Performance Report

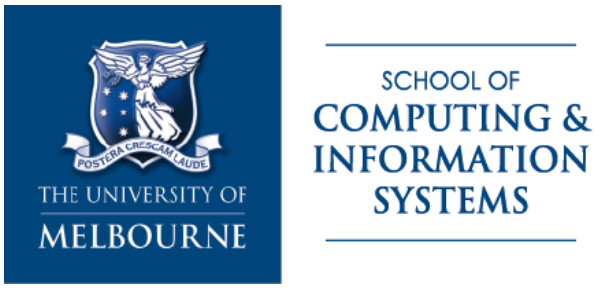
SWEN90007 Semester 2 - Part 4

TomcatBypass

Marketplace System

Team: Team 2

Name	UoM ID	UoM Username	GitHub	Email
James Hollingsworth	915178	jhollingwor	JaeholsUni	jhollingwor@student.unimelb.edu.au
James Vinnicombe	916250	jvinnicombe	jvinn	jvinnicombe@student.unimelb.edu.au
Sable Wang-Wills	832251	lawsonw1	saybb	sable.w@student.unimelb.edu.au
Thomas Woodruff	834481	twoodruff	twoodruff01	twoodruff@student.unimelb.edu.au



Contents

Methodology

We followed a systematic approach and used several different tools.

Approaches

Application performance was analysed with two main approaches:

Load Testing

We used a tool called “Locust” https://locust.io/ which allows you to write load tests programmatically in Python. This was somewhat involved to set up, however the result was realistic load tests which could be scaled up to thousands of users. Locust does not have the same throughput of tools like Jmeter, however it does have the advantage of being able to run realistic scenarios like a bidding war on the same listing.

Although Jmeter is commonly used in industry, we were not particularly interested in “writing” test scenarios by clicking buttons in a GUI and recording the HTTP traffic.

Because of the increased complexity associated with writing load tests in Python, we did not write many of them and used this tool only to target important performance cases.

Java Profiling

We experimented with these two profiling tools:

- JProfiler: https://www.ej-technologies.com/products/jprofiler/overview.html
- YourKit: https://www.yourkit.com/

Both options are proprietary and require licences for anything more than an initial trial. We used JProfiler for all profiling because of its simpler user interface. YourKit is a capable tool, probably too capable for the scale or timeline of this project.

Tools & Environment

We created a Docker environment which runs our PostgreSQL database, Tomcat application, and Locust. JProfiler can connect to a Docker container running on localhost, so for profiling our application we just connect it to the “sda-tomcat” container. Details can be found in the README file: https://github.com/SWEN900072022/tomcatbypass.

Process

Combining the above tools and environments, a full system test involves running a specific load test with Locust while profiling the Tomcat application with JProfiler, then analysing the results. This has only been done for important patterns or cases where we can change code to illustrate performance differences of different implementations.

Patterns Used

Unit of Work

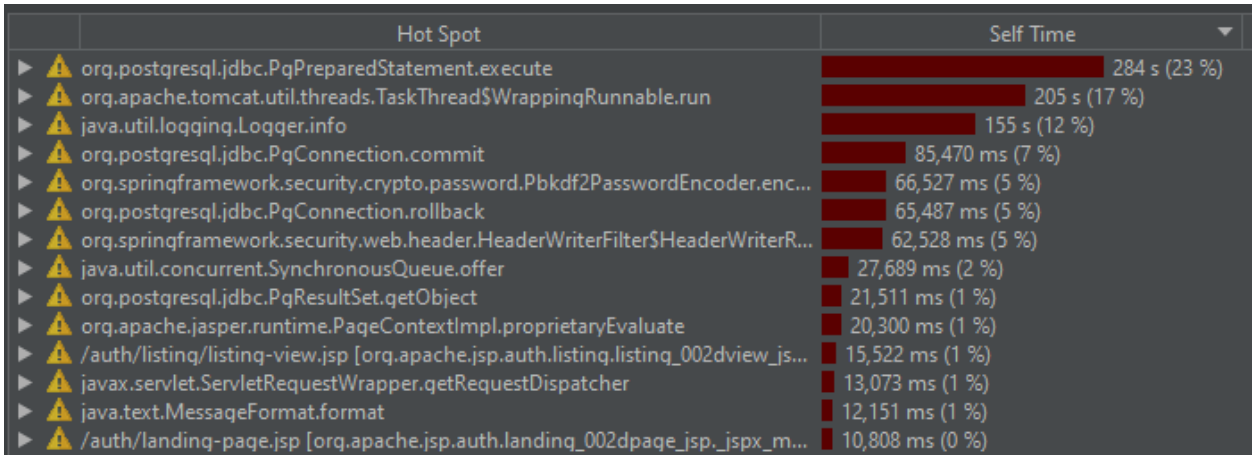
Description

Unit of Work is a pattern describing a method of demarcating boundaries of a transaction, and ensuring the changes made during it are atomic. This generally involves tracking objects that are changed or created, and committing those changes all at once on completion of a transaction.

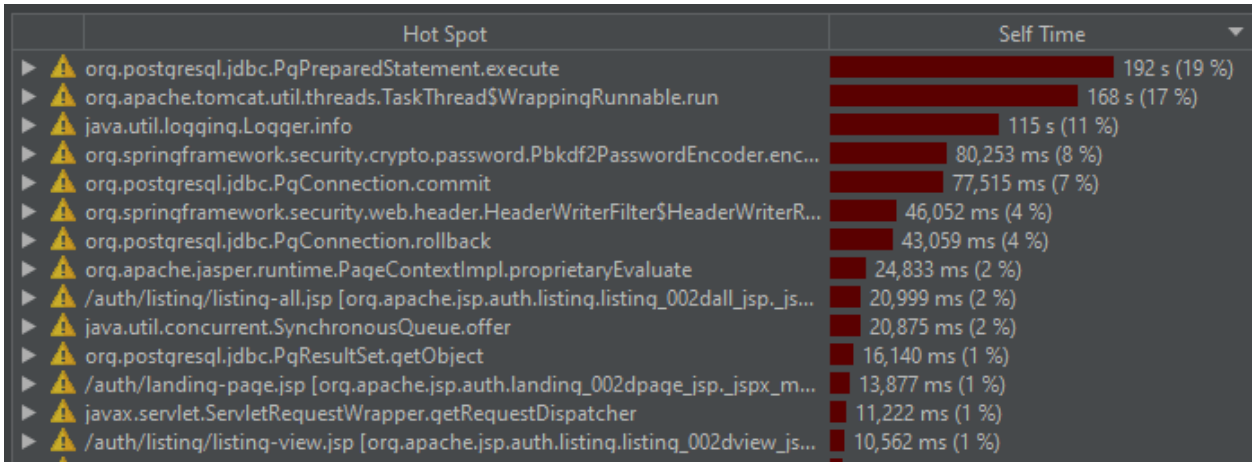
Unit of Work was implemented as thread-local instances and integrated with our ManagedTransaction class to manage our transaction changes. Any objects that are created, deleted or changed self-register themselves with the thread-local instance, and when the Unit of Work is told to commit, it will use the current DB Connection to execute the changes atomically, rolling back all changes if any operations fail. This is done by setting ‘autocommit’ to ‘false’, and reusing the existing connection for all changes.

Performance Impact

The Unit of Work pattern had a significant positive impact on performance. Two load tests were performed, one without the Unit of Work pattern enabled and one with it disabled. The load tests used the bidding war scenario where multiple users would bid on the same auction listing. Disabling the Unit of Work pattern was achieved by setting autocommit to false in the commit method used by the Unit of Work pattern. Below are the results of these tests.



Unit of Work disabled (autocommit=true in commit method) test of 7000 users spawning at 70 per second



Unit of Work enabled (autocommit=false in commit method) test of 7000 users spawning at 70 per second

In the case where Unit of Work was disabled, the system dedicated 23% of its time interacting with the database. When Unit of Work was enabled, only 19% of the system's time was spent interacting with the database. We can also see that the overall time spent interacting with the database has reduced dramatically going from 284s with UnitOfWork disabled to 192s when it is enabled - nearly a 30% difference. These results are to be expected because the purpose of the Unit of Work pattern is to reduce the number of individual connections to the database and instead process all database queries for a given system transaction at the same time. Requiring fewer connections naturally requires a smaller amount of time to be spent on setting up database connections, thus the system is able to spend more time on other tasks, improving the overall performance of the system.

Although Unit of Work is not specifically a performance pattern, it is clear that using it streamlines the process of executing a transaction and has a positive impact on performance.

System Transactions

Description

Transactions that make changes to the database are called Database Transactions or System Transactions. From the perspective of the database, they span a single series of changes that are either all committed or none are committed.

Business Transactions are bounded at the application level, and may comprise many System Transactions. They can be more complex, and involve user think-time. Applications make changes to the Database via Business Transactions.

All of our system's Business Transactions were implemented as single System Transactions with our ManagedTransaction class. Thus, all concurrency issues have been handled solely by choosing an appropriate PostgreSQL isolation level.

Performance Impact

Implementing concurrency control always comes at some cost to performance, be it throughput, uptime or similar. Our simple, minimal approach minimises the performance cost of concurrency control by avoiding locking the system with pessimistic locks, and judiciously using online optimistic approaches where conflicts are less likely.

For transactions implemented with isolation levels with no potential for rollback, there is no performance impact from rolling back or locking parts of the system. However, PostgreSQL does need to take snapshots and apply the isolation levels, which inherently has a small performance cost, but this happens regardless of the isolation level chosen.

For transactions that were implemented using an isolation level that could trigger a rollback, generally the isolation level was chosen because conflicts were unlikely. However, in the event of a conflict, there is of course a performance cost associated with retrying a transaction.

Concrete Table Inheritance

Description

There are numerous ways to map object-oriented inheritance to database schemas. The challenge is that relational databases do not support inheritance, and traits of objects that are unique to subtypes must be stored in the database somehow.

Single Table Inheritance involves representing all possible attributes of all subtypes in a single table. Concrete Table Inheritance involves representing all possible concrete subtypes as separate tables, even if common fields are repeated. Class Table Inheritance involves dividing up information over multiple tables, so that there is a table for each class, and fields are unique to each table, with inheriting tables linked to their parent table via primary key.

We chose to implement Concrete Table Inheritance for listings and orders. There is an abstract Listing class, with concrete Fixed Price Listing and Auction Listing classes, in addition to an abstract Order class, with concrete Fixed Price Order and Auction Order classes. In the database, there are only tables for Fixed Price Listings, Fixed Price Orders, Auction Listings and Auction Orders.

Performance Impact

Class Table Inheritance has the performance disadvantage of requiring multiple queries to the database or joins in order to fetch information about an object. It also creates a bottleneck at superclasses. If we had used Class Table Inheritance, this would have adversely impacted performance since both types of listings are frequently queried, meaning that the Listing table would be queried constantly and become a bottleneck.

Concrete and Single Table Inheritance are similar in that all the information about an object is stored in a single table, and therefore queries are simpler and require fewer joins. It has the advantage of splitting the query load over multiple tables, which is relevant for Listings. Fixed Listings are frequently bought and Auction Listings receive many bids, increasing the chance of non-serially equivalent interleaving of transactions in the case of Single Table Inheritance, as both listing types would all be checked against the same table. This could lead to more rollbacks or more time spent locking the database, and therefore lower throughput and uptime.

Connection Pooling

Description

Creation and destruction of database connections is expensive especially when the application and the database are not located on the same machine. A connection pool eliminates this overhead by creating a fixed amount of connections on start up and never closing them. The connections are instead kept in a ‘pool’ and reused by different database sessions. Most implementations of connection pooling will have one thread per connection, so the pool will have a number of threads each with one connection always ready and waiting.

Unfortunately Java does not have built in support for connection pooling (some languages like Go do), so you either need to write your own pool or use an external library. We chose the latter.

The default database connection example given in the subject notes was a bit funny in hindsight, and probably deliberately so. Not only did it create a new connection for every single database operation, it actually registered the database driver every time too!

Load Test Used

A single load test was used for all connection pooling performance analysis. The test involves a user registering themselves, logging in, and then performing several different read-only operations. Importantly, each operation is a different business transaction and therefore requires a new database connection. This is by design, and is intended to show the significant performance improvement from reusing connections through pooling rather than just creating new connections for every interaction with the database.

The Python class name of the test is ‘ManyConnectionsRandomNewLoggedInUser’.

Each load test has a total of 3000 users, with the number of users ramping up at 20 users per second. Because of the slow performance of registration, every test is initially slow and then speeds up once all the new users have registered themselves.

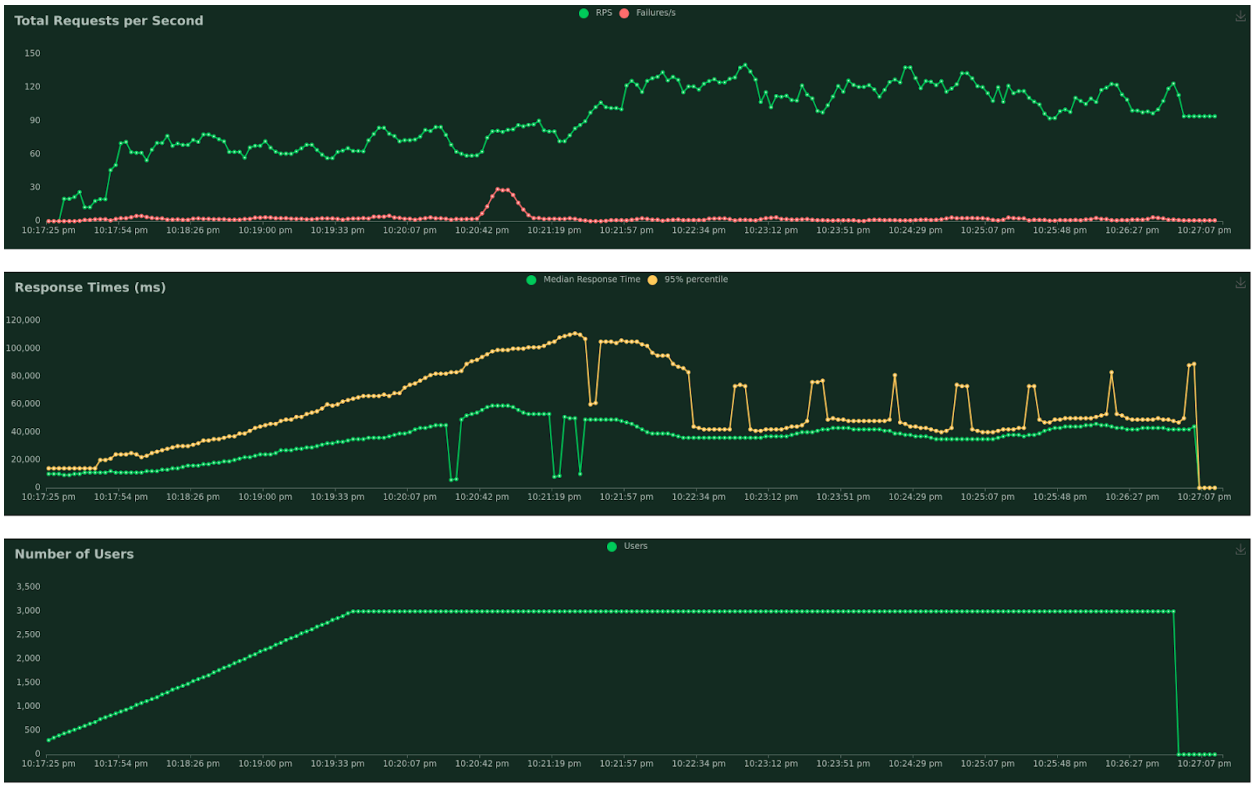
Performance Impact

The application had two alternate implementations for connecting to the database. These could be turned on or off using the boolean attribute ‘hikariEnabled’ in the class ‘Pool’. The implementations are discussed below.

1. DefaultPool

This one is somewhat misleadingly named since it’s not a connection pool at all. It registers the database driver once on startup, but any subsequent calls to getConnection() will create an entirely new connection. This is a great benchmark because it’s exactly the sort of naive implementation we would have used if we hadn’t been tipped off about connection pooling.

The below image provides a summary of the load test run with Locust using the DefaultPool. The test ran for 10 minutes and had a failure rate of 3% of requests.



Viewing the response times (in milliseconds) and number of users, it is obvious that there is a linear relationship between the two. This hints at the abysmal performance caused by not using a connection pool. Even after the initial spike from new users registering themselves the subsequent baseline or “normal” median response time is 40 seconds, with random spikes up to 80 seconds for the 95th percentile of users.

From the first graph “Total Requests per Second” we can also see that there are a number of failures. Failures occurred in clusters throughout the test and there was a particularly large spike around the 3 minute mark. There is also an unexpected drop in the response time only a minute

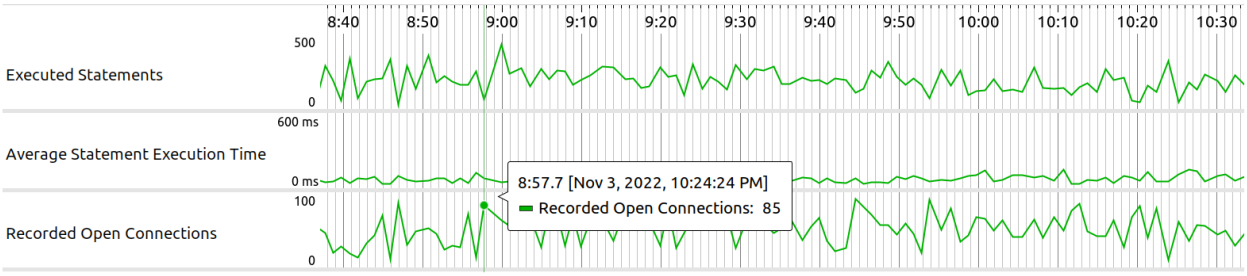
before this failure. This is caused by several database connection requests being rejected at once, which suddenly frees up resources for already running requests. These connection failures then show up as HTTP failures about a minute later when their relevant requests time out.

A quick look through the database logs confirms this hypothesis. There are hundreds of individual failures to acquire a database connection as shown in the following log screenshot.

```
sda-tomcat     | 03-Nov-2022 11:27:03.566 INFO [http-nio-8080-exec-151] com.unimelb.tomcatbypass.control.ServletUtils.logHttpRequest GET /tomcatbypass-1.0-SNAPSHOT/auth/landing-page
sda-postgres   | 2022-11-03 11:27:04.273 UTC [44321] FATAL:  sorry, too many clients already
```

The root cause of this performance issue is the failure to limit the number of simultaneous connections to the database. Requests are essentially fighting each other for access to the database, and once the maximum number of connections has been reached, any new connections fail until an already running connection finishes and relinquishes access.

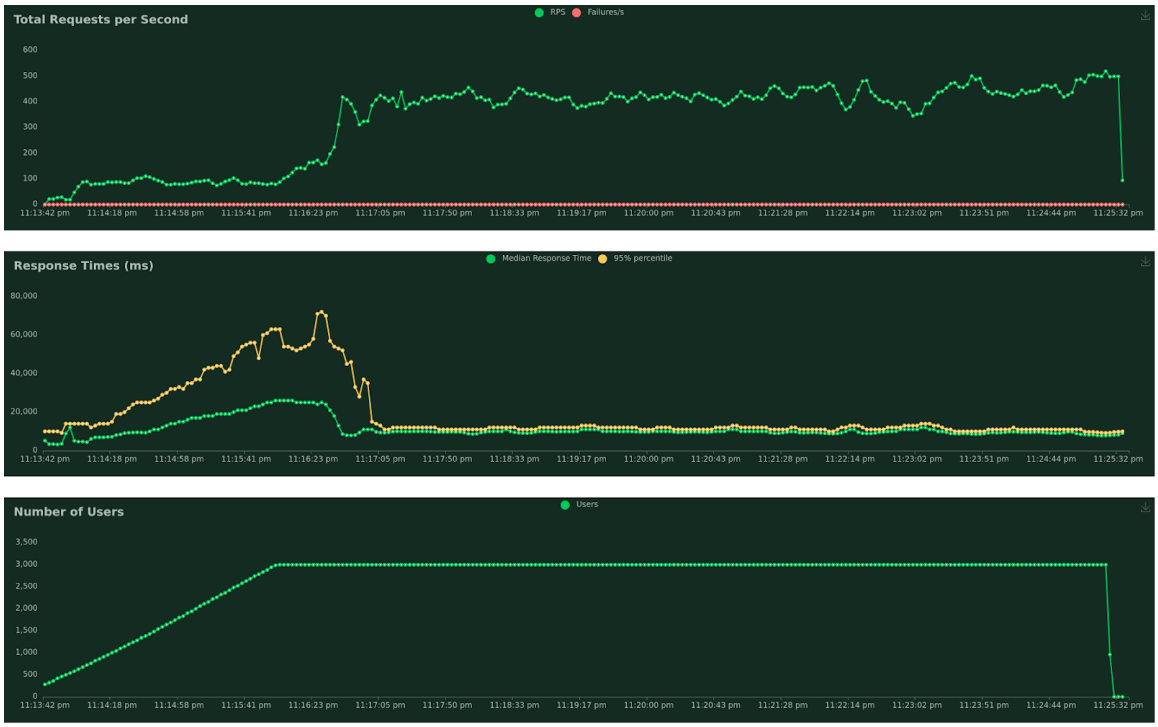
Viewing the following diagram of “Recorded Open Connections” we can see that the number of open connections never exceeds 100. This is the default maximum number set by PostgreSQL and is to be expected. The graph shows that database connection numbers are somewhat “spiky” however (presumably because values are truncated) it does not show the extreme spikes which cause failures to group together as seen in the initial Locust diagram.



2. HikariPool

This uses a state-of-the-art external library called HikariCP used in many enterprise applications. Its source code can be found here: https://github.com/brettwooldridge/HikariCP.

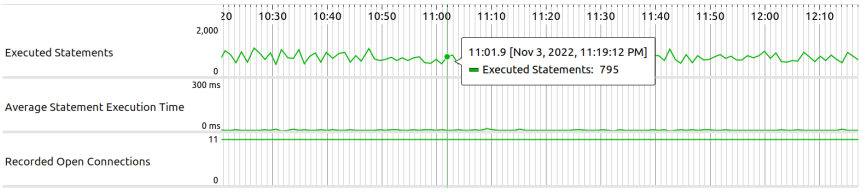
The below image provides a summary of the load test run with Locust using the HikariPool class. The test ran for 12 minutes and had no request failures.



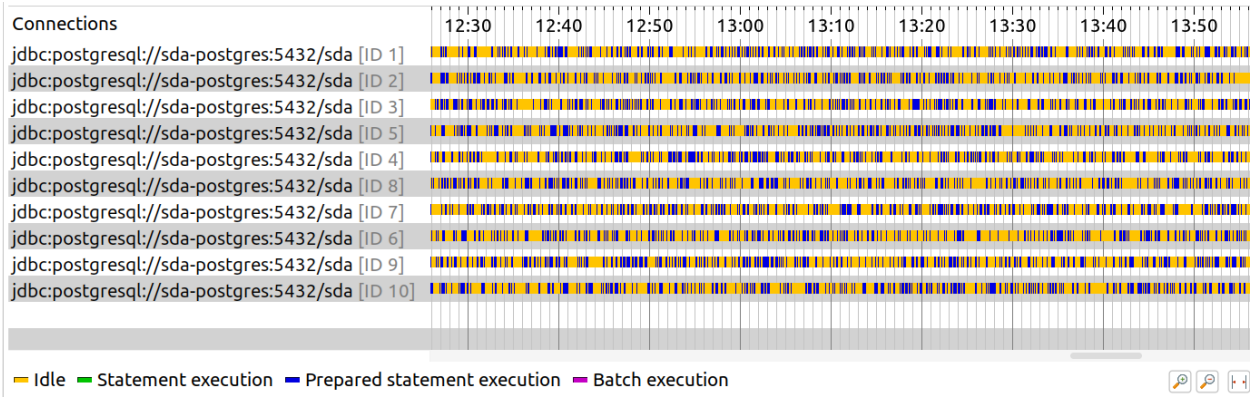
Relative to the performance of our application with the previous implementation, this is fantastic. There is an initial spike due to registration (discussed later in the report) followed by a significant drop in response time to a median of 8 seconds. Response times are also stable with the 95th percentile close to the actual median. The maximum requests per second serviced by our application is now 500, compared to the previous value of 135.

Although 8 seconds of latency for each request would be unacceptable for real users, we're pretty happy with it, especially given that every single request eventually gets fulfilled.

As shown below, the number of open connections is stable at 10, which is the default maximum pool size set by HikariCP. This confirms the application is behaving as intended. Please note that the value of 11 is just the y axis and not the actual value.



Because there are a finite number of reused connections to the database, we can now examine them to see how heavily they are used. The following diagram shows that even under heavy load, each connection spends more time idle than being used.

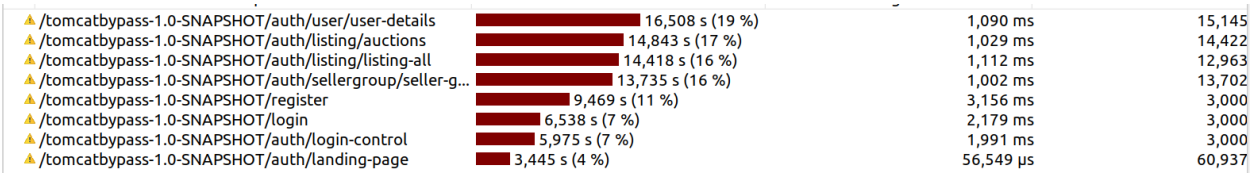


There are two possible reasons for this:

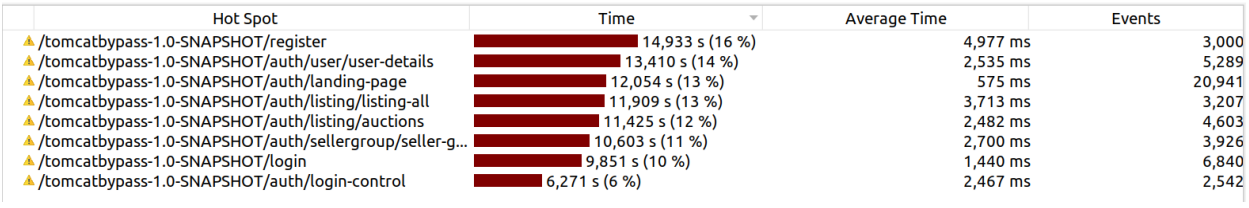
1. The Docker container doesn't have 10 cores, so each connection has to time share and is inherently going to be blocked waiting for CPU resources.
2. Our application has bottlenecks unrelated to database connections.

Both are probably happening.

The following diagram shows a summary of the response time for each HTTP endpoint, when HikariPool is enabled.



This is in stark contrast to the next diagram which shows the same statistics but with the DefaultPool enabled. Not only is each endpoint faster with a proper connection pool, but the simpler pages like landing-page load faster, as they should.



Authentication and Authorization

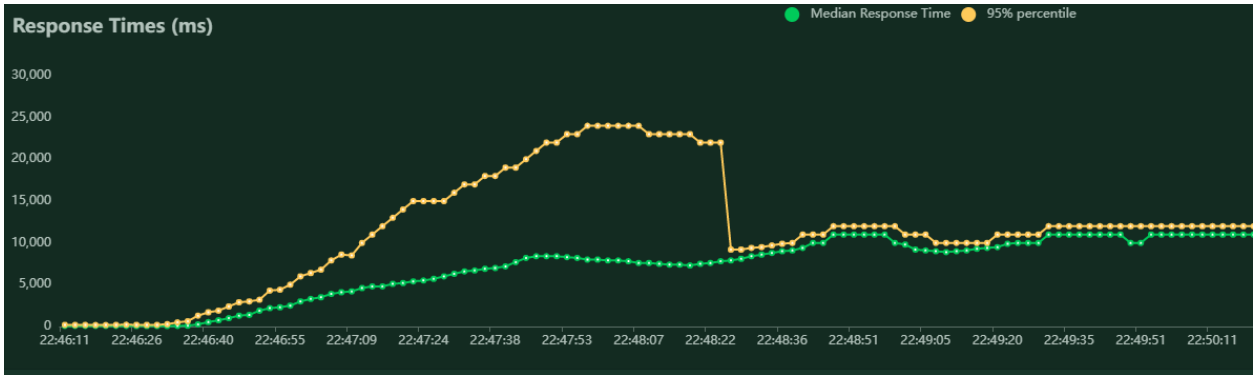
Description

Authentication and authorization is a high level pattern that determines what a given user has access to within a system. The first part of authentication is the act of verifying, from the application's perspective, who a user is. This is usually done through a username and password combination. Once authenticated a user can then be authorised to interact with different elements of the application depending on their roles.

We implemented this pattern using Spring Security. Users were authenticated through a username, and a password that was encoded and saved to the database. Spring Security enforced controls on the types of interactions each user could have with the application depending on their authorization role.

Performance Impact

The initial user registration process was identified as a major performance bottleneck. Our implementation of load testing involved many threads creating an account and signing in with that account, then performing other actions. It became immediately clear that the initial registration of users was massively slowing down the performance of these load tests. During load tests there would be a spike in response times while the users were registering themselves and then logging in. Response times would subsequently drop and the relevant part of a given load test would begin.

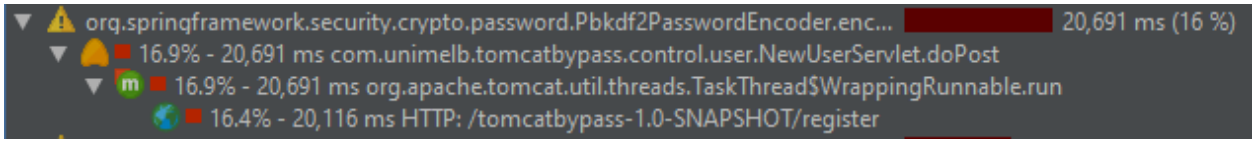


Test of 7000 users spawning at 70 per second. Observe the decrease in response time after users have finished registering and logging in.

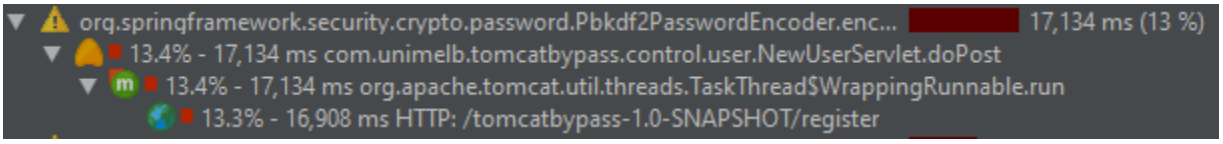
By profiling the application with JProfiler while a load test was running, we identified the source of the bottleneck to be the slow performance of Spring's password encoder.

We had employed a pbkdf2 encoder through Spring Security and implemented this using the default 185,000 iterations per encoding. This amount of iterations for the encoding of a new password and logging in through the password checking process was straining the system.

Especially for load tests comprising thousands of users this tended to be the biggest impact on performance during a test. We have since dropped the number of iterations to 1,850, a decrease of 2 orders of magnitude.



Pbkdf2PasswordEncoder 185,000 iterations time of ~21 seconds during test of 3000 users spawning at 30 per second



Pbkdf2PasswordEncoder 1,850 iterations time of ~17 seconds during test of 3000 users spawning at 30 per second

While the change in performance wasn't significant in absolute terms from 185,000 down to 1,850 the practical change when running load tests was apparent, especially when resources were being consumed by other load testing activities after the login process.

We can see through these tests the way that authentication & authorization negatively affects application performance. The login process using complex hashing functions is critical to ensuring the security of the platform however it is intensive for the system to handle and therefore must be a tradeoff of complexity to enforce security and the impact that has on the performance and user experience. In enterprise systems this is commonly solved with the implementation of an authentication service which can run as a separate process with resources allocated to ensure that performance is maintained during spikes in potential customer signup. However even in enterprise systems it is common to see authorisation as a performance bottleneck.

Caching and Memory

Description

There are various patterns which have a similar effect on performance because they all relate to the performance increase gained from storing objects in memory rather than querying the database each time objects are referenced. The patterns which fall into this category are as follows.

Domain Model

The Domain Model pattern involves storing objects in memory and using methods to allow interactions between these objects. The nature of objects being cached in memory when they are being referenced allows for much faster computation when compared with querying objects from a database every time they are required.

Association Table Mapping

The Association Table Mapping pattern deals with the problem of representing many to many relationships between tables in the database and objects in memory. In the database, many to many relationships are represented using an intermediate table which contains the foreign keys to represent the relationships. These relationships are represented in memory by storing a list of object references that are referenced by a given object. Storing many to many relationships using this method allows for much faster response times when querying the database. It is simple to retrieve the list of related objects to be queried and doesn't require redundant information in table rows.

Lazy Load

The Lazy Load pattern works by storing dummy objects in memory and only loading the data associated with these dummy objects as they are needed. This pattern improves performance because information which is never used will never be queried from the database, which reduces the overall amount of data that is loaded from the database. As is the case with the other patterns in this section, the Lazy Load pattern is characterised by storing data in memory to allow for faster retrieval.

Our Inclusion

We have implemented the Domain Model and the Association Table Mapping model in our application. We have created classes for all the necessary domain objects and we store many to many associations by storing lists of references where required, namely in the SellerGroup class. As for the Lazy Load pattern, we initially did not see any reason to implement it because our implementation of the domain model wasn't entirely correct. Our implementation of the domain model meant that in memory associations were linked using database IDs rather than in memory object references. Linked objects would be retrieved using database queries as they were needed rather than being loaded at the same time as the object that links to them. Doing

so makes Lazy Load redundant because this implementation unintentionally achieved the same effect of only loading referenced objects as they are needed.

Performance Impact

Referencing in memory objects is much faster than querying the database every time that object is needed within a system transaction. For example, in the createBid system transaction method in the BidService class, the appropriate listing object is retrieved and stored in memory at the start of the transaction. This object is subsequently referenced five times in the same system transaction. If this object were not retrieved and stored as a domain object at the start of the transaction, five database queries would be required instead of one. Clearly the use of far fewer database queries will improve the performance of the system.

If we had implemented the Domain Model pattern correctly we would have recognised the benefit of using the Lazy Load pattern. If this were the case, the Lazy Load pattern would undoubtedly have a significant positive impact on performance. For example, with our current Domain Model implementation when a user is loaded, the seller groups that the user is a part of aren't loaded alongside them, but they would be if we had implemented the Domain Model pattern correctly. In the case that we had implemented the Domain Model correctly, if a user wanted to do something unrelated to the seller groups they are a part of after they are logged in (i.e. purchasing a fixed listing), then it would be pointless to load the users seller groups when the user is loaded. This is where the Lazy Load pattern would have been very effective. We would have used lazy initialization in this case so that the list of seller groups referenced by the user object would only be loaded when they are needed (i.e. the user wants to sell an item) and when information related to the users seller groups isn't needed then these seller group objects would not be loaded at all, as would be the case if the user simply wanted to purchase a fixed listing. Clearly avoiding loading vast amounts of referenced objects when they are not needed will have a positive impact on performance.

Patterns Not Used

These patterns were not used within the application, and are discussed in the context of any potential performance improvements or disadvantages.

Pessimistic Locking

Description

Pessimistic locking is a form of database locking to handle concurrent transactions within a system. This form of locking takes the strongest approach to concurrency management. When a transaction could potentially change a row in a table it procures a lock on that row preventing any other transaction from accessing the row.

Our application did not use pessimistic locking. This strict form of access control has a negative impact on performance and should be avoided wherever possible. We found throughout our application we had no functions that required the strict control of pessimistic locking so opted not to use it.

Performance Impact

The primary reason for excluding pessimistic locking from our system was performance based. This is because the impact of pessimistic locking is extreme. When a table row is locked this means that until the transaction concludes no other transaction will be able to access this table row.

This means any table row that may be commonly accessed will bring all relevant transactions to a halt during an edit. Any concurrent transaction depending on the locked resource will have to wait until the lock has been released. This waiting creates a performance bottleneck as there will be transactions waiting on a user to finish editing. This is dependent on the user's think time and thus not controllable. This increases the response time of any transaction that needs to read from the locked row as it must wait for the lock to be released. It's clear to see the negative impact on performance creating intermittent yet extreme increases in time to completion on any functionality utilising affected rows.

Embedded Value

Description

Embedded value is used when an object in the application is not suitable to be its own table within the database. This can be for a variety of reasons. Instead of creating a table to house the information, the various attributes of the object are stored within table rows for the object(s) which contain the originally specified object. In this way when the table row is read a set of the attributes from that row are turned into an object which is then contained within the primary object for the table.

We did not opt to include embedded value within our system. In the initial system design we proposed a usage of embedded value for order addresses, however primarily due to time constraints we removed the pattern from the system in part 2 and did not subsequently add it back.

Performance Impact

Implementation of embedded value in the correct circumstances will cause a marginal increase in performance. When correctly implemented the object utilising this pattern is never called from the database on its own. The information is only retrieved as part of an object that contains it. Embedding this value into the tables of these other objects saves an additional database read when reading the object data into memory. The embedded information is required to be retrieved when creating the object (not taking into account any lazy loading implementations) and as such if the value weren't embedded then this would require an additional query to the table containing the now embedded data.

A poorly thought through implementation of embedded value can adversely affect performance If the embedded value is often queried independently of the table it is embedded in. This poor implementation would require unnecessary reads of information into memory thus hampering performance.

The performance impact of embedded value is slight for a single transaction however the cumulative impact would be noticeable to users in a system which heavily uses this pattern.