shaf435

SE325 Assignment 1

Scalability

One of the ways scalability was improved was through a sub-select fetch mode when loading collections on the one/many-to-many side. In this fetch mode, associated entities can be loaded with a single round trip to the database, thereby minimising the number of DB accesses. As such, as the number of resource requests increase, the number of DB accesses will increase linearly and not overload the DB.

Another property which makes the service scalable is its statelessness. The web service implements the request-per-resource model, and as such there is no local state in the resources themselves (apart from the subscriptions). This would allow the service to be arbitrarily replicated to different machines and distributed across the world to provide load balancing as well, quicker access times, and increased availability.

Eager and Lazy Loading

In the Booking domain class, there is a Set<Seat> with a one-to-many association with Seat objects. This collection is eager loaded, with a sub-select fetch mode. Booking objects are accessed in endpoints to get all bookings, and get booking by ID. In both situations, the Seat entity is also accessed (to convert them to SeatDTOs). As such, eager loading them is appropriate to avoid the n+1 selects problem (in the get all bookings endpoint). To avoid large cartesian products, a sub-select fetch mode is used, which also helps with scalability as mentioned previously.

In the Concert domain class, the Set of LocalDateTime and Performer objects are fetched lazily with a sub-select fetch mode. These collections are lazily loaded since not every access of a Concert will also access its Performers and Dates (e.g. these are not accessed when getting concert summaries). As such, to reduce the amount of minimum Concert data loaded, these collections are lazily loaded. The fetch mode is sub-select for these collections for the same reason as above: reducing number of DB accesses whilst avoiding large cartesian products.

Concurrent Access

The only part of the system that could suffer from concurrency issues is in the booking process. To avoid this issue, when a Booking is about to be created, the Seat entities being booked are locked with an optimistic lock. To enable this, a version field was also added to the Seat domain class. With optimistic locking, if a double booking is about to occur, the transaction which creates the booking first (by updating the Seat objects to mark them as booked) will be the transaction that succeeds in booking. The transaction which commits second will fail since it will have a record version mismatch. This first-come-first-served model is appropriate since **valid** (correct concert, date, and unbooked seats) Booking requests should not be subject to any conditions; the User which requests the Booking first should be the one who receives it. If a transaction fails due to an OptimisticLockException (indicating that the seats have already been booked), then no further attempts are made at re-booking the seats; another User has booked them and they are unavailable to all other Users now.

This method of optimistic locking prevents the issue of double-bookings, and preserves a first-come-first-served model.

Extensions to Service

Different prices per concert

This assumes that all concerts still have the same 3 pricing bands, but with different price values. A new Price entity would be created which would contain price values for each band. Each Concert would include a Price entity as an attribute (as a FK)(in a many-to-one relationship). The “price” attribute would be removed from the Seat class, and replaced with a “band” attribute, which is a String indicating the price band of that Seat. To determine the price of a Seat, you would go to the Seat’s Concert, look at the Concert’s PriceBand, and determine the numerical price based on the Seat’s price band. Pricing for a concert can be modified by creating a new Price object with the price band values set as desired.

Multiple venues

This assumes a single Concert can have different venues for each date. A new Venue entity would be created. This entity would have venue-specific information such as name, location, no. of rows, and no. of columns (for seating layout). Another entity named ConcertVenue would be created which would have a composite key (which would be an @Embeddable class with venue_id and concert_id fields), a concert date, a reference to Venue, and a reference to Concert (with @MapsId setting the foreign key in the composite key). Essentially, ConcertVenue would track the many-to-many relationship between Concert and Venue with a relationship attribute of “concert date”. Then, each Concert would have a Set<ConcertVenue> which would track the different venues the Concert would have on each date. This creates a uni-directional one-to-many relationship between Concert and ConcertVenue.

Support for “held” seats

A new endpoint would be created to allow holding seats while processing payment (e.g. POST to “/hold-booking” with a suspended AsyncResponse); when a user selects their seat, they send the requested seats to that endpoint, and a Booking is persisted (provided the request is valid), and their seats are marked as booked with an optimistic lock. After the specified time period, the booking is deleted and the seats are marked as available again, and a 400 error code is returned. If they pay, then another POST request is made to the current “/bookings” endpoint, where the Booking made earlier (in “/hold-booking”) is fetched with an optimistic force-increment lock with an extended-scope and returned to the user. If the user pays within time, then the force-increment nature of the lock means that deleting the Booking in “/hold-booking” will fail due to an OptimisticLockException, thereby preserving the user’s booking despite a timeout.

If they cancel payment, then a DELETE request would be made to “/release-booking” (with the same BookingRequestDTO as the one used to hold booking). The Booking object matching the request DTO will be fetched and deleted, and all the held seats will be marked as available.