SOEN 387: Web-Based Enterprise Application Design

Chapter 6. Session State

- Any **stateful** server object needs to keep all its state while waiting for a user to ponder a Web page.
- A stateless server object, however, can process other requests from other sessions.
- **Example**: A server with hundred people who want to know about books; processing a request about a book takes one second. Each person makes one request every ten seconds.
- If we want to track a user's requests with a **stateful** server object, we must have one server object per user: one hundred objects.
- But 90 percent of the time these objects are sitting around doing nothing. If just use **stateless** server objects to respond to requests, we can get away with only ten server objects fully employed all the time.

The point is that, if we have no state between method calls, it doesn’t matter which object services the request, but if we do store state we need to always get the same object.

Statelessness allows us to pool our objects so that we need fewer objects to handle more users. The more idle users we have, the more valuable stateless servers are.

Stateless servers are very useful on high-traffic Web sites. Statelessness also fits in well with the Web since HTTP is a stateless protocol.

Can we make everything stateless?

The problem is that many client interactions are inherently stateful.

Consider the shopping cart metaphor, the user’s interaction involves browsing several books and picking which ones to buy. The shopping cart needs to be remembered for the user’s entire session.

Essentially, we have a stateful business transaction, which implies that the session has to be stateful. If I only look at books and don’t buy anything, my session is stateless, but if I buy, it’s stateful.

Session State

The details of the shopping cart are session state, meaning that the data in the cart is relevant only to that particular session. This state is within a business transaction, which means that it’s separated from other sessions and their business transactions.

Since session state is within a business transaction, it has many of the properties that people usually think of with transactions, such as ACID (Atomicity, Consistency, Isolation, and Durability).

One interesting consequence is the effect on **consistency**. While the customer is editing an insurance policy, the current state of the policy may not be legal. The customer alters a value, uses a request to send this to the system, and the system replies indicating invalid values. Those values are part of the **session state**, but they are not valid.
Session state is often like this—it isn’t going to match the validation rules while it’s being worked on; it will only when the business transaction commits.

The biggest issue with session state is dealing with isolation. A number of things can happen while a customer is editing a policy. The most obvious is two people editing the policy at the same time.

Consider that there are two records, the policy itself and the customer record. The policy has a risk value that depends partially on the zip code in the customer record. The customer begins by editing the policy and after ten minutes does something that opens the customer record so he can see the zip code. However, during that time someone else has changed the zip code and the risk value—leading to an inconsistent read.

Ways to Store Session State

Client Session State stores the data on the client. There are several ways to do this: encoding data in a URL for a Web presentation, using cookies, serializing the data into some hidden field on a Web form, and holding the data in objects on the client.

Server Session State *:* may be as simple as holding the data in memory between requests. Usually, however, there is a mechanism for storing the session state somewhere more durable as a serialized object. The object can be stored on the application server's local file system, or it can be placed in a shared data source. This could be a simple database table with a session ID as a key and a serialized object as a value.

Database Session State is also server-side storage, but it involves breaking up the data into tables and fields and storing it in the database much as you would store more lasting data.

Using ***Client Session*** *State* means that session data needs to be transferred across the wire with every request.

At every request you have to transfer all the data the server uses for it, even if it isn't needed by the client for display. All in all this means that you don't want to use ***Client Session State*** unless the amount of session state you need to store is small. You also have to worry about security and integrity. Unless you encrypt the data, you have to assume that any malicious user could edit your session data

Session data has to be isolated. In most cases what's going on in one session shouldn't affect what's going on in another. If we book a flight itinerary there should be no effect on any other user until the flight is confirmed. Indeed, part of the meaning of session data is that it's unseen to anything outside the session.
This becomes a tricky issue if you use ***Database Session State***, because you have to work hard to isolate the session data from the record data that sits in the database.

If you have a lot of users,you'll want to think about whether you need session migration.

Session migration allows a session to move from server to server as one server handles one request and other servers take on the others.

Its opposite is **server affinity**, which forces one server to handle all requests for a particular session. Server migration leads to a better balancing of your servers, particularly if your sessions are long. However, that can be awkward if you're using ***Server Session State*** because often only the machine that handles the session can easily find that state.

Server affinity can lead to bigger problems. In trying to guarantee server affinity, the system can't always inspect the calls to see which session they're part of. As a result, it will increase the affinity so all calls from one client go to the same application server. Often this is done by the client's IP address. If the client is behind a proxy, that could mean that many clients are all using the same IP address and are thus tied to a particular server.

If you use ***Server Session State*** , the session state is pretty much there.

If you use ***Client Session State*** , it's there, but often needs to be put into the form you want.

If you use ***Database Session State*** , you need to go to the database to get it.

This implies that each approach can have different effects on the system's responsiveness. The size and complexity of the data will have an effect on this time.

If you have a public retail system, you probably don't have that much data going into each session, but you do have a lot of mostly idle users. For that reason, ***Database Session State*** can work nicely in performance terms.

For a leasing system you run the risk of heavy masses of data in and out of the database with each request. That's when *Server Session State* can give you better performance.

One of the big issues in many systems is when a user cancels a session and says forget it. This is particularly awkward with B2C applications because the user usually doesn't actually say forget it, it just disappears and doesn't come back. *Client Session State* certainly wins here because you can forget about that user easily. In the other approaches you need to be able to clean out session state when you realize it's canceled, as well as set up a system that allows you to cancel after some timeout period.

What happens when a system cancels?

A client can crash, a server can die, and a network connection can disappear.

Database Session State can usually cope with all three pretty well.

Server Session State may or may not survive, depending on whether the session object is backed up to a nonvolatile store and where that store is kept.

Client Session State won’t survive a client crash, but should survive the rest going down.