Multithreading Concerns, SQL Injection, Scaling Up

Last time: Multithreading Concerns

- When our code is running in a web app, supporting a busy website, multiple requests can be executing concurrently
- Each request for a servlet runs in a thread supplied by tomcat
- Requests use transactions when accessing the database, and the transactions are started and ended in a single service-layer call.
- The thread's execution stack is thread-private, so local variables are too.
- We take advantage of this in handling the crucial Connection object for a transaction: we hold it in a local variable in the service layer.
- However, the DataSource is a singleton, so accessed by all the threads, and so needs to be thread-safe. Luckily, it's all implemented for us.

Concurrency and "thread-safe" code

DataSource is a Spring bean, a singleton, so needs to be thread-safe.

So do the other Spring beans: the Controllers, the API singletons, the CommandLineRunner: they are all singletons accessed by multiple threads.

And these hold code that we have written, so we need to be careful to keep them thread-safe.

The main concern is their fields. We saw that even a simple int counter as a field in one of these is not thread-safe, as was pointed out by Murach in Chap. 5.

Any mutable data structure in a field of a singleton needs "mutex protection", but we are hoping not to have to write such code. The JDK helps with this, providing thread-safe counters and collections.

We saw that all the beans are created before the flurry of requests come in, so we don't have worry about changes involved in initial setup.

Other objects in our system: domain objects

More on domain objects: they are private to the request thread

As covered last time, each request comes in and gets its own JDBC Connection from the connection pool in the DataSource that was set up by Spring as a Spring bean.

Recall we called the domain objects "scratch copies" of database data. They are created in the DAO for the currrent request and filled with fresh DB data. Or created in the service layer for the request's upcoming DB inserts, etc.

Either way, they are thread-private, not shared with executions of other requests in other threads.

Each request gets fresh copies of the DB data, so if an admin deletes a topping, that will be evident in the following requests.

These thread-private domain objects are important to the argument that we don't have to worry about multithreading issues in our code. That's a big feature!

Add domain objects to the request story

Domain objects are created usually near the start of the request execution in the server, to represent database data that is needed:

Or a bit later when the service layer decides to add something to the database:

They are all garbage-collected at the end of the request

Domain objects

Since all our domain objects are request-private, they can't cause any multithreading problems.

Of course we are re-creating the same Product objects over and over, a possible performance problem. But these are lightweight objects, very fast to create.

And we are asking the database over and over for the same Product info.

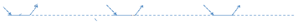
Luckily, databases optimize for repeated requests for the same data, keeping such data in memory for quick return.

Concurrency and Performance

Consider request-response cycles for various users:

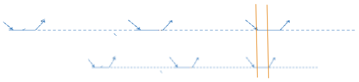
We are assuming that each user causes a sequence of request cycles usually well separated in time. Concurrent requests come about because of different user’s requests happening to occur at the same time, as follows:

Timeline showing requests for one user over time (they should be further apart)



Session object continuing existence of session object, until, say, 20 minutes of idleness by user-->

Multiple users: some requests are concurrent

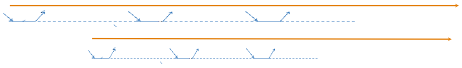


In fact, the requests are much further apart, so the probability of concurrency is very low with just two users.

With more users, it does happen more frequently

Multiple users: each has a session object

session object lifetimes:



Unlike the domain objects, the session objects live on after the request is done.

Rough Performance Analysis

Suppose 10000 session objects, for 10000 users in last 20 minutes

1000 active sessions, each lasting 2 minutes (so 10000 in 20 min)

Each user session does 10 requests over the 2 minutes, so 10000 requests in 2 min = 120 sec

So about 100 requests per sec.

Suppose each

That’s 5ms of CPU time per request, so 100 req/sec * 5ms = 500 ms CPU per sec, means 50% of one CPU to handle this load, OK.

Rough Performance Analysis, cont.

We said 50ms elapsed time per request. There are 20 such periods in each second, in which 100 requests are positioned, so about 5 requests in each 50ms period.

Each request takes 50 ms and uses 10% of a CPU during this time

So we see about 5 concurrent requests occurring at each point in time with this load.

Most requests have transactions, so also about 5 concurrent transactions, each with its own Connection.

That means we are involved in multithreaded execution.

But all the changeable shared domain data is held in the database, so we are using the database to do the hard work of concurrency control to shared data.

That’s one of the main secrets of good web application design.

Multithreading Issues??

Multi-threading: Let the DB handle concurrent access to domain data

Five concurrent transactions, each with its own thread, means we’re definitely doing a multi-threaded application. Need to worry about concurrent access to shared data.

Multithreading issues: we claim to be free of them!

Well, if we follow some rules…

Multithreading in our web apps is safe

Our rule for multithreading safety: make sure any fields of a singleton are thread-safe, or simply non-existent, remembering that bean-to-bean refs are thread-safe.

We also need the assumption that each session (i.e. requests from one user) involves one request at a time.

And that all our data shared between users is stored in a database.

Multithreading problems come up when two threads act on the same object, i.e., the data is shared in memory (in tomcat's JVM). So we have to look for objects shared between threads, i.e., request executions.

We are considering the web case of execution here. The non-web executions run in the client just like pizza1 and pizza2, so multiple executions only interact in the database, which of course is expert in handling concurrent access.

Objects Shared between threads

--no domain objects obtained from DB, because each thread has its own domain objects

--no fields of service objects by statelessness, and immutability of DAO references, and the rule about fields of singletons

--recall that the Connection is held in a local variable in the service layer, and all local variables are thread-private.

--no fields of other singletons (DAO, Controllers, etc), unless the objects are themselves thread-safe

--no request objects/variables, because the request object is specific to one thread. So is the response.

--no session variables, by one-at-a-time request assumption for each user

Objects Shared between threads, cont.

--we are not using "application" scope variables (this is possible, but leads to multithreading issues. We use the DB for shared data)

--we do share the DataSource object, but it is thread-safe, that is, has internal mutex to guard actions against race conditions.

-- The CommandLineRunner is another bean, but it's used only at startup in the web case

So we have an argument for all objects used in our programs. Right?

Conclusion: we are doing multithreaded programming without a single mutex that we set up ourselves!

Summary

With our architecture, the database system takes care of concurrent access to shared data. The only memory objects that receive concurrent access in our app are the API objects and Controller objects, where we are very careful about fields, and the DataSource, which is thread-safe (it has to be!). We are assuming that a *single user* does not have concurrent requests, so we don't have to worry about synchronizing access to the HTTPSession object.

That's a big win. Having multiple mutexes in code leads to the dreaded deadlock situation, when the software system freezes up, and needs true multithreaded debugging (Java has some support for this).

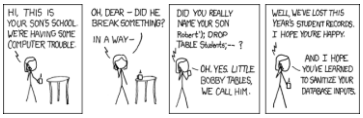
Web attack by SQL Injection: What it is and how to avoid it

SQL Injection means a break-in to a SQL-using application (usually a webapp) by entering just the right user input, causing the SQL in the application to provide hidden data, or allow a user to log in without knowing the credentials.

There is a flaw the admin app of music1 that should be considered

But first proof that this flaw is well known to proper nerds…

This problem is fairly well known… From XKCD webcomics, at https://xkcd.com/327/ with title Exploits of a Mom


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Web attack by SQL Injection: What it is and how to avoid it: music project case

There is a flaw the admin app of music1 that should be considered

Login UI takes in username and password for an admin user, for example user andrea, password sesame (this relates to a row in userpass).

DAO does

```
select * from userpass where username='andrea' and
password='sesame'
```

(where the underlined parts were input by the user) and allows the user to log in if such rows exist.

Sounds OK, but is prone to "SQL injection" ploy--

Adding on to app's SQL by putting the right text in a user input field

My break in

User input:

Username: ' or 'a' = 'a
Password: ' or 'a' = 'a

Success login!

This input made the query into:

select * from userpass
where username='' or 'a' = 'a' and password='' or 'a' = 'a'

(user input underlined) which retrieves every row in the table, resulting in a successful login.

SQL Injection Fixes

We could turn ' into ' ' for user input strings being used for SQL values (in string literals), since two apostrophes in a string literal stand for an actual ' in the string value by the SQL standard. This allows proper compare for O'Neil, for example. But this assumes a certain kind of break-in, so not a great solution.

or, another more common approach, that handles all the usual SQL Injection breakins:

Use a PreparedStatement with ? for each of username and password. This will bind the username string and the password string to the query without using surrounding quotes, so quotes are not special characters any more, and the user input cannot get any extra SQL in the query. Also O'Neil works properly.

See the Wikipedia article on SQL injection.

How can a malicious insert break into a database?

See http://amolnaik4.blogspot.com/2012/02/sql-injection-in-insert-query.html

- Basic idea: insert with a subquery can retrieve something (admin password for example), put it in the user table as "email"
- Later, have app print out user "email", see retrieved info
- Email entry: ' || (select max(password) from admins where user='root')||' (note single quotes at start and end entered by user)
- Plugged into: insert into users values ("'"+user+ "'", "'" +email+"')"
- Yields: insert into users values ('xxx',''||(select …)||'');
- Last value: nullstring||(select…)||nullstring, equals select result
- Ends up storing admin password as user's "email"
- Note this is not using a PreparedStatement.

Scaling Up: where we are starting from

Layered: call-down layers, no up-calls. Works great for answering web requests, but can't notify client between requests.

Domain Objects created for each request. Domain code itself is free of infrastructure calls (they are simple POJOs, unaware of how they are persisted).

Strong concept of Service API defining what the web app can do. Service API is "stateless", so each call stands on its own.

Use of a single database to hold all long-term changeable data and handle multi-user access to shared data.

Use a single executable running in application server: redeploy if need to change anything in the software. This use of a single Java executable for the webapp is known as the "monolithic" approach, a term of some derision by some "microservices" proponents. More on this later.

Stretching the basic setup

The single database may be very powerful, handling up to 100 TPS or somewhat higher. That's 360,000 TP/hour, a huge load. Ref: TPC non-clustered performance data.

For high performance, the database server is on a different machine than the application servers.

The single Java executable is not a problem for small or medium-sized sites. All the long-term data is in the database, which usually doesn't need to change for a software upgrade.

The only users who could be affected are those with long-running sessions.

Long-running sessions and redeployment

Web containers are required to maintain session objects across redeployments of the web app (Servlet spec, sec. 10.8).

However, with embedded tomcat, redeployment means restart of tomcat itself, so I don’t think this promise holds.

Even if the session object survives redeployment, objects hanging off the session object may be no longer recognized if their classes were edited (an argument for using just numeric ids, Strings, etc. in session objects).

25

Scaling up: first steps

First use a decent multiprocessor for the app server, and another one for the DB server.

Note that a DB caches hot data in its database cache, for fast access, so get enough memory for all important database data.

You will find that the app server gets overloaded before the DB server (assuming small DB requests of course).

So scale up the app server first (more CPUs). Eventually one system can't handle the load.

So replicate the app servers up to 25 ways (requires a request router up front.) Get a 40-way (or more) multiprocessor for the DB, with lots of memory.

This config is fully supported by our software setup, because of the independence of all the request environments. The request router needs to send all the requests of one user to the same server, unless we switch over to Spring session support.

26

Big systems for DB: example

Only when the biggest system can’t provide the needed TPS should you replicate the DB, unless the work can be perfectly partitioned in a way the request router can understand.

Example huge system for DB: Running Oracle on a 32 socket server with 24T of memory

- CPUs: Intel(R) Xeon(R) CPU E7-8890 v2 @ 2.80GHz – (Ivy Bridge EX)
- 32 sockets
- 480 cores (15 cores/socket)
- 960 threads (intel hyper threading)

Luckily, when your site is this big, you should have enough money to hire real experts!

27

Scaling up to seriously large apps

There are two directions for scaling up, system size (TPS) or software complexity.

For 1000 TPS, you need to go to machine clusters for the database, and caching. One way to get all those machines is to go to the cloud.

Code complexity: more code means more developers. For more than 75 developers (one estimate), the monolithic code becomes a problem. One little change by group A of say 8 groups means rebuild and redeployment.

Breaking up the Monolith. At some point, the system needs to be re-architected into multiple executables (components) that cooperate and provide not only services to the world, but also services to each other. Each part has its own database.

28

Netflix

For example, Netflix had a monolithic code for years, and then it was broken up into components with services.

Effort started in 2008, after a 3-day outage caused by a failure in the monolithic Oracle database.

They already had about 15 million subscribers for DVDs-by-mail, so this was already a huge and successful webapp.

They set up its new component-based system in the AWS cloud.

Microservices. This multi-component architecture has become known as the "microservice architecture" (terminology since about 2013).

Here each component has exclusive use of its part of the database data, so can be redeployed with database changes if necessary.

The other components can access the data via the services offered by that component.

29

Splitting the Music Database into two parts: how to break up a monolithic DB.

To illustrate the basics of microservice architecture, the music database could be refactored to allow it to turn into the back ends of two separate cooperating webapps.

We could change the database setup to accommodate use of one database for "sales" (users and invoices) and another for the "catalog" (products, downloads, etc.).

No foreign key can bridge the gap between the databases, so the user associated with a download no longer has a foreign key, and similarly, neither has the product associated with a lineitem. Instead of ids with foreign keys, the username is used in a download, and the product code is used in a lineitem.These are unique ids that often can be used without lookup of full details.

30

Splitting the Music Database into two parts, cont.

Old way: user_id in download (now part of catalog db), with foreign key to site_user (now part of sales db) to keep it correct
New way: email_address in download (which is a unique id), no foreign key

Old way: product_id in lineitem (now part of sales db), with foreign key to product (now part of catalog db) to keep it correct
New way: product_code in lineitem (which is a unique id), no foreign key

We could continue to use user_id in download (and product_id in lineitem) here, but this way, a Download object will have a email_address for the user, much more useful than a user_id, given that the user table is in a different DB.

Splitting the Music Database into two parts, cont.

You can see certain changes to break consistency would not be detected now, for example, deleting a user or changing the email_address for a user would leave download rows hanging. There is a cost in consistency for the split database. It can be fixed with additional code that "catches up" with updates, for "eventual consistency".

The service API can be split into CatalogServiceAPI and SalesServiceAPI, one for each database. They are implemented so that the catalog service only uses its DAOs (users, invoices, lineitems), and the catalog service only its DAOs (product, track, download). The sales service code calls the catalog service API to get information on a certain product, rather than the product DAO.

Splitting the Music Database into two parts, cont.

With these changes, we can have separate "Sales" and "Catalog" webapps, each with its own controllers and service API.

This could be useful to save on database costs: we could use Oracle (expensive) for the sales webapp, for safer money-handling, and mysql (free) for the catalog.

Or it could be used to stretch the database resources if the database server (already expanded to 640 CPUs and 10TB of memory, say) has become overloaded.

Or it could be used to allow two development groups to work and deploy new code independently, or almost independently (they do call each other through the service APIs).

Microservice architecture Wikipedia article

More info on Microservices

Microservice architecture Wikipedia article

Netflix video: Mastering Chaos
Start watching this video about 5 minutes from the start to miss a long intro. Netflix was a pioneer in the microservices area, and they are quite proud of it.

Another video with less negative treatment of the monolithic approach: Split That Monolith
Message: start with a monolith, monitor it to know when to split it up.

More details:https://hub.packtpub.com/how-netflix-migrated-from-a-monolithic-to-a-microservice-architecture-video/

How to do microservices on AWS cloud: "Cloud Native Development Patterns and Best Practices: Practical architectural patterns for building modern, distributed cloud-native systems", by John Gilbert, 2018, Packt Publishing. (at Amazon). Shows how hard it is to handle relationships between entities handled by different microservices.