# SOFTENG 325 Assignment 1 Report

## Scalability

I have tried to account for scalability in my web service with the design of my domain model, caching, fetch plans, and allowance for concurrent requests.

My domain model maps the object-oriented architecture of the program to the relational architecture of the database. This allows us to use object-oriented logic for the system logic and makes the system more maintainable and extendable if the system were to be of a larger scale.

We use JPA persistence contexts that automatically support caching for entities that had been loaded previously. This allows us to reduce the number of queries to the database and improve performance at the cost of memory. Performance is important when considering scalability as we need our system to operate efficiently at a large scale.

I have designed fetch plans for different queries in my system depending on its use case, so I don't use queries that retrieve unnecessary data or make too many queries. This increases performance and therefore, scalability.

Concurrent requests are allowed for using my current system design so multiple clients can access the system at the same time, which is extremely important for scalability as we will likely have more than one client accessing our system at the same time. Our system is RESTful so no session states are stored, which allows for horizontal scalability as we can have several servlets running concurrently.

## Fetch Plans

I have set all the default fetch plans for my mapped persistent collections as lazy loading so that no unnecessary information is fetched. This reduces the size of query results and increases performance when we don't need information from the collections, but it can potentially cause the n+1 problem where if I need all the collections in an entity, I need to make n+1 queries to the database, where n is the number of the collections. This is addressed by dynamic eager fetching.

When I need all or most of the mapped persistent collections of an entity, such as when I'm making a ConcertDTO from the server to send to the client, and I need to include all the performers in the object, I dynamically eager fetch the collections in the query. This reduces the number of queries that I make to the database to increase performance. However, eager fetching can result in the Cartesian product problem where a lot of duplicated data is produced by joining and returning. To resolve this problem, I use the DISTINCT option when making my query, so only unique data is returned.

## Concurrency

Currently, the only entities that can change after creation in my domain model are Seat and User. I use optimistic locking to ensure concurrency for both these entities as it gives better performance when there are few collisions, and there are no deadlock problems or necessity to wait for locks when the application is run, since our client should not have to wait for other users to let go of seats and users. The locking ensures data integrity while allowing for concurrent clients.

Seat has a boolean field isBooked that is changed after we book the seat. When there are concurrent clients, it is possible for a client to book a seat while another client is also trying to book the same seat. In this case, I want the client that completes the booking first to get the seat, and the other client should be unable to make the booking. I use optimistic locking on Seat to achieve this. There is a version number in Seat. The version for a single seat is updated when it has been changed and committed to the database. When booking a seat, we check its isBooked status and only continue if it hasn't been booked. When we make a new booking and commit it to the database, we check the version number in the seats of our booking and ensure that they haven't changed. If they have, it means that isBooked has been changed and the seat has been booked, so our new booking should be invalid.

Users have a list of bookings. This could cause concurrency issues if the same user tries to make several bookings at the same time. These kinds of collisions are unlikely though, and thus we use optimistic locking to account for it. When a user has a new booking added to it, we forcibly increment its version number so that when two bookings are concurrent, the one that is sooner to commit succeeds and the later one fails.

I synchronize the access to the static list of subscriptions to ensure that concurrent servlets will share the same list of subscribers and to prevent concurrency issues when two servlets access the subscriptions at the same time.

# New Features

## Support for different ticket prices per concert

My web service currently creates seats for all concerts. The seat is specific to the concert and contains its individual price. Therefore the web service is able to support different ticket prices per concert; It just needs to create seats with different prices for the concert.

## Support for multiple venues

Seats in my web service for a particular concert currently uses the seat label and date to identify it. This can be a problem if we have different concerts on the same date, multiple venues for concerts on the same date that have the same seat label, or a combination of both. Seats that are in reality for different concerts could be seen by the database as the same seat.

To resolve this problem, I would map my seats to a specific concert. Since we can't have the same concert at different venues at the same time, our seats should be unique. Another resolution could be to give seats a different seat label for each different venue, but this doesn't solve the problem of seats on the same day for different concerts being seen as one seat.

## Support for "held" seats

My web service does not currently provide this support. There are two methods where we could extend the web service to provide it.

The first one and the one I recommend is by changing the IS_BOOKED column in a seat to an enumeration of SeatStatus with different statuses booked, free, and held. We would need to also store the DateTime startHeldTime that the seat has started being held in a different column. Where we check for if a seat is booked, we would now also need to check if it has been held. If it has, we then check if the difference between the current time and the startHeldTime is over the holding period, in which case we would treat the seat as free.

Another method is to lock the seat for the holding period and keep the lock with the client so that when they pay for the booking, they can update the seat to booked. The problem with this method is that it is not obvious to the system whether a seat is being held, so we cannot give an appropriate indication to the client about the status of the seat. Another problem is that the client may wait for a long time (the holding period) to get the lock for the seat. We could use a timeout for the client, but this raises other issues such as whether the timeout implies that the seat has been held, a server error, or a network failure.

# Extra Note

For my domain model, I understood that we should include Cascade annotations for our relationships. However, for the current functionality that our services provide, I believe that mappings in my domain model barely need any cascading. When more functionality is added to the system such as the ability to delete a concert, we can add the cascade logic easily, such as deleting all seats related to the concert using CascadeType.DELETE on the map between the concert and the seat.