SOEN 387: Web-Based Enterprise Application Design

Chapter 17. Session State Patterns

Client Session State

Stores session state on the client.

With some applications you can consider putting all of the session data on the client, in which case the client sends the full set of session data with each request and the server sends back the full session state with each response. This allows the server to be completely stateless.

Most of the time you'll want to use *Data Transfer Object* to handle the data transfer. The *Data Transfer Object* can serialize itself over the wire and thus allow even complex data to be transmitted.

With an HTML interface, there are three common ways to do client session state: URL parameters, hidden fields, and cookies.

URL parameters are the easiest to work with for a small amount of data. Essentially all URLs on any response page take the session state as a parameter.

The clear limit to doing this is that the size of an URL is limited, but if you only have a couple of data items it works well, that is why it is a popular choice for something like a session ID.

Some platforms will do automatic URL rewriting to add a session ID.

Changing the URL may be a problem with bookmarks, so that is an argument against using URL parameters for consumer sites.

Another alternative is to use a hidden field is a field sent to the browser that is not displayed on the Web page. You get it with a tag of the form <INPUT type = “hidden”>.

To make a hidden field work you serialize your session state when you make a response and read it back in on each request.

The last, and sometimes controversial, choice is cookies, which are sent back and forth automatically. Just like a hidden field you can use a cookie by serializing the session state into it. You are limited in how big the cookie can be.

Also, many people don not like cookies and turn them off. If they do that, your site will stop working.

Cookies also work only within a single domain name, so if your site is separated into different domain names the cookies won’t travel between them.

Some platforms can detect whether cookies are enabled; and if not, they can use URL rewriting. This can make client session state very easy for very small amounts of data.

Client Session State contains a number of advantages. In particular, it reacts well in supporting stateless server objects with maximal clustering and failover resiliency. Of course, if the client fails all is lost, but often the user expects that anyway.

The arguments against *Client Session State* vary exponentially with the amount of data involved.

With just a few fields everything works nicely. With large amounts of data the issues of where to store the data and the time cost of transferring everything with every request become prohibitive.

There is also the security issue. Any data sent to the client is vulnerable to being looked at and altered. Encryption is the only way to stop this, but encrypting and decrypting with each request are a performance burden.

Server Session State

Keeps the session state on a server system in a serialized form.

In the simplest form of this pattern a session object is held in memory on an application server.

You can have some kind of map in memory that holds these session objects keyed by a session ID; all the client needs to do is to give the session ID and the session object can be retrieved from the map to process the request.

The great appeal of *Server Session State* is its simplicity.

Serializing a BLOB to a database table may turn out to be much less effort than converting the server objects to tabular form.

How servers remember a user?

Servlets typically deal with multiple users. When a servlet receives data from a browser, it must somehow figure out which user sent the message, what the user's privileges are, etc.

Each request from the browser to the server starts a new connection, and once the request is served, the connection is torn down.

However, typical web transactions involve multiple request–response pairs. This makes the process of remembering the user associated with a connection somewhat difficult without extra support from the system.

The system provides the necessary support by means of what are known as **sessions**, which are of type `HttpSession`. When it receives a request from a browser, the servlet may call the method `getSession()` on the `HttpServletRequest` object to create a **session object**, or if a session is already associated with the request, to get a reference to it.

To check if a session is associated with the request and to optionally create one, a variant of this method `getSession(boolean create)` may be used. If the value `false` is passed to this method and the request has no valid `HttpSession`, this method returns `null`.

When a user logs in, the system creates a session object as below.

```
HttpSession session = request.getSession();
```

When the user logs out, the session is removed as below.

```
session.invalidate();
```

Database Session State

Stores session data as committed data in the database.

When a call goes out from the client to the server, the server object first pulls the data required for the request from the database. Then it does the work it needs to do and saves back to the database all the data required.

In order to pull information from the database, the server object will need some information about the session, which requires at least a session ID number to be stored on the client.

One of the key issues to consider here is the fact that session data is usually considered local to the session and shouldn't affect other parts of the system until the session as a whole is committed.

Thus, if you are working on an order in a session and you want to save its intermediate state to the database, you usually need to handle it differently from an order that is confirmed at the end of a session. This is because you do not want pending orders to appear that often in queries run against the database for such things as book availability and daily revenue.

So how do you separate the session data? Adding a field to each database row that may have session data is one route. The simplest form of this just requires a Boolean **isPending** field.

A second alternative is a separate set of pending tables. So if you have orders and order lines tables already in your database, you would add tables for pending orders and pending order lines. Pending session data you save to the pending table; when it becomes record data you save it to the real tables.

Often the record data will have integrity rules that do not apply to pending data. In this case the pending tables allow you to forgo the rules when you do not want them but to enforce them when you do. Validation rules as well typically are not applied when saving pending data. You may face different validation rules depending on where you are in the session, but this usually appears in
server object logic.

If you use pending tables, they should be exact clones of the real tables. That way you can keep your mapping logic as similar as possible. Use the same field names between the two tables, but add a session ID field to the pending tables so you can easily find all the data for a session.

You'll need a mechanism to clean out the session data if a session is canceled or abandoned. Using a session ID you can find all data with it and delete it. If users abandon the session without telling you, you'll need some kind of timeout mechanism.

A daemon that runs every few minutes can look for old session data. This requires a table in the database that keeps track of the time of the last interaction with the session.

When to Use It

Database Session State is one alternative to handling session state; it should be compared with *Server Session State* and *Client Session State*.

The first aspect to consider with this pattern is performance. You will gain by using stateless objects on the server. However, you will pay with the time needed to pull the data in and out of the database with each request.

You can reduce this cost by caching the server object so you won't have to read the data out of the database whenever the cache is hit, but you'll still pay the write costs.

In a choice between *Database Session State* and *Server Session State* the biggest issue may be how easy it is to support clustering and failover with *Server Session State* in your particular application server. Clustering and failover with *Database Session State* are usually more straightforward.