Cookieless ASP.NET

57 out of 101 rated this helpful

Dino Esposito
Wintellect

May 2005

Summary: Dino reviews the pros and cons of cookieless sessions and discusses why you should avoid storing valuable information in the session state. (6 printed pages)

Contents

Let's admit it—we're so accustomed to the idea of session state that we forget that session state is an artifice introduced with Active Server Pages (ASP) back in 1997. Session state gives you as the developer the ability to persist a piece of information about a user for the duration of time the user interacts with your application. The user-specific information is generally preserved for a 20-minute period that is renewed each time the user comes back to the site.

The first time the user connects to the site a brand new session state is created in the form of a block of memory to hold data, plus an ID to uniquely link it to the current user. On the next request, the user is expected to present the session ID so that the session state can be retrieved and properly restored. The session ID is an alphanumeric string that ASP and ASP.NET generate in total autonomy. How can the user manage it and make sure it is wrapped up with each subsequent request?

The HTTP protocol is stateless in nature, and nobody has done anything to change this fact. Almost two decades ago, while developing their first browser, Netscape Corporation "invented" a persistence mechanism to work over HTTP. They called it an HTTP cookie. It is interesting to note that the term "cookie" in computer science jargon just indicates an opaque piece of data held by an application that affects users but is never directly managed by users.

So cookies store the ID of the session and browsers transparently move their contents back and forth between the Web server and the local user's machine. When a cookie-enabled browser receives a response packet, it looks for attached cookies and stores their content to a text file in a particular folder in the local Windows directory. The cookie also contains information about the site of origin. Next, when the browser sends a request to the site, it looks in the cookies folder for a cookie that originated from that domain. If found, the cookie is automatically attached to the outgoing packet. The cookie hits the server application where it is detected, extracted, and processed.

In the end, cookies make Web sites much easier to navigate because they provide the illusion of continuity on top of a user's experience that necessarily spans over multiple requests.

Are Cookies a Problem?

For several years, cookies were simply considered a technical feature and largely ignored. A few years ago, the worldwide push on Web security focused the spotlight on cookies. Cookies were alleged to contain dangerous programs capable of stealing valuable information even beyond the physical boundaries of the machine.

It goes without saying that cookies are not programs and therefore can't collect any information on their own, let alone any personal information about users. More simply, a cookie is a piece of text that a Web site can park on a user's any personal information about users. More simply, a cookie is a piece of text that a Web site can park on a user's machine to be retrieved and reused later. The information stored consists of harmless name-value pairs.

The point is, cookies are not part of the standard HTTP specification, so they imply a collaboration between browsers and Web sites to work. Not all browsers support cookies and, more importantly, not all users may have cookie support enabled in their own copy of the browser.

There are Web site features that are historically so tightly bound to cookies that they make it hard to distinguish which really came first. On the one hand, session state management and user authentication are much easier to code with cookies. On the other hand, if you take a look at your site's statistics regarding browsers used to access pages, you might be surprised to discover that a significant share of users connect with cookies disabled. This poses a point for you as a developer.

Summarizing, cookies are not a problem per se but their use undoubtedly gives some server code the ability to store a piece of data on client machines. This prefigures some potential security risks and an overall situation less then ideal. (In some cases and countries, it's even illegal for an application to require cookies to work.)

Enter Cookieless Sessions

In ASP.NET, the necessary session-to-user link may optionally be established without using cookies. Interestingly enough, you don't have to change anything in your ASP.NET application to enable cookieless sessions, except the following configuration setting.

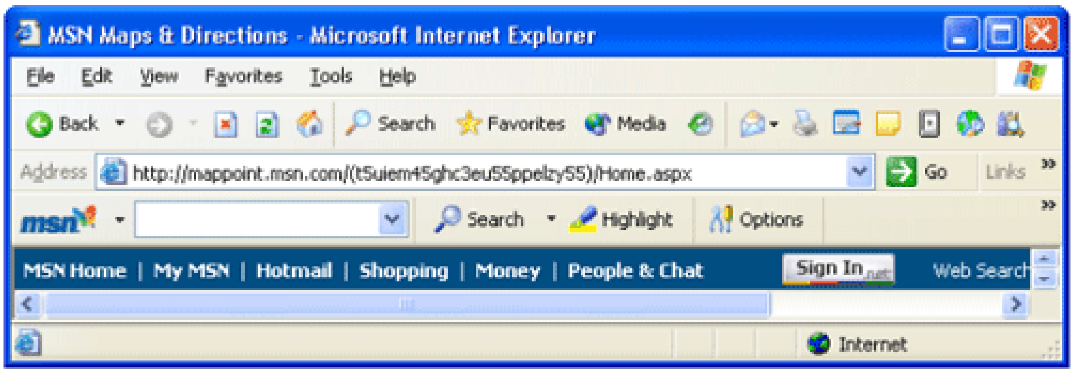
```
<sessionState cookieless="true" />
```

Default settings for ASP.NET session state are defined in the `machine.config` file and can be overridden in the `web.config` file in the application's root folder. By ensuring that the above line appears in the root web.config file, you enable cookieless sessions. That's it—easy and effective.

The **<sessionState>** node can also be used to configure other aspects of the session state management, including the storage medium and the connection strings. However, as far as cookies are concerned, setting the **cookieless** attribute to true (default is false) is all that you have to do.

Note that session settings are application-wide settings. In other words, all the pages in your site will, or will not, use cookies to store session IDs.

Where is ASP.NET storing the session ID when cookies are not being used? In this case, the session ID is inserted in a particular position within the URL. The figure below shows a snapshot from a real-world site that uses cookieless sessions.


MSN Maps & Directions - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go Links
msn Search Highlight Options
MSN Home | My MSN | Hotmail | Shopping | Money | People & Chat Sign In Web Search
Internet

http://mappoint.msn.com/(t5uiem45ghc3eu55ppelzy55)/Home.aspx

Figure 1. MapPoint using cookieless sessions

Imagine you request a page like http://yourserver/folder/default.aspx. As you can see from the MapPoint screenshot, the slash immediately preceding the resource name is expanded to include parentheses with the session ID stuffed inside, as below.

```
http://yourserver/folder/(session ID here)/default.aspx
```

The session ID is embedded in the URL and there's no need to persist it anywhere. Well, not exactly.. Consider the following scenario.

You visit a page and get assigned a session ID. Next, you clear the address bar of the same browser instance, go to another application and work. Then, you retype the URL of the previous application and, guess what, retrieve your session values as you get in.

If you use cookieless sessions, in your second access to the application you're assigned a different session ID and lose all of your previous state. This is a typical side effect of cookieless sessions. To understand why, let's delve deep into the implementation of cookieless sessions.

Implementation

The implementation of cookieless sessions results from the efforts of two runtime modules—the standard Session HTTP module named **SessionStateModule** and an executable known as aspnet_filter.dll. The latter is a small piece of Win32 code acting as an ISAPI filter. HTTP modules and ISAPI filters realize the same idea, except that HTTP modules are made of managed code and require ASP.NET and CLR to trigger and work. Classic ISAPI filters like aspnet_filter.dll are invoked by Internet Information Services (IIS). Both intercept IIS events fired during the processing of the request.

When the first request of a new browser session comes in, the session state module reads about the cookie support in the web.config file. If the **cookieless** attribute of the **<sessionState>** section is set to true, the module generates a new session ID, mangles the URL by stuffing the session ID just before the resource name, and redirects the browser to the new URL using the HTTP 302 command.

When each request arrives at the IIS gate—far before it is handed over to ASP.NET—aspnet_filter.dll is given a chance to look at it. If the URL embeds a session ID in parentheses, then the session ID is extracted and copied into a request header named **AspFilterSessionId**. The URL is then rewritten to look like the originally requested resource and let go. This time the ASP.NET session state module retrieves the session ID from the request header and proceeds with session-state binding.

The cookieless mechanism works great as long as the URL contains information that can be used to obtain the session ID. As you'll see in a moment, this poses some usage restrictions.

Let's review the pros and cons of cookieless sessions.

Thumbs Up

In ASP.NET, session management and forms authentication are the only two system features that use cookies under the hood. With cookieless sessions, you can now deploy stateful applications that work regardless of the user's preferences about cookies. As of ASP.NET 1.x, though, cookies are still required to implement forms authentication. The good news is that in ASP.NET 2.0 forms authentication can optionally work in a cookieless fashion.

Another common reason advanced against cookies is security. This is a point that deserves a bit more attention.

Cookies are inert text files and as such can be replaced or poisoned by hackers, should they gain access to a machine. The real threat lies not much in what cookies can install on your client machine, but in what they can upload to the target site. Cookies are not programs and never run like programs; other software that gets installed on your machine, though, can use the built-in browser support for cookies to do bad things remotely.

Furthermore, cookies are at risk of theft. Once stolen, a cookie that contains valuable and personal information can disclose its contents to malicious hackers and favor other types of Web attacks. In summary, by using cookies you expose yourself to risks that can be zeroed off otherwise. Really?

Thumbs Down

Let's look at security from another perspective. Have you ever heard of session hijacking? If not, take a look at the TechNet Magazine article Theft On The Web: Prevent Session Hijacking. In brief, session hijacking occurs when an attacker gains access to the session state of a particular user. Basically, the attacker steals a valid session ID and uses that to get into the system and snoop into the data. One common way to get a valid session ID is stealing a valid session cookie. That said, if you think that cookieless sessions put your application on the safe side, you're deadly wrong. With cookieless sessions, in fact, the session ID shows up right in the address bar! Try the following:

1. Connect to a Web site that uses cookieless sessions—for example, MapPoint—and get a map. At this point, the address is stored in the session state.
2. Grab the URL up to the page name. Don't include the query string but make sure the URL includes the session ID.
3. Save the URL to a file and copy/send the file to another machine.
4. On the second machine, open the file and paste the URL in a new instance of the browser.
5. The same map shows up as long as the session timeout is still valid.

With cookieless sessions, stealing session IDs is easier than ever.

I believe we all agree that stealing sessions is a reprehensible action from an ethical point of view. But is it injurious as well? That depends on what is actually stored in the session state. Stealing a session ID per se doesn't execute an action out of the code control. But it could disclose private data to unauthorized users and enable the bad guy to execute unauthorized operations. Read Wicked Code: Foiling Session Hijacking Attempts for tips on how to block session hijacking in ASP.NET applications. (And, yes, it doesn't rely on cookieless sessions!)

Using cookieless sessions also raises issues with links. For example, you can't have absolute, fully qualified links in your ASP.NET pages. If you do this, each request that originates from that hyperlink will be considered as part of a new session. Cookieless sessions require that you always use relative URLs, like in ASP.NET postbacks. You can use a fully qualified URL only if you can embed the session ID in it. But how can you do that, since session IDs are generated at run time?

The following code breaks the session:

```
<a runat="server" href="/test/page.aspx">Click</a>
```

To use absolute URLs, resort to a little trick that uses the **ApplyAppPathModifier** method on the **HttpResponse** class:

```
<a runat="server"
    href=<% =Response.ApplyAppPathModifier("/test/page.aspx")%> >Click</a>
```

The **ApplyAppPathModifier** method takes a string representing a URL and returns an absolute URL that embeds session information. For example, this trick is especially useful in situations in which you need to redirect from a HTTP page to an HTTPS page. Finally, be conscious that every time you type a path to a site from within the same browser you're going to lose your state with cookieless sessions. As a further warning, be aware that cookieless sessions can be problematic with mobile applications if the device can't handle the specially formatted URL.

Summary

The main reason for cookieless sessions in ASP.NET is that users—for whatever reasons—may have cookies disabled on their browsers. Like it or not, this is a situation you have to face if your application requires session state. Cookieless sessions embed the session ID in the URL and obtain a two-fold result. On the one hand, they provide a way for the Web site to correctly identify the user making the request. On the other hand, though, they make the session ID clearly visible to potential hackers who can easily steal it and represent themselves as you.

To implement cookieless sessions you don't have to modify your programming model—a simple change in the **web.config** file does the trick—but refactoring your application to avoid storing valuable information in the session state is strongly recommended too. At the same time, reducing the lifetime of a session to less than the default 20 minutes can help in keeping your users and your site safe.

About the Author
Dino Esposito is a Wintellect instructor and consultant based in Italy. Author of Programming Microsoft ASP.NET and the more recent Introducing Microsoft ASP.NET 2.0 (both from Microsoft Press), he spends most of his time teaching classes on ASP.NET and ADO.NET and speaking at conferences. Tour Dino's blog at http://weblogs.asp.net/despos.