Robert L. Bogue
MS MVP, MCSE, MCSA:Security
317-844-5310
Rob.Bogue@ThorProjects.com

Performance Improvement: Session State

In the first part of this series a discussion was presented on what performance is, and some of the techniques that can be used to improve or monitor performance in your application. In this article the focus is specifically on managing session state and the things that you can do to maintain performance in your application.

There are two key areas to understand in session state management. First, you need to understand the options you have for maintaining session state. Second, you have to consider the different kinds of information that need to be managed in session state and how the different needs for maintaining session state impact how you might choose to manage it.

Background

Just as we had to review some background concepts in order to understand the broad performance discussion, there are a few key concepts related to the communication between the client and the server. This includes the weight of the request in terms of the bytes transferred to the server and transferred from the server to the client. We'll talk about the request/response weight as well as the benefits and weakness of various encryption techniques that may allow you to leverage the users' machines for some session state management.

Request/Response Weight

In the old days, when most access, particularly from consumers, was expected to be from dial up lines, a ton of attention was paid to page weight. Some folks got aggressive by using programs to remove the unnecessary whitespace from JavaScript and HTML files. Others just intelligently managed the way that files were included to minimize the amount of data loaded on the first page. Some implemented splash pages to hide some of the initial weight of their main page.

Page weight really breaks down into two components: first request, and subsequent requests. The first request is where the browser doesn't have any content loaded. Every JavaScript file, CSS, image, etc., has to come across the wire. This initial page weight becomes everything that must be downloaded to make this happen.

The subsequent requests become much lighter because the images can be cached as can the JavaScript and CSS files. As a sidebar, you want to use a tool like Fiddler (http://www.fiddlertool.com), to make sure your browsers are only re-requesting the files they need.

Some of the techniques for managing session state rely upon an increase in page weight. It's possible to transmit some data in the form of hidden fields on a form or as a cookie to the browser. The browser will dutifully transmit this data back to the server when it's asked to. This increases the communications in both directions, however, if the amount of data

being transmitted back and forth isn't that large it can be a good technique for managing the amount of back end IO on a database server.

An argument that was made a while ago, which isn't really an issue today is that users may turn off cookies. It's certainly true that you need to check for this because if they do they'll break any mechanism you use to send data to them. However, because most sites require cookies, most users leave them turned on meaning that you don't have to be concerned about sending them session data via cookies. This is much more practical than the URL encoding strategies and form posting strategies that can be used to make sure you can get posted data back. I won't go into these two techniques but they essentially encode the information into the URL or force each page to be a post so that the client browser posts back hidden form fields.

Related Articles

- Browser Compatibility Development Guide
- Performance Improvement: Understanding
- A Two-Way Requirements Verification Process during Design Phase
- The Rules of the Project Management Game
- Hardware's Dirty Little Secret, or Why Software Can be Mass Produced

With any data transmitted to the client there's the concern for its sensitivity. You wouldn't want a cookie with the users password to be transmitted because even if you're running the site over SSL the cookies can get persisted to disk and thereby snooped. It's possible to specify that cookies are session cookies and shouldn't be persisted by not every browser honors these requests. As a result you have to assume that whatever you send as a cookie can end up on a disk — and therefore snooped or tampered with.

Encrypting Transmitted Data

Frequently when speaking with web developers, I hear that they don't need to worry about security because they're using SSL. While SSL is designed to prevent snooping and tampering during transmission, it doesn't do anything about securing the information that is on the user's computer. Securing the information that's on the user's computer takes two unique dimensions.

The first dimension is the protection of the user's data from snooping eyes. In other words, a cookie that has the user's social security number that they just entered on a form should be encrypted so that it's protected on the user's computer.

The other dimension is the malicious user who's trying to tamper with the information they have so that they can gain access to things that they shouldn't have. For instance, storing an unencrypted user id value in a cookie is a bad idea because a malicious person could just change the value and become someone else — if you were using that for their authentication.

Microsoft ASP.NET handles the session key encryption process for you if you use the ASP.NET Membership provider — and has built in tamper resistance for the session ID that is created, however, if you step outside of this to manage your own client storage you'll definitely need to be concerned with how you're going to secure the information being stored on the local system. That means understanding a few encryption basics.

Encryption Basics

In computer encryption there are two basic types: symmetric and asymmetric. Symmetric encryption is fast but requires that both the sender (encrypter) and the receiver (decrypter) have the same key. This key management problem — making sure that only the right people have the keys necessary to encrypt and decrypt data is where most encryption mechanisms throughout history have fallen apart. One an enemy (or bad guy) has the encryption key then your messages are no longer safe.

Technologies like Secure Sockets Layer (SSL) do the bulk of their encryption with symmetric encryption because it is so fast. The keys for the symmetric encryption are exchanged through asymmetric encryption which isn't as fast but doesn't have the same problem that symmetric encryption has with key distribution.

Asymmetric encryption relies on what's called a public-private key pair. One key is kept secret and the other key is made public. For instance, let's take a simplified exchange where we want to exchange some symmetric keys.

The client (requestor) requests a copy of the public key from the server (responder). The client then creates a random symmetric key and encrypts the information with the public key of the server. The server decrypts the package with its private key and gets the symmetric key the client randomly created. Now that both sides have the symmetric key they can exchange information using a symmetric encryption mechanism which is smaller and faster to decode than an asymmetric approach.

This understanding of encryption is necessary because it's impractical to encrypt all of the information going to the client to the client with asymmetric approaches. Since this isn't practical it's necessary to use a symmetric encryption mechanism for data sent to the client. The processing required to do asymmetric encryption and decryption just isn’t practical with current hardware. This introduces a key management problem.

If you're going to send data to the client using a symmetric key that key will need to change periodically in order to keep the data secure from prying eyes and from tampering. Changing the keys is difficult because it's mostly done on the fly and any data existing on the client when the change is made will end up being invalidated — because the old key it was encrypted with won't match the new key.

It is issues like these which push folks away from storing information on the client workstation and keeping all session state on the server.