At Your Service

**Using ASP.NET Session State in a Web Service**

Matt Powell  
Microsoft Corporation

August 6, 2002

*Correction: IncrementSessionCounterX call is corrected in this document*

**Introduction**

One of the most common challenges that Web developers encounter is maintaining state in the stateless world of HTTP. There have been a number of clever means used to get around the stateless issue, from reposting application data with each request, to using HTTP authentication to map requests to specific users, to using HTTP cookies to preserve the state of a series of requests. One particularly clever way of maintaining state that hides all the challenging work below is to simply use the Microsoft® ASP.NET **System.Web.SessionState.HttpSessionState** class. You can use the ASP.NET **HttpSessionState** class from a Web method just as you can from ASPX pages, but things work a little differently for Web methods.

**Quick Overview of ASP.NET Sessions**

ASP.NET session state is maintained by using one of two underlying mechanisms. The first is by using HTTP cookies. The idea behind HTTP cookies is that when the client sends a request, the server sends back a response with an HTTP Set-Cookie header that has a name/value pair in it. For all subsequent requests to the same server, the client sends the name/value pair in an HTTP Cookie header. The server then can use the value to associate the subsequent requests with the initial request. ASP.NET uses a cookie that holds a session ID to maintain session state. Then that ID is used to find the corresponding instance of the **HttpSessionState** class for that particular user. The **HttpSessionState** class provides just a generic collection in which you can store any data that you want.

The other mechanism that ASP.NET uses for maintaining session state works without cookies. Some browsers do not support cookies or are not configured to keep and send cookies. ASP.NET provides a mechanism for getting around this problem by redirecting a request to a URL that has the ASP.NET session ID embedded in it. When a request is received, the embedded session ID is simply stripped out of the URL and is used to find the appropriate instance of the session object. This works great for browsers that are doing HTTP GET requests, but creates issues when writing Microsoft® .NET code that consumes an XML Web service.

It should be noted that sometimes it makes sense to store state information in cookies themselves instead of in the ASP.NET session object. By avoiding the session object, you use fewer resources on the server, and you do not have to worry about issues like locating a specific instance of the session object across a Web farm, instances of the session object being cleaned up because of a long delays between requests, or session instances lingering around for no reason until their timeout period expires. However, if you have data that includes implementation information that you do not want to share with the consumers of your service, or is private data that you do not want to send across an unencrypted channel, or if the data would be impractical to serialize into an HTTP header, then it may make sense to take advantage of the **HttpSessionState** class in ASP.NET. The **HttpSessionState** class returns an index key that is used to map a particular user to an instance of the **HttpSessionState** class that holds information stored for that user. Both the ASP.NET **HttpSessionState** class and HTTP cookies are available to users writing ASP.NET Web services.

**Why Use an HTTP Mechanism for Maintaining State in an XML Web Service?**

There are many ways to maintain state between SOAP requests. Certainly one feasible option would be to include something like the ASP session ID in the SOAP header of your SOAP message. The problem is that you have to: 1) still write the server side code yourself, and 2) make sure your clients treat your session ID header like an HTTP cookie and send it back to you with each request. There are certainly cases where using the SOAP header approach makes a lot of sense, but there are situations where using the HTTP approach can make sense as well.

ASP.NET session state is already done for you. The **HttpSessionState** class is available for easily storing your session objects. Most HTTP clients already understand that they must return the cookies that are set by the server and **HttpSessionState** happens to support the underlying transport most frequently used for SOAP communications—HTTP. Thus it makes sense that using ASP.NET session support could be a smart decision to meet many state management requirements.

**Enabling Session Support on the Server**

By default, ASP.NET session support for each Web method is turned off. You must explicitly enable session support for each Web method that wants to use session state. This is done by adding the **EnableSession** property to the **WebMethod** attribute of your function. The code for a Web method with the **EnableSession** property set to true, and which accesses the **HttpSessionState** object, is shown below.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

<WebMethod(EnableSession:=True)> \_

Public Function IncrementSessionCounterX() As Integer

Dim counter As Integer

If Context.Session("Counter") Is Nothing Then

counter = 1

Else

counter = Context.Session("Counter") + 1

End If

Context.Session("Counter") = counter

Return counter

End Function

As you might expect, if you enable session support for one Web method, that does not imply that it is enabled for another Web method. In fact, the **Context.Session** property will be null if **EnableSession** is not explicitly set to True for a particular Web method.

Be aware that it is possible to disable sessions by way of a web.config setting, so that even if you use the **EnableSession** property in your **WebMethod** attribute, **Context.Session** will always be null. The **/configuration/system.web/sessionState** element has a mode attribute that is used to configure how session state is maintained for your ASP.NET application. By default the mode is set to "InProc," which means that the **HttpSessionState** objects will simply be held in the ASP.NET process' memory. If the mode is set to "Off," then there will be no session state support in the ASP.NET application.

From the HTTP server standpoint, the scope of an ASP.NET session is that it lives within a given ASP.NET application. This means that the same instance of the **HttpSessionState** class will be used for all session-enabled ASP.NET requests within a single virtual directory for a particular user. A request to a different virtual directory with the same session ID cookie will result in ASP.NET being unable to find the corresponding session object—because the session ID was set for a different ASP.NET application. ASP.NET does not differentiate between ASPX and ASMX requests as far as sessions are concerned, so you could theoretically share session state between a Web method call and a normal ASPX file. However, there are client-side issues that we will look at in a little bit that might make this tricky.

When setting an HTTP cookie, you can associate an optional expiration time with it. The expiration time indicates how long the client should continue sending the cookie back to the server. If a cookie is set without the optional expiration, it will only be returned for the life of the process making the requests. For instance, Microsoft® Internet Explorer will return the cookie until you close that particular instance of your browser. The session ID cookies used by ASP.NET do not have expiration times. Therefore, if multiple processes on a client machine are making HTTP requests to your server, then they will not share the same **HttpSessionState** object. This is true even if the two processes are running at the same time.

If you are making simultaneous Web service calls from the same process, the requests will be serialized at the server so that only one will execute at any one time. Unlike .ASPX pages that have support for read-only access to the **HttpSessionState** object, which allows for simultaneous processing of multiple requests, there is no such capability with ASP.NET Web services. All Web method calls with sessions enabled have read/write access and will be serialized within each session.

**Client-Side Issues**

Successfully using the **HttpSessionState** capabilities in your Web service does rely upon some assumptions about the consumers of your Web service. First and foremost, if you are using the default HTTP cookie mode of maintaining session state, then your clients must support HTTP cookies. If you are using the cookieless mechanism for supporting sessions, then your clients must be able and willing to redirect their requests to the modified URLs with the session IDs in them. As it turns out, this is not a trivial assumption, even with a .NET client application.

**Everything Works from the Browser**

If you develop an ASP.NET Web service in Microsoft® Visual Studio® .NET, the default debugging behavior is to launch Internet Explorer and browse to your .ASMX file. This usually will result in a friendly HTML interface for invoking your Web methods. This turns out to be a nice way to debug your Web service code, and if you have set the **EnableSession** property to True for your Web method, it tends to work out beautifully. Even if you turn on cookieless session support, the browser client will work perfectly, and your session will work in the manner that you expect it to.

However, most Web service requests do not come from a browser. What happens when you create a client application that uses the "Add Web Reference" feature of the .NET Framework? Let's take a look at the results.

**Problems Using Add Web Reference**

I created a simple XML Web service using the code snippet that we saw earlier. If you recall, the Web method is called **IncrementSessionCounter** and simply stores an integer in the **HttpSessionState** object, increments it with each call, and returns the current value. From the browser client, we see that the number increases by one with each invocation as we expect.

Next, I created a simple Microsoft® Windows Form application and added a Web reference for my Web service. The code for invoking my Web service looks like this:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl02_code');" \o "Copy Code)

' Does NOT work with ASP.NET Sessions

Private Sub Button1\_Click(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) Handles Button1.Click

Dim proxy As New localhost.Service1()

Dim ret As Integer

ret = proxy.IncrementSessionCounterX()

Label1.Text = "Result: " & CStr(ret)

End Sub

When I invoke the Web service the first time, everything works as expected. The Web method returns the initial value for my session variable, which is 1. Now if I click on Button1 to invoke my Web method again, I expect to see a returned value of 2. However, no matter how many times I click on Button1, I always see a value of 1 returned.

You might suspect the cause of this is that I'm creating a new instance of the proxy class for my Web service, so each time I click on the button, I am losing my cookies (so to speak). Unfortunately, even if you move the proxy initialization code into the constructor for your **Form** class and use the same instance of the proxy for each Web method call, you still will not see the session variable return with a value greater than 1.

The problem is with the cookies. The Web service code does not see a valid session ID with the request, so it creates a brand new **HttpSessionState** object for each call, and returns the initial value of 1. The reason for this is that the client proxy class, which inherits from the **System.Web.Services.Protocols.SoapHttpClientProtocol** class does not have an instance of the **System.Net.CookieContainer** class associated with it. Basically, there is no place to store cookies that are returned. To fix this problem, I changed my code as follows with the new code highlighted:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

' Works with cookied ASP.NET sessions but NOT with

' cookieless sessions.

**Private Cookies As System.Net.CookieContainer**

Private Sub Button1\_Click(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) Handles Button1.Click

Dim proxy As New localhost.Service1()

Dim ret As Integer

**' Set the Cookie Container on the proxy**

**If Cookies Is Nothing Then**

**Cookies = New System.Net.CookieContainer()**

**End If**

**proxy.CookieContainer = Cookies**

ret = proxy.IncrementSessionCounterX()

Label1.Text = "Result: " & CStr(ret)

End Sub

And now the code works as expected! With each click of Button1, I see the returned value increase by 1. Note that the Cookies variable is not declared inside my function. It is a private member of my form class. I need to use the same instance of the **CookieContainer** class with each request if I expect the same session ID cookie to be returned to the server. This explains why a default cookie container is not automatically associated with an instance of the **SoapHttpClientProtocol** class. There is a good chance that you would want to use a separately managed cookie container that could be shared among multiple instances of the **SoapHttpClientProtocol** class, instead of automatically creating a new cookie container for each instance.

**Cookieless Sessions**

From the standpoint of the Web service developer, you might think that quite a few people trying to consume your service will forget to add a cookie container to their client proxies. With a clever twinkle in your eye, you also might think that cookieless sessions may be the perfect solution to this problem. If you set the **cookieless** attribute of the **sessionState** element to "true" in your web.config, you will notice that sessions still work perfectly when invoking your Web methods using the browser interface. Unfortunately, there are still issues if you use the "Add Web Reference" capabilities within Visual Studio .NET.

To investigate cookieless sessions, I decided to take the client code I used above and simply see if it would work for a Web service that was configured for cookieless sessions. I did not bother to delete the cookie container code, because I wanted to have code that would work with traditional cookied sessions as well as cookieless sessions. Being a bit of an optimist, I simply ran the code as is. Disappointingly, but not completely unexpectedly, I witnessed the following exception:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

An unhandled exception of type 'System.Net.WebException' occurred in system.web.services.dll

Additional information: The request failed with the error message:

--

<html><head><title>Object moved</title></head><body>

<h2>Object moved to <a href='/HttpSessionState/(l2z3psnhh2cf1oahmai44p21)/service1.asmx'>here</a>.</h2>

</body></html>

What happened is that the HTTP request received a response that was not a "200 OK" HTTP response. For those of you familiar with HTTP, you probably can correlate the HTML listed in the response shown as indicating that this was a "302 Found" HTTP response. This means that the request was redirected to the URL indicated in the hyperlink. The HTML returned is actually just a nice thing that a browser can show if for some reason it does not support redirects, or until the redirected request completes. If you look at the hyperlink, you will notice that the href includes an interesting substring of "(l2z3psnhh2cf1oahmai44p21)". If you have been paying attention, you have probably correctly deduced that this is the ASP.NET session ID, and it has been embedded in the URL that we have been redirected to. What we need is for our client proxy class to resend the request to this new URL.

Having done more than my share of programming with the old Win32 WinInet API, I went looking for a property on our proxy class that would allow me to turn on auto redirects. In layman's terms, this simply means that if we received an HTTP response of "302 Found," we would simply resend the request to the URL indicated by the HTTP Location header in the response. I was feeling pretty smart when the Microsoft® IntelliSense® in Visual Studio .NET showed me the **AllowAutoRedirect** property on my proxy class. I quickly added the following line to my code:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

proxy.AllowAutoRedirect = True

I gave my program another try, thinking this was still slightly easier than creating a **CookieContainer** class, and assigning it to my proxy. I got the following exception (truncated for brevity):

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

An unhandled exception of type 'System.InvalidOperationException' occurred

in system.web.services.dll

Additional information: Client found response content type of 'text/html; charset=utf-8',

but expected 'text/xml'.

The request failed with the error message: …

If you looked at the contents of the error message, you would find that you were looking at the HTML page that you see when you browse to your .ASMX file. The question you might have is: Why it is returning HTML when I am posting XML (in the form of a SOAP envelope) to the Web service? As it turns out, you did not send an HTTP POST request with a SOAP envelope, you simply sent an HTTP GET request with no body, and your Web service appropriately assumed you were a browser and returned its normal HTML response. How could this happen?

If you read the [HTTP specification](http://www.ietf.org/rfc/rfc2616.txt), you will find that it is appropriate for an HTTP client to send an HTTP GET request to the indicated URL in reaction to an HTTP "302 Found" response, even if the initial request was an HTTP POST. This works great with browsers, because just about all of their requests are HTTP GET requests in the first place. It does not work well when you see this result when you are posting data to a URL.

The justification for this is that potentially sensitive data may be contained in the posted data, so you need to confirm with the user if they really want to send the data to the new resource. If you are going to the new location based off an auto-redirect setting, you are obviously failing to confirm with the user whether it is okay to post their data to a new location. Therefore the data is not sent, and a simple HTTP GET request is sent instead.

I made the following modifications to set the URI on the proxy, catch the "302 Found" **WebException**, prompt the user for permission to redirect their request, and call my function again with the new location (changes from the previous code are highlighted):

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

' Works with both cookied and cookieless ASP.NET sessions.

Private Cookies As System.Net.CookieContainer

**Private webServiceUrl as Uri**

Private Sub Button1\_Click(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) Handles Button1.Click

Dim proxy As New localhost.Service1()

Dim ret As Integer

' Set the Cookie Container on the proxy

If Cookies Is Nothing Then

Cookies = New System.Net.CookieContainer()

End If

proxy.CookieContainer = Cookies

**' Set the Url on the proxy**

**If webServiceUrl Is Nothing Then**

**webServiceUrl = New Uri(proxy.Url)**

**Else**

**proxy.Url = webServiceUrl.AbsoluteUri**

**End If**

**Try**

ret = proxy.IncrementSessionCounterX()

**Catch we As WebException**

**' We need an HttpWebResponse if we expect to**

**' check the HTTP status code.**

**If TypeOf we.Response Is HttpWebResponse Then**

**Dim HttpResponse As HttpWebResponse**

**HttpResponse = we.Response**

**If HttpResponse.StatusCode = HttpStatusCode.Found Then**

**' This is a "302 Found" response. Prompt the user**

**' to see if it is okay to redirect.**

**If MsgBox(String.Format(redirectPrompt, \_**

**HttpResponse.Headers("Location")), \_**

**MsgBoxStyle.YesNo) = \_**

**MsgBoxResult.Yes Then**

**' It is okay. Set the new location and**

**' try again.**

**webServiceUrl = New Uri(webServiceUrl, \_**

**HttpResponse.Headers("Location"))**

**Button1\_Click(sender, e)**

**Return**

**End If**

**End If**

**End If**

**Throw we**

**End Try**

Label1.Text = "Result: " & CStr(ret)

End Sub

And now the ASP.NET session code works as expected. For the purposes of your own application, you can determine whether you need to prompt a user for redirecting their HTTP POST request or not. For instance, if you were calling this code from a service, you would not want to create a dialog that could not be seen.

This may seem to be a lot of work for getting ASP.NET sessions to work properly, but be aware that the code shown is useful for other things as well. For instance, any Web service on any platform that uses HTTP cookies would require the cookie container code. Similarly, there may be a host of other reasons why you might receive a "302 Found" response in reply to your request to a Web service. In a robust application, there will probably be a number of special scenarios that you will want to handle when invoking a Web service. Handling cookies and redirects are two such scenarios you may want to include in your Web service invocation code on a regular basis.

**Conclusion**

ASP.NET sessions can be very useful for maintaining state between Web method calls in your Web service. You do need to be aware that there may be issues that must be handled by client applications that you may not see when testing your Web service with the convenient browser interface. Fortunately, these issues are not particularly hard to handle.

**At Your Service**

**Matt Powell** is a member of the MSDN Architectural Samples team, where he helped develop the groundbreaking SOAP Toolkit 1.0. Matt's other achievements include co-authoring *Running Microsoft Internet Information Server* from Microsoft Press, writing numerous magazine articles, and having a beautiful family to come home to every day.