SE 325 Scalability Report

elee353, 840454023

There are several aspects that improve the scalability of my design.

Separating NewsItemResource class from ConcertResource class promotes horizontal scaling by allowing duplicate servers to run the same service. The ConcertResource class is stateless while the NewsItemResource is not. Encapsulating the states only in the NewsItemResource class promotes scalability because the ConcertResource class can be scaled separately, while NewsItemResource still maintains its necessary state information.

Many aspects of the program are stateless, but the news item component has to be stateful. Asynchronous response is utilised to increase the scalability of this particular module. With the asynchronous response, the client is not blocked while waiting for a response. Different threads may be assigned to the same request at different times with asynchronous responses. This enables the server to use limited threads to process more responses to improve scalability. In contrast, the same thread is devoted for the lifetime of the request with synchronous responses.

Fine-grained access control promotes scalability. In my design, the Performer and Concert domain classes do not store references to the bookings and reservations. This achieves fine-grained access control by preventing the lockout problem. The lockout problem can happen in a coarse-grained access model. An entire table can be potentially blocked out by an increased number of users trying to book and reserve concerts. An example of this is a model which a concert class contains seats. This model allows more users to reserve and book concerts without having to block out some user when demands increase.

My concert application utilises a three-tier architecture. In a three-tier system, the three main components involved are the client, server, and database. A three-tier architecture is more scalable because an application server can be used to cache persistent data to increase performance and scalability.

In general, the fewer database calls, the better the performance. Caching has been used to achieve this objective. When the first time the user tries to download the performer images, the images are downloaded and cached in the DefaultService. The cached images are retrieved when needed later, instead of downloading them from the AWS Bucket again. Another example of caching is where some domaining class objects are cached in DefaultService. This promotes scalability because the database calls are reduced by retrieving the cached objects instead. Using the cached objects also improve response time by reducing database round trip time and reduce the number of database calls.

Optimistic concurrency control (OCC) is used to promote scalability. It allows fast performance and high concurrency when access by multiple users, at the cost of occasional conflicts. Pessimistic locking requires overhead for every operation, whether or not multiple users are actually trying to access the same record. The overhead is small, but adds up because every row that is updated requires a lock. Furthermore, every time that a user tries to access a row, the system must also check whether the requested rows are already locked by another user or connection. In a highly concurrent system such as our concert application, these locks can impact scalability.

The ‘first commit wins‘ characteristic of OCC matches the ‘first in first served' nature of booking a seat in the concert. Multiple users can attempt to book the same seats at the same time, and the first user who completes confirmation can successfully own the booking. If pessimistic locking is used, only one user can attempt to book seats each time, while others are unable to the same table. Furthermore, it is also common for a user to start booking, and then leave without completing the entire process. If locking is used, the selected rows will stay idle for a long time until the first user's lock is released. This will significantly impact scalability.

In the send() method inside the NewsItemResource class, an Executor instance is used. The Executor instance is instantiated by a newSingleThreadExecutor() call. This method guarantees the tasks execute sequentially, and no more than one task will be active at any given time. This is suitable for the concert news as the number of news items should not expand rapidly and only a single thread can send the news item. Computing resources such as threads can be used for other computing tasks.

At the moment, there is a single thread instance that times the five-second session in the Default Service class. This ensures only one thread instance is active each time and prevents excessive amounts of thread to be created to drain the thread resources. In the future, this can be done by logging the start and end sessions of booking and comparing if the difference is within 5 seconds. This will minimise the use of the thread resources, hence increasing scalability.

Lazy loading is used to improve scalability in my design. In most situations when a user is booking a concert, the user is likely to view the list or overview of the concert attributes instead of the very fine-grained attribute details. Therefore, lazy loading allows the fetching of a relationship to be deferred until it is accessed. Lazy loading helps to prevent the Cartesian product problem by avoiding loading of every related object through the relationship hierarchy. This prevents wasting memory and computation resources on loading unused objects.