimal; as with Form Post, this data is sent back and forth with every page |  |
| Config file | All users | Until the configuration file is updated | Can hold a lot of data; usually organized as many small strings or XML structures |  |

**Application**

Let's set the object use criteria by answering the state questions I asked earlier. Who needs this data? All users need access to it. How long does this data need to be persisted? It has to live forever, or for the life of the application. How large is this data? It can be almost any size—only one copy of the data will exist at any given time.

In classic ASP, the Application object provided a great place to store frequently used pieces of data that changed infrequently, such as the contents of menus or other reference data. While the Application object is still available as a data container in ASP.NET, other objects are generally better suited for the kinds of data that would have been stored in the Application collection of a classic ASP application.

In classic ASP, the Application object was an ideal choice if the data that was to be stored did not vary at all (or very rarely) for the life of the application (such as read-only or read-mostly data). Connection strings were one of the more common pieces of data stored in Application variables, but in ASP.NET such configuration data is best stored in the Web.config file. One thing to consider if you are using the Application object is that any writes to it should be done either in its Application\_OnStart event (in global.asax) or within an Application.Lock section. While using Application.Lock is necessary to ensure that writes are performed properly, it also serializes requests for the Application object, which can be a serious performance bottleneck for the application. **Figure 2** demonstrates how to use the Application object; it consists of a Web form and its code-behind file. An example of its output is shown in **Figure 3**.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 2 Accessing the Application Object in ASP.NET

Application.aspx

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl40_ctl00_ctl00_code');" \o "Copy Code)

<form id="Application" method="post" runat="server"> <asp:validationsummary id="valSummary" Runat="server"> </asp:validationsummary> <table> <tr> <td colSpan="3">Set Application Variable:</td> </tr> <tr> <td>Name</td> <td><asp:textbox id="txtName" Runat="server"></asp:textbox> </td> <td><asp:requiredfieldvalidator id="nameRequired" runat="server" Display="Dynamic" ErrorMessage="Name is required." ControlToValidate="txtName">\* </asp:requiredfieldvalidator></td> </tr> <tr> <td>Value</td> <td><asp:textbox id="txtValue" Runat="server"> </asp:textbox></td> <td><asp:requiredfieldvalidator id="valueRequired" Runat="server" Display="Dynamic" ErrorMessage="Value is required." ControlToValidate="txtValue">\* </asp:requiredfieldvalidator></td> </tr> <tr> <td colSpan="3"><asp:button id="btnSubmit" Runat="server" Text="Update Value"></asp:button></td> </tr> </table> <asp:Label ID="lblResult" Runat="server" /> </form>

Application.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl40_ctl00_ctl01_code');" \o "Copy Code)

private void btnSubmit\_Click(object sender, System.EventArgs e) { if(IsValid) { Application.Lock(); Application[txtName.Text] = txtValue.Text; Application.UnLock(); lblResult.Text = "The value of <b>" + txtName.Text + "</b> in the Application object is <b>" + Application[txtName.Text].ToString() + "</b>"; } }

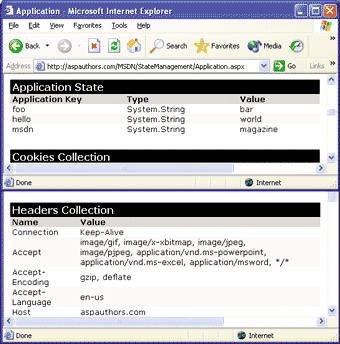


Figure 3**Contents of Application Object**

Note that in **Figure 3** the contents of the Application object are displayed in the trace output. Tracing is a really great debugging tool, but you can expect that at some point, a page with tracing turned on will accidentally make it into your production environment. When that happens, you really won't want anything sensitive to be shown. This is one of the primary reasons why the Application object is no longer the recommended place to store sensitive information like connection strings.

**Cookies**

Cookies are handy when a particular user needs a specific piece of data, and it needs to be persisted for a variable period of time. It can be as brief as the life of the browser window, or as long as months or even years. As far as size goes, cookies are very small. Cookies can be as small as only a few bytes of data, and since they are passed with every browser request, their contents should be kept as small as possible.

[**Editor's Update - 1/11/2005:** The best way to secure sensitive state that should not be viewed or modified by a hostile user is to store that state on the server. If sensitive data must be sent to the client, it should be encrypted beforehand, regardless of the storage mechanism employed.]

A particular named cookie can store a single value or a collection of name/value pairs. **Figure 4** shows an example of both single- and multi-value cookies, as output by the built-in trace features of ASP.NET. These values can be manipulated within an ASP.NET page by using the Request.Cookies and Response.Cookies collections, as **Figure 5** demonstrates.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 5 Accessing Cookies in ASP.NET

Cookies.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl41_ctl00_ctl00_code');" \o "Copy Code)

// Setting a cookie's value and/or subvalue using the HttpCookie class HttpCookie cookie; if(Request.Cookies[txtName.Text] == null) cookie = new HttpCookie(txtName.Text, txtValue.Text); else cookie = Request.Cookies[txtName.Text]; if(txtSubValueName.Text.Length > 0) cookie.Values.Add(txtSubValueName.Text, txtSubValueValue.Text); cookie.Expires = System.DateTime.Now.AddDays(1); // tomorrow Response.AppendCookie(cookie); // Retrieving a cookie's value(s) if(!Request.Cookies[txtName.Text].HasKeys) lblResult.Text = "The value of the <b>" + txtName.Text + "</b> cookie is <b>" + Request.Cookies[txtName.Text].Value.ToString() + "</b>"; else { lblResult.Text = "The value of the <b>" + txtName.Text + "</b> cookie is <b>" + Request.Cookies[txtName.Text].Value.ToString() + "</b>, with subvalues:<br>"; foreach(string key in Request.Cookies[txtName.Text].Values.Keys) { lblResult.Text += "[" + key + " = " + Request.Cookies[txtName.Text].Values[key].ToString() + "]<br>"; } }

Delete a Cookie

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl41_ctl00_ctl01_code');" \o "Copy Code)

// Set the value of the cookie to null and // set its expiration to some time in the past Response.Cookies[txtName.Text].Value = null; Response.Cookies[txtName.Text].Expires = System.DateTime.Now.AddMonths(-1); // last month

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 4 Single-value and Multi-value Cookies

**Form Post / Hidden Form Field**

Form data is needed by a particular user, and it must be persisted for any period from a single request to the life of the application. The data can be virtually any size; it's sent back and forth over the network with each form post.

In classic ASP, this was a common way to retain state within an application, especially in a multi-page form. However, in ASP.NET, this technique is rarely appropriate since Web controls and ViewState handle this automatically as long as you use the postback model (that is, pages that post back to themselves). ViewState is an ASP.NET implementation of this technique, which I will describe later in this article. Access to form values sent through a POST is done using the HttpRequest object's Form collection. In **Figure 6**, a user ID is set by one ASP.NET page, after which it is persisted in a hidden form field. Subsequent requests to either page retain the value as long as the pages use Submit buttons to link to each other.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 6 Using Hidden Form Fields to Persist Data in ASP.NET

Form1.aspx

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl43_ctl00_ctl00_code');" \o "Copy Code)

<h1>Form 1</h1> <form id="Application" method="post" runat="server"> <p>Your username: <asp:Label ID="lblUsername" Runat="server" /> </p> <asp:Panel Runat="server" ID="pnlSetValue"> <asp:validationsummary id="valSummary" Runat="server"> </asp:validationsummary> <TABLE> <TR> <TD colSpan="3">Set Hidden Form Username Variable:</TD></TR> <TR> <TD>Username</TD> <TD> <asp:textbox id="txtName" Runat="server"></asp:textbox></TD> <TD> <asp:requiredfieldvalidator id="nameRequired" runat="server" ControlToValidate="txtName" ErrorMessage="Name is required." Display="Dynamic">\*</asp:requiredfieldvalidator></TD></TR> <TR> <TD colSpan="3"> <asp:button id="btnSubmit" Runat="server" Text="Set Value"> </asp:button></TD></TR></TABLE> </asp:Panel> <asp:Label ID="lblResult" Runat="server" /> </form> <form action="form2.aspx" method="post" name="form2" id="form2"> <input type="hidden" name="username" value="<%# username %>" > <input type="submit" value="Go to Form2.aspx" </form>

Form1.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl43_ctl00_ctl01_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e) { if(!IsPostBack) // new request or request from form2.aspx { // check Forms collection if(Request.Form["username"] == null) pnlSetValue.Visible = true; else { // need to set the username value pnlSetValue.Visible = false; username = Request.Form["username"].ToString(); lblUsername.Text = username; // Databind to set the hidden form field's value this.DataBind(); } } } private void btnSubmit\_Click(object sender, System.EventArgs e) { if(IsValid) { // Hide the form to set the value. pnlSetValue.Visible = false; username = txtName.Text; lblResult.Text = "Username set to " + txtName.Text + "."; lblUsername.Text = username; this.DataBind(); } }

Form2.aspx

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl43_ctl00_ctl02_code');" \o "Copy Code)

<h1>Form 2</h1> <form id="Application" method="post" runat="server"> <p>Your username: <asp:Label ID="lblUsername" Runat="server" /></p> </form> <form action="form1.aspx" method="post" id="form2" name="form2"> <input type="hidden" name="username" value="<%# username %>" > <input type="submit" value="Go to Form1.aspx" </form>

Form2.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl43_ctl00_ctl03_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e) { if(Request.Form["username"] != null) { username = Request.Form["username"].ToString(); lblUsername.Text = username; this.DataBind(); } }

In ASP.NET, only one server-side form can exist on a page, and that form must submit back to itself (client-side forms can still be used, without limitations). One of the major reasons that hidden form fields are no longer used to pass data around applications built on the Microsoft® .NET Framework is that all .NET Framework controls are capable of maintaining their own state automatically using ViewState. ViewState simply encapsulates the work involved in setting and retrieving values using hidden form fields into a simple-to-use collection object.

**QueryString**

The data stored in the QueryString object is used by the individual user. Its lifetime can be as brief as a single request, or as long as the user continues to use the application (if architected appropriately). This data is typically less than 1KB. Data in a QueryString is passed in the URL and is visible to the user, so as you might guess, sensitive data or data that can be used to control the application should be encrypted when using this technique.

That said, the QueryString is a great way to send information between Web forms in ASP.NET. For example, if you have a DataGrid with a list of products, and a hyperlink in the grid that goes to a product detail page, it would be an ideal use of the QueryString to include the product ID in the QueryString of the link to the product details page (for example, productdetails.aspx?id=4). Another advantage of using QueryStrings is that the state of the page is contained in the URL. This means that a user can put a page in their Favorites folder in its generated form when it's created with a QueryString. When they return to it as a favorite, it will be the same as when they actually made it a favorite. Obviously, this only works if the page doesn't rely on any state outside the QueryString and nothing else changes.

Along with sensitive data, any variable that you don't want the user to be able to manipulate should be avoided here (unless encryption is used to remove human-readability). Also, keep in mind that characters that are not valid in a URL must be encoded using Server.UrlEncode, as **Figure 7** shows. When dealing with a single ASP.NET page, ViewState is a better choice than QueryString for maintaining state. For long-term data storage, Cookie, Session, or Cache are more appropriate data containers than QueryStrings.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 7 Using QueryStrings to Pass Data in ASP.NET

Querystring.aspx

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl44_ctl00_ctl00_code');" \o "Copy Code)

<form id="Querystring" method="post" runat="server"> <asp:validationsummary id="valSummary" Runat="server"> </asp:validationsummary> <table> <tr> <td colSpan="3">Set Querystring Variable:</td> </tr> <tr> <td>Name</td> <td><asp:textbox id="txtName" Runat="server"></asp:textbox> </td> <td><asp:requiredfieldvalidator id="nameRequired" runat="server" Display="Dynamic" ErrorMessage="Name is required." ControlToValidate="txtName">\* </asp:requiredfieldvalidator></td> </tr> <tr> <td>Value</td> <td><asp:textbox id="txtValue" Runat="server"> </asp:textbox></td> <td><asp:requiredfieldvalidator id="valueRequired" Runat="server" Display="Dynamic" ErrorMessage="Value is required." ControlToValidate="txtValue">\* </asp:requiredfieldvalidator></td> </tr> <tr> <td colSpan="3"><asp:button id="btnSubmit" Runat="server" Text="Update Value"></asp:button></td> </tr> </table> <asp:Label ID="lblResult" Runat="server" /> <a href="http://querystring.aspx?x=1">Set querystring x equal to 1</a> </form>

Querystring.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl44_ctl00_ctl01_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e) { // Retrieving a cookie's value(s) if(Request.QueryString.HasKeys()) { lblResult.Text = "The values of the <b>" + txtName.Text + "</b> querystring parameter are:<br>"; foreach(string key in Request.QueryString.Keys) { lblResult.Text += "[" + key + " = " + Request.QueryString[key].ToString() + "]<br>"; } } } private void btnSubmit\_Click(object sender, System.EventArgs e) { if(IsValid) { string url = "querystring.aspx?"; foreach(string key in Request.QueryString.Keys) { url += key + "=" + Request.QueryString[key].ToString() + "&"; } Response.Redirect(url + txtName.Text + "=" + Server.UrlEncode(txtValue.Text)); } }

**Session**

Session data is specific to a particular user. It lives for as long as the user continues to makes requests plus some period of time afterward (typically 20 minutes). The Session object can hold large or small amounts of data, but total storage should be kept minimal if the application is intended to scale to hundreds of users or more.

Unfortunately, the Session object earned itself a very bad name in classic ASP because it tied an application to a particular machine, precluding the use of clustering and Web farms for scalability. In ASP.NET, this is less of an issue, since it is a simple matter to change the location where the session is stored. By default (and for best performance), session data is still stored in the memory of the local Web server, but ASP.NET also supports an external state server or database for managing Session data.

Using the Session object is easy and its syntax is identical to classic ASP. However, the Session object is one of the less efficient ways of storing user data, since it is held in memory for some time even after the user has stopped using the application. This can have serious effects on scalability for a very busy site. Other options allow more control over the release of memory, such as the Cache object, which may be better suited for some large data values. Also, ASP.NET sessions rely on cookies by default so if the user disables or doesn't support cookies, sessions won't work. Support for cookie-free sessions can be configured, however. For small amounts of data, the Session object can be a perfectly valid place to store user-specific data that needs to persist only for the duration of the user's current session. The following example demonstrates how to set and retrieve values from the Session object:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl20_code');" \o "Copy Code)

private void btnSubmit\_Click(object sender, System.EventArgs e) { if(IsValid) { // Set the Session value. Session[txtName.Text] = txtValue.Text; // Read and display the value we just set lblResult.Text = "The value of <b>" + txtName.Text + "</b> in the Session object is <b>" + Session[txtName.Text].ToString() + "</b>"; } }

The Web form is almost identical to the one used for the Application object, and the contents of the Session collection are also visible when page tracing is enabled.

You should be aware that even when not in use, sessions carry some overhead for an application. You can squeeze a little bit more performance out of your pages if you turn off sessions on pages that do not use it. Also, setting session state to read-only can also optimize pages that read but do not write data to sessions. Configure sessions in this fashion by adding an attribute to the @Page directive in one of these two ways::

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl21_code');" \o "Copy Code)

<%@ Page EnableSessionState="false" %> <%@ Page EnableSessionState="readonly" %>

ASP.NET sessions can be configured in the Web.config or Machine.config with the sessionState element. This element supports the attributes listed in **Figure 8**.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 8 sessionState Attributes

| **Attribute** | **Options** | **Description** |
| --- | --- | --- |
| mode | Off | Disables sessions. |
| Inproc | Same as classic ASP. Stored in the Web server's local memory. This option provides the best performance, but it is not clusterable. This is the default option. |
| StateServer | Session data is stored in memory on another server. |
| SqlServer | Session data is stored in a SQL Server database. |
| cookieless | True | Enables cookieless sessions. Session ID is automatically passed in the QueryString instead of in a cookie for all relative URLs in the application. |
| False | This is the default setting. Sessions use cookies. |
| timeout | | Session timeout in minutes. The default is 20. |
| sqlConnectionString | | Connection string used for SqlServer mode sessions. |
| stateConnectionString | | Connection string used for StateServer mode sessions. |

Here is an example of the settings in the Web.config:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl23_code');" \o "Copy Code)

<sessionState timeout="10" cookieless="false" mode="Inproc" />

**New State Containers in ASP.NET**

As mentioned earlier, ASP.NET adds several new ways to store data during and between user requests. This gives you much finer control over how state information is persisted. These new techniques narrow the scope down to as small as a single request (the Context object), or widen the scope to as large as the whole Web server and all applications on that server (the Machine.config file). In many cases, you will have several options available to you when you need to store a particular piece of data—use the questions and answers provided with each method description to determine if it is appropriate for your situation.

**Cache**

Cache data is specific to the single user, a subset of users, or even all users. This data persists for multiple requests. It can persist for a long time, but not across application restarts. Also, data can expire based on time or other dependencies. It can hold both large and small amounts of data effectively.

The Cache is one of the coolest objects in all of ASP.NET. It offers incredible flexibility, versatility, and performance, and is therefore often a better choice than Application or Session for persisting data within an ASP.NET application. A complete description of the ways in which the Cache object can be used (both declaratively and programmatically) is beyond the scope of this article, but suffice to say, it's a versatile object. Like the other collection objects, it is simply a name-value collection, but by using a key value that is specific to a user, you can make the cached values user-specific. Similarly, you can cache multiple sets of data for different related data, like several sets of car data with keys like "fordcars", "chevycars", and "gmcars". Data in the Cache can be given an expiration period that is absolute, sliding, or based on changes to a file. They also implement a callback function that is invoked whenever the cached value is ejected from the cache, which is useful because you can then check to see if there is a more recent version of the data available, and if not (or if the data source is unavailable), re-cache the value that was just expired.

Adding and accessing data in the cache is done using a syntax similar to what I have already covered. However, in addition to the standard indexer method of accessing this collection's contents, Cache also supports a number of methods to allow more control over the data that is cached. The method you will most often use is Insert, which supports several overloads that allow you to specify dependencies, timeouts, priority, and callbacks. Some simple examples are shown in the following code:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl24_code');" \o "Copy Code)

// Add item to cache Cache["myKey"] = myValue; // Read item from cache Response.Write(Cache["myKey"]); // Set a CacheDuration of 10 seconds and add item to cache Cache.Insert("myKey",myValue, null, System.DateTime.Now.AddSeconds(10), System.Web.Caching.Cache.NoSlidingExpiration);

One of the more powerful features of the Cache object is its ability to execute a callback when an item in the cache expires. This uses delegates or function pointers, a fairly advanced topic that I won't be covering in this article. Fortunately, once you have a bit of sample code showing how this technique works, you can take advantage of it in your applications by simply cutting and pasting, without knowing all the intricacies of how delegates work. There are many reasons why you might use this functionality, the most common being to refill the cache with current data whenever it expires, or restoring the old cache data if the data source to repopulate the cache is unavailable.

In my example, I am simply going to cache the current time, and whenever the cache expires, I am going to add an asterisk character (\*) to the end of the string in the cache. Over time, you will be able to determine how many times the cache has expired by counting the asterisks. **Figure 9** demonstrates the important concept of callbacks and provides a good template for building more functional callback routines into your use of the cache.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 9 Caching Callback Example

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl50_ctl00_ctl00_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e) { string cacheKey = "myKey"; string data = ""; // Check to see if the data is in the cache already if(Cache[cacheKey]==null) { // Get the data since it isn't in the cache data = System.DateTime.Now.ToString(); //create an instance of the callback delegate CacheItemRemovedCallback callBack = new CacheItemRemovedCallback(onRemove); Label1.Text = "Generated: " + data; Cache.Insert(cacheKey,data,null, System.DateTime.Now.AddSeconds(5), System.Web.Caching.Cache.NoSlidingExpiration, System.Web.Caching.CacheItemPriority.Default, callBack); } else { Label1.Text = "Cached: " + Cache[cacheKey].ToString(); } } private void onRemove(string key, object val, CacheItemRemovedReason reason) { //create an instance of the callback delegate CacheItemRemovedCallback callBack = new CacheItemRemovedCallback(onRemove); Cache.Insert(key,val.ToString() + "\*",null,System.DateTime.Now.AddSeconds(5),Cache.NoSlidingExpiration, System.Web.Caching.CacheItemPriority.Default, callBack); }

One important feature to note in **Figure 9** is the pattern used in the Page\_Load to determine whether or not to use the data in the cache. You will always want to use this pattern when you deal with items in the cache. Use an if statement to check if the current contents of the cache are null (use a variable for your cache key since you'll be referencing it several times). If it is null, generate the data from its source and place it in the cache. If it is not null, return the data from the cache. If you have some very complex data access logic, you should place the whole if statement in a separate function that's tasked with retrieving the data.

The Cache object has a lot more functionality than most of the other objects I have discussed. It is one of the more powerful features of ASP.NET, and I would definitely recommend reading more on it. The summary at the beginning of this article lists some places to start your search for more information.

**Context**

The Context object holds data for a single user, for a single request, and it is only persisted for the duration of the request. The Context container can hold large amounts of data, but typically it is used to hold small pieces of data because it is often implemented for every request through a handler in the global.asax.

The Context container (accessible from the Page object or using System.Web.HttpContext.Current) is provided to hold values that need to be passed between different HttpModules and HttpHandlers. It can also be used to hold information that is relevant for an entire request. For example, the IBuySpy portal stuffs some configuration information into this container during the Application\_BeginRequest event handler in the global.asax. Note that this only applies during the current request; if you need something that will still be around for the next request, consider using ViewState.

Setting and getting data from the Context collection uses syntax identical to what you have already seen with other collection objects, like the Application, Session, and Cache. Two simple examples are shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl27_code');" \o "Copy Code)

// Add item to Context Context.Items["myKey"] = myValue; // Read an item from the Context Response.Write(Context["myKey"]);

**ViewState**

ViewState holds the state information for a single user, for as long as he is working with this ASPX page. The ViewState container can hold large amounts of data, but care must be taken to manage the size of ViewState since it adds to the download size of every request and response.

ViewState is one of the new containers in ASP.NET that you're probably already using, even if you don't know it. That's because all of the built-in Web controls use ViewState to persist their values between page postbacks. This is all done behind the scenes, so you don't need to worry about it most of the time. You should be aware of it, though, since it does impose a performance penalty on your application. How big this penalty is depends on how much ViewState you are carrying between postbacks—for most Web forms the amount of data is quite small.

The easiest way to determine the amount of ViewState being used by each control on a page is to turn on page tracing and examine how much ViewState each control is carrying. If a particular control doesn't need to have its data persisted between postbacks, turn off ViewState for that control by setting EnableViewState to false. You can also see the total size of the ViewState on a given ASP.NET page by viewing the HTML source of the page in a browser and examining the hidden form field, \_\_VIEWSTATE. Note that the contents are Base64-encoded to prevent casual viewing and manipulation. ViewState can also be disabled for an entire page by adding EnableViewState="false" to the @Page directive.

A typical Web form won't need to manipulate ViewState directly. If you build custom Web controls, however, you will want to understand how ViewState works and implement it for your controls so that they work similarly to the Web controls that ship with ASP.NET. Reading and writing values to and from ViewState is done using the same syntax I've used for the other collection objects:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl28_code');" \o "Copy Code)

// Add item to ViewState ViewState["myKey"] = myValue; // Read an item from the Context Response.Write(ViewState["myKey"]);

When building your own custom Web controls, you may also want them to take advantage of ViewState. This is simply done at the property level in your controls. **Figure 10** shows how you might store the PersonName property of a simple custom control in ViewState, and use it in the control's Render method.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 10 Storing a Property in ViewState

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl53_ctl00_ctl00_code');" \o "Copy Code)

namespace MSDN.StateManagement { public class HelloPerson : System.Web.UI.Control { public string PersonName { get { string s = (string)ViewState["PersonName"]; return ((s == null) ? "" : s); } set { ViewState["PersonName"] = value; } } protected override void Render(System.Web.UI.HtmlTextWriter writer) { writer.Write("Hello " + PersonName); } } }

**Web.config and Machine.config Files**

The data in these files is available to all users of an application. Data stored in the Web.config file is available for the life of the application. The data is generally small and the object works well for holding strings for file locations and database connections. Larger pieces of data are better kept elsewhere.

In addition to the various collection objects available, ASP.NET introduces a set of XML configuration files that are used to manage many of the settings for your applications, and even for your whole server. Each ASP.NET application uses a Web.config file to set many of its properties, and each server has a Machine.config file located in its system folder that is used as a basis for all applications. These settings are used as defaults unless overridden. In addition to storing configuration data, these files can store data that your application (or many applications, in the case of the Machine.config) needs.

Configuration information is read whenever the application starts, and is then cached. Since it is cached, the application can read this data very quickly, so you should not be concerned that your application will have a bottleneck because it has to constantly refer to a text file for some integral information. In addition, changes to the Web.config result in an application restart for that application (or all applications on the machine with Machine.config). This ensures that changes to configuration information are always reflected immediately by the application.

Database connection information, default image paths, and paths to XML data files are some of the more common pieces of data that are stored in the Web.config file. The syntax for storing data in the Web.config file is as follows, although ideally you might want to use integrated SQL authentication:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl31_code');" \o "Copy Code)

<configuration> <!-- application specific settings --> <appSettings> <add key="connectionString" value="server=myDBServer; uid=myUID;pwd=myPassword;database=myDB" /> </appSettings> <system.web> <!-- all wsb settings go here --> </system.web> </configuration>

To access these values within your ASP.NET pages, you use the ConfigurationSettings collection, which is in the System.Configuration namespace. The following simple example demonstrates how to extract the previous connection string into a local variable:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl32_code');" \o "Copy Code)

using System.Configuration; ••• String strConnString = ConfigurationSettings.AppSettings["connectionString"];

Adding a reference to the System.Configuration namespace reduces the amount of code required to reference these values. Since changes to the Web.config or Machine.config result in an immediate application restart, these values are typically only modified by the server administrator, usually by hand. Thus, you should think of these files as being a good place to store read-only data, not data that you will need to modify within your application.

**Conclusion**

Effective state management can mean the difference between a frustrating user experience with the potential for data corruption and a smooth, fast page or transaction process. While state management in ASP 3.0 was somewhat unwieldy, ASP.NET brings it under control with the state objects discussed here. With their careful use, you'll be on your way to presenting the best Web experience possible to your customers.