SOFTENG 325

Assignment 1 Report

Name: Sahana Srinivasan

UPI: ssri365

AUID: 677618824

In web-based applications, it is very important that the time taken to return a response is reasonable. However, it is intuitive that when the number of users for an application increase, its performance time increases to accommodate the new load. By allowing room for scalability for a web-based application, more users are able to process requests in shorter spans of time. This implementation can be either on the hardware side of the server, or on the software side.

For the scope of this assignment, scalability is implemented in the software aspect with the help of asynchronous responses and by keeping the system stateless. It is possible to also scale a server using components like additional CPUs or other hardware with larger memory spaces to promote multitasking and hinder overloading.

Asynchronous REST responses in JAX-RS ensure that several sections of a request in a program are handled by different threads. The advantage of this feature is that more requests can be processed simultaneously as the threads free up faster. However, this feature is only relevant when there's more than one client accessing the application.

Statelessness in REST API means that the server has memory cached of the client's requests. Each client request must contain all the necessary for the handling of the request. It is similar to that of a Moore finite state machine, where output depends only on inputs and not its previous state. This feature can be taken advantage of to improve application scalability.

Smaller JPQL database tables also maintain brisk response times. JPQL database tables require memory so that they can be stored. Therefore, when the tables contain more records and become more detailed, they can be very expensive on memory resources. This expense can however be worked around with, by using lazy-loading over eager fetching.

Response time, and the performance of an application, also depends on the method of fetching used to acquire data from their relational schema. By default, most of the application relies on lazy-loading, where data is not fetched until a request triggers it. This operation saves quite a lot of memory space, by only loading sections of a domain model at a time. As a result, when multiple users access the web application, server loading is minimised.

It can also be noted that for the concert reservation application, certain fields of the JPQL tables, like all the concert dates while retrieving a concert, requires eager-fetching to be implemented on it for correct functionality of the application. Only when such an explicit call is created can the user be allowed to view all possible options they can reserve seats for.

Double-bookings, where one or more users want to book the same seats, is well-implemented for the scope of this concert reservation application. Since testing only happens on one machine, only one user can be authenticated at a time. Therefore, when a user makes a reservation, it is processed before another user wants to book those same seats, in which case the latter request is not processed as the selection is now occupied.

For concurrent access, however, where multiple users want to book the same seats at the same time, provision has not yet been made. It is possible to create such implementations due to the RESTful web service being stateless, which implies other methods of scalability could be applied in the future. Such requests can be dealt with by using lock modes like Pessimistic_Read, or Pessimistic_Write on the required query, which has not been taught in the course yet and could be an additional implementation if there wasn't a restriction on time.

Different ticket prices for different concerts can exist if the individual prices were inserted as another JPQL table column in the generated database tables. However, that would increase the memory the server would need to reserve in order to create the tables on which reservation queries can be processed.

To support multiple venues at which concerts could be held, one way could be initialising multiple layouts in the Utils class folder. The different venues could be value type classes and related to one another using the annotation *@EmbeddedId* and *@Embeddable* annotations on the respective domain classes. A corresponding DTO could ensure mapping between the domain class on the server and the client.

In order to create a time-restriction, during which a reservation can be blocked but must be released when the time elapses, the LocalDateTime class can be used to calculate the time of reservation. Since the response is asynchronous, an implicit counter could run in the background of the application for a certain period of time, after which it can time out using the setTimeOut method. Then, the seats could be released. This is one of the possible implementations.