This report will reflect on SoftEng 325 assignment 1 and discuss scalability, uses of lazy loading and eager fetching, concurrent access possibility and any future changes that will add additional features. The aim of this assignment was to build a Restful web service for a concert booking application. The application layer has been developed with JAX-RS implementation and RESTEasy, while the persistence layer is provided with JPA and Hibernate.

An important attribute required for projects such as this, where the intention is to provide a useful concert booking service for N number of people is scalability. As more people become interested in this service the service should handle a growing number of potential clients as well as keep up with spikes of request when popular concerts become available. The system may become unresponsive if it cannot keep up with these requestion giving customers a bad experience.

Scalability:

Web Services are horizontally scalable making them very scalable.

The CAP theorem states that availability, consistency, and partition tolerance cannot be achieved at the same time. Relational Database's provide solid, mature services according to ACID properties but has poor horizontal scalability due to poor partition tolerance. By giving up ACID, scalability can be increased, however databases tend to contain central state of the system and are not easily replaced. Thus, the Relational database bottleneck can be minimized by decreasing database strain. This is done by keeping database requests to a minimum.

In general, only the minimum amount of data necessary to perform the request was fetched using lazy loading.

Another scalability optimisation was made using an in-memory cache of commonly accessed (transient entities). This lowers the strain on the database by lowering database requests and the use of a non0persisted cache is justified for transient user sessions since the users can login again with minuscule cost if the session has been lost.

Asynchronous Responses were also used for the scalability of notification/subscribe updates. This meant that requests to the database could be made without holding up client traffic.

Long polling was used to ensure notification updates made an optimized minimum number of requests. Since one request is made for each response. Regular polling can make many requests without receiving a response. Thus, this decision increases the scalability of the system.

Fetch Strategies:

Two primary fetch strategies were taught during this course and thus considered for this project, lazy loading, and eager fetching.

Eager fetching loads an entity and all related entities at the same time. An advantage of eager fetching is that in best case scenario one call can be made. A disadvantage is that related entities might be redundantly loaded.

Lazy loading fetches an entity and proxies of related entities. An advantage of this is that the entire object graph is not loaded and in best case scenario does not redundantly load anything. A disadvantage is that many more database calls might be needed.

Examples of code and their justification in relation to the concert booking service are discussed below:

Extract from lectures: Persistent collections that are mapped with @ElementCollection, @OneToMany and @ManyToMany are, by default, lazily loaded

Concert Domain, Booking Domain and Performer Domain code examples are shown below.

```
    //Because we are using a collection with a many to many relationship it is standard to use lazy fetching.
    //When we load a date we only want to load the dates that relate to the concert not all the dates.
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "CONCERT_DATES")
```

```
//Because we are using a collection with a many to many relationship it is standard to use lazy fetching.
//When we load a Concert we only want to load the concert that relate to the performer not all the concerts.
@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
private Set<Concert> concerts;
```

```
//Because we are using a collection with a many to many relationship it is standard to use lazy fetching.
//When we load a seat we only want to load the seats that relate to the booking.
@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
@org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SUBSELECT)
private List<Seat> seatList;
```

Concurrent access possibility:

Because of high traffic it is almost certain that booking for the same seats will occur. If prevention methods are not taken double booking will occur, this is a major breach to the data integrity of the system. Overbooking can lead to more seats being booked then are possible leading to extreme customer dissatisfaction, bad reviews and issued refunds. To avoid these situations optimistic concurrency control will be used.

OptimisticLock exception ensures that the first transaction will succeed, and all subsequent transactions will fail. It does this by utilizing version control to impose a virtual lock (not a real database lock).

Examples of optimistic concurrency control and their justification in relation to the concert booking service are discussed below:

Optimistic Locking was used to change the last commit wins problem to first commit wins. It was used over Pessimistic Locking as the risks associated with pessimistic locking, deadlock, was deemed too dangerous as it results in a complete system crash. The tools associated with managing this risk are not covered. Also, Pessimistic locking is more heavy handed than optimistic locking. Although optimistic locking has more database requests.

On further assessment with the right implementation Pessimistic locking can improve scalability reducing overall database requests through a queue-like structure which only gives one request database permissions.

Future Changes:

Possible improvements to concert booking service are discussed below.

Different price, different concert:

Create a Ticket entity and assign the price attribute to this object. This object will reference a concert, seat, and date. This means a set price can be associated with specific concerts. In this case a Seat would be a physical seat in the venue and Tickets will reference exactly one seat, however seat could be referenced by multiple tickets. Thus, Tickets will have a one to many relationship with seats and one to one relationship with concert.

Multiple venues, venues, venues:

Create a Venue entity that references seats and concerts. Venue will have a one to many relationship with seats and concerts. A venue can reference many seats, but a seat only reference one venue and a venue reference many concerts, but a concert only references one venue.

If a concert can have different venues on different dates then the venue entity must contain a list of concerts and refer to LocalDateTime.

Seat reservation:

Create a Reservation entity that references a Seat and a User and contains a Expiry Time value. When the reservation has elapsed the Expiry Time value it will be removed from the database.

A booking attempt will now also check that there are no Reservations for a specified seat unless that User owns the Reservation. If a user books a reserved entity the Reservation will be removed from the seat and the seat will be made unavailable. Additionally, notification can also be updated to notify users a set time before their reservations expire.