

Software Architecture Design Report

Submission 3 SAD

NüRoom

SWEN90007 SM2 2021 Project



NüROOM

Made for You

In charge:

| Name | Student ID | UoM Username | GitHub Username | Email |
|----------------|------------|--------------|-----------------|--|
| Guo Xiang | 1227317 | XIAGUO2 | moneynull | xiaguo2@student.unimelb.edu.au |
| Basrewan Irgio | 1086150 | IBASREWAN | irgiob | ibasrewan@student.unimelb.edu.au |
| Bai Zhongyi | 1130055 | zhongyib | Marvin3Benz | zhongyib@student.unimelb.edu.au |
| Hernán Romano | 1025543 | hromanocuro | hromanoc | hromanocuro@student.unimelb.edu.au |

Release Tag: SWEN90007_2022_Part3_Nuroom



SCHOOL OF
**COMPUTING &
INFORMATION
SYSTEMS**

Revision History

| Date | Version | Description | Author |
|------------|-----------|---|----------------|
| 02/10/2022 | 01.00-D1 | Initial document | Hernán Romano |
| 05/10/2022 | 01.00-D2 | Restructure of document | Team |
| 06/10/2022 | 01.00-D3 | Adding summary of concurrency issues | Hernán Romano |
| 07/10/2022 | 01.00-D4 | Updated class diagram | Zhongyi Bai |
| 08/10/2022 | 01.00-D5 | Added testing info for BookHotel and ModifyHotel | Irgio Basrewan |
| 09/10/2022 | 01.00-D6 | Adding Optimistic Lock description | Hernán Romano |
| 10/10/2022 | 01.00-D7 | Added description for concurrency issues | Zhongyi Bai |
| 12/10/2022 | 01.00-D8 | Added implementation of lock | Xiang Guo |
| 13/10/2022 | 01.00-D9 | Added implementation of DB constraint | Xiang Guo |
| 15/10/2022 | 01.00-D10 | Added testing info for use CustomerSignUp and CreateHotel | Irgio Basrewan |
| 16/10/2022 | 01.00-D11 | Adding Pessimistic Lock description | Hernán Romano |
| 17/10/2022 | 01.00-D12 | Added testing info for ModifyBooking and RemoveHotel | Irgio Basrewan |
| 19/10/2022 | 01.00-D13 | Updated conclusion for the testing | Zhongyi Bai |
| 19/10/2022 | 01.00-D14 | Updated system class diagram | Xiang Guo |
| 20/10/2022 | 01.00 | Version 1 & Create Release tag | Team |

Contents

| | |
|--|-----------|
| 1. Introduction | 3 |
| 1.1 Proposal | 4 |
| 1.2 Target Users | 4 |
| 1.3 Conventions, terms and abbreviations | 4 |
| 2. Actors | 4 |
| 3. Expanded Use Cases | 4 |
| 3.1 Use Case Diagram | 5 |
| 3.2 List of Use Cases | 6 |
| 4. Class Diagram | 7 |
| 4.1 Description | 7 |
| 5. Concurrency Issues | 9 |
| 5.1 Summary table | 9 |
| 5.2 Optimistic Offline Lock | 11 |
| 5.2.1 Description | 11 |
| 5.2.2 Implementation Details | 12 |
| 5.2.3 Testing Strategy & Outcome | 15 |
| 5.3 Pessimistic Offline Lock | 16 |
| 5.3.1 Description | 16 |
| 5.3.2 Implementation Details | 18 |
| 5.3.3 Testing Strategy and Outcome | 20 |
| BookHotel | 21 |
| 5.4 Database Constraints | 24 |
| 5.4.1 Description | 24 |
| 5.4.2 Implementation Details | 25 |
| 5.4.3 Testing Strategy and Outcome | 26 |
| 6. References | 28 |

1. Introduction

1.1 Proposal

This document specifies the SWEN90007 project's detailed use cases, actors, updated domain model, Class Diagram and the description of each Design Pattern implemented in the system.

1.2 Target Users

This document is mainly intended for SWEN90007 students and the teaching team.

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

| Term | Description |
|----------------|--|
| UML | Unified Modelling Language |
| Hotelier Group | A set of Hoteliers under a common hotel business interest |
| Amenity | A desirable or useful feature or facility of a hotel or room |

2. Actors

| Actor | Description |
|---------------|--|
| Customer | A person who makes use of the Hotel Booking System and acquires services |
| Administrator | A person who manages the Hotel Booking System |
| Hotelier | A person who manages hotel(s) |

3. Expanded Use Cases

3.1 Use Case Diagram

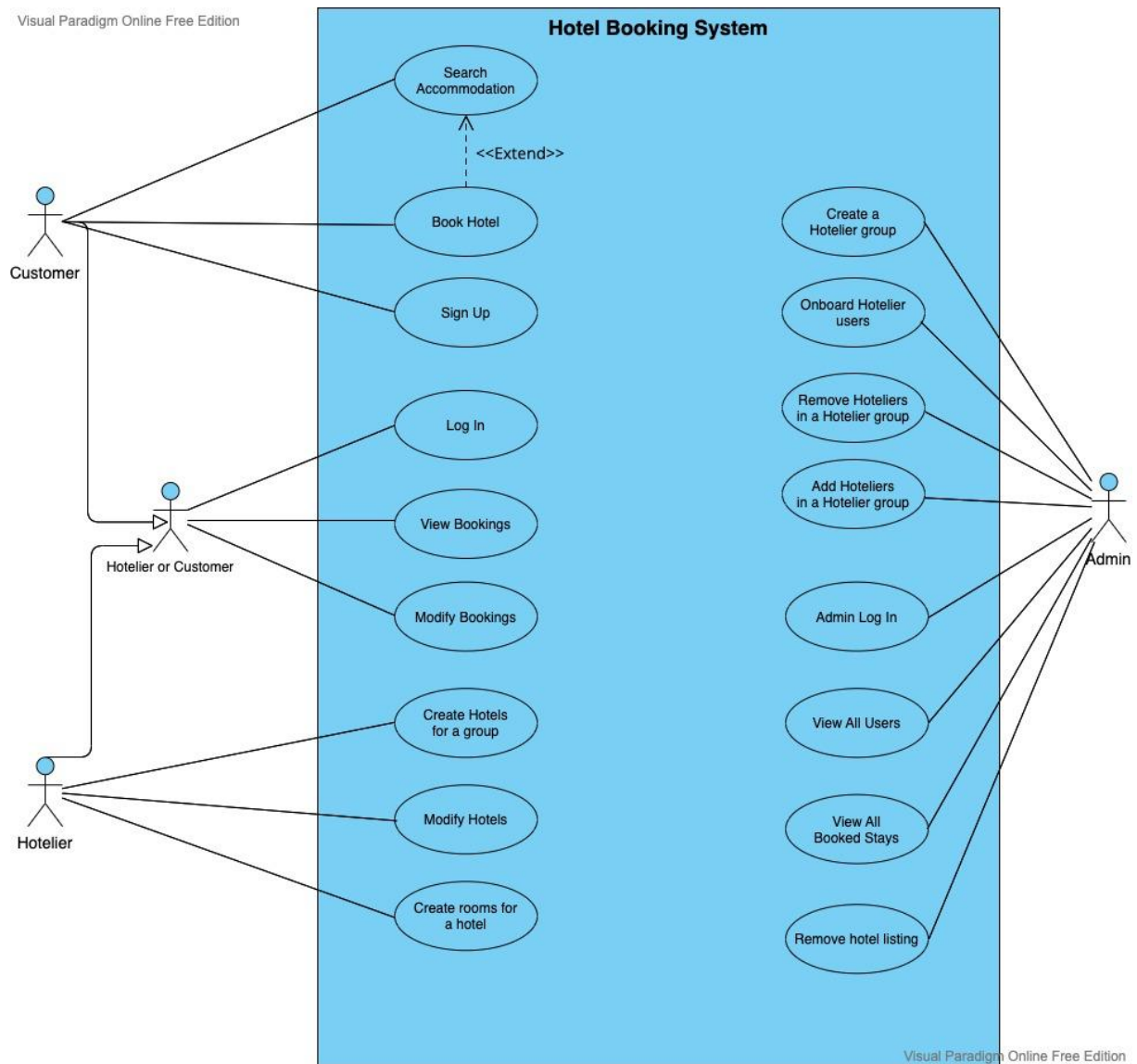


Figure 3.1: Use Cases Diagram

3.2 List of Use Cases

The **Table 3.1** shows a summary of the use cases considered in the project:

Table 3.1: List of Use Cases

| Use Case ID | Use Case Name |
|-------------|---|
| 01 | Customer Signs Up |
| 02 | Customer Searches for accommodations |
| 03 | Customer Books Hotel Rooms |
| 04 | Hotelier or Customer Logs In |
| 05 | Hotelier or Customer Views Bookings |
| 06 | Hotelier or Customer Modifies Bookings: Alternative flows(Add more people, Changes dates, Cancel Booking) |
| 07 | Hotelier Creates Rooms for a Hotel |
| 08 | Hotelier Creates Hotels for a group |
| 09 | Hotelier Modifies Hotels |
| 10 | Admin Logs In |
| 11 | Admin Remove hotel listing |
| 12 | Admin Views All Users |
| 13 | Admin Views All Booked Stays |
| 14 | Admin Onboards Hoteliers |
| 15 | Admin Remove Hoteliers in a Hotelier group |
| 16 | Admin Adds Hoteliers in a Hotelier group |
| 17 | Admin Creates a Hotelier group |

4. Class Diagram

4.1 Description

In our project, we mainly have 3 layers, the Controller/Service layer, the Domain layer, and the Data Source layer. The business logic flow and authentication will be handled in Controller/Service layer. In the Domain layer, we declared all objects that will be used in the system. We put all necessary SQL queries in our Data Source layer so that we can simply invoke the methods to insert/update data in the database. Using these layers, our system architecture will be clearer and have high modularity.

In this updated system class diagram, we added the Concurrency package, which includes two lock managers: **HotelLockManager**, and **BookingLockManager**. These two lock managers can help us properly manage the transactions with concurrency issues.

As **Figure 4.1** shows, SecurityFilter class is involved in every controller package because most controller classes use SecurityFilter's doFilter to check authentication and limit the actions allowed per user type, therefore improving the security of our system. All classes in controller packages (e.g. hotelControl, accountControl, etc.) consume different services (e.g. accountService, roomService, etc.) in the service package. Each service class invokes a mapper (e.g. UserMapper, RoomAmenityMapper, etc.) in the data source layer. All mappers inherit DataMapper class. Additionally, RemoveHotel, ModifyBooking, and BookHotel class acquire lock from **BookingLockManager**. HotelMapper acquires lock from **HotelLockManager**.



5. Concurrency Issues

5.1 Summary table

As recommended by teaching staff on [edForum](#), and considering the expertise on the source code of the application, the team decided to adopt the “Hazard and Operability Study (**HAZOP**)” approach to identify test cases that can be impacted by concurrency access to the application by our users.

To evaluate each available action allowed by the platform, we broke up the overall design of the Hotel Booking Process by following a structured tree, which is based on the available Business transactions allowed per user type.

Additionally, The **HAZOP** technique encouraged the team to have multiple meetings to discuss and stimulate our imagination to identify any potential concurrency problem that could be generated in the application. Therefore, we updated the possible Business transactions at each meeting and discussed any new or old scenarios.

Table **5.1** shows a summary of the concurrency scenarios analysed.

Table 5.1: Identified concurrency issues

| Actor | Business transaction | Use Case ID | Issue | Design pattern | Justification |
|---------------|---|-------------|--|--------------------------|--|
| Customer | Book a Hotel | 03 | Multiple Customers try to book rooms in the same hotel, filling the capacity of the Hotel. | Pessimistic Offline Lock | Correctness over liveness |
| | Modify a booking: <ul style="list-style-type: none"> • Adding more people to an existing booking • Changing dates | 06 | Multiple Customers could modify the booking dates, this can lead to Customers selecting the same periods at the same time for a particular Room. | | |
| | Sign Up | 01 | Multiple Customers try to sign up simultaneously with the same email. | Database Constraints | Concurrency is managed by the database |
| Hotelier | Create a hotel for a group | 08 | Multiple Hoteliers can create the same hotel group at the same time. | | |
| | Modify a hotel | 09 | Multiple Hoteliers can modify a Hotel at the same time with the same hotel properties. | Optimistic Offline Lock | Liveness over Correctness |
| Administrator | Remove a Hotel | 15 | A hotel can be removed from the listing while a Customer is booking a Hotel | Pessimistic Offline Lock | Correctness over liveness |

5.2 Optimistic Offline Lock

5.2.1 Description

The optimistic lock design pattern helped us to implement concurrency control by validating the changes about to be committed in one session do not conflict with changes in another. If they do, the changes cannot be committed, and the business transaction fails. It does this by confirming that, since a record has been loaded, no other session has changed that record.

The team decided to implement an Optimistic lock after analysing each of its properties for the selected business transactions shown in **table 5.1**.

- Conflict detection vs Conflict prevention:

Allowing concurrent customers to progress with a business transaction and detect any conflict at the end of it is a risk we can take for specific actions which are not critical for the business.

For this reason, **we decided to use Optimistic Lock in such cases where the chances of conflicts due to concurrent sessions are low**, such as the case where a Hotelier tries to modify a specific Hotel. In the case of modifying a hotel, two simultaneous update requests to the same hotel would most **likely be the same update** anyway, so the chance of two or more conflicting simultaneous requests

- Liveness vs Correctness:

Certain business transactions are not critical to keep the Customer experience at a high standard and do not directly impact the business. For this reason, certain non-critical use cases could benefit by allowing multiple transactions access to the same resource by keeping track of the version of the accessed resource. In this manner, **everyone could load the data simultaneously (liveness), but when writing back to the database, the transaction could be rolled back if the version differs**.

Cases such as modifying the profile properties of a Hotel such as a name, street and others are not as critical as modifying a booking characteristic which could impact multiple Customers concurrently, degrading in this way the user experience and provoking economic loss to the business due to a poor user service experience when not being able to modify a booking by then of its transactions.

For this reason, **we decided to use Optimistic Lock in cases where a transaction is not critical for the business** and where liveness is more important than correctness, **a higher throughput** (transactions per second) **will be expected, as Professor Maria Rodriguez mentioned** during the lecture of Week 10, Oct 5th during minute 49 of the video lecture in LMS.

- Version persistence

In order to keep the versions of each shared resource for certain business transactions, we decided to persist the version information in the database, in this way the system can confirm that, since a record has been loaded, no other session has changed that record.

For this, each user request will read and return the version number when triggering a business transaction.

After discussing each property, we decided to control concurrent transactions with Optimistic lock for the following Business transaction:

- **Modify Hotel**

When more and more hoteliers come to our website to host their hotels and rooms, it may raise a concurrent issue that happens when modifying hotel. Although it is rare to see hoteliers frequently modify hotel information, we still need to consider this scenario. When multiple hoteliers try to modify the same hotel simultaneously, a concurrency issue occurs: all hoteliers receive a successful modification message, but the database only stores one of the modifications. It will confuse some hoteliers since they get successful messages but their modifications are not updated in the database.

The reasons why we don't use a pessimistic lock for this case:

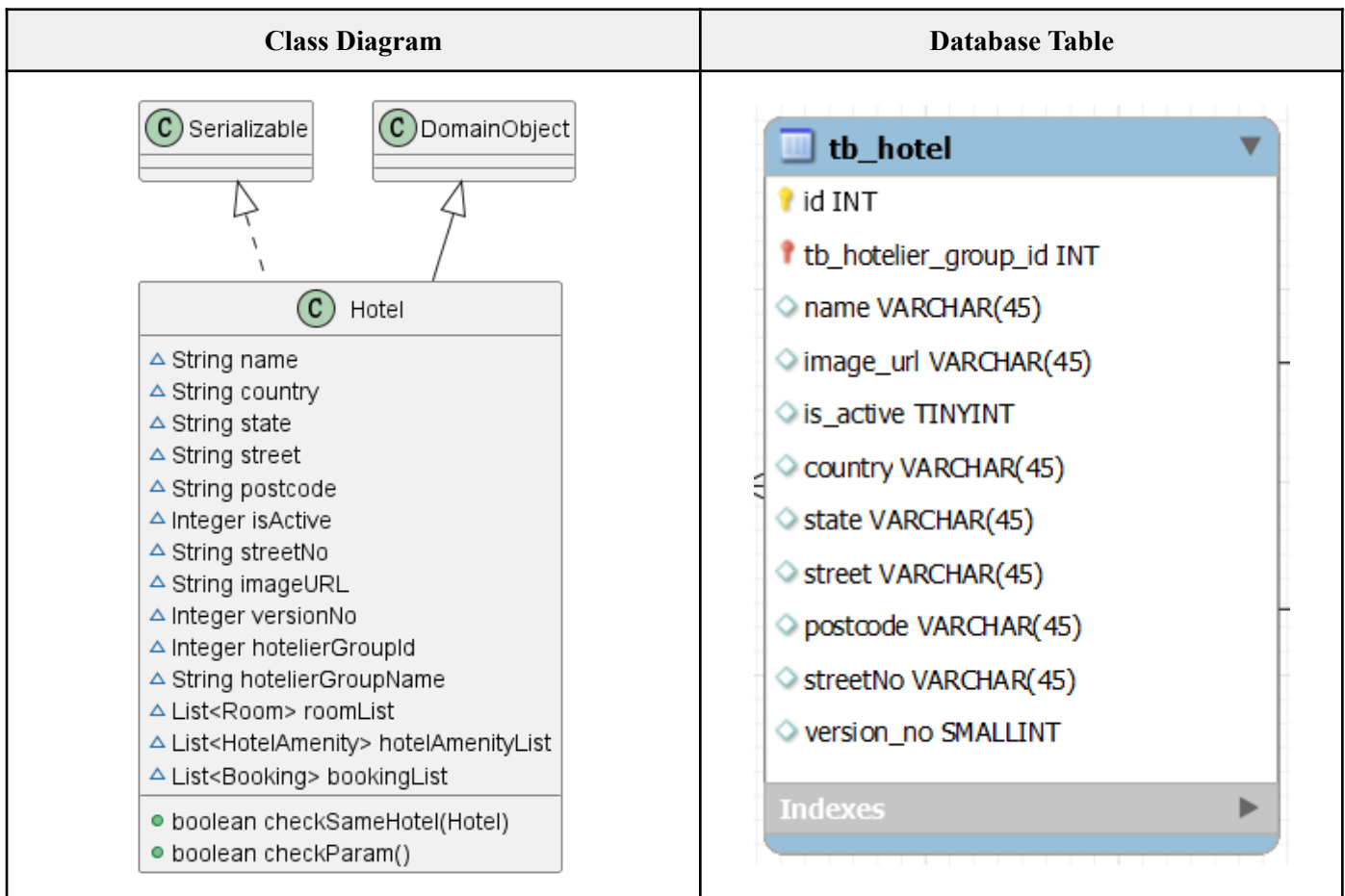
- Using a pessimistic lock will impact the usability of modifying a hotel. If multiple hoteliers try to modify the same hotel, they all have to wait, which is a bad user experience.
- The correctness of the hotel's information is not as important as the booking. The hotelier can simply update the hotel's information again if it is failed to update. Next time customers search for the hotel or view their bookings, they can easily get the latest information about the hotel.

5.2.2 Implementation Details

To implement this optimistic lock, we make several steps:

Step 1: Add the version number to the table and class of hotel

The version number will be used for checking whether the hotel is being modified or has been modified by other hoteliers.



Step 2: Modify SQL statement and mapper function logic of updating the hotel

Adding 'version_no=?' as an extra condition to guarantee the SQL statement could only update the current version of the requested hotel. Besides, the count of the updated rows could be used to capture the concurrent update's failure.

Therefore, the process of optimistic offline lock implemented in modify hotel shows below, together with the sequence diagram, **figure 5.1**:

1. The two hoteliers will get the same hotel data with the same version
2. Hotelier A and Hotelier B send the *modify hotel* request concurrently
3. The DB will receive two SQL update statements concurrently, but DB will handle these two statements respectively
4. The first one executed by DB will succeed, and the version number will be plus one.
5. The latter one will update nothing because the version number has been updated and return 0 as an updated result and response "wrong version number".
6. The Hotelier who got the "wrong version number" error could refresh the website to get the newest hotel data with the newest version number. And then the Hotelier could try to modify the hotel again.

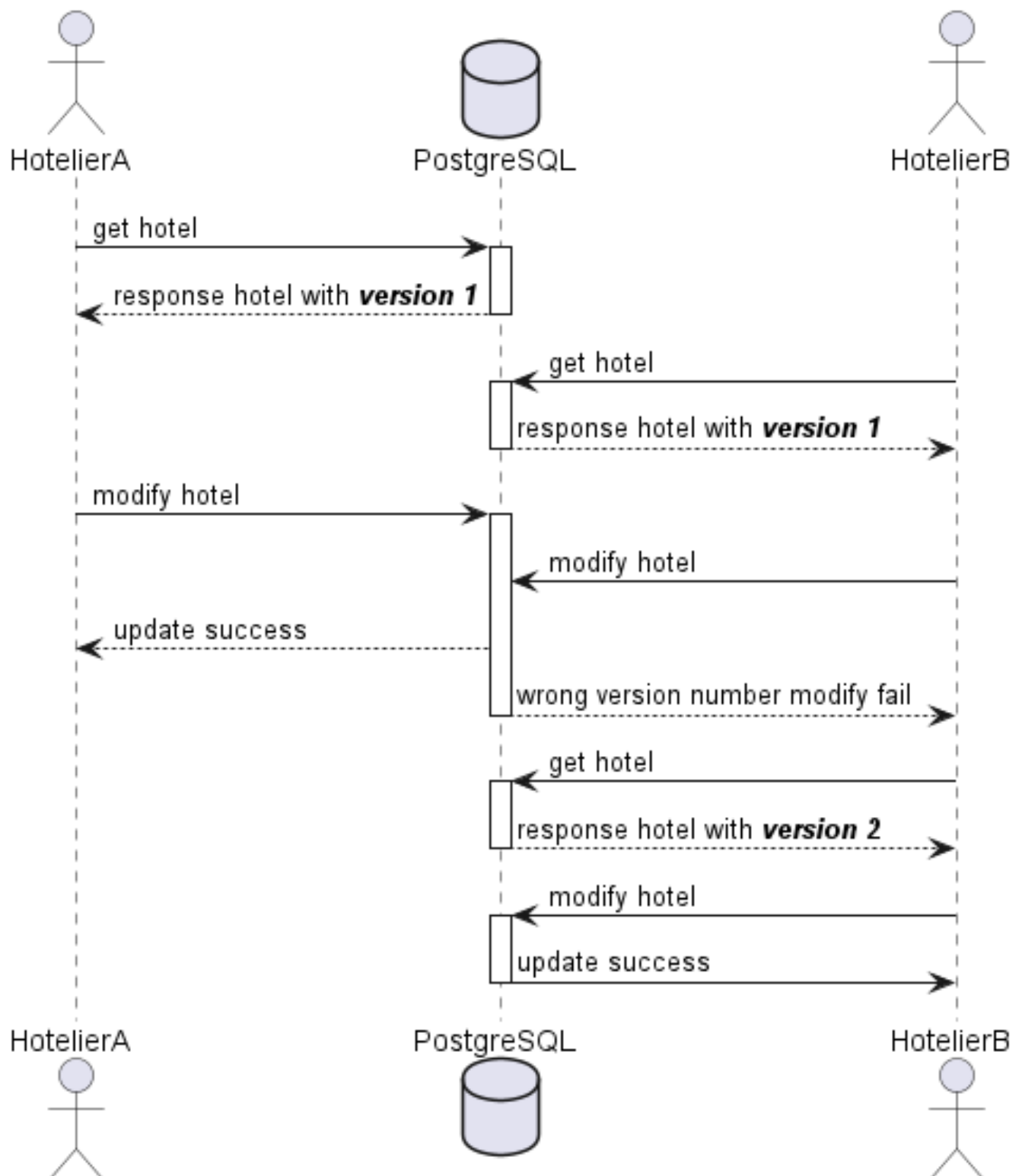


Figure 5.1: Sequence Diagram - Optimistic Offline Lock

5.2.3 Testing Strategy & Outcome

As recommended during the lectures, **we decided to adopt JMeter** as the primary tool to send concurrent requests to test the proposed concurrency implementation of our application. Additionally, we focus our efforts on **testing edge cases** for each domain input value **based on the limit of 20 concurrent connections allowed by our hosting provider Heroku**. [6]

After implementing Optimistic Lock, when multiple hoteliers modify the same hotel at the same time, and only one hotelier can successfully modify the hotel's information, this hotelier will get a success message. Other hoteliers will get an error code that indicates they failed to modify the hotel, and need to refresh the page to get the latest data and then modify again.

To test this, we ran experiments with a single update, ten concurrent updates on the non-concurrent version of the system, and multiple concurrent updates on the concurrent version of the system. For the concurrent version of the system, we did tests with varying amounts of concurrent requests for boundary testing, including 2, 10, 20, and 21 requests. Each request has roughly the same body content with varying values (marked by `${}`) extracted from a CSV file to help check if the final request is the one that was actually kept.

```
{
  "id": 7,
  "name": "${name}",
  "hotelierGroupId": 1,
  "country": "${country}",
  "state": "${state}",
  "street": "${streetName}",
  "postcode": "${postcode}",
  "imageUrl": "",
  "isActive": 1,
  "streetNo": "${streetNumber}",
  "hotelAmenityList": [
    {
      "id": 100042,
      "name": "Lounge",
      "description": "Nam dui.",
      "imageUrl": "",
      "hotelId": 7
    }
  ]
}
```

We expect that the non-current version of the system will accept all incoming update requests, which may not be ideal as hoteliers will not know if another hotelier had also updated the hotel at the same time, and they will not be able to track which update was able to actually succeed (since they all say they do). The concurrent system should block all requests while one request is currently being processed, avoiding this concurrency problem. We can see this behaviour validated by our tests done

in Jmeter. Errors are defined as when requests are rejected, either due to the server being busy or because another request is currently updating the hotel. As such, for every block of simultaneous requests, only one request should not return an error. The results of the test are recorded in **table 5.2**:

Table 5.2: Modify Hotel Jmeter Testing Result

| Result | Experiment 1: Single update without concurrency | Experiment 2: Multiple updates without concurrency | Experiment 3: Multiple updates with concurrency | | | |
|--|--|--|--|-------------|------|------|
| Number of requests | 1 | 10 | 2 | 10 | 20 | 21 |
| Avg Time (ms) | 1991 | 2036 | 1364 | 1342 | 1803 | 1507 |
| Succeeded | 1/1 | 9/10 | 1/2 | 1/10 | 1/20 | 1/21 |
| Only one receive successfully update result? | Yes | No | Yes | Yes | Yes | Yes |

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|---|-----------|---------|------|------|-----------|---------|------------|
| One update to one hotel | 1 | 1991 | 1991 | 1991 | 0.00 | 0.000% | .50226 |
| 2 concurrent updates to one hotel (w/ concurrency) | 2 | 2036 | 2036 | 2037 | 0.50 | 50.000% | .98135 |
| 10 concurrent updates to one hotel (w/ concurrency) | 10 | 1364 | 1341 | 1411 | 21.30 | 90.000% | 7.05716 |
| 20 concurrent updates to one hotel (w/ concurrency) | 20 | 1803 | 1265 | 3169 | 789.06 | 95.000% | 6.30517 |
| 21 concurrent updates to one hotel (w/ concurrency) | 21 | 1507 | 1324 | 1584 | 65.46 | 95.238% | 13.24086 |
| 10 concurrent updates to one hotel (no concurrency) | 10 | 1342 | 1242 | 1383 | 39.81 | 10.000% | 7.22543 |

From **table 5.2** we can conclude:

- Without an optimistic offline lock, 90% of hoteliers will get a successful update message, but actually only one of the modifications is updated in the database, which raises inconsistent results to hoteliers.
- On the other hand, with an optimistic offline lock, only one hotelier gets the successful update result which matches the real database update. Using optimistic offline lock helps us to offer a consistent result to all hoteliers.

5.3 Pessimistic Offline Lock

5.3.1 Description

The pessimistic offline lock design pattern helped us to implement concurrency control by locking the required data over the entire business transaction. Thus, it prevents conflicts between concurrent business transactions as it only allows one business transaction to access the data at a time.

The team decided to implement Pessimistic offline lock after analysing each of its properties for the selected business transactions shown in **table 5.1**.

- Conflict Detection vs Conflict Prevention:

Maximizing the isolation of a business transaction by preventing any conflict and locking down every resource used from any other session will ensure the correctness of the transaction and ensure data integrity.

For this reason, **we decided to use the Pessimistic Lock design pattern in such cases where the chances of conflicts due to concurrent sessions are high**, such as the case where multiple Customers try to modify a booking, **even though it limits the system's concurrency capacity**.

- Liveness vs Correctness:

Certain business transactions are critical to keep the Customer experience at a high standard and directly impact the business running as usual. For this reason, certain critical use cases must keep data integrity at a high standard by **preventing access (correctness) to a shared resource in use, allowing a single transactions access to the resource by locking the resource and not allowing any other session access to the same result until every system transaction is finished**. In this manner, we keep the data consistent even though a user should wait until the shared data is available again.

Cases such as modifying the profile properties of a Hotel such as a name, street, and others are not as critical as modifying a booking or booking a hotel characteristic which could impact many Customers simultaneously, degrading in this way the user experience and provoking economic loss to the business due to multiple customers accessing the data but not being able to ensure that its booking is being accepted when finalising its business transaction.

In this regard, we considered applying the Pessimistic Offline Lock design pattern for the following business transactions:

- **Book Hotel**

It is common to see that before a holiday, many travellers will book accommodation for their lovely trips. It brings a situation that thousands or millions of customers may try to book hotels at the same time, which will cause concurrency issues to the system. The possible concurrency issue is that the rooms can be overbooked.

For instance, when two or more customers try to book the same room, and we need to check the availability of this room, if we don't have a proper mechanism to manage these two transactions, these two will read the same availability (e.g. 1 room available), and all book successfully. However, we should only allow one room to be booked, and the remaining bookings should be failed.

- **Modify Booking**

When many customers are trying to change the check-in/out date at the same time, a possible concurrency issue can occur. Some rooms may be overbooked, for example, two or more customers try to change their check-in/out date to 2022-12-25 — 2022-12-27 with Room A in Hotel B. The original available rooms for Room A is 2, if these customers change the check-in/out date simultaneously, they all read the same availability of this room and if they all book successfully, this room is overbooked.

It might be fine if only a few rooms are overbooked, the support team need to contact one of two customers and tell them the bookings are invalid and they need to rebook other rooms. However, considering there are thousands of rooms are overbooked, it will be a disaster for the support team. Thus, we need to manage the “modify booking” transaction properly to prevent the rooms from being overbooked.

- **Remove Hotel**

When the admin tries to remove a hotel, several customers may try to book a hotel at the same time. This situation will cause a serious problem if customers book an inactive hotel successfully. Because the customers will not know the hotel has been inactive until they check-in, in other words, those bookings are invalid but still ongoing.

This problem will dramatically affect user experience. Imagine a customer who has prepared everything for a trip, but when he comes to the hotel with his family, he is told that this hotel is not available anymore. It is definitely a very bad experience for our customers. Thus, we need to manage the “remove hotel” transaction properly to prevent customers from booking an inactive hotel.

5.3.2 Implementation Details

To implement a Pessimistic offline lock for each of the Use Cases explained in **section 5.3.1**. To explain the implementation details we will take “Book Hotel” as an example of the steps, we made to properly implement it:

Step1: Choosing the exclusive write lock

First of all, we thought the most important point is to **keep the correctness and integrity of data while the customer is booking the hotel**, we need to guarantee that customer will not overbook the room of the hotel.

Besides, according to our business logic, we need to check whether there are still enough rooms before having a new booking (insert a new record in the booking table). And we are using SQL with grouping by the booking room table to get the available rooms. Thus, we need to implement the **exclusive write lock** to avoid the inconsistent data check caused by the new insertion of new records.

The reason why we do not use an exclusive read lock is if we implement an exclusive read lock, it will dramatically reduce the usability of the entire system. The exclusive read lock will prevent inconsistent reads; however, it is common to see some customers booking a hotel while some other customers are searching for the hotel. At that time, it may raise some inconsistent data related to room availability. If we prevent inconsistent reads, the search feature will be impacted (e.g. wait longer time), which is not acceptable from the customer perspective. The read process should not be locked; in that case, even the result might be inconsistent. From the customer’s view, getting search results faster is more significant than getting slight wrong results. So exclusive read lock is not suitable for this case.

The reason why we do not use read/write lock. In a read/write lock, a write lock can only be obtained if no process is reading. However, we need to allow the process to acquire the write lock while other process is reading the data. Similar to what we discussed above, we need to

ensure the search functionality is not impacted when some customers book hotels. If we implement a read/write lock, an extreme scenario could happen: the customers can never book a hotel just because there are always some other customers searching for accommodation. So we cannot use a read/write lock.

Step 2: Define the lock manager

Next, we define a lock manager based on the exclusive write lock we decided. The **figure 5.2** shows the class diagram of the Lock Manager:

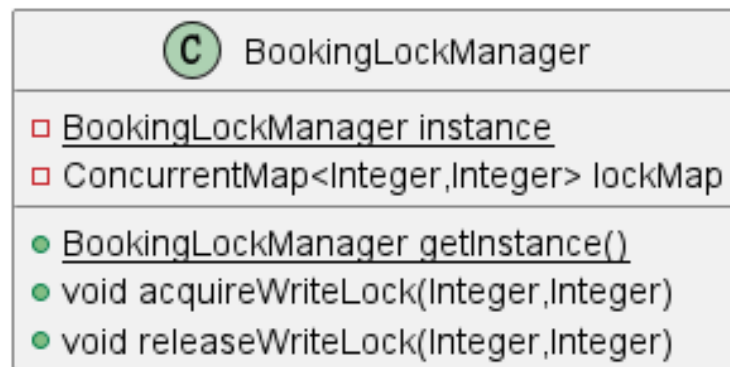


Figure 5.2: Class diagram of booking lock manager

We decided only to use a **concurrent map** to store the lock in JVM temporarily to avoid frequent access to the database to insert and delete the lock. Moreover, the structure of the concurrent map contains **two integers**: the first integer takes the hotel ID as lockable, and the second integer takes the User ID as the owner of the lock.

For the owner, according to our structure that we are using restful API and developing Frontend separately from Backend, it is better and clearer for us to allow each User holds the lock. This also makes the User have such responsibility to release the lock as well.

Step 3: Define the lock protocol

The lock protocol for the booking hotel business transaction is to lock at the beginning, lock the hotel requested by the user, make the user become the owner of the lock and release the lock at the end of the business transaction.

The reason for acquiring a lock at the start of the controller and releasing the lock at the end of the controller is that every API will only execute only one business transaction. But locking the whole business logic and making it execute serially and influence the usability. Thus, we decided to lock each hotel requested by the user and make the hotel ID as the first integer in the concurrent map as lockable, and whenever the concurrent map contains the hotel ID, the other requests on the same hotel will wait until it is released by the previous user.

Based on the consideration of the request order from users, the system should at least allow users to wait in line instead of getting a “*concurrency exception*” directly while concurrency happens. So, we decided to use the strategy of waiting until the lock can be acquired instead of rejecting the request from the other users.

Therefore, the following steps show the process of pessimistic offline lock together with a sequence diagram, **figure 5.3**, below:

1. Two users send the request to book a hotel concurrently
2. The first request will start the business transaction and acquire a lock on the hotel from the lock manager
3. The later request acquire a lock on the same hotel, but the lock manager will let this request wait until the first request to release the lock because the same lock has already been granted
4. After the first business transaction has been completed, it will allow the lock manager to release the lock, and the lock manager will notify the next waiting request and grant the lock.
5. The request which gets the lock could continue to complete the business transaction and then tell the lock manager to release the lock.

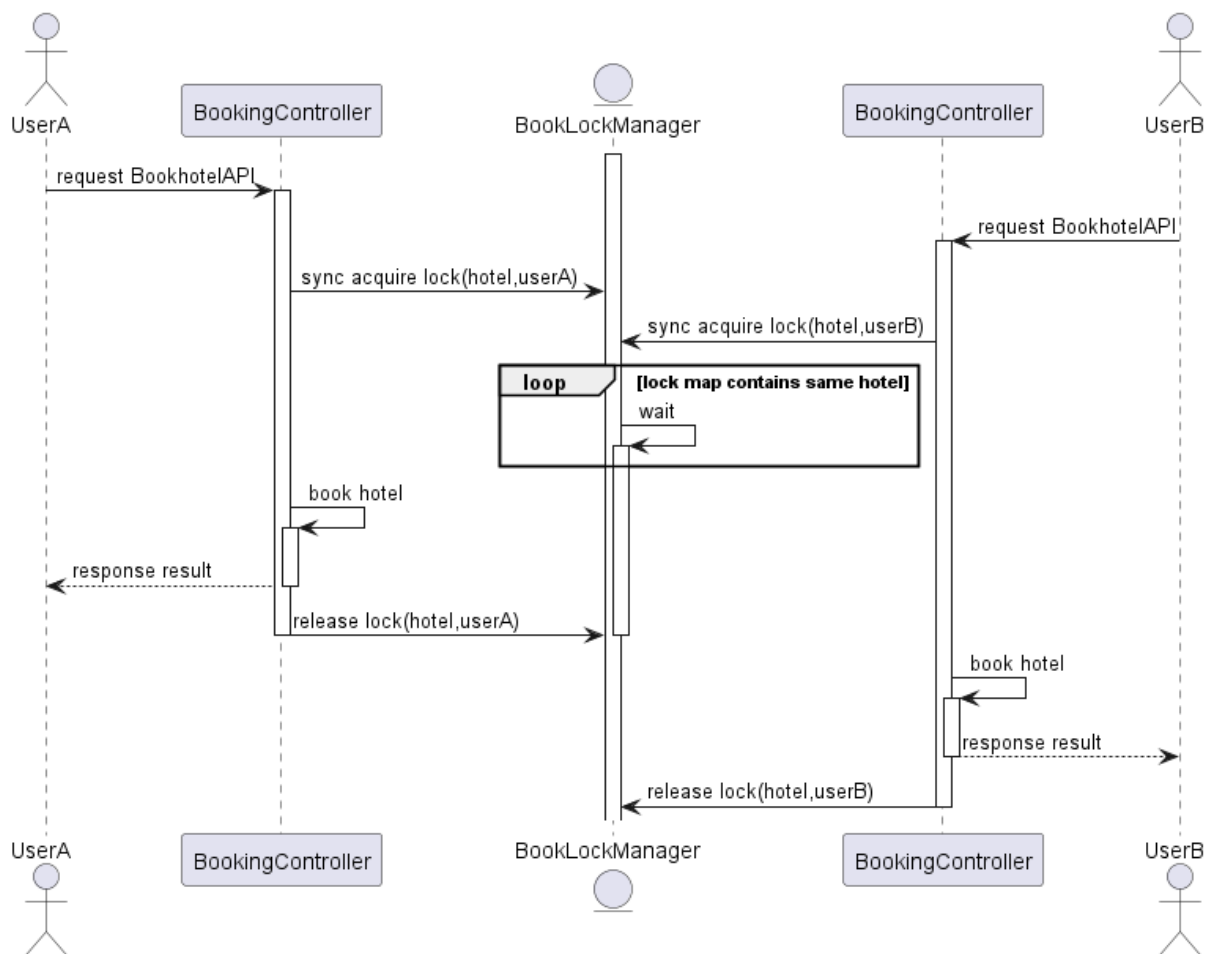


Figure 5.3: Sequence Diagram - Pessimistic Offline Lock

5.3.3 Testing Strategy and Outcome

As recommended during the lectures, **we decided to adopt JMeter** as the primary tool to send concurrent requests to test the proposed concurrency implementation of our application. Additionally,

we focus our efforts on **testing edge cases** for each domain input value **based on the limit of 20 concurrent connections allowed by our hosting provider Heroku**. [6]

BookHotel

The following test was done using a hotel with ID 7 and room with ID 30. There are 110 available rooms of ID 30 and all tests were done on days without existing bookings. Each request has the same following body (with varying dates):

```
{
  "userId": 100010,
  "totalNumGuest": 1,
  "status": "PENDING",
  "startDate": "2024-02-21",
  "endDate": "2024-02-22",
  "hotelId": 7,
  "bookingRoomList": [
    {
      "roomId": 30,
      "numberOfRooms": 20,
      "hotelId": 7
    }
  ]
}
```

Three (03) sets of tests were done: one with a single request, one with multiple requests on a version of the system without concurrency patterns, and one with multiple requests on a version with concurrency patterns. The tests with multiple requests both had 10 simultaneous requests. Since there are 110 total available rooms and each request books 20, the system should accept 5 requests (totalling up to 100 booked rooms) and reject the other 5. Without concurrency, however, all requests are accepted and 200 rooms end up being booked on the same day, much more than are available. The tests done in Jmeter are shown in **table 5.3** below shows the results obtained after running each experiment.

Table 5.3: Results - Pessimistic Offline Lock Scenario

| Result | Experiment 1: Single request without concurrency | Experiment 2: Multiple requests without concurrency | Experiment 3: Multiple requests with concurrency | | | |
|--------------------|--|---|---|------|------|------|
| | | | | | | |
| Number of requests | 1 | 10 | 2 | 10 | 20 | 21 |
| Avg Time (ms) | 1667 | 1128 | 1804 | 1360 | 1269 | 2402 |
| Succeeded | 1/1 | 10/10 | 2/2 | 5/10 | 5/20 | 5/21 |

| Result | Experiment 1: Single request without concurrency | Experiment 2: Multiple requests without concurrency | Experiment 3: Multiple requests with concurrency | | | |
|----------------------|--|---|---|-----------|----|----|
| Hotel Overbooked? | No | <i>Yes</i> | No | <i>No</i> | No | No |

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|--|-----------|---------|------|------|-----------|---------|------------|
| One booking to same hotel | 1 | 1667 | 1667 | 1667 | 0.00 | 0.000% | .59988 |
| Multiple bookings to same hotel (no concurrency) | 10 | 1128 | 1078 | 1167 | 29.00 | 10.000% | 8.55432 |
| Multiple bookings to same hotel (with concurrency) - 2 simultaneous users | 2 | 1804 | 1793 | 1815 | 11.00 | 0.000% | 1.10193 |
| Multiple bookings to same hotel (with concurrency) - 10 simultaneous users | 10 | 1360 | 1324 | 1379 | 17.52 | 50.000% | 7.24113 |
| Multiple bookings to same hotel (with concurrency) - 20 simultaneous users | 20 | 1269 | 1251 | 1306 | 18.40 | 75.000% | 15.30222 |
| Multiple bookings to same hotel (with concurrency) - 21 simultaneous users | 21 | 2402 | 1707 | 4464 | 945.91 | 76.190% | 4.70009 |
| TOTAL | 64 | 1656 | 1078 | 4464 | 762.18 | 57.812% | 3.98010 |

Figure 5.4: Results - Pessimistic Offline Lock Scenario

From **table 5.3** we can conclude:

- Without concurrency, the proper logic for checking if there are enough available rooms does not work properly, and all simultaneous requests are approved even if they go over the number of rooms availability.
- Implementing concurrency patterns solves this issue and returns the correct output, approving requests until the limit is reached, then rejecting the rest. That helps us solve the overbooking problem.

ModifyBooking

Similarly to booking the same hotel and date multiple times simultaneous, modifying multiple bookings to the same hotel and date was also tested. The same hotel, room, and availability is used from the book hotel tests, which is why we have similar results from the previous test.

Table 5.4: Results - Pessimistic Offline Lock Scenario - ModifyBooking

| Result | Experiment 1: Single request without concurrency | Experiment 2: Multiple requests without concurrency | Experiment 3: Multiple requests with concurrency | | | |
|-----------------------|--|---|---|-------------|------|------|
| Number of requests | 1 | <i>10</i> | 2 | <i>10</i> | 20 | 21 |
| Avg Time (ms) | 1754 | <i>1132</i> | 1605 | <i>1271</i> | 1747 | 1493 |
| Succeeded | 1/1 | <i>9/10</i> | 2/2 | <i>5/10</i> | 5/20 | 5/21 |
| Hotel Overbooked? | No | <i>Yes</i> | No | <i>No</i> | No | No |

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|---|-----------|---------|------|------|-----------|---------|------------|
| Update 1 booking to same hotel and dates | 1 | 1754 | 1754 | 1754 | 0.00 | 0.000% | .57013 |
| Update 10 bookings to same hotel and dates (no concurrency) | 10 | 1132 | 1115 | 1141 | 8.92 | 10.000% | 8.75657 |
| Update 2 bookings to same hotel and dates (w/ concurrency) | 2 | 1605 | 1602 | 1608 | 3.00 | 0.000% | 1.24301 |
| Update 10 bookings to same hotel and dates (w/ concurrency) | 10 | 1271 | 1237 | 1302 | 22.18 | 50.000% | 7.66284 |
| Update 20 bookings to same hotel and dates (w/ concurrency) | 20 | 1747 | 1265 | 3510 | 781.57 | 75.000% | 5.68990 |
| Update 21 bookings to same hotel and dates (w/ concurrency) | 21 | 1493 | 1309 | 1613 | 76.82 | 76.190% | 13.01922 |

Figure 5.5: Results - Pessimistic Offline Lock Scenario - ModifyBooking

Conclusion is the same as the Book Hotel use case shown above.

Delete Hotel

For delete hotel we want to test the concurrency scenario where a hotel is cancelled at the same time bookings to said hotel are made. Without concurrency patterns, all the booking requests should succeed and not be set to cancelled despite the hotel being made inactive, which is incorrect behaviour. The tests done show this behaviour in the non-concurrent system, and also shows that the correct behaviour is displayed in the concurrent version. We can see that in the test, the remove hotel request and 10 booking requests were made at the same time, and the remove hotel request is not necessarily going to be the first actually processed by the system. From the image below, we can see that in that case, all bookings approved before the hotel were removed become cancelled (which we verified using SQL queries to the database) and all bookings after are simply rejected.

Table 5.5: Results - Pessimistic Offline Lock Scenario - Delete Hotel

| Result | Experiment 1: Remove hotel at the same time as make 10 simultaneous booking requests without concurrency | Experiment 1: Remove hotel at the same time as make 10 simultaneous booking requests without concurrency |
|------------------------------------|--|--|
| Number of requests | 11 (1 /RemoveHotel + 10 /BookHotel) | 11 (1 /RemoveHotel + 10 /BookHotel) |
| Avg Time (ms) | 2026 | 1642 |
| Succeeded | 10/10 returned success and were not cancelled | 4/10 returned success but were cancelled and 6/10 returned failure |
| All bookings cancelled or rejected | No | Yes |

No concurrency pattern in the system (but all requests sent simultaneously):

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|---------------------|-----------|---------|------|------|-----------|---------|------------|
| Remove Hotel | 1 | 2916 | 2916 | 2916 | 0.00 | 0.000% | .34294 |
| 10 Booking Requests | 10 | 2026 | 2019 | 2041 | 6.26 | 0.000% | 4.89956 |

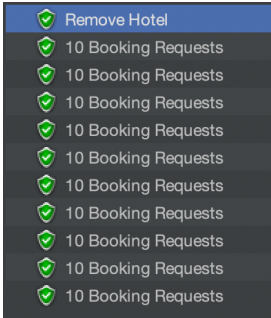
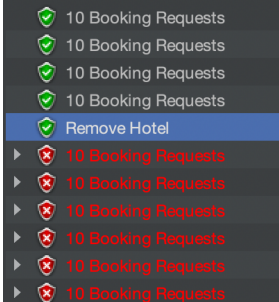


Figure 5.6: Results - No concurrency Patter - Delete Hotel

With concurrency pattern in system (and all requests sent simultaneously):



| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|---------------------|-----------|---------|------|------|-----------|---------|------------|
| 10 Booking Requests | 10 | 1642 | 1580 | 1671 | 27.34 | 60.000% | 5.98444 |
| Remove Hotel | 1 | 2554 | 2554 | 2554 | 0.00 | 0.000% | .39154 |

Figure 5.7: Results - Pessimistic Offline Lock Scenario - Delete Hotel

Based on above results, we conclude that

- Without pessimistic offline lock, even if a hotel has been removed, customers still can book the hotel successfully, which will confuse customers.
- With a pessimistic offline lock, all bookings after removing the hotel will be rejected and all bookings that are related to the removed hotel will be cancelled.

5.4 Database Constraints

5.4.1 Description

Apart from the optimistic and pessimistic lock, the system also uses Database Constraints to handle some concurrency issues. As **Table 5.1** shows, there are three use cases that use Database Constraints.

- **Customers sign up**

When the system becomes popular very fast, there are maybe over thousands of users who try to sign up almost at the same time. In this scenario, if multiple customers try to sign up with the same email, we need to ensure that only one customer can sign up successfully. Otherwise, it will lead to information exposure security issues and hard to identify different customers.

- **Hoteliers create a hotel for a group**

When multiple hoteliers create a hotel with the same name and address at the same time, if they all can create successfully, it will make customers confused since two hotels have exact same attributes. Additionally, it is hard for hoteliers to manage hotels with the same attributes.

Thus, we need only to allow one hotelier to create the hotel if there are multiple hoteliers creating hotels with the same attributes.

The reason why we don't use Database Constraints for creating rooms for a hotel is the correctness of creating a room is less important than creating a hotel, hotelier can easily manage and remove a room in the future if multiple hoteliers accidentally create two same rooms.

5.4.2 Implementation Details

- Customers sign up

Because sometimes the name could be duplicated with both first name and last name, we decided to use the email address as the unique identifier of users, which means every email could only be registered once.

Thus, adding a constraint to the database to guarantee the uniqueness of every email address when new records are inserted:

“CONSTRAINT tb_user_email_key UNIQUE (email)”

- Create hotel

Because sometimes the same hotel name might exist in different countries or regions, the unique identifier for a hotel should be represented by multiple fields. Therefore, we need to combine and use several fields of hotel together to distinguish the duplicated hotel.

As a result, we decided to add a constraint to guarantee the uniqueness of every hotel by using the fields “name, country, state, street” of the hotel:

“CONSTRAINT tb_hotel_same_key UNIQUE (name, country, state, street)”

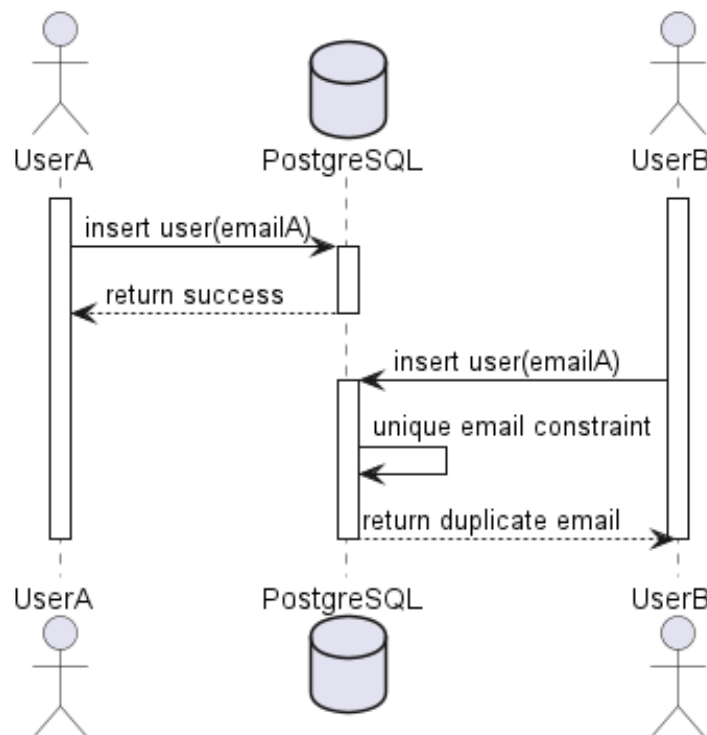


Figure 5.8: Sequence Diagram - Database level concurrency

5.4.3 Testing Strategy and Outcome

As recommended during the lectures, **we decided to adopt JMeter** as the primary tool to send concurrent requests to test the proposed concurrency implementation of our application. Additionally, we focus our efforts on **testing edge cases** for each domain input value **based on the limit of 20 concurrent connections allowed by our hosting provider Heroku**.

CustomerSignUp

As database constraints are a concurrency pattern implemented on the database itself, we expect that both the version of the codebase with and without concurrency patterns will handle concurrent requests the same way. Because of this, we only need to test 1 version of the system, as both will behave identically. We will send varying amounts of identical simultaneous requests, and expect that only one request will succeed. **Table 5.6** shows the results obtained after running each experiment:

Table 5.6: Results - Database Constraint Scenario - Customer Sign Up

| Result | Experiment: Multiple requests with database constraints | | | | |
|--------------------|---|------|------|------|------|
| Number of requests | 1 | 2 | 10 | 20 | 21 |
| Avg Time (ms) | 3142 | 1325 | 1489 | 1425 | 1459 |
| Succeeded | 1/1 | 1/2 | 1/10 | 1/20 | 1/21 |
| Duplicate User? | No | No | No | No | No |

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|------------|-----------|---------|------|------|-----------|---------|------------|
| 1 Sign Up | 1 | 3142 | 3142 | 3142 | 0.00 | 0.000% | .31827 |
| 2 Sign Up | 2 | 1325 | 1314 | 1337 | 11.50 | 50.000% | 1.49589 |
| 10 Sign Up | 10 | 1489 | 1455 | 1545 | 23.61 | 90.000% | 6.46412 |
| 20 Sign Up | 20 | 1425 | 1290 | 1539 | 61.21 | 95.000% | 12.99545 |
| 21 Sign Up | 21 | 1459 | 1350 | 1563 | 77.75 | 95.238% | 13.39286 |

Figure 5.9: Results - Database Constraint Scenario - Customer Sign Up

From **table 5.6** we can conclude:

As shown in **table 5.6**, using database constraints and locking each transaction at the database level is useful in such cases where the Business Transaction maps directly to a single System transaction and the probability of concurrency issues is low; furthermore it reduces the logic complexity by allowing the database engine to deal with concurrency calls.

CreateHotel

Similar to sign up, database constraints are implemented on the database level, not on the system level, and so we only need to test 1 version of the system. Once again we will send varying amounts of simultaneous requests to make the same hotel, and all but 1 should fail due to the constraint. **Table 5.7** shows the results obtained after running each experiment:

Table 5.7: Results - Database Constraint Scenario - Create Hotel

| Result | Experiment: Multiple requests with database constraints | | | | |
|--------------------|---|------|------|------|------|
| Number of requests | 1 | 2 | 10 | 20 | 21 |
| Avg Time (ms) | 2740 | 1367 | 1416 | 1392 | 1992 |
| Succeeded | 1/1 | 1/2 | 1/10 | 1/20 | 1/21 |
| Duplicate Hotel? | No | No | No | No | No |

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput |
|-----------------|-----------|---------|------|------|-----------|---------|------------|
| 1 Create Hotel | 1 | 2740 | 2740 | 2740 | 0.00 | 0.000% | .36496 |
| 2 Create Hotel | 2 | 1367 | 1359 | 1376 | 8.50 | 50.000% | 1.45349 |
| 10 Create Hotel | 10 | 1416 | 1366 | 1474 | 31.13 | 90.000% | 6.78426 |
| 20 Create Hotel | 20 | 1392 | 1286 | 1449 | 43.54 | 95.000% | 13.77410 |
| 21 Create Hotel | 21 | 1992 | 1330 | 3225 | 821.26 | 95.238% | 6.50356 |

Figure 5.10: Results - Database Constraint Scenario - Create Hotel

6. References

- [1] Larman, C. (2005). *Applying UML Patterns: an introduction to object-oriented analysis and design and iterative development (3rd ed.)*. Pearson Education, Inc.
- [2] Cockburn, A. (2016). *Writing effective use cases*. Boston Addison-Wesley
- [3] Oliveira, Eduardo and Rodriguez, Maria. Lecture Notes, Topic: “*SWEN90007. Software Design and Architecture*”. Engineering, Melbourne, Victoria: Department of Computing and Information Systems, Melbourne University, 2022
- [4] PlantUML. *Domain Model Tool Creator*. <https://plantuml.com/commons>
- [5] Visual Paradigm Online.
<https://online.visual-paradigm.com/diagrams/solutions/free-visual-paradigm-online/>
- [6] Heroku. Concurrency and Database Connection. “*Maximum Database Connection*”. The last date accessed: 2022-10-19
<https://devcenter.heroku.com/articles/concurrency-and-database-connections#maximum-database-connections>