SE325 Assignment 1, cpat430, 194789725

Scalability

Scalability is an essential feature for a booking system, regardless of what for. Therefore, it should be able to deal with any range of users all booking at the same time. One issue with scalability is concurrency. A booking system will have many people all booking at the same time, so consistency is vital to making sure there are no issues with double-booking. An attempt to deal with this issue is using locking to ensure concurrent transactions don't affect each other. To do this, pessimistic locking is used in this web service.

Pessimistic locking means that any number of transactions can obtain the read lock and they can all read as much as they like whilst they have the lock. However, there can only be one transaction with a write lock, and then no other transactions can read or write when it holds the lock.

The map is synchronized and is a ConcurrentHashMap which both deal with concurrent access. Therefore when being access by more than one user, it will be thread-safe. A linked list is used for storing the subscribers because this means there can be O(1) insertion and O(1) deletion. This favours the subscriber system where they are removed once a notification is sent to them.

Aside from the subscriber map, the system is entirely stateless meaning there is no server-side synchronization logic. This allows the system to be replicated, and this can increase the number of potential users that can concurrently book seats/purchase tickets. All that would be required to achieve this would ensure consistency between the replicas.

By using HTTP through the web service, it implements caching, which in turn increases the overall performance of the website. Each time a user enters the website, they aren't required to send a request to the root server and back again. The process of finding a URL is timely, and by caching, it will save it in the local DNS server, which is much closer to the client.

Fetching

Firstly, within the booking domain class, the set of seats uses eager fetching. The purpose of eager fetching is when a booking is requested, the seats within the booking will be needed by the user. This will prevent a potential n + 1 problem as we know that the seats will be required when the booking is requested.

Secondly, for most of the other collections within the entities, lazy loading was preferred. Lazy loading is preferred for performance, especially when the main entity doesn't use the collections. I.e. the dates and the performers in the concerts aren't immediately required until the concert is clicked on. If eager was used for this, then it would load all the dates and performers of all the concerts at the start, but the user might only want to check out one concert. For example, when creating a concert summary, the dates and the performers aren't required. Therefore, it would be inefficient to load in all the dates if they weren't needed.

Finally, for the dates, we have a sub-select fetch mode which will get all the dates at the same time as getting one of the dates. The same goes for the performers. Sub-select is perfect for this because if none of the dates was requested, then there is no need to load any of the dates, whereas when one is loaded, all the dates would be required and only uses two queries to the database.

Concurrent Access

Double bookings can occur when two bookings are made at the same time, and both bookings request the same seats. Without any intervention, this will cause issues when both users show up at the door wanting to get into the concert after booking the same seat. This is dealt with by locking the seats. Locking the seats will prevent some booking requests getting the same seats that another user is wanting. If the seats are booked or currently locked, then the user will be notified that they must change their seats because it is currently locked. Throughout the code, pessimistic reading and writing were used to prevent multiple bookings writing to the seats at the same time. When no bookings are writing to it, then all the other transactions can read the data.

If the scalability were to increase, the number of venues could increase. To deal with this issue, a ConcurrentHashMap is used to store all the subscribers. The key to this is the LocalDateTime and this stores the AsyncReponse and the Subscription info. When a concert on a specific date changes, it will check this against all the subscriptions for that particular date. As the date stores all of the subscriptions for that date, it will check against the id.

Extensions

Different Ticket Prices

To support different ticket prices, I would consider creating a theatre layout object for each of the various concerts which would take costs for each of the seat types. The concert class would then have a constructor that had the option to add costs; otherwise, a default value would be used. The concert could then have 'tickets from $XXX', which will use the cheapest ticket price and this could be stored in the concert class. This will help the user to know whether they can afford the concert before clicking on it.

Multiple Venues

To deal with multiple venues, I would create a venue class which would have the venue name, capacity and the theatre layout. The venue information could be kept in the database and then retrieved whenever there was something hosted at that venue. The venues will be stateless to be scalable. The venues can be inputted using either an SQL insert or using a REST method which only allows authorized access to use. For the database, it would use eager fetching because when the venue is selected, the number of seats will be required at the same time.

Held Seats

To detect holding a seat, there must be somewhere we can retrieve held seats. The seats could be held in a database table called "HELD" which could be persisted into when a user was to select some seats. Once the user selected the seats, it can send a request to a /hold place. Now, if another user wants to book some seats when selecting and trying to proceed, it will check in the hold method to see if the seats are taken. If the seats are currently held, the current user would be notified that they should choose another seat or they can wait X minutes to try again. A time-to-live will determine the time that each user gets. The time-to-live will be specified by the admin and could be any reasonable amount of time that it would take to books seats. If at any point the time-to-live expires or the user cancels the payment, the seats will be available again.