Software Architecture Design (SAD) Report

Submission 3

Team LANS

SWEN90007 SM2 2022 Project

In charge:

Mohammad Saood **Abbasi** <mohammadsaoo@student.unimelb.edu.au>

Arman **Arethna** <aarethna@student.unimelb.edu.au>

Navdeep **Beniwal** <nbeniwal@student.unimelb.edu.au>

Levi **McKenzie-Kirkbright** <levim@student.unimelb.edu.au>

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

1 REVISION HISTORY

Date	Version	Description	Author(s)
30/09/22	01.00-D01	Created document outline	Levi
08/10/22	01.00-D02	Identify concurrent use cases	Everyone (as a team)
09/10/22	01.00-D03	Create template for concurrent use case discussions	Levi
09/10/22	01.00-D04	Introduction for Section 4 Discussion of Concurrency Issues	Levi
09/10/22	01.00-D05	Draft discussion for “make a booking”	Levi
10/10/22	01.00-D06	Draft description for: - Customer – make a booking - Customer – modify a booking – add more - Customer – modify a booking – change dates - Customer modify a booking – cancel booking - Hotelier – create a hotel - Hotelier – modify a booking - Hotelier – cancel a booking - Admin – remove hotel listing - Admin – remove Hotelier	Levi
10/10/22	01.00-D07	Draft rationale for Admin “remove hotel listing”	Levi
11/10/22	01.00-D08	Refine description for each use case.	Team
20/10/22	01.00-D08	Concurrency strategy section	Levi

2 TABLE OF CONTENTS

3 INTRODUCTION

3.1 Proposal

This document contains is a Software Architecture and Design (SAD) Report focusing on **concurrency**. It is the report accompanying the **third** submission of the project for SWEN90007 Software Modelling and Design, 2022.

3.2 Target Users

This document is intended for SWEN90007 teaching staff and other SWEN90007 students, particularly the LANS team.

3.3 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.

Table 1: terms used in this report.

Term	Description
"Team" or "the Team" or "LANS"	The project team.
Primary user	In discussing concurrency, the primary user is the user who is the focus of the use case under discussion.
Interfering user	In discussing concurrency, the interfering user is the user whose simultaneous interactions may interfere with the interactions of the primary user of the use case under discussion.

3.4 Links

Heroku deployment: https://swen90007-2022-lans.herokuapp.com/

Repository: https://github.com/SWEN900072022/SWEN90007-2022-LANS

Tagged release: SWEN90007_2022_Part3_LANS

3.5 User accounts for teaching staff

We have created several accounts for teaching staff to interact with our system and test our system's concurrency handling. If teaching staff do not want to use these accounts, i.e., create new accounts, please proceed to create accounts using standard account creation in the Auth0 redirect window. All customer accounts can be upgraded to hotelier accounts via the admin dashboard.

Please be aware that users cannot be elevated to admin through the frontend system.

Use the admin dashboard to:

1. Elevate customers to hoteliers
2. Add and remove hoteliers to/from groups.
    a. Relevant data will not be displayed in a hotelier's dashboard until they are added to a group.
3. Delisted hotels
4. Create hotel groups

Use the hotelier dashboard to:

1. View and modify hotels of a hotel group
2. Add rooms to a hotel
3. View and cancel bookings for each hotel in the hotel group

As a customer:

1. Search for stays
2. View available rooms in hotels
3. Create bookings at hotels
4. View and modify existing bookings
    a. Change dates
    b. Update the number of guests
    c. Cancel

Table: username and passwords for testing accounts. Note: you can create new accounts if you wish – simply follow the signup link in the Auth0 popup when you click on Log in/Sign up.

Role	Username	Password
Admin #1	admin@admin.com	Admin123$
Admin #2	admin2@admin.com	Admin123$
Luke Hotelier†	luke@hotelier.com	Hotelier123$
Max Hotelier†	max@hotelier.com	Hotelier123$
Eduardo Hotelier†	eduardo@hotelier.com	Hotelier123$
Luke Customer	luke@customer.com	Customer123$
Max Customer	max@customer.com	Customer123$
Eduardo Customer	eduardo@customer.com	Customer123$

†Note: hoteliers are not currently added to a hotel group. To add them to a hotel group, simply go to the admin dashboard of one of the admin accounts and elevate them.

4 Executive Summary

Concurrency is one of the most challenging aspects of software development. Inappropriately handled concurrency concerns can render a system unusable in many ways, from hard-to-debug livelock situations to data corruption that does not cause runtime errors. Concurrency is especially important in enterprise systems that support many users with different use cases that interact with the system simultaneously over asynchronous connections, viewing and updating the system's data independently. In sophisticated enterprise software systems, two key concerns must be balanced. On the one hand, the system must guarantee the correct execution of complex business rules and the integrity of the underlying data, whilst on the other hand, the system must maximize concurrent access for the many diverse users of the system.

We present a detailed analysis of an implementation of concurrency handling patterns in the LANS Hotels, a web application built with enterprise software architecture and design patterns. LANS Hotels is a platform that connects customers to hotels. Customers can book stays at hotels, hoteliers can manage their businesses, and the system admin can manage the overall platform. Concurrency is a crucial concern for LANS Hotels as a web platform: it must correctly apply many business rules, protect data integrity, and maintain availability whilst being used by many users simultaneously.

Applying several key engineering techniques, with a particular emphasis on Optimistic Offline Locking, enables LANS Hotels to concurrently serve multiple users without compromising data integrity and system liveness. Our implementation handles concurrent use at multiple levels of the system's architecture, including leveraging native database features, applying enterprise concurrency patterns in the backend of the compute layer, and delivering responsive error handling in the frontend. Repeatable tests are systematically documented to verify the success of the overall concurrency strategy and implementation. We discuss of the trade-offs and design rationale in for our concurrency strategy, explaining specific patterns with accompanying diagrams. Overall, our implementation delivers a high-availability web service that guarantees the integrity of the system's data.

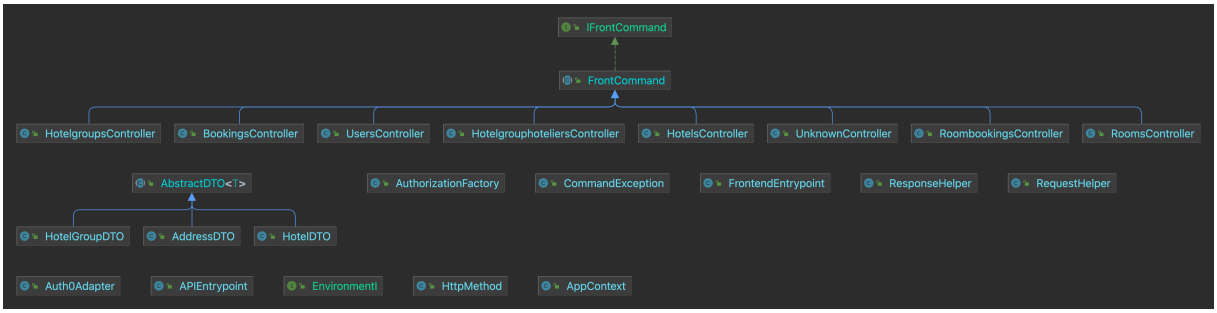
5 Updated Class Diagram

5.1 Full scale images

Full scale, zoomable images can be found in the project directory under: /docs/class-diagrams

5.2 Improved API layer

Image: API layer class diagram (please see image in /docs/part3/class-diagrams for full image)



5.2.1.1 *Request and response helpers*

We created two classes to encapsulate common operations related to extracting JSON data from requests and injecting JSON data into responses.

These can be found in: /src/main/java/lans/hotels/api/utils

5.2.1.2 Authorization framework

We created a high-level custom framework for handling authorization. This framework acts as an adapter over the auth0.

It can be found in: /src/main/java/lans/hotels/api/auth

5.2.1.3 Data Transfer Objects

To encapsulate the complexity of marshalling and unmarshalling domain objects, we introduced some DTOs.

These can be found in: /src/main/java/lans/hotels/api/DTOs

5.3 Use Cases

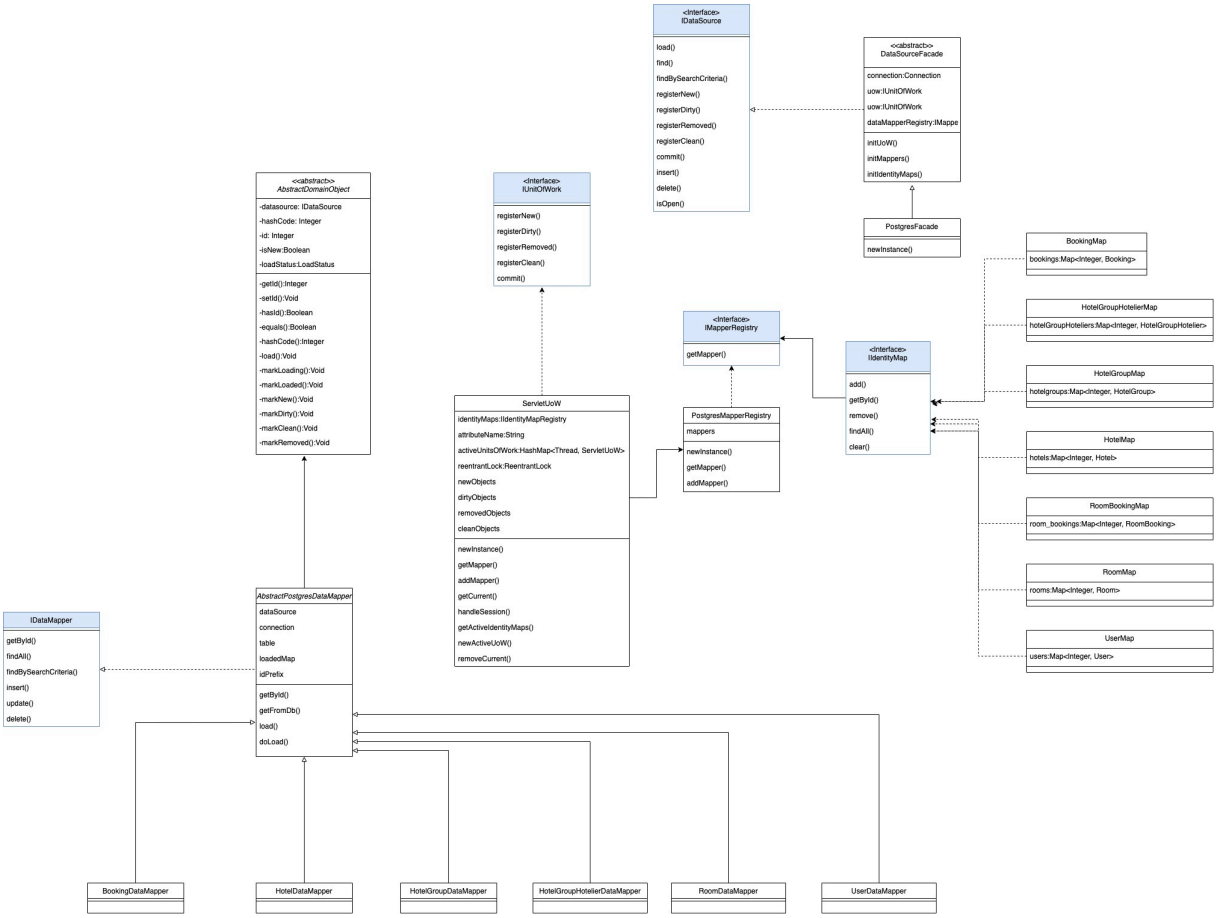
We introduced a new abstraction: classes for use cases. These classes helped us improve the separation of concerns. Controllers simply map request patterns (path and/or request body params) to the corresponding use case with appropriate authorization (if any).

These can be found in: /src/main/java/lans/hotels/use_cases

Image: use cases class diagram (please see image in /docs/part3/class-diagrams for full image)

5.4 Data source layer

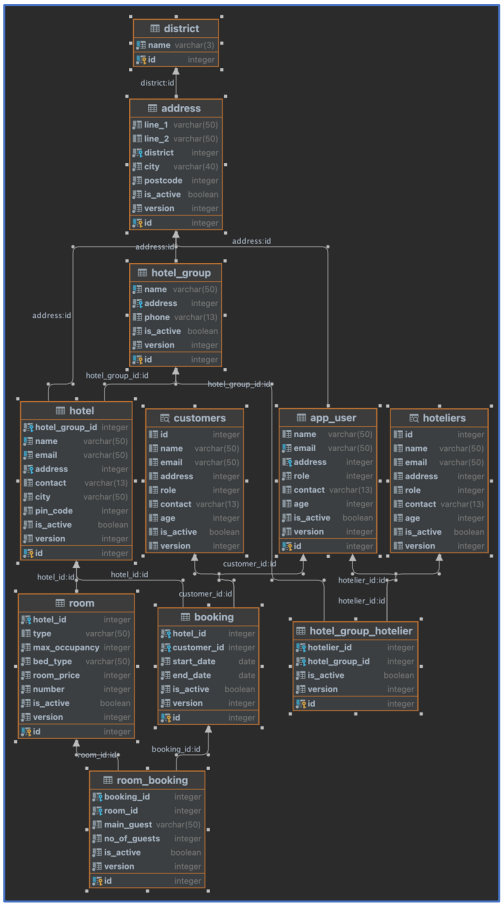
Image: data source layer class diagram



5.5 Database schema

We simplified our database schema. The most significant change that we made was that we changed our class inheritance pattern for the user types from Class Table Inheritance to Single Table Inheritance. We made this change because it simplified our integration with auth0.

Image: updated database schema



6 Concurrency Strategy Overview

A concurrency strategy is one of the most critical subsets of architectural and design decisions that must be made for an enterprise software system. Ideally, engineering decisions should maximize the benefits of tradeoffs to align with the needs of users and the goals of the business, with careful consideration of the constraints and sacrifices that accompany each decision. An enterprise system's concurrency strategy will have a direct impact on many functional and non-functional aspects, including availability and complexity of development.

As a web-based two-sided marketplace, LANS Hotels is both a B2B and a B2C application. It does not have any "mission critical" or "safety critical" functionality, such as handling financial assets (banking systems) or controlling machinery (aeroplane controls). Instead, a consumer-facing platform, LANS prioritizes the availability of data for users and system developability and maintainability. We achieved these outcomes by implementing Optimistic Offline Locking and leveraging the locking capabilities of PostgreSQL in our schema design. Despite its benefits, our strategy has one primary downside: the risk of reduced usability because of lost work. Given the user needs and business goals, occasional lost work was considered an acceptable tradeoff to achieve data integrity with a simple, cohesive design.

Our design can be summarized as such:

- Request isolation: one thread per request.
- Transaction isolation: one transaction per request.
- No longer transactions.
- We delegate as much concurrency control to the database as possible.
- Optimistic offline locking with integer version numbers when locking could not be appropriately delegated to the database.
- Mitigating usability issues in the case of failures due to concurrency with consistent and informative error messages.

In the following subsections, we discuss the various high-level tradeoffs of our design, including:

- Protecting against business logic violation
- Lost work
- Availability of data
- Developability and maintainability of source code
- Inconsistent read
- Dirty read
- Lost update
- Uncommitted data

6.1 Motivation

Simultaneous access to shared data may cause users concurrently operating on the same data to lose their work. The system designers must make decisions considering two critical dimensions of lost work: frequency and severity.

- **Frequency**: the expected or observed rate of work lost, for example, the proportion of commits that failed because of lost work compared to the total number of commits attempted.
- **Severity**: the cost of losing a unit of work, either qualitatively (e.g., emotional pain to the user) or quantitatively (e.g., lost revenue).

Furthermore, we prioritised the availability of *viewable* data. One of the key considerations was that at such an early stage in the development of the system, there might not be many hotels to populate the supply side of our marketplace. As a growing startup with our sights set on world domination, we

could not risk damaging our image of hypergrowth by being a ghost town when users land on our site. As such, we want all hotels and rooms that are available to appear in searches and not be hidden – even transiently – behind pesky read locks, even if the occasional user bounces when they try to make a booking. It is a sensible decision for a startup in our position to accrue some design usability debt (in the form of potential lost work) to communicate how undeniably popular our platform is amongst the budding post-Airbnb nouveau crypto-riche indie hotelier market.

6.1.1 CAP Theorem analysis

We also briefly analyse our system in terms of the CAP theorem. This is a relevant analysis vector as our system is a distributed system. It has two three major distributed architectural components:

1. **Frontend**: the user interface is a single-page application. The client downloads all the code to render all the views of the application in their browser. The interface is populated with data by sending requests to the backend.
2. **Backend**: the backend is a multitiered monolith with a single SQL database for persistence. It has two main functions: sending the client the JavaScript code to render the UI and sending JSON responses to API requests.
3. **Auth0**: provides user authentication via JSON Web Tokens (JWTs), and role-based authorisation is handled by a custom authorisation framework in the API layer. Authenticated user accounts are replicated to the backend database, where they can be enriched with roles (such as elevating a customer to a hotelier). The backend updates the user's roles in the browser cache for conditional client-side rendering for frontend security. For API security, access policies are applied to each request.

LANS Hotels is not a highly distributed, so we prioritised availability and consistency above partition tolerance. We have two places where partitions may occur: between the client and Auth0 or between the client and the API. If there is a network partition between the client and auth0, then they will not be able to be authorised and will only be able to interact with public parts of the user interface. If the network partitions between the client and the API, they will not be able to submit updates or receive new data until the connection is restored. By prioritising availability and consistency, we protect the integrity of the core shared data at the expense of occasional lost work for users attempting to make updates using stale data (discussed in detail later) during a prolonged network partition.

6.2 Data-related concurrency concerns

6.2.1 Lost Updates

A lost update occurs when two (or more) transactions attempt to update the same data. The second update may overwrite the first update. If this is not prevented, a user may believe that their update has been committed when, in fact, it has not been.

Our system prevents lost updates by checking the version number when executing update commands to the database. Our database stores the version number of each row. On all update requests, the domain object is first loaded into memory from the database, including its relevant version number. When the transaction attempts to commit the object back to the database, the update command uses both the ID column and the version number column to locate the resource. When two transactions attempt to update the same row simultaneously, the first transaction to commit will increment the version number of the record. When the second transaction attempts to update the record, it will not find a record with the same ID and version number pair as the in-memory object, and the command will be rejected. As such, the second transaction will roll back, and an appropriate error will be displayed to the user.

6.2.2 Dirty reads

A dirty read occurs when one transaction reads data involved in another transaction that has not yet been committed. Our one transaction per request model means that our system never allows dirty reads. As discussed in 6.1.1 Lost Updates, when two requests co-occur, the first transaction to commit will increment record version numbers, and the second transaction will roll back. Our system does not support long transactions, so persistent data is never in a non-committed state.

6.2.3 Inconsistent Reads

Inconsistent reads occur when multiple values are read from the database in one transaction while another transaction updates one or more of those same values. In most situations, our strategy does not explicitly prohibit inconsistent reads. The two primary methods of preventing inconsistent reads are coarse-grained locking and pessimistic offline locking. Pessimistic offline locking can cause data to be unreadable for extended periods across multiple requests.

Hotel booking platforms must constantly provide consumers with views of the data in the system and ensure that the data in the database is never in an invalid state. With optimistic offline locking and one transaction per request, inconsistent reads in our application may lead to the occasional loss of work when a user attempts an update using stale data. This is a minor usability issue with low expected frequency and severity, but it will never cause data corruption. As such, in the context of our application, data availability and integrity are more important than the occasional low-severity loss of work.

6.2.4 Uncommitted Work

Our system does not support long transactions, so data in our database never has uncommitted work.

6.3 Impact on functional requirements

6.3.1 Lost Work

A user of our system may lose the work they have performed over several interactions. In general terms, the user is attempting to execute a business transaction comprised of multiple system transactions. This issue is a direct consequence of our one-transaction-per-request model.

For example, a customer may try booking a hotel room. Firstly, they search for hotels and navigate to a hotel of interest. Whilst constructing their booking out of the rooms displayed as available on the hotel's page, another customer may also navigate to the same page. The second customer may submit their booking first, making the data the first customer uses stale. When the first customer attempts to make a booking, the booking may be rejected if it conflicts with the second customer's booking (i.e., the bookings contain one or more of the same rooms with overlapping date ranges).

There is a tension between providing constant availability of data to be viewed by users and incurring usability debt when work is lost. We could prevent this sort of event from occurring if we used a strategy to prevent users from reading data simultaneously, such as long transactions with pessimistic offline locking (read/write lock). However, that would seriously deplete the amount of data for viewing, such as the number of available rooms in a hotel. *Casual* browsing is typical in applications like LANS Hotels, such as looking at multiple hotels and comparing prices. As such, preventing concurrent reads could significantly impact the business strategy of the platform, as many casual users acquiring read locks across long transactions could make it seem like there is less liquidity on the platform than there is.

If we continue developing our application, one of the main areas of improvement would be to introduce discrete user flows with appropriate pessimistic locking for areas that require read locks. Mixing optimistic and pessimistic locking schemas for different parts of the user experience would

yield the best user experience whilst maintaining data integrity. At scale, similar platforms likely apply pessimistic read-and-write locks to a separate user flow related to creating a booking. For example, a platform like Airbnb may have optimistic locking for browser accommodation *before* a customer starts a booking flow. Once the customer has begun creating a booking, they may apply pessimistic locking to prevent their booking from being rejected at the end of the flow. Furthermore, Airbnb may apply techniques such as timeouts to prevent the lock from holding the data for too long.

6.3.2 Data consistency

A critical component of any real-world enterprise system is data consistency. Our system never allows the data in persistent storage to enter an inconsistent state. As discussed extensively in section 6.1, an optimistic offline lock ensures that updates to records in our database are consistent.

The only place inconsistent data will occur is in the user's browser. A user may read some data on one page and then continue to another page with that data in the browser's cache. With this inconsistent data, the user may attempt a particular use case and will not find out that their data is inconsistent until they try to make an update (this is discussed more thoroughly in section 6.2.2 Lost Work).

For example, a customer may view all their bookings on their "My Bookings" page and then navigate to a particular booking to edit it. Whilst the customer is making modifications to the booking on their user interface, a hotelier may cancel the booking. At this moment, the data in the user's browser is inconsistent. When the customer attempts to commit the update, the request will fail, and they will be presented with an error message. To prevent the buildup of stale data in the user interface across requests, when a collision like the one described is detected by the front end, the single-page application reloads the page, resetting the user's locally cached data.

6.4 Comparison of concurrency handling patterns

Optimistic Offline Lock: used for conflict *detection* when the chance of conflict is low, and the consequences of conflict are acceptable. Fowler recommends Optimistic Offline Locks as the default approach because it is easier to implement and debug. Other strategies should only be considered if there is sufficient reason given the requirements. The primary downside is that victims of conflict may lose their work for a business transaction and only find out when they attempt to commit changes.

Pessimistic Offline Lock: used for conflict *prevention* when the chance of conflict is high or the consequences of conflict are unacceptable. A more robust form of concurrency control than optimistic locks, pessimistic locks ensure that inconsistent reads and lost work are minimised or eliminated. However, pessimistic locks are more complex to implement and debug and introduce the risk of hard-to-reproduce contention issues such as livelock. Furthermore, they reduce the overall throughput of the system by lowering concurrency. They are not recommended for systems with business transactions that fit inside a single system transaction.

Course-grained lock: lock multiple related objects simultaneously when more complex locking helps satisfy specific business requirements, such as locking the child objects of an aggregate root. Coarse-grained locks are understandable mechanisms that allow the system to lock related data without loading all group members into memory. An interesting and subtle aspect of complex software systems is the intractable problem of *indirect* dependencies, i.e., changes in one part of the system affect another part of the system even though neither explicitly depends on the other. Coarse-grained locks may help reveal and mitigate the risk of indirect dependencies corrupting the data. This is both an advantage and a possible disadvantage. On the one hand, preventing data corruption is a high priority in enterprise systems, so raising the safety of the system will improve the assurance of data integrity. On the other hand, it could create hard to predict conflicts and potentially reduce throughput. In the case of optimistic offline locking, these conflicts would be revealed as lost work when operating on data that might seem unrelated to the user's current workflow. As the complexity of the system

grows over time, in conjunction with domain-driven design, the advantages of consolidating locking would likely outweigh the reduced throughput and possible risk of increased frequency of lost work.

Implicit Lock: *enforce the locking mechanism* by acquiring the lock through the layer supertype. Implicit locks are used in large applications with complex logic to minimise the risk of a forgotten lock. They can dramatically simplify the locking mechanism by centralising the design and abstracting the lock from the concrete mappers. However, they introduce the risk of difficult-to-understand bugs if the development team is unaware of the locking mechanism.

6.4.1 Trade-offs of our strategy

We selected optimistic offline lock as the primary mechanism for locking, supplemented by schema design and constraints in the database layer. Our system has the following characteristics that make optimistic offline lock the most appropriate choice between the four alternatives discussed in Fowler:

1. Limited time to implement and test the system, so simplicity and understandability of the design were essential.
2. There are few users, so the likelihood of collisions is very low.
3. The consequences of lost work are very low because the system is not mission-critical. Additionally, the user flows to create a business transaction are short and straightforward so that any lost work can be attempted again with minimal effort.
4. Few domain objects, so the minor benefit of an implicit lock was not worth the increased risk of debugging complexity.

Perhaps the most apparent reason these trade-offs are acceptable is that the current load on the system is very low. Arguably, when a system is starting, such as for a new startup, it would be advisable to ensure data consistency from the beginning. As Donald Knuth says, *premature optimisation is the root of all evil*. For a small, greenfield project, it is wasteful of development resources to engineer a system to scale for an audience that does not yet exist. Still, it must protect the integrity of its data, no matter how few users. Had we been building a system with more throughput at a larger scale, we may have made different trade-offs to meet the needs under those circumstances. As such, we next consider how engineering for concurrent usage may change as the platform grows.

6.4.2 Future considerations: platform growth and concurrency

As an initial build of a system, the system meets minimal use case and concurrency requirements. However, architectural and design decisions create path dependence for a project. As such, to explore the consequences of our strategy, it is worth considering the downstream enhancements that may be required under different circumstances. For discussion, we imagine that LANS Hotel is a new startup and are now considering some (hypothetical) downstream enhancements:

1. Although the direct consequences of lost work are not severe, they still negatively impact usability and user experience. For a new startup, this may be acceptable at the beginning, but as the user base grows, the tolerance lost work will decrease, and usability debt will need to be paid down. As discussed in section 6.3.1, we could gradually minimise or eliminate lost work by strategically applying pessimistic offline locking. For example, we could extract the process of booking a room into a standalone user flow with a long (multi-request) transaction. We could then apply a read lock to the rooms in the booking and timeout to ensure that the lock is released.
2. As the system grew in complexity, and perhaps the team grew with lots of Silicon Valley venture funding (or maybe an *ICO* – LANS Hotels *to the moon*), concrete locks could be abstracted into an implicit lock in the abstract data mapper supertype. This would reduce the cognitive overhead for developers not explicitly working on concurrency concerns.

3. An advanced engineering technique to control domain complexity is applying *domain-driven design*. As the system grew to support more use cases, we could refactor our domain model into aggregates, as discussed in *Domain Driven Design* (Evans, 2003). As such, we could then minimise the complexity of the locking schema by applying coarse-grain locks to the aggregates, which would also minimise the risk of conflict with indirect dependencies in the object graph.
4. As LANS Hotels scaled globally and revolutionised the accommodation industry, we would need to support low latency interaction for users anywhere on the planet with five-9 availability. A standard tool for low latency, high-availability distributed systems uses distributed databases that can be scaled incrementally, such as by sharding. To take advantage of distributed database technologies and delight our growing international user base, we could relax the data consistency constraint for some data subsets or split the data based on region. Document databases, such as MongoDB, would be a natural fit. Using chronological or cryptographically ordered versioning of records would enable eventual consistency, where updated versions are propagated to users asynchronously. According to the CAP theorem, this modification would allow us to increase availability (and throughput) at the cost of some inconsistency.

7 Identification of Concurrency Scenarios

The approach taken by our team to identify all possible concurrency scenarios within the system used the following steps:

1. Use of 2-dimenional matrix with both row and column values representing individual use cases (in the case of our system, each use case is associated with one API request to the back-end server).
2. Each cell within the matrix maps to the concurrent execution of the use cases represented by the row and column headers. Each cell is used to identify which use cases when paired with another use case would generate a possible concurrency conflict. The way we define a conflict is a scenario where the effects of one use case affect the execution or current data view provided by another use case.
    a. Example: Admin delisting a particular hotel would affect the ability of a customer to change dates of a booking of that same hotel.
3. The following details the way the matrices were populated
    a. Cases where no conflicts were identified are marked as green.
    b. Cases where conflicts were identified are marked in red.
4. We have created 2 such matrices (shown below) with different purposes
    a. The first matrix **Matrix A** has both rows and columns as use cases/requests that insert, modify or delete data within the database. Concurrency conflicts in this table would affect the actual execution and result of the other use case. Description of the conflicts along with handling pattern details and further elaboration is provided in **Section 9** below.
    b. The second matrix **Matrix B** has column names as use cases that insert, modify or delete data within the database and row names as use cases that only present data views on the front-end using data from the database. Concurrency conflicts in this case would affect the consistency of the data view being provided to the front-end. These cannot be directly handled due to rendering on client side being refreshed only on client refresh command. If data insert or update action occurs before rendering, there is no concurrency mismatch, if rendering occurs before data insert or update, refreshing a page would negate the concurrency mismatch.

		admin					hotelier			Customer				
		create hotel group	delist hotel	upgrade user to hotelier	add hotelier to hotel group	remove hotelier from hotel group	create hotel for group	create room for hotel	cancel hotel booking	Register on system	create booking	cancel booking	change booking dates	change no of guests in room booking
admin	create hotel group													
	delist hotel													
	upgrade user to hotelier													
	add hotelier to hotel group													
	remove hotelier from hotel group													
hotelier	create hotel for group													
	create room for hotel													
	cancel hotel booking													
Customer	Register into system													
	create booking													
	cancel booking													
	change booking dates													
	change no of guests in room booking													

Matrix A

		admin					hotelier			Customer				
		create hotel group	delist hotel	upgrade user to hotelier	add hotelier to hotel group	remove hotelier from hotel group	create hotel for group	create room for hotel	cancel customer booking	Register into system	create booking	cancel booking	change booking dates	change no of guests in room booking
admin	view all hotels													
	view all users													
	view all hotel groups													
	view all hoteliers													
hotelier	view hotels of hotel group													
	view rooms of hotel													
	view bookings of hotel													
customer	search hotels by city													
	view available rooms of hotel													
	view customer bookings													
	view room bookings within booking													

Matrix B

8 Description of Concurrency Conflict Scenarios & Pattern Implementations

Our team has used the following format to describe the various concurrency conflicts identified and how the system deals with them.

1. A brief description of each use case.
2. Details of conflicting use cases that may give rise to concurrency issues (derived from **Matrix A**) along with the reason for potential conflict occurring.
3. Choice of patterns used by us to deal with identified conflicts.
4. Implementation details and characteristics of the chosen patterns.
5. Rationale behind choosing the patterns.
6. Sequence diagrams to show how the system behaves when receiving conflicting concurrent requests (1 example per use case, a lot of conflicts have similar sequences so we have tried to give diagram examples of all the different kinds of scenarios).
7. Test cases for each conflicting use case with parent use case.
    a. Description of the two actors performing conflicting use cases (A & B).
    b. Details of the use cases (A&B).
    c. Conditions to ensure the conflict occurs when performing use cases concurrently.
    d. Expected results to try to recreate, test and validate outcome of conflicting scenarios on the system. One result for if Use Case A executes first and subsequent execution of Use Case A, one result for if Use Case B executes first and subsequent execution of Use Case A.

Discussions of concurrency involve two or more users using the system at the time. The users in question may be of the same or different user type. To make our concurrent use case discussion easy to follow, we follow the below conventions for distinguishing between users. The purpose of these conventions is to differentiate between the *primary* user and the *interfering* user (see Table 1 in *Conventions, terms, and abbreviations*).

1. The user performing an interaction will be gendered *female*, using the following default names for each type of user:
    a. User **Ursula**
    b. Customer **Carly**
2. The user performing a concurrent interaction that interferes with (1) will be gendered *male*, using the following default names for each type of user:
    a. Customer **James**
    b. Hotelier **Harry**
    c. Hotelier **Bob**
    d. Admin **Adam**

e. Admin **Aaron**

We also use the following default hotel and hotel group names:

1. Default hotel name: **Heartbreak Hotel**
2. Default hotel group name: **Presley Hotels**

As an example, the use case "make a booking" can be discussed as follows:

1. Customer Carly attempts to make a new booking at Heartbreak Hotel. At the same time:
    a. Customer Collin tries to book one or more of the same rooms in Heartbreak Hotel with dates that overlap with Customer Carley's booking, or
    b. Hotelier Harry tries to book one or more of the same rooms at Heartbreak Hotel with dates that overlap with Customer Carley's booking.

The fictional names for users, the hotel, and the hotel group are used purely for illustrative purposes to make the discussions more readable. The concurrent use case discussions can be easily generalised by substituting general words for specific fictional names. For example, "Customer Carly attempts to make a new booking at Heartbreak Hotel, and at the same time Customer Collin tries to book one or more of the same rooms in Heartbreak Hotel" can be interpreted as "a customer attempts to make a new booking at a hotel, and at the same time another customer tries to book one or more of the same rooms in the same hotel."

8.1 Admin Use Cases

8.1.1 Create a hotel group.

1.1. Description: Admin Adam creates a hotel group.

1.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Admin Aaron creates a hotel group with the same name.	A hotel group must have a unique name.

1.3. Choice of pattern: Handled via controller checks using database requests.

1.4. Pattern implementation:

1.4.1. At the time of create hotel group request, the admin session first checks if there already exists a hotel group with the same name using database query.

1.4.2. If a duplicate is found then no insert is attempted, returns bad request error.

1.4.3. If no duplicate is found, then insert is attempted.

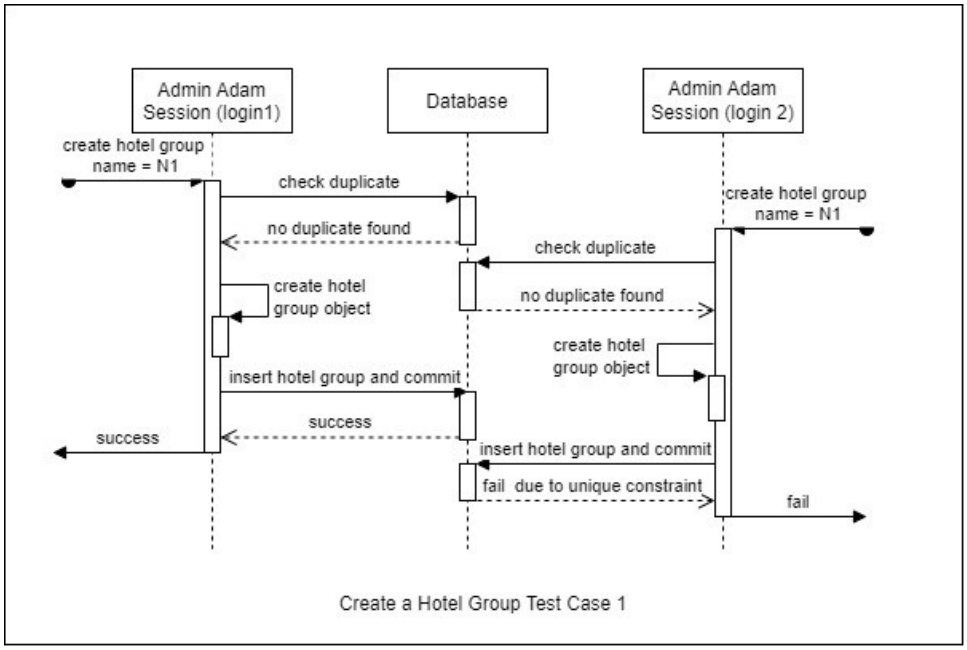
1.4.4. If by chance a create hotel group request gets executed in between initial check and insert having same name, the database still will not accept the insert due to unique condition on both name and email. Returns internal server error.

1.5. Rationale:

1.5.1. The use case requires only inserting data, hence no locking is required.

1.5.2. Checks using queries and database constraints provide enough support to always uphold ACID properties.

1.6. Sequence Diagram:



Create a Hotel Group Test Case 1

1.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results

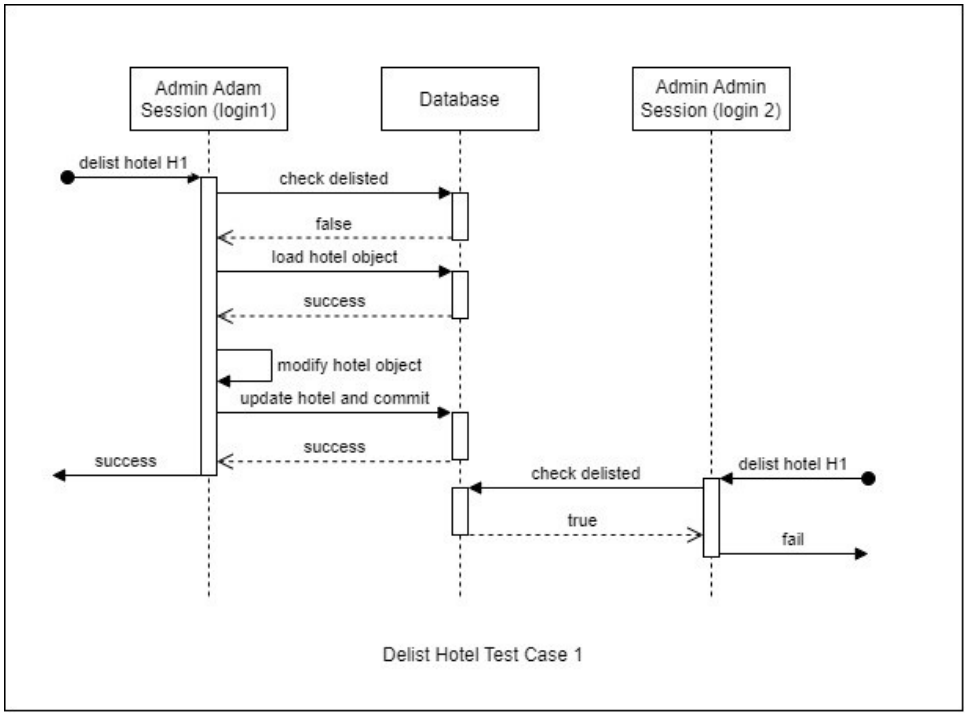
1	Admin Adam	Create new hotel group HG1 with name N1.	Admin Adam	Create new hotel group HG2 with name N2.	1) Both instances of Admin Adam are logged in simultaneously from different access points. 2)name N1 = name N2	1) Use Case A : Hotel group HG1 created by Actor A -> Use Case B : Show error hotel group name exists to Actor B. 2) Use Case B : Hotel group HG2 created by Actor B-> Use Case A : Show error hotel group name exists to Actor A.

8.1.2 Delist hotel

2. Discussion:
   2.1. Description: Admin Adam delists Heartbreak Hotel.
   2.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Admin Aaron delists same hotel.	A hotel can be delisted only once and using the correct version number.

   2.3. Choice of pattern: Combination of the following
      2.3.1. Controller checks using database requests.
      2.3.2. Optimistic offline lock
   2.4. Pattern implementation:
      2.4.1. At the time of delist hotel request, the admin session first checks if the hotel is already delisted using database query.
      2.4.2. If delisted, returns bad request.
      2.4.3. If not delisted, updates hotel object to reflect delisting and then sends an update query to database.
      2.4.4. If by chance a delist request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match the object number currently in database and returns server error.
   2.5. Rationale:
      2.5.1. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
      2.5.2. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the database.
      2.5.3. No data is deleted, only is_active field in hotel table is changed.
   2.6. Sequence Diagram:



2.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Delist hotel H1.	Admin Adam	Delist hotel H2.	1) Both instances of Admin Adam are logged in simultaneously from different access points. 2) hotel H1 = hotel H2	1) Use Case A : Hotel H1 delisted by Actor A. -> Use Case B : Show error hotel H2 delisted error to Actor B. 2) Use Case B : Hotel H2 delisted by Actor B. -> Use Case A : Show error hotel H1 delisted to Actor A.

8.1.3 Upgrade user to hotelier

3. Discussion:

3.1. Description: Admin Adam upgrades User Ursula to role of hotelier.

3.2. Possible Concurrency Conflicts

No.	Conflicting Use Case	Reason
1	Admin Aaron upgrades the same user.	A user can be upgraded to hotelier only once and using the correct version number.

3.3. Choice of pattern: Combination of the following

3.3.1. Controller checks using database requests.

3.3.2. Optimistic offline lock

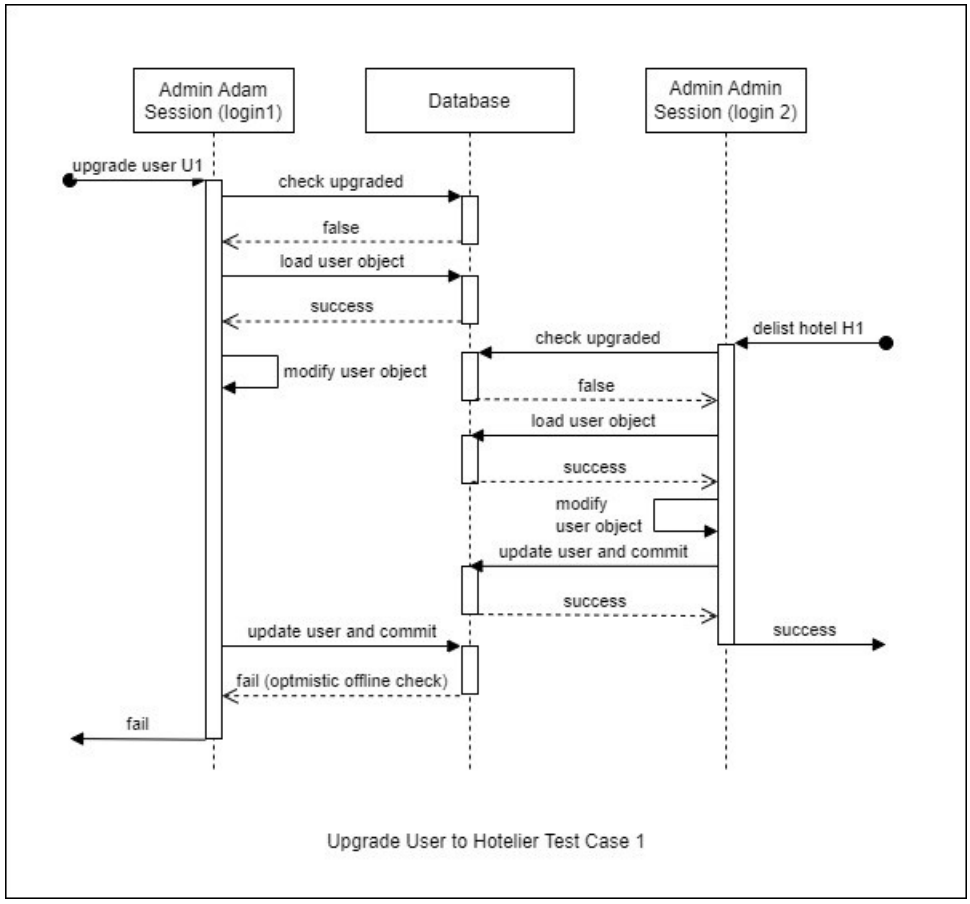
3.4. Pattern implementation:

3.4.1. At the time of upgrade user request, the admin session first checks if the user is already upgraded using database query.

3.4.2. If upgraded, returns bad request.
3.4.3. If not upgraded, updates user object to reflect upgrade and then sends an update query to database.
3.4.4. If by chance an upgrade request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match and returns server error.

3.5. Rationale:
3.5.1. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
3.5.2. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the database.

3.6. Sequence Diagram:



Upgrade User to Hotelier Test Case 1

3.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results

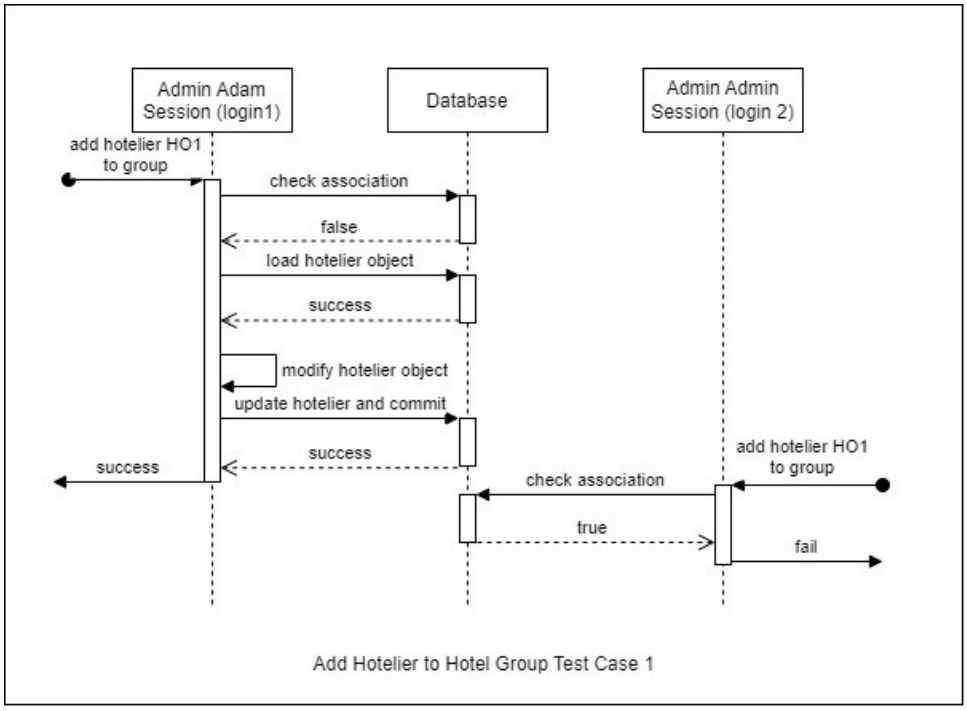
1	Admin Adam	Upgrade user U1 to hotelier.	Admin Adam	Upgrade user U2 to hotelier.	1) Both instances of Admin Adam are logged in simultaneously from different access points. 2) user U1 = user U2	1) Use Case A : Upgrade to hotelier request successful for Actor A. -> Use Case B : Upgrade to hotelier request unsuccessful, error user not a customer shown to Actor B. 2) Use Case B : Upgrade to hotelier request successful for Actor B. -> Use Case A : Upgrade to hotelier request unsuccessful, error user not a customer shown to Actor A.

8.1.4 Add hotelier to hotel group

4. Add hotelier to hotel group.
   4.1. Description: Admin Adam adds Hotelier Harry to Hotel Group Atlantis.
   4.2. Possible Concurrency Conflicts

No.	Conflicting Use Case	Reason
1	Admin Aaron adds the same hotelier to a hotel group.	A hotelier can be associated with only one hotel group, needs association to be removed before being added to another hotel group.

   4.3. Choice of pattern: Handled via controller checks using database requests.
   4.4. Pattern implementation:
      4.4.1. At the time of add hotelier to group request, the admin session first checks if the hotelier is already associated to a hotel group using database query.
      4.4.2. If a row is found then no insert is attempted, returns bad request error.
      4.4.3. If no duplicate is found, then insert is attempted.
      4.4.4. If by chance a add hotelier to group request gets executed between initial check and insert, the database still will not accept the insert due to unique condition on the hotelier id within hotel_group_hotelier table. Returns internal server error.
   4.5. Rationale:
      4.5.1. The use case requires only inserting data, hence no locking is required.
      4.5.2. Checks using queries and database constraints provide enough support to always uphold ACID properties.
   4.6. Sequence Diagram:



Add Hotelier to Hotel Group Test Case 1

4.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Add hotelier HO1 to hotel group HG1.	Admin Adam	Add hotelier HO2 to hotel group HG2.	1) Both instances of Admin Adam are logged in simultaneously from different access points. 2) hotelier HO1 = hotelier HO2	1) Use Case A : Add hotelier request by Actor A is successful. -> Use Case B : Request by Actor B is unsuccessful, show error hotelier already associated to hotel group. 2) Use Case B : Add hotelier request by Actor B is successful. -> Use Case A : Request by Actor A is unsuccessful, show error hotelier already associated to hotel group.

8.1.5 Remove hotelier from hotel group

5. Remove hotelier from hotel group.
   5.1. Description: Admin Adam removes Hotelier Harry to Hotel Group Atlantis.
   5.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason

1	Admin Aaron removes the same hotelier from hotel group.	A hotelier can be associated removed from hotel group only if they are currently associated to a hotel group.

5.3. Choice of pattern: Handled via controller checks using database requests.

5.4. Pattern implementation:

5.4.1. At the time of remove hotelier from group request, the admin session first checks if the hotelier is not associated to a hotel group using database query.

5.4.2. If no association to hotel group is found then no delete is attempted, returns bad request error.

5.4.3. If an association with hotel group is found, then insert is attempted.

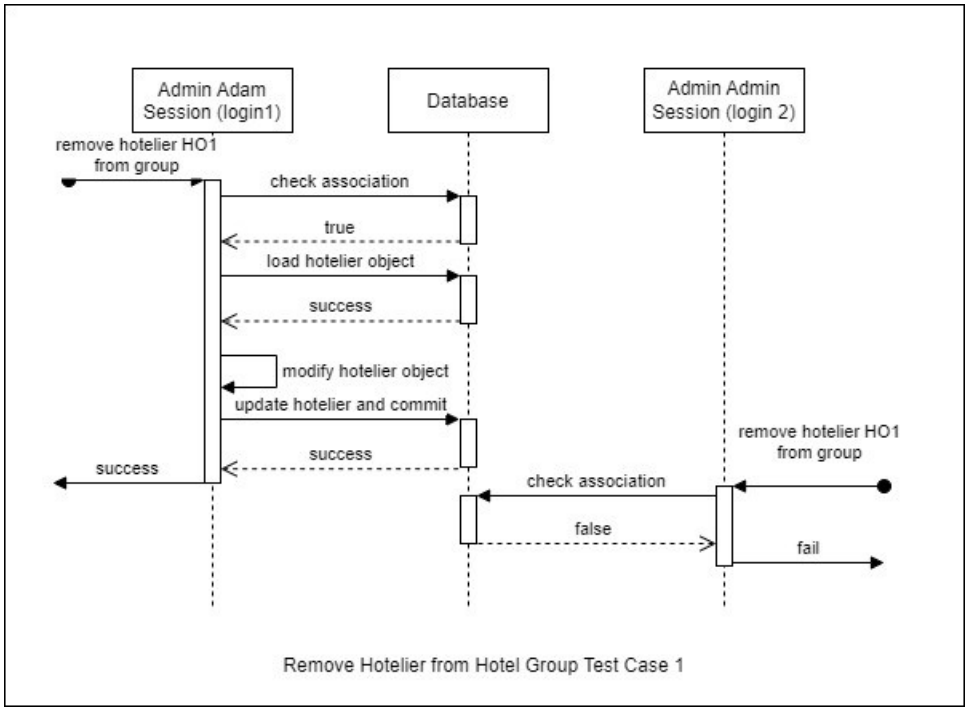
5.4.4. If by chance a remove hotelier from group request gets executed between initial check and delete, the delete will not delete anything as no row exists.

5.5. Rationale:

5.5.1. The use case requires only deleted of row data from single table with no dependencies, hence no locking is required.

5.5.2. Checks using queries and database constraints provide enough support to always uphold ACID properties.

5.6. Sequence Diagram:



Remove Hotelier from Hotel Group Test Case 1

5.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results

1	Admin Adam	Remove hotelier HO1 from Hotel group.	Admin Adam	Remiove hotelier HO2 from hotel group.	1) Both instances of Admin Adam are logged in simultaneously from different access points. 2) hotelier HO1 = hotelier HO2	1) Use Case A : Remove hotelier request successful for Actor A. -> Use Case B : Request unsuccessful, show error hotel group hotelier does not exist to Actor B. 2) Use Case B : Remove hotelier request successful for Actor B. -> Use Case A : Request unsuccessful, show error hotel group hotelier does not exist to Actor A.

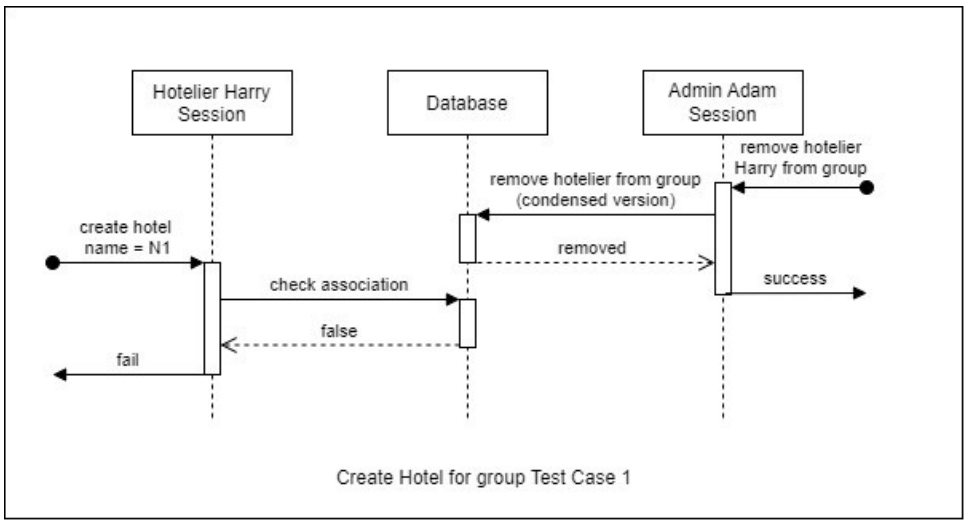
8.2 Hotelier Use Cases

8.2.1 Create hotel for group

1. Create hotel for group
    1.1. Description: Hotelier Harry tries to create a hotel for Hotel Group Atlantis.
    1.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Admin Adam removes Harry's association to Hotel Group Atlantis.	Hotel can be created only by a hotelier who has an associated hotel group.
2	Hotelier Bob creates a hotel with the same name.	A hotel must have a unique name.
3	Hotelier Bob creates a hotel with the same email.	A hotel must have a unique email.

    1.3. Choice of pattern: Handled via controller checks using database requests.
    1.1. Pattern implementation:
        1.1.1. At the time of create hotel request, the hotelier session first checks if the hotelier is associated to a hotel group using database query.
        1.1.2. If not, then sends a bad request error.
        1.1.3. If yes, the hotelier session checks if there already exists a hotel with the same name or email using database query.
        1.1.4. If a duplicate is found then no insert is attempted, returns bad request error.
        1.1.5. If no duplicate is found, then insert is attempted.
        1.1.6. If by chance an add hotel to group request gets executed in between initial check and insert having same hotel name or email, the database still will not accept the insert due to unique condition on the name. Returns internal server error.
    1.4. Rationale:
        1.1.1. Use case requires only inserting data, hence no locking is required.
        1.1.2. Checks using queries and database constraints provide enough support to always uphold ACID properties.
    1.5. Sequence Diagram:



Create Hotel for group Test Case 1

1.6. Test Cases:

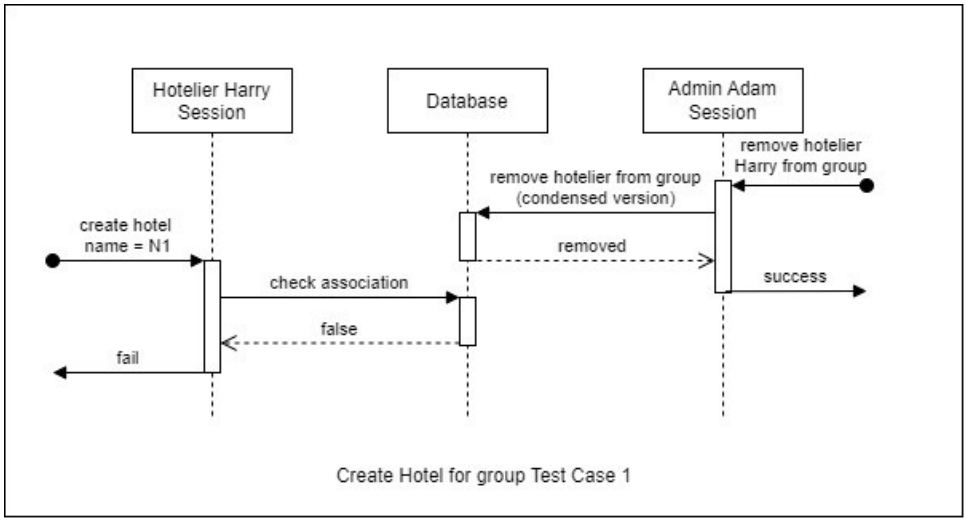
	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Remove hotelier HO1 from Hotel group.	Hotelier Harry	Create new hotel H1 for group.	1) hotelier HO1 = hotelier Harry	1) Use Case A : Remove hotelier request successful for Adam. -> Use Case B : Request unsuccessful, hotel H is not created, show unauthorized error to Harry. 2) Use Case B : Request successful, hotel H is created by Harry. -> Use Case A : Remove hotelier request successful for Adam.
2	Hotelier Harry	Create hotel H1 with name HN1.	Hotelier Bob	Create hotel H2 with name HN2.	1) Hotelier Harry and Hotelier Bob are associated to the same hotel group. 2) name HN1 = name HN2	1) Use Case A : Hotel H1 is created by Harry. -> Use Case B : Show error hotel name exists to Bob. 2) Use Case B : Hotel H2 is created by Bob. -> Use Case A : Show error hotel name exists to Harry.
3	Hotelier Harry	Create hotel H1 with email EM1.	Hotelier Bob	Create hotel H2 with email EM2.	1) Hotelier Harry and Hotelier Bob are associated to the same hotel group. 2) email EM1 = email EM2	1) Use Case A : Hotel H1 is created by Harry. -> Use Case B : Show error hotel email exists to Bob. 2) Use Case B : Hotel H2 is created by Bob. -> Use Case A : Show error hotel email exists to Harry.

8.2.2 Create room for hotel

2. Create room for hotel
    2.1. Description: Hotelier Harry tries to create a room for Heartbreak Hotel of Hotel Group Atlantis.
    2.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Heartbreak Hotel is delisted by Admin Adam before Hotelier Harry creates room.	New rooms can only be created only for active/ non-delisted hotels.
2	Admin Adam removes Harry's association to Hotel Group Atlantis.	New room for Heartbreak Hotel can be created only by a hotelier who has an associated its parent hotel group.
3	Hotelier Bob creates a room with the same number for Heartbreak Hotel.	A room within a hotel must have a unique number.

    2.3. Choice of pattern: Handled via controller checks using database requests.
    2.4. Pattern implementation:
        1.1.1. At the time of create room request, the hotelier session first checks if the hotel where room is being added has any room with same number.
        1.1.2. If yes, returns bad request error.
        1.1.3. If no, the hotelier session checks if the hotelier is associated to the same hotel group as hotel where room is being added using database query.
        1.1.4. If not, then sends a bad request error.
        1.1.5. If yes, then insert is attempted.
        1.1.6. If by chance an add room to hotel request gets executed in between initial check and insert having same room number, the database still will not accept the insert due to unique condition on the room number. Returns internal server error.
    2.5. Rationale:
        1.1.1. Use case requires only inserting data, hence no locking is required.
        1.1.2. Checks using queries and database constraints provide enough support to always uphold ACID properties.
    2.6. Sequence Diagram:



    2.7. Test Cases:

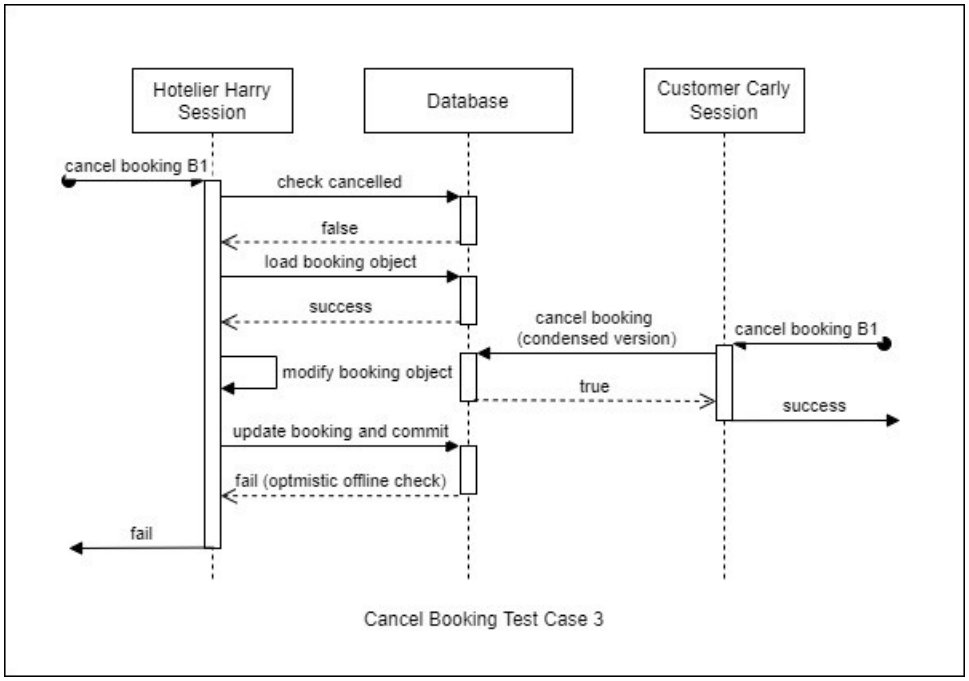
	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Delist hotel H1.	Hotelier Harry	Create room R1 for hotel H2 of hotel group HG1.	1) hotel H1 = hotel H2	1) Use Case A : Hotel H1 delisted by Adam. -> Use Case B : Show hotel delisted error to Harry. 2) Use Case B : Room successfully created by Harry. -> Use Case A : Hotel H1 delisted by Adam.
2	Admin Adam	Remove hotelier HO1 from Hotel group.	Hotelier Harry	Create room R1 for hotel H1 of hotel group.	1) hotelier HO1 = hotelier Harry	1) Use Case A : Remove hotelier request successful for Adam. -> Use Case B : Request unsuccesssful, show error unauthorized to Harry. 2) Use Case B :-> Use Case A : Request successful, room R1 created by Harry.
3	Hotelier Harry	Create room R1 for hotel H1 with room number RN1.	Hotelier Bob	Create room R2 for hotel H2 with room number RN2.	1) Hotelier Harry and Hotelier Bob are associated to the same hotel group. 2) hotel H1 = hotel H2 3) room number RN1 = room number RN2	1) Use Case A : Room R1 is successfully created by Actor A. -> Use Case B : Show room number exists error to actor B. 2) Use Case B : Room R2 is successfully created by Actor B.-> Use Case A : Show room number exists error to actor A.

8.2.3 Cancel hotel booking

3. Cancel hotel booking
    3.1. Description: Hotelier Harry tries to cancel booking of Heartbreak Hotel of Hotel Group Atlantis.
    3.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Admin Adam removes Harry's association to Hotel Group Atlantis.	Cancellation of bookings for a hotel can be done only by a hotelier who has an associated its parent hotel group.
2	Harry's profile cancels the same booking from a separate login.	A booking can only be cancelled once and using correct version number.
3	Customer Carly cancels the same booking.	A booking can only be cancelled once and using correct version number.
4	Customer Carly's profile changes dates of the same booking.	A booking can only be cancelled once and using correct version number.

    3.3. Choice of pattern: Combination of the following
        1.1.1. Controller checks using database requests.
        1.1.2. Optimistic offline lock
    3.4. Pattern implementation:
        1.1.1. At the time of cancel booking request, the hotelier session first checks if the booking is already cancelled using database query.
        1.1.2. If cancelled, returns bad request.
        1.1.3. If not cancelled, checks if hotel group id of booking being cancelled matches that of the hotelier sending the request.
        1.1.4. If not matching, then responds with bad request error.
        1.1.5. If matching, then updates booking and room bookings objects to reflect cancellation and then sends an update query to database.
        1.1.6. If by chance a cancel booking or change dates request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match the object number currently in database and returns server error.
    3.5. Rationale:
        i. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
        ii. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the
    3.6. Sequence Diagram:



Cancel Booking Test Case 3

3.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Remove hotelier HO1 from Hotel group.	Hotelier Harry	Cancel booking B1 of hotel H1 of hotel group HG.	1) hotelier HO1 = hotelier Harry	1) Use Case A : Remove hotelier request successful for Adam. -> Use Case B : Request unsuccesssful, show error unauthorized to Harry. 2) Use Case B : Request successful, booking B1 cancelled by Harry. -> Use Case A : Remove hotelier request successful for Adam.
2	Hotelier Harry	Cancel booking B1.	Hotelier Harry	Cancel booking B2.	1) Both instances of Hotelier Harry are logged in simultaneously from different access points. 2) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Actor A. -> Use Case B : Show error Booking already cancelled to Actor B. 2) Use Case B : Booking B2 cancelled by Actor B. -> Use Case A : Show error Booking already cancelled to Actor A.

3	Hotelier Harry	Cancel booking B1.	Customer James	Cancel booking B2.	1) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Harry. -> Use Case B : Show error Booking already cancelled to James. 2) Use Case B : Booking B2 cancelled by James. -> Use Case A : Show error Booking already cancelled to Harry.
4	Hotelier Harry	Cancel booking B1.	Customer Carly	Change dates of booking B2 from S1-E1 to S2-E2.	1) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Harry. -> Use Case B : Dates change for Booking B2 unsuccessful, show booking cancelled error to Carly. 2) Use Case B : Dates changed for booking B2 successfully by Carly to S2-E2. -> Use Case A : Booking B1 cancelled by Harry.

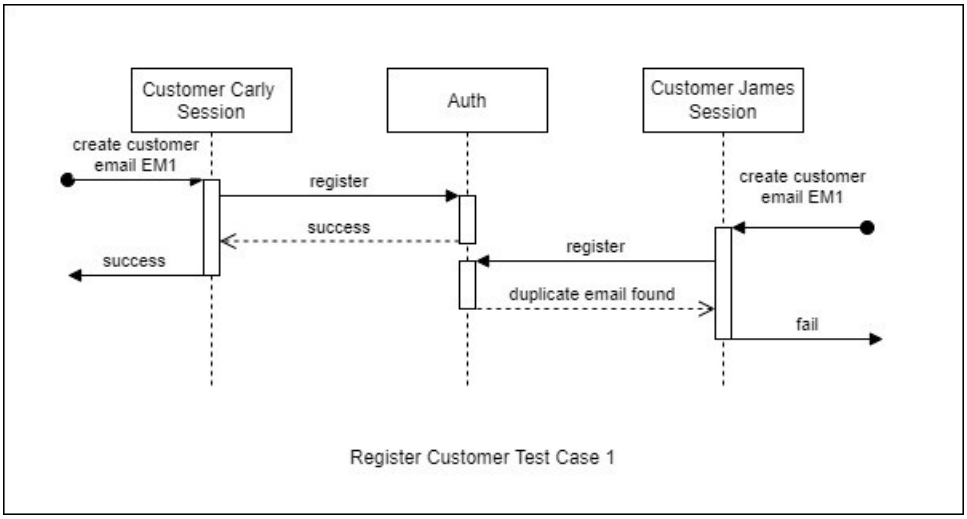
8.3 Customer Use Cases

8.3.1 Register on system

1. Register on system
    1.1. Description: User Ursula tries to sign up on the LANS hotels website using her email ID.
    1.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Another Customer Carly has already signed up using the same email ID provided by Ursula.	Email ID used for sign up for any system user must be unique.

    1.3. Choice of pattern: Directly handled within authorization mechanism.
    1.4. Pattern implementation : The registration conflicts are handled implicitly by authentication mechanism (auth0), which ensures that unique email addresses are used at the time of user sign up on system. Both duplicate checking and error communication is handled by the authorization mechanism at the time of sign up.
    1.5. Rationale:
        1.5.1. Reasoning: We chose this pattern as the authentication mechanism itself stores all user emails (user for login), as well as sending the user emails after registration to the database to store in app_user table (User details).
    1.6. Sequence Diagram:



Register Customer Test Case 1

    1.7. Test cases:

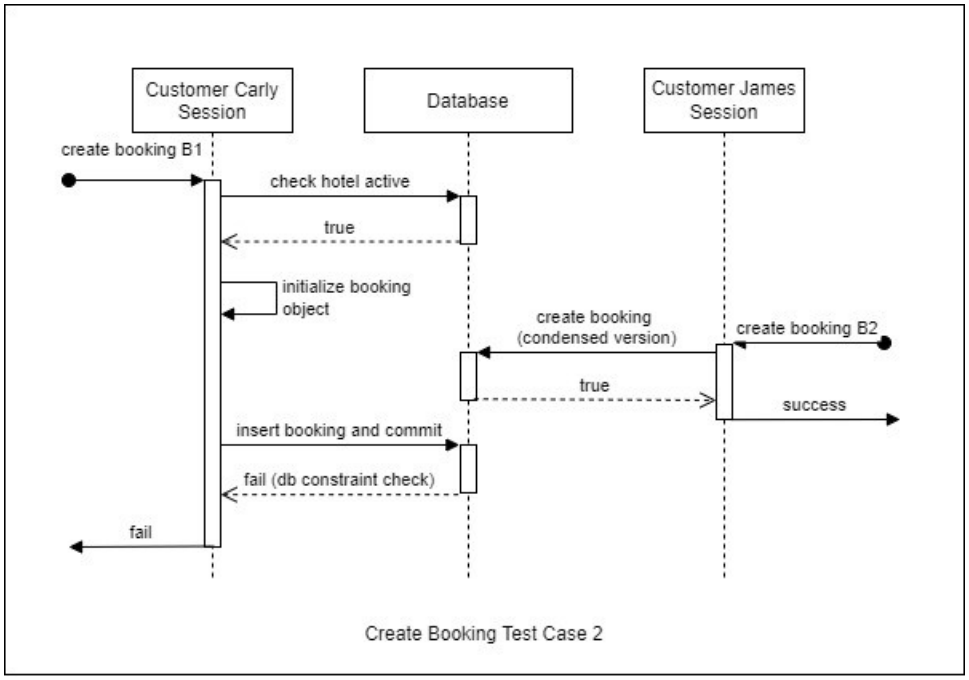
	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Customer Carly	Registers on website with email EM1.	Customer James	Registers on website with email EM2.	1) email EM1 = email EM2	1) Use Case A : Carly gets registered. -> Use Case B : James gets error saying email already used. 2) Use Case B : James gets registered, -> Use Case A : Carly gets error saying email already used

8.3.2 Create booking

2. Create booking
    2.1. Description: Customer Carly tries to make a booking for Heartbreak Hotel between for date range (S1-E1) with room set {R}. Each element of R has values of number of guests and main guest name associated to it.
    2.2. Possible Concurrency Conflicts:

No.	Conflicting Use Case	Reason
1	Heartbreak Hotel is delisted by Admin Adam before Customer Carly confirms booking.	New bookings can be made only for active/ non-delisted hotels.
2	Another Customer James makes a booking for Heartbreak Hotel with overlapping dates and common rooms to the booking of Carly.	For any single date, only one room booking can exist for any single room.
3	Another Customer changes the dates of an existing booking for Heartbreak Hotel which has common rooms with Carly's booking, to now have overlapping dates to the booking of Carly.	For any single date, only one room booking can exist for any single room.

    2.3. Choice of pattern:
        2.3.1. Controller checks using database requests.
        2.3.2. Database constraints.
    2.4. Pattern Implementation :
        2.4.1. At the time of create booking request, the customer session first checks if the hotel is delisted using database query.
        2.4.2. If delisted, returns bad request.
        2.4.3. If not delisted, then initializes bookings and room bookings and then sends an insert query to database.
        2.4.4. If the create booking conflicts due any of the rooms being requested not being available on those dates or hotel being delisted, the insert query will not execute due to the constraints defined to check room availability and hotel active status and results in an internal server error. If constraint is passed, insert goes through and gets committed.
    2.5. Rationale:
        2.5.1. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the database.
        2.5.2. The database constraints ensure that only valid data can be inserted into or updated within the database.
        2.5.3. Only while booking is being created (room validity and room availability checks and insertion) tables are locked, before being instantly released after transaction.
    2.6. Sequence Diagram:



Create Booking Test Case 2

2.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Delist hotel H1.	Customer James	Create booking B1 for hotel H2.	1) hotel H1 = hotel H2	1) Use Case A : Hotel H1 delisted by Adam. -> Use Case B : Booking request by James is unsuccessful. 2) Use Case B : Booking request by James is successful. -> Use Case A : Hotel H1 delisted by Adam.
2	Customer Carly	Create booking for hotel H1 for date range S1-E1 with rooms set {R1}.	Customer James	Create booking for hotel H2 for dates S2-E2 with rooms set {R2}.	1) hotel H1 = hotel H2 2) rooms set {R1} and rooms set {R2} have common elements 3) date range S1-E1 overlaps date range S2-E2	1) Use Case A : Carly's booking is successfully made. -> Use Case B : James booking request fails. 2) Use Case B : James's booking is successfully made. -> Use Case A : Carly's booking request fails.
3	Customer Carly	Create booking for hotel H1 for date range S1-E1 with rooms set {R1}.	Customer James	Change dates of booking B2 for Hotel H2 with room set {R2} from date range	1) hotel H1 = hotel H2 2) rooms set {R1} and rooms set {R2} have common elements 3) date range	1) Use Case A : Carly's booking is successfully made. -> Use Case B : James change date request fails. 2) Use Case B : James's change date is successfully made. -> Use

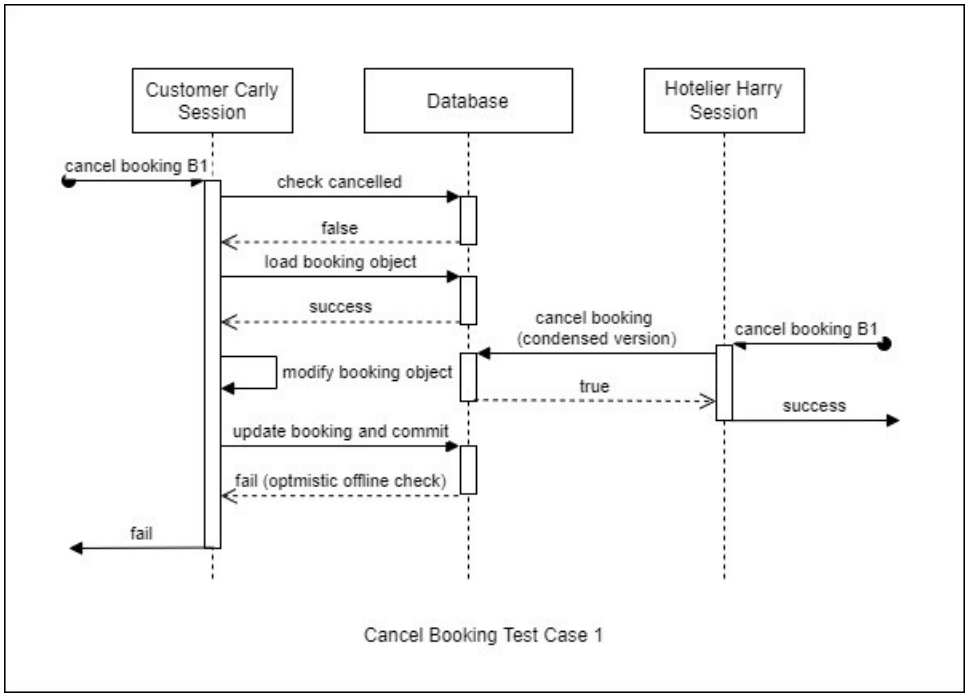
				S2-E2 to S3-E3.	S1-E1 overlaps date range S3-E3	Case A : Carly's booking request fails.

8.3.3 Cancel booking

3. Cancel booking
    3.1. Description: Customer Carly tries to cancel a booking at Heartbreak Hotel.
    3.2. Possible Concurrency Conflicts

No.	Conflicting Use Case	Reason
1	Hotelier Harry cancels the same booking.	A booking can only be cancelled using the correct version number.
2	Carly's profile cancels the same booking from a separate login.	A booking can only be cancelled using the correct version number.
3	Carly's profile changes dates of the booking from a separate login.	A booking can only be cancelled using the correct version number.

    3.3. Choice of pattern: Combination of the following
        3.3.1. Controller checks using database requests.
        3.3.2. Optimistic offline lock
    3.4. Pattern implementation:
        3.4.1. At the time of cancel booking request, the customer session first checks if the booking is already cancelled using database query.
        3.4.2. If cancelled, returns bad request.
        3.4.3. If not cancelled, checks customer id of booking being cancelled matches that of the user sending the request.
        3.4.4. If not matching, then responds with bad request error.
        3.4.5. If matching, then updates booking and room bookings objects to reflect cancellation and then sends an update query to database.
        3.4.6. If by chance a cancel booking or change dates request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match the object number currently in database and returns server error.
    3.5. Rationale:
        3.5.1. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
        3.5.2. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the
    3.6. Sequence Diagram:



Cancel Booking Test Case 1

3.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Hotelier Harry	Cancel booking B1.	Customer James	Cancel booking B2.	1) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Harry. -> Use Case B : Show error Booking already cancelled to James. 2) Use Case B : Booking B2 cancelled by James. -> Use Case A : Show error Booking already cancelled to Harry.
2	Customer Carly	Cancel booking B1.	Customer Carly	Cancel booking B2.	1) Both instances of Customer Carly are logged in simultaneously from different access points. 2) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Actor A. -> Use Case B : Show error Booking already cancelled to Actor B. 2) Use Case B : Booking B1 cancelled by Actor B.-> Use Case A : Show error Booking already cancelled to Actor A.

3	Customer Carly	Change dates of booking B1 for Hotel H1 with room set {R1} from date range S1-E1 to S2-E2.	Customer James	Cancel booking B2 for hotel H2 with room set {R2} for date range S3-E3.	1) hotel H1 = hotel H2. 2) room sets {R1} and {R2} have common elements. 3) date range S2-E2 overlaps S3-E3.	1) Use Case A : Dates changed for Booking B1 is unsuccessful for Carly. -> Use Case B : James cancel booking request is successful. 2) Use Case B : James cancel booking request is successful. -> Use Case A : Dates changed for Booking B1 is successful for Carly.

8.3.4 Change booking dates

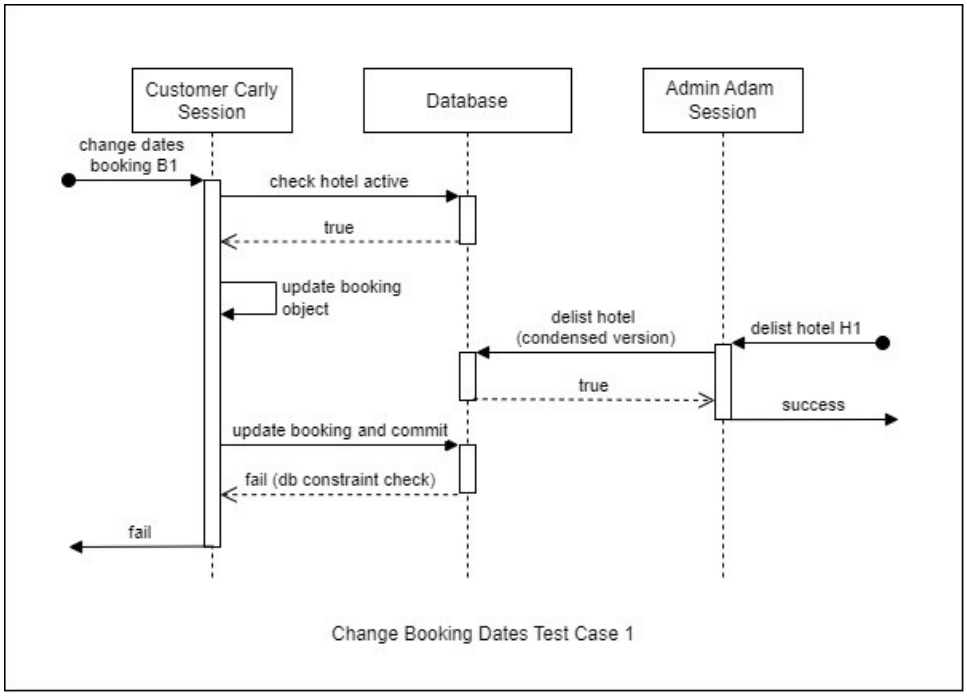
4. Change booking dates
    4.1. Description: Customer Carly tries to change booking dates of a booking at Heartbreak Hotel.
    4.2. Possible Concurrency Conflicts

No.	Conflicting Use Case	Reason
1	Heartbreak Hotel is delisted by Admin Adam before Customer Carly confirms booking date change.	Change of booking dates can be made only for active/ non-delisted hotels.
2	Hotelier Harry cancels the same booking.	Change of booking dates can be made only for active bookings.
3	Another Customer James makes a booking for Heartbreak Hotel with overlapping dates and common rooms to the new booking dates of Carly.	For any single date, only one room booking can exist for any single room.
4	Carly's profile cancels the same booking from a separate login.	Change of booking dates can be made only for active bookings.
5	Carly's profile changes booking dates for the same booking from a separate login.	Date change for a booking can only be made using the most recent version number.

    4.3. Choice of pattern: Combination of the following
        4.3.1. Controller checks using database requests.
        4.3.2. Optimistic offline lock
        4.3.3. Database constraints.
    4.4. Pattern implementation:
        4.4.1. At the time of change dates of booking request, the customer session first checks if the booking is already cancelled using database query.
        4.4.2. If cancelled, returns bad request.
        4.4.3. If not cancelled, checks if customer id of booking being date changed matches that of the user sending the request.
        4.4.4. If not matching, then responds with bad request error.
        4.4.5. If matching, then updates booking object to reflect date change and then sends an update query to database.
        4.4.6. If by chance a change dates request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match the object number currently in database and returns server error.
        4.4.7. If the date change conflicts due any of the rooms being requested not being available on those dates or hotel being delisted, the insert query will not execute due to the constraints defined to check room availability and hotel active status and results in an internal server error. If constraint is passed, update goes through and gets committed.
    4.5. Rationale:
        4.5.1. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
        4.5.2. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the database.

4.5.3. The database constraints ensure that only valid data can be inserted into or updated within the database. Locks booking and room bookings during update transaction, similar to create booking.

4.6. Sequence Diagram:



Change Booking Dates Test Case 1

4.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Delist hotel H1.	Customer James	Change dates of booking B for hotel H2 with room bookings set {RB} for dates S1-E1 to S2-E2.	1) hotel H1 = hotel H2	1) Use Case A : Hotel H1 delisted by Adam. -> Use Case B : Dates change request for booking B is unsuccessful, James receives error saying hotel is delisted. 2) Use Case B : Date change request for booking B by James is successful. -> Use Case A : Hotel H1 delisted by Adam.
2	Hotelier Harry	Cancel booking B1.	Customer Carly	Change dates of booking B2 from S1-E1 to S2-E2.	1) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Harry. -> Use Case B : Dates change for Booking B2 unsuccessful, show booking cancelled error to Carly. 2) Use Case B : Dates changed for booking B2 successfully by Carly to

						S2-E2. -> Use Case A : Booking B1 cancelled by Harry.
3	Customer Carly	Create booking for hotel H1 for date range S1-E1 with rooms set {R1}.	Customer James	Change dates of booking B2 for Hotel H2 with room set {R2} from date range S2-E2 to S3-E3.	1) hotel H1 = hotel H2 2) rooms set {R1} and rooms set {R2} have common elements 3) date range S1-E1 overlaps date range S3-E3	1) Use Case A : Carly's booking is successfully made. -> Use Case B : James change date request fails. 2) Use Case B : James's change date is successfully made. -> Use Case A : Carly's booking request fails.
4	Customer Carly	Change dates of booking B1 for Hotel H1 with room set {R1} from date range S1-E1 to S2-E2.	Customer Carly	Cancel booking B2.	1) Both instances of Customer Carly are logged in simultaneously from different access points. 2) booking B1 = booking B2	1) Use Case A : Dates changed for Booking B1 is successful for Actor A. -> Use Case B : Actor B cancel booking request is successful. 2) Use Case B : Actor B cancel booking request is successful. -> Use Case A : Dates changed for Booking B1 is unsuccessful for Actor A.
5	Customer Carly	Change dates of booking B1 having rooms set {R1} from date range S1-E1 to S2-E2.	Customer James	Change dates of booking B2 having rooms set {R2} from date range S3-E3 to S4-E4.	1) room sets {R1} and {R2} have common elements 2) date range S2,E2 overlaps S4,E4	1) Use Case A : Dates changed for booking B1 to date range S2-E2 for Carly. -> Use Case B : James's booking date change is unsuccessful. 2) Use Case B : Dates changed for booking B2 to date range S4-E4 for James -> Use Case A : Carly's booking date change is unsuccessful.

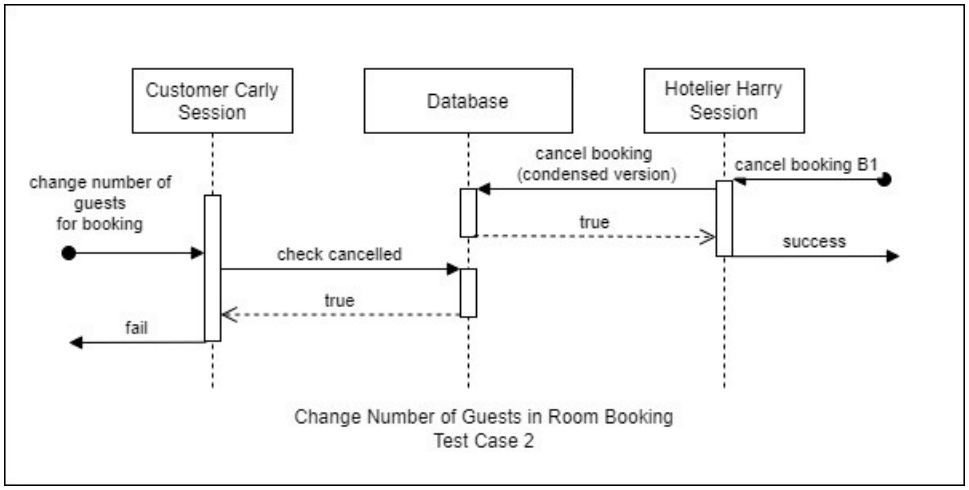
8.3.5 Change number of guests in room booking

5. Change number of guests in room booking
   5.1. Description: Customer Carly tries to change the number of guests of a room booking within a booking at Heartbreak hotel.
   5.2. Possible Concurrency Conflicts

No.	Conflicting Use Case	Reason
1	Heartbreak Hotel is delisted by Admin Adam before Customer Carly confirms room booking number of guests change.	Change of number of guests can be made only for active/ non-delisted hotels.
2	Hotelier Harry cancels the same booking.	Change of number of guests can be made only for active bookings.
3	Carly's profile cancels the same booking from a separate login.	Change of number of guests can be made only for active bookings
4	Carly's profile changes the number of guests of the room booking from a separate login.	Change of number of guests of a room booking can only be made using the most recent version number.

   5.3. Choice of pattern: Combination of the following
      5.3.1. Controller checks using database requests.
      5.3.2. Optimistic offline lock
      5.3.3. Database constraints.
   5.4. Pattern implementation:
      5.4.1. At the time of change number of guests of booking request, the customer session first checks if the booking is already cancelled using database query.
      5.4.2. If cancelled, returns bad request.
      5.4.3. If not cancelled, checks if the customer id of booking being changed matches that of the user sending the request.
      5.4.4. If not matching, then responds with bad request error.
      5.4.5.
      5.4.6. If matching, checks if the new value of number of guests is same as before.
      5.4.7. If yes, then sends bad request error.
      5.4.8. If no, then updates the room booking object to reflect new number of guests and then sends an update query to database.
      5.4.9. If by chance a n update number of guests request executes in between initial check and update, the database will use the optimistic offline checking to see that version number of object does not match the object number currently in database and returns server error.
   5.5. Rationale:
      5.5.1. The optimistic offline locking maintains the desired property of liveness and prevents scenarios involving dirty reads or lost updates.
      5.5.2. The checks using database requests uphold ACID properties and ensure that the update request is valid before going through to the
      5.5.3. The database constraints ensure that only valid data can be inserted into or updated within the database.

5.6. Sequence Diagram:



5.7. Test Cases:

	Actor A	Use Case A	Actor B	Use Case B	Conditions	Expected Results
1	Admin Adam	Delist hotel H.	Customer James	Change number of guests from N1 to N2 for room booking RB in booking B of hotel H.	1) hotel H1 = hotel H2	1) Use Case A : Hotel H1 delisted by Adam. -> Use Case B : Number of guests change request by James is unsuccessful, receive error saying hotel is delisted. 2) Use Case B :-> Use Case A : Number of guests change request by James is successful.
2	Hotelier Harry	Cancel booking B1.	Customer Carly	Change number of guests of room booking RB1 in booking B2.	1) booking B1 = booking B2	1) Use Case A : Booking B1 cancelled by Harry. -> Use Case B : Number of guests change for booking B2 unsuccessful, show booking cancelled error to Carly. 2) Use Case B : Number of guests changed for Booking B2 successfully by Carly. -> Use Case A : Booking B1 cancelled by Harry.

3	Customer Carly	Cancel booking B1.	Customer Carly	Change number of guests of room booking RB2 in booking B2 from NG1 to NG2.	1) Both instances of Customer Carly are logged in simultaneously from different access points. 2) booking B1 = booking B2	1) Use Case A : Cancel booking successful by Actor A. -> Use Case B : Number of guests change request for booking B1 unsuccessful, show booking cancelled error to Actor B. 1) Use Case B : Number of guests change request by Actor B for booking B1 is successful.-> Use Case A : Cancel booking successful by Actor A.
4	Customer Carly	Change number of guests from NG1 to NG2 of room booking RB1 in booking B1.	Customer Carly	Change number of guests from NG3 to NG4 of room booking RB2 in booking B2.	1) Both instances of Customer Carly are logged in simultaneously from different access points. 2) booking B1 = booking B2 3) room booking RB1 = room booking RB2	1) Use Case A : Number of guests changed for room rooking to NG2 by Actor A.-> Use Case B : Number of guests changed for room rooking to NG4 by Actor B. 2) Use Case B : Number of guests changed for room rooking to NG4 by Actor B. -> Use Case A : Number of guests changed for room rooking to NG2 by Actor A.

8.4 Summary of Outcomes

All the above test cases for concurrency have been validated to match expected outcomes taking into account execution order of the use cases involved. The team has made sure to cover all major concurrency conflict scenarios and while there is potential of some not being identified, the controller checks, database checks and constraints and optimistic pattern implementations should still work in the right manner to deal with other scenarios.

9 Conclusion

Our application uses a cohesive and simple approach to handling concurrency. Its simplicity means that it is easy to maintain and extend. However, limitations of our one transaction per request model creates the risk of frequent lost work for users. The lost work was considered a reasonable tradeoff to make it as simple to reason and debug about complex concurrent usage. Given initial low throughput of usage in the platform there is an expected low frequency of collisions. Furthermore, lost work in this our application is considered low severity, so complex locking schemas (such as pessimistic offline locking) were deemed unnecessary. However, should throughput increase as system usage grows, frequency of lost work may increase and degrade usability. As such, we discussed opportunities for improving the concurrency strategy over time, for example introducing pessimistic offline read locks with timeouts for booking flows.

Extensive and thorough concurrency tests were executed and documented. Throughout the testing process we were able to improve our implementation, guided by iteratively identifying and fixing failing tests. As of submission, our application has passed all tests described in our test document. Furthermore, we conducted and documented further non-concurrent tests to verify the full functionality of our system. Our test suite demonstrates that our system guarantees security and data integrity and gives useful feedback to users when their work is lost.

Our implementation, tests and accompanying documentation – including this report – thoroughly demonstrate our team's understanding of concurrency problems and patterns for solutions in an enterprise application.

10 References

Fowler, M. (2015). *Patterns of enterprise application architecture*. Boston, Mass. ; Munich: Addison-Wesley.

Evans, E. (2014). *Domain-driven design : tackling complexity in the heart of software*. Boston, Mass. ; Munich: Addison-Wesley.