Architecture Document

SWEN90007

SWEN90007 Software Design and Architecture

Hotel Booking System
Team: Alphecca

In charge of:
Xiaotian Li 1141181 xiaotian4@student.unimelb.edu.au
Haimo Lu 1053789 haimol@student.unimelb.edu.au
Edison Yang 1048372 lishuny@student.unimelb.edu.au
Yifei Wang 1025048 yifewang2@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
10/10/2022	01.00-D01	Initial draft	Xiaotian Li
11/10/2022	01.00-D02	Added some concurrency issue, test	Xiaotian Li
13/10/2022	01.00-D03	Report flow rearranging	Edison Yang
14/10/2022	0.1.00-D04	Add details to solution and test	Xiaotian Li
15/10/2022	0.1.00-D05	Add details to class diagram	Xiaotian Li
15/10/2022	01.00-D06	Add some details to rationale	Haimo Lu
16/10/2022	01.00-D07	Add system sequence diagram for section 3.2.4, 3.2.5	Haimo Lu
16/10/2022	0.100-D08	Add details to solutions description	Xiaotian Li
17/10/2022	0.100-D09	Improve quality of sequence diagram	Xiaotian Li
18/10/2022	0.100-D10	Adding testing for each of the concurrency issue	Edison Yang, Yifei Wang
18/10/2022	0.100-D11	Add sequence diagrams	Xiaotian Li
19/10/2022	0.100-D12	Add a Jmeter test result	Xiaotian Li
20/10/2022	01.00	Finalizing the report	Xiaotian Li, Edison Yang, Haimo Lu, Yifei Wang

Contents

1. Introduction

This document specifies the Alphecca hotel booking system's architecture, describing its main concurrency issues, solutions, choice of patterns, details and test strategy and outcome.

1.1 Proposal

In this report document, some updated application class diagrams will firstly be discussed. Then this report will be focusing on discussing the concurrency functionality of the hotel booking system. It starts with introducing the concurrency patterns that have been implemented in the system, then it addresses the different concurrency issues and scenarios depending on the project, choices, design and the implementation of the application. After that, it demonstrates some tests according to different concurrency scenarios and the corresponding test results.

2. Logical View

Updated application class diagrams are shown in this section. The updated information are summarized as below:

- The whole application class diagram was separated into several parts to ensure that they are reader friendly. It also makes the system easier to understand.
- Domain classes have association with each other directly, instead of IDs. Source codes are modified to match the relationship for domain patterns, by adding fields with specific class type.
- Removed redundant information.

The application class diagram covers: Controllers, Business Logic Objects, Data Mappers and domain entity objects.

2.1 Controller and Business Logic Object Class Diagram

Controller classes are responsible for handling incoming user requests, authentication, parsing request parameters and constructing response objects.

Business logic object(Blo) classes are responsible for wrapping business logic codes, and exposing methods for Controllers to use, so as to make the project convenient to maintain.

The Figure below illustrates the UML class diagram of Controllers and Business Logic Objects in the hotel booking application system. For presentability and simplicity of the diagram, relationships with minor helper classes, Data mapper classes and domain classes are omitted.

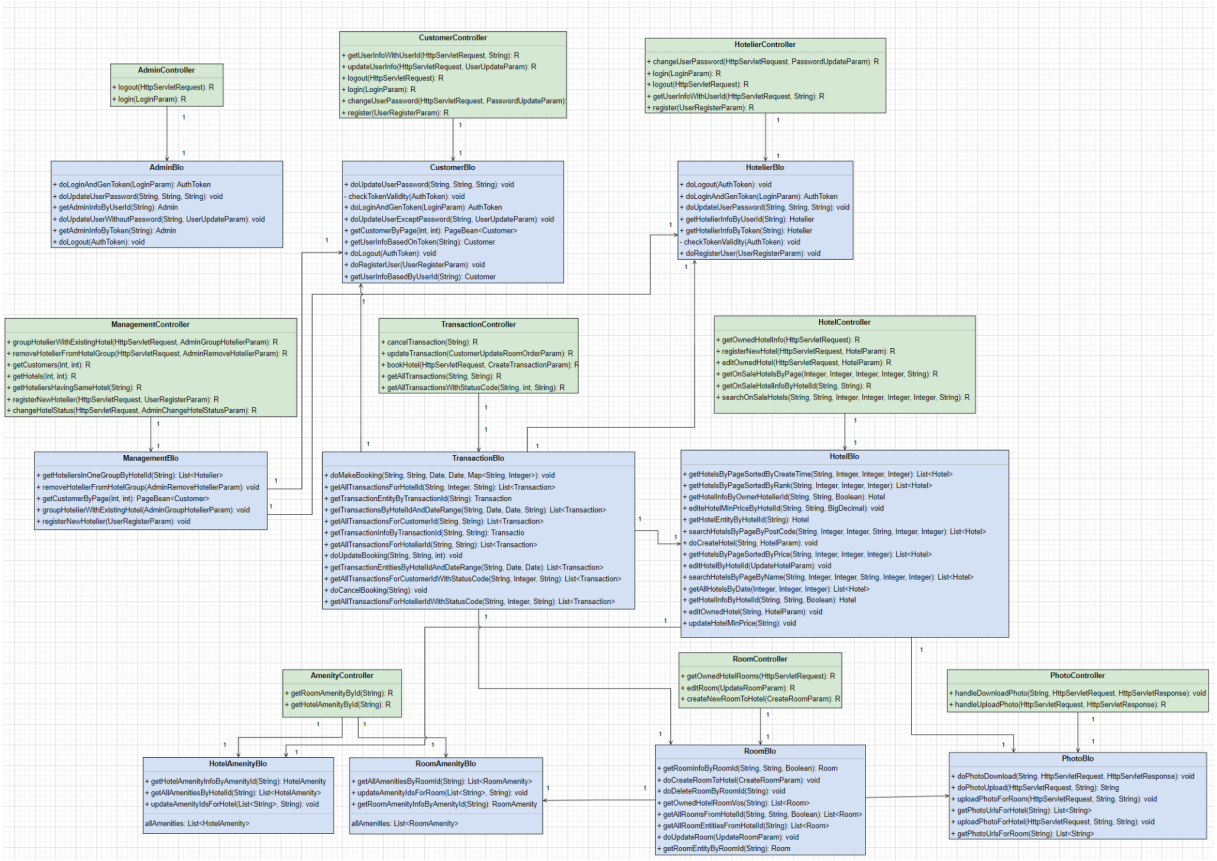


Figure 2.1 *Controllers and Business Logic Objects Class Diagram*

The system is using dependency injection. All Blo classes are injected into corresponding Controller classes as the one to one association relationship. Also, Blo classes have association relationships between each other:

- Admin, Customer, Hotelier Blo: handles user register, profile, and authentication;
- Management Blo: handles administrator management;
- Hotel/Room Blo: handles hotels/rooms creation, information editing and so on;
- Hotel/Room Amenity Blo: handles amenity logic;
- Transaction Blo: “Transaction” in transaction layer means “Booking” or “Trade”, as is used in the project specification. It is NOT relevant to the Transaction in the database.

2.2 Domain Objects and Data Access Object Class Diagram

Updated UML Class Diagram for domain classes are shown below. In the diagram, for simplicity, some getters and setters are omitted.

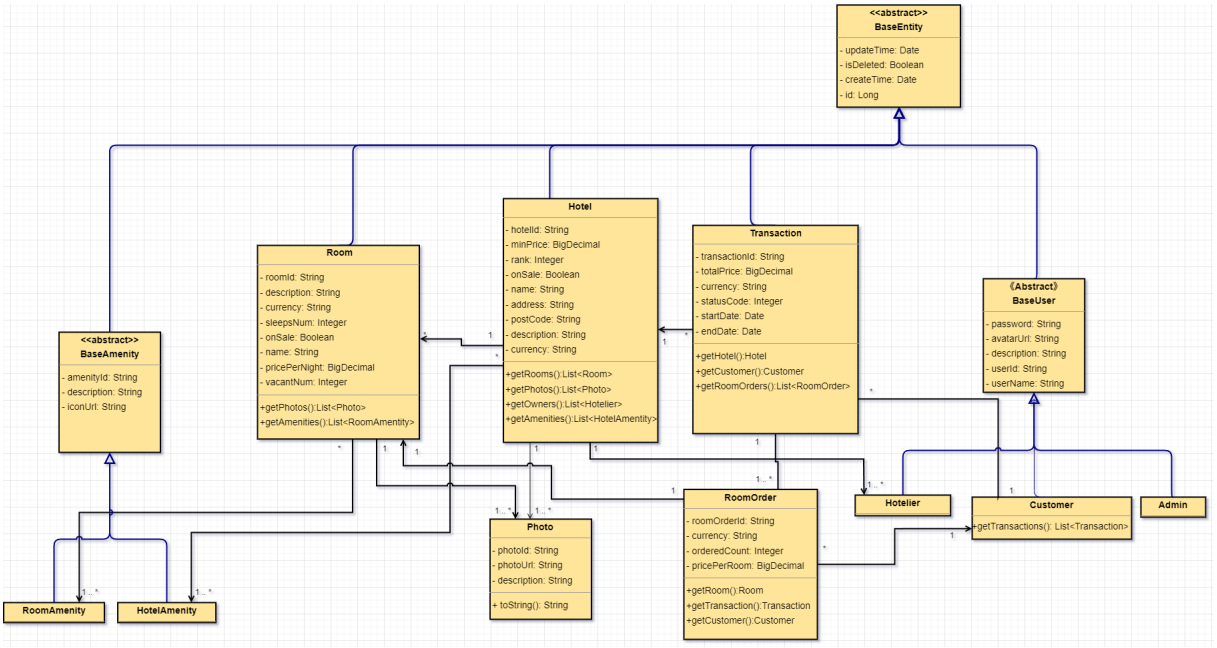


Figure 2.2.1 *Domain Objects and Data Access Object Class Diagram*

As for updating, we changed source codes of entity classes by adding class fields to achieve associate coupling requirements of the domain pattern, rather than using string IDs to achieve coupling. Since we have already defined domain's business behavior in business logic objects and we have limited time to remodify them, these concrete entity classes remain to hold data, but without some complex business behavior methods.

For the system, Data mapper is used as the data source. Data Access Object (Dao) acts as the Data Mapper of the system. Dao specifies a mediator/mapper that separates the in memory domain objects from the underlying database, and it will be used by Blo.

The updated diagram below shows relationships among Dao, Blo and Domain objects. Since Blo object's details have been descriptive in the section 2.1, details and relationships of Blo objects in the classes diagram below are omitted for simplicity.

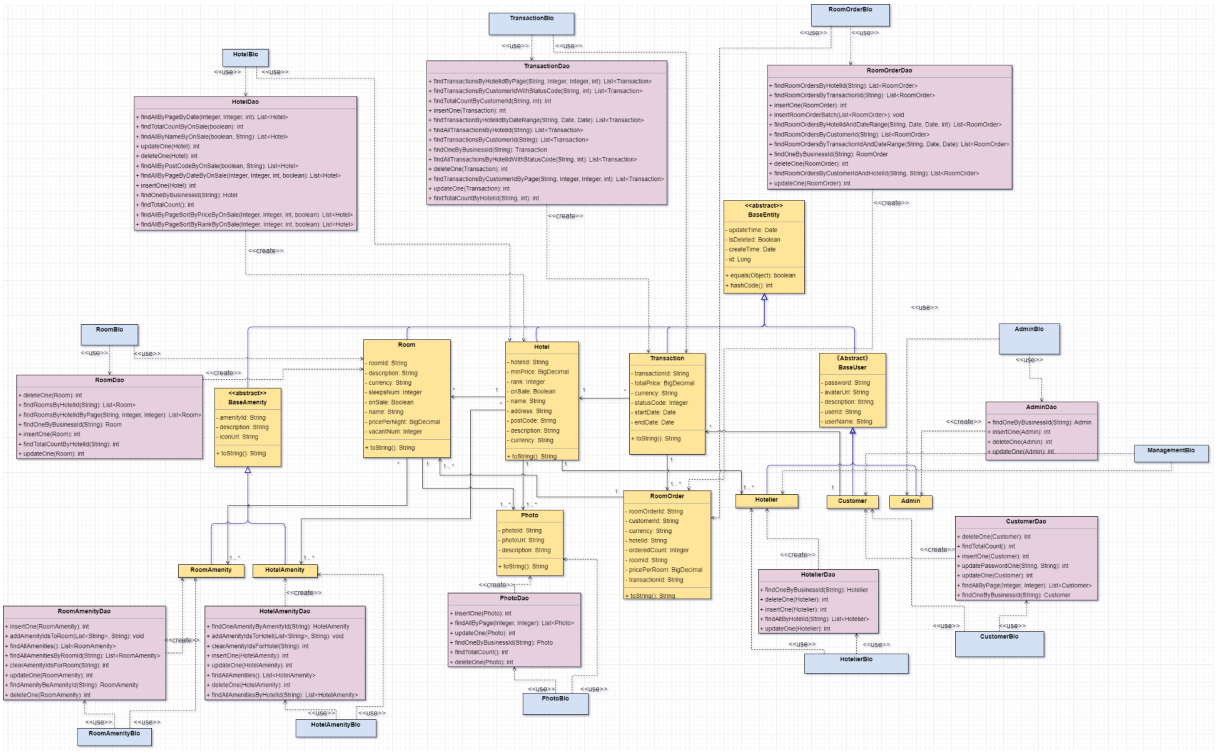


Figure 2.2.2 *Class Diagram of Dao, Blo and Domain Objects*

As we can see from the diagram above. Dao objects take responsibility to perform database I/O and create domain objects in the system. Blo objects use Dao objects to create domain objects, then use created domain objects to perform business logic.

3. Process View

3.1 Concurrency Patterns

We used 2 concurrency patterns to handle potential concurrency issues that may arise during the operation of our system which are the Optimistic offline lock and Pessimistic offline lock.

3.1.1 Optimistic Offline Lock

Optimistic offline lock associates a version number to records in the database. Conflicts are detected at commit time. The version number will be checked before the transaction commits. Otherwise, the transaction will rollback. This pattern can be used when the probability of conflict between concurrent transactions is low. We implemented the pattern in the Dao objects' update methods, and added version fields to data records. We have details in sections below

3.1.2 Pessimistic offline lock

It prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data. Unlike optimistic offline lock, the conflicts are avoided by using pessimistic offline lock.

For pessimistic offline lock, we use exclusive read lock with lock manager in the system. This lock type is bad in liveness, but it is easy to implement.

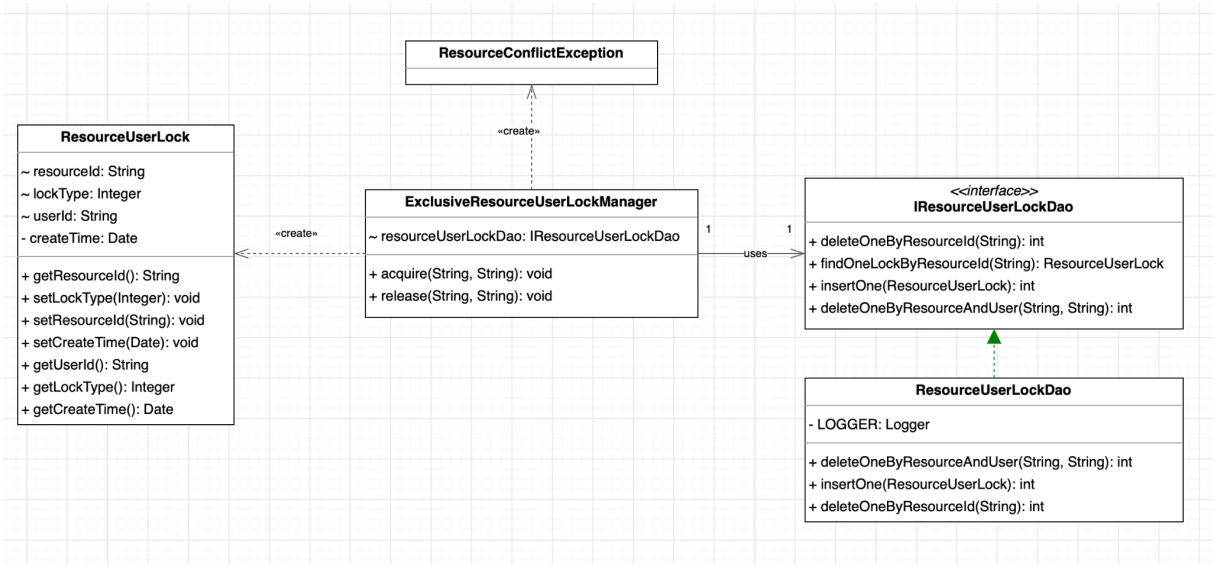


Figure 3.1.2 *Class Diagram of Pessimistic Offline Lock*

In our project, we implemented ExclusiveResourceUserLockManager and relevant classes, to achieve the pessimistic offline lock. Please refer to the details below for system sequence diagrams.

3.2 Concurrency Issues and Solutions

3.2.1 [Lose Updates] Hoteliers Update shared Hotel information

The system can be accessed by multiple hoteliers at the same time. They may edit the shared hotel information simultaneously. In this case, lost updates will occur.

Supposed Hotelier A and Hotelier B are updating the hotel info at the same time. Since they start the editing transaction in the system simultaneously, the problem of lost update could occur as shown in the diagram below:

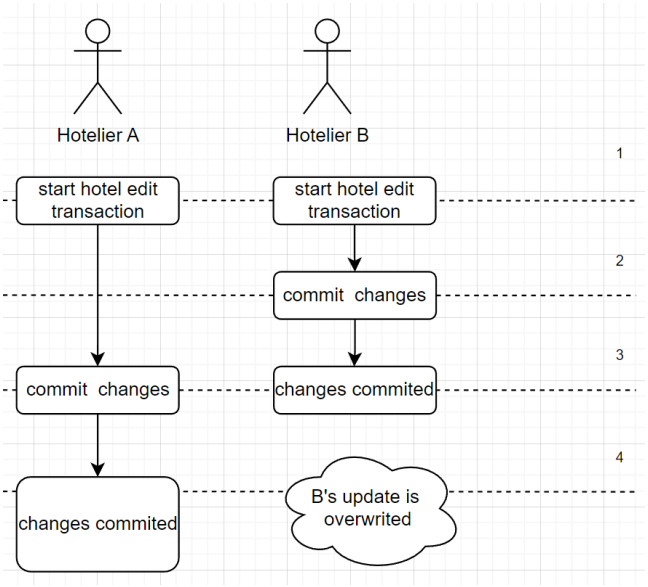


Figure 3.2.1.1 *Sequence Diagram of Concurrency Issue*

As shown in the diagram, updating transactions simultaneously started at step 1. At this stage, two transactions are accessing shared hotel data from the database at the same time. Then, B's updates are persisted in the database before A does. Then, A's updates will be persisted in the database later. As a result, Hotelier A's change overwrites Hotelier B's change in the database.

Solution:

Optimistic Offline Lock can be used to handle this issue.

Rationale:

- Frequency of conflicts is low: The hotel information, such as hotel name, address, postcode are usually more stable compared to the room info. Modifying these types of information by hoteliers will be less frequent.
- Severity of conflicts is low: Hotel information is often brief. The hotelier can try to commit again once he notices the submission is unsuccessful.
- Simplicity: Just add logic to several methods of data mapper and add the version field in the data record.

The corresponding solution is shown in the figure below:

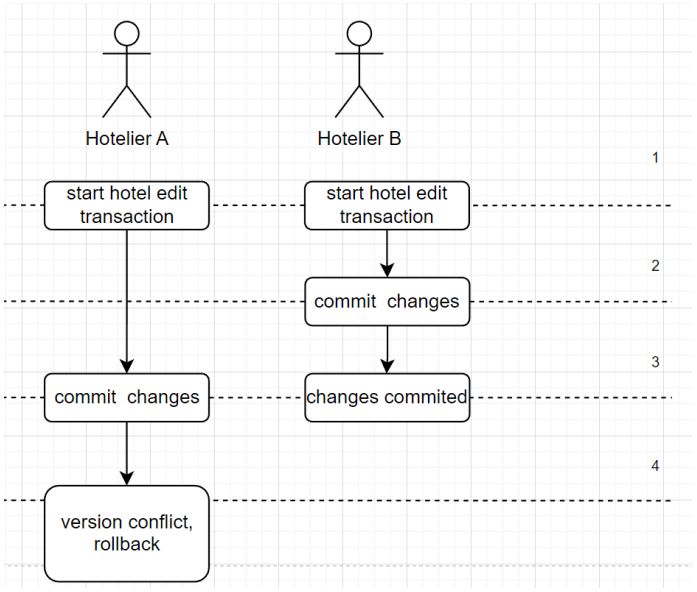


Figure 3.2.1.2 *Example of Optimistic Offline Lock*

As shown in *figure 3.2.1*, two transactions start at the same time in step 1. They both access shared hotel data simultaneously to get original information with version, and B's changes are persisted before A did. In step 3 and 4, when Hotelier A tries to perform an update in the commit stage, the system will check the record version from the database. Since the version is not the same as the data version in step 1, Hotelier A's transaction will rollback. Hotelier A will receive a notice saying that "The data have been updated, please try again".

As a result, hotelier A is able to know that the data has changed, and check the latest data by refreshing his webpage. If hotelier A starts the updating transaction again, the updating will be successful because there are no conflicts.

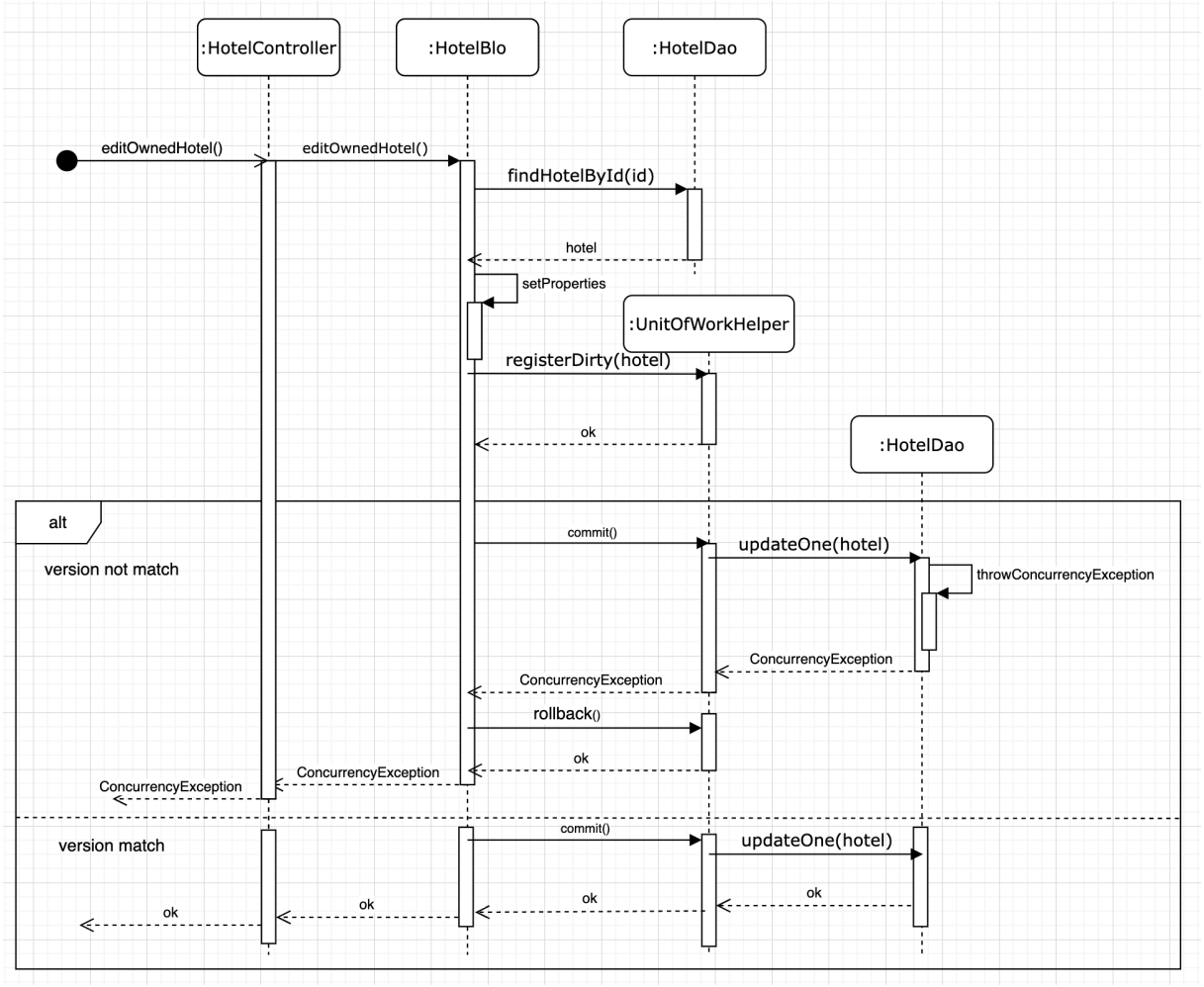


Figure 3.2.1.3 *Sequence Diagram of Backend System*

As the system sequence diagram illustrates, at the start of the editing system transaction, the shared hotel information is read from the database with a version number, the system can record its version. When the transaction tries to commit the updates, HotelDao's UpdateOne method will try to modify the target data record with the specific version.

If the target version is not found in the database in the UpdateOne method, HotelDao will call throwConcurrencyException method, so that it can fetch the latest record from the database to find out the details of the issue. In the throwConcurrencyException method:

- If there are no records in the database, the data has already been deleted by other users;
- If the latest version from db is greater than the target version, the data has already been modified by other users;
- If the latest version from db is less than the target version, this is an unexpected exception,and could be caused by a mistaken implementation for the version increase logic.

Once the exception has been raised, the unit of work helper will rollback the whole transaction.

On condition that there are no conflicts, i.e. if two versions are matched, the data record is not modified by other users, the update can be committed. The version field of the record will be added by 1.

Please refer to Section 4.1 for the corresponding testing and outcome.

3.2.2 [Lose Updates] Hotelier Group Edit the Room Information

Editing room info is a business transaction, which is composed of multiple system transactions including getting room info to render the editing page, and submit the updated form.

Updating conflicts will occur when multiple hoteliers are editing the same room they own. For example, Hotelier A finds an issue related to the description of room R. At the same time, Hotelier B finds an issue related to the pricing of the same room. They both start to edit the room info with the same original data as shown in the figure below:

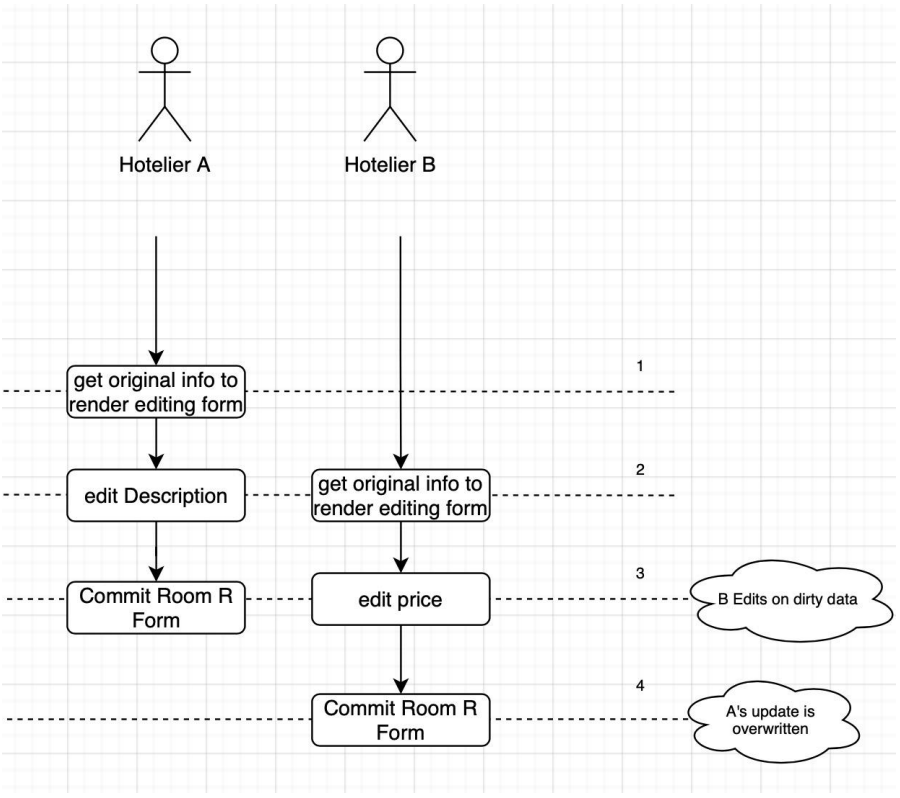


Figure 3.2.2.1 *Sequence Diagram of Concurrency Issue in Editing Room Info*

As is shown in *figure 3.2.2.1*, both Hotelier A and Hotelier B want to edit one shared room info but in different data fields. To render the editing form, they all access the API to get original room info. In step 3, Hotelier A finishes his work and commits the change. However, Hotelier B doesn't know the changes from Hotelier A and continues on working on the dirty original data. In step 4, Hotelier B's commit will overwrite the changes made by Hotelier A in step 3.

Solution:

Pessimistic offline lock's exclusive read lock type can be used here to handle the issue.

As for reasons:

- Severity of conflicts could be high: Room info contains more data fields than hotel info does. Work/data loss caused by Optimistic one will be painful. A hotelier might want to edit lots of information about a room.
- Frequency of conflicts is high: Room info can be changed more frequently than Hotel due to likely sales events and business seasons, especially in the price. In addition, it is very likely that more than one hotelier is editing the room information at the same time, for example, it is common hotelier group distribute their work in editing hotel information or room information (one could edit the description of a room, and the other could edit the price of a room or number of bed it has).
- Low liveliness can be tolerated: There are not so many hoteliers managing the hotel and room, so low liveliness may not cause big trouble.
- Exclusive read lock can prevent inconsistent reads in this scenario. Also, since we have limited time for development, implementing exclusive read lock can be easier.

The workflow of the pessimistic offline lock is shown in the figure below:

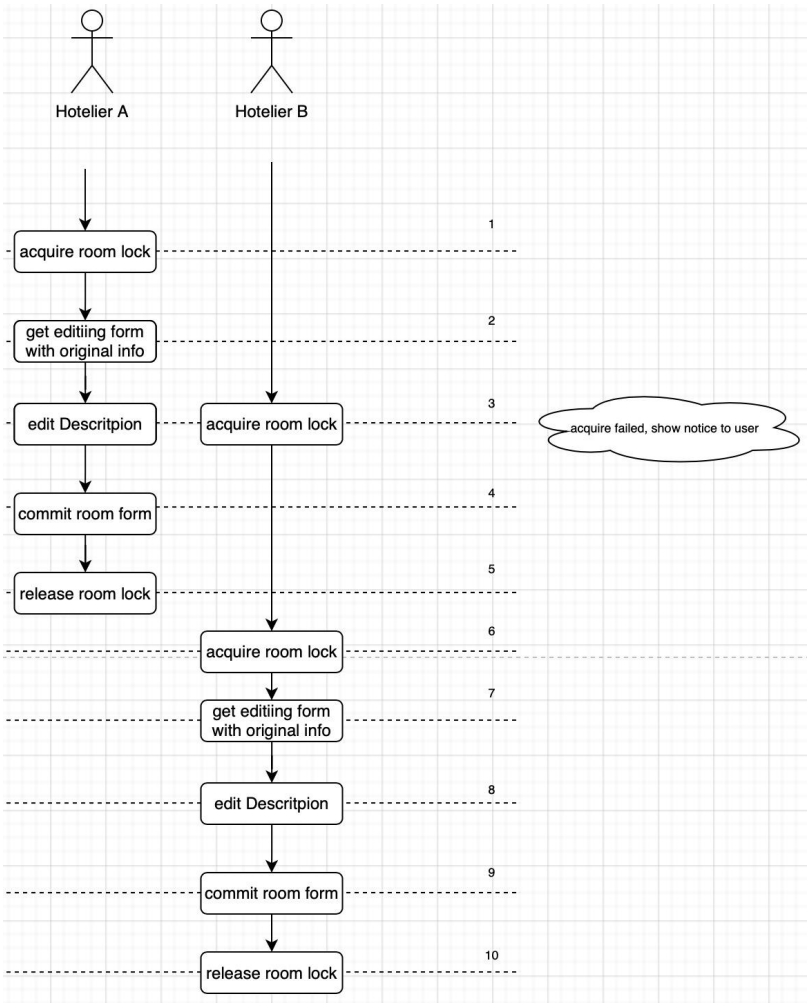


Figure 3.2.2.2 *Example of Pessimistic Offline Lock*

As is shown in the *figure 3.2.2.2*, in step 1, hotelier A will acquire an exclusive lock for the shared room, and he is working on the editing page. If hotelier B wants to access the room editing page, he

needs to acquire a lock, but the system will reject it by throwing a concurrency exception in step 3. After hotelier A submits and releases the room lock in step 5, the other hotelier can access the editing page.

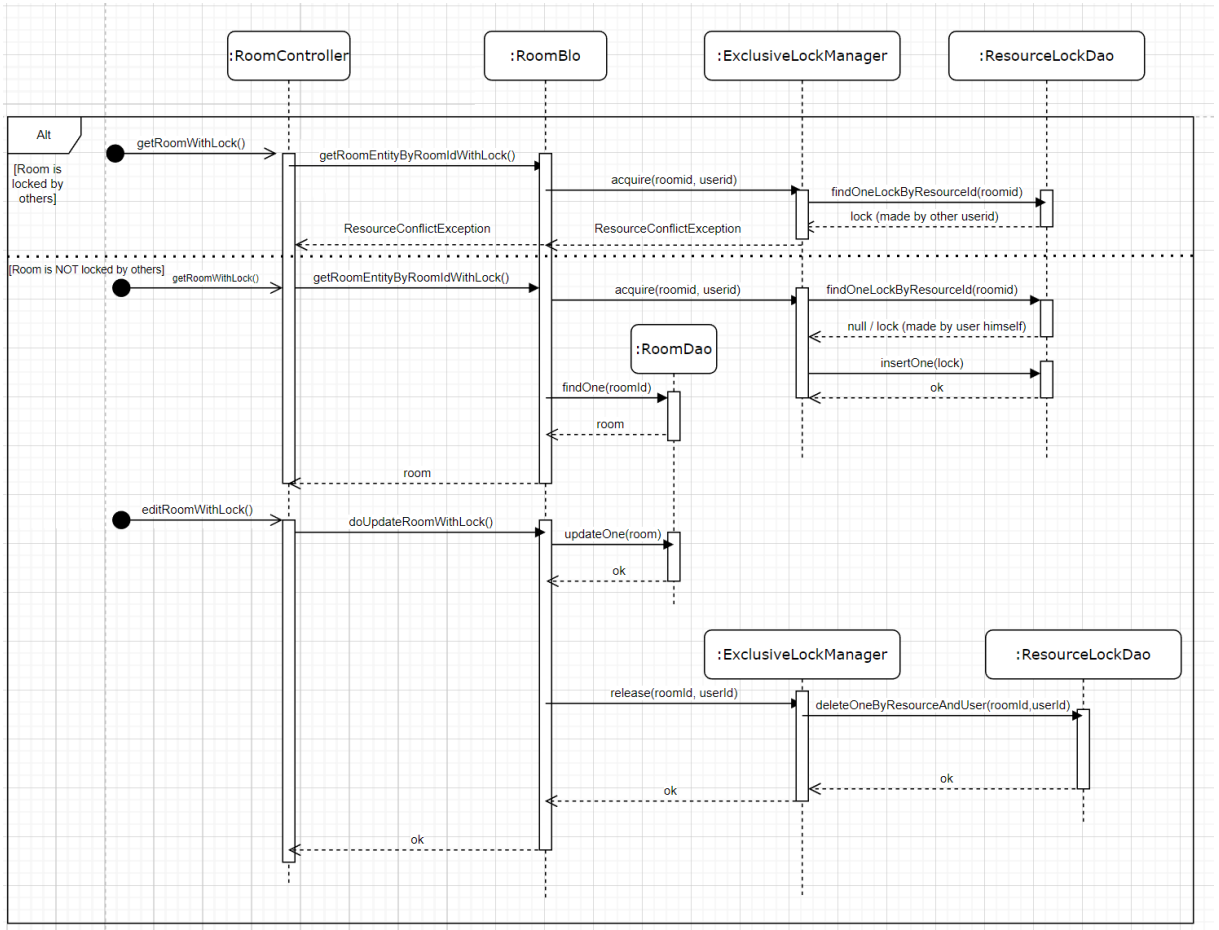


Figure 3.2.2.3 *Sequence Diagram of Backend System*

The sequence diagram here shows two alternatives. The first alternative happens when the room the hotelier is trying to edit is already locked by another hotelier. In this case, after receiving the request from the frontend app, the RoomController calls [GET]editRoomWithLock() with relevant parameters to pass the request to lock room to the business logic layer (RoomBlo). In the corresponding method, the RoomBlo calls acquire() to the ExclusiveResourceUserLockManager, trying to acquire the lock.

Since the room is already locked by others, the ResourceLockDao will return an existing lock record after findOneLockByResourceId called by ExclusiveResourceUserLockManager, which triggers the ResourceConflictException. The exception then sends back to the frontend app, blocking the hotelier from accessing the locked resource.

The second alternative differs after the acquire() phase, after findOneLockByResourceId(resourceId) called by ExclusiveResourceUserLockManager, there is no lock on that room, so the lock can be acquired by the hotelier. The RoomBlo then sends back the message about the initial information for the editing page.

After editing and clicking the submission button, the frontend then sends a request to submit the form. RoomController will call doUpdateRoomWithLock() in the RoomBlo, and it calls the RoomDao to update the room information, and commit the changes in the UnitOfWorkHelper. Lastly, the RoomBlo calls ExclusiveResourceUserLockManager to release the lock.

Please refer to Section 4.2 for the corresponding testing and outcome.

3.2.3 [Deadlock] Hotelier Group Edit the Room Information

This lock protocol we mentioned in 3.2.2 can cause deadlock. For example, if Hotelier A is editing a room's information but his browser or computer crashes suddenly. As a result, he can not submit the work. This scenario will lock the resource for a long time. If the pessimistic lock only locks on the resource ID, Hotelier A will never be able to re-enter the editing form and release the lock. This scenario can be illustrated by figure shown below:

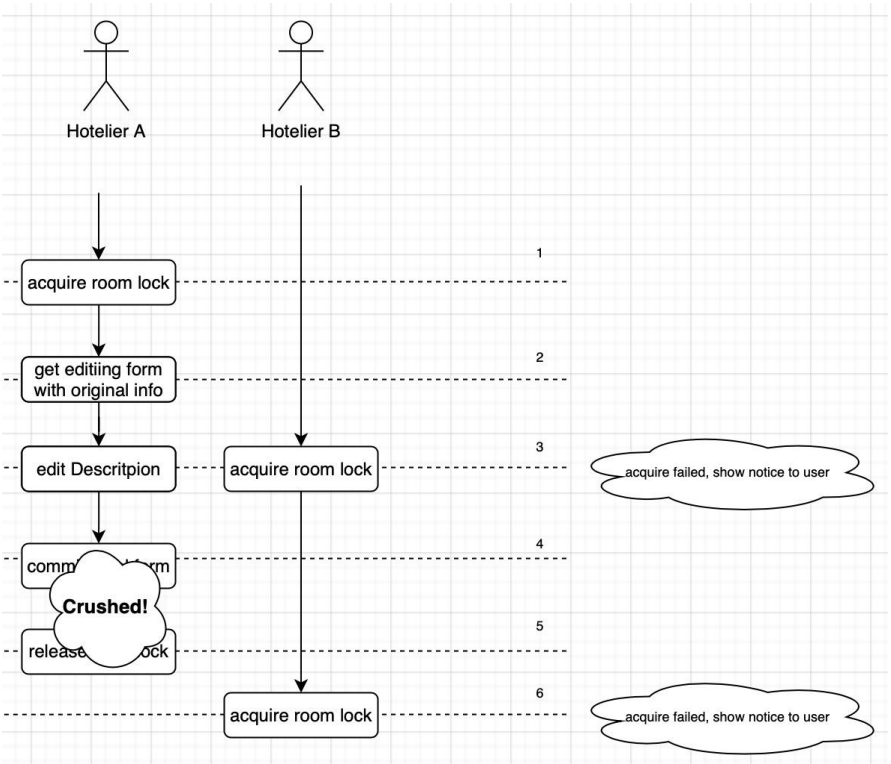


Figure 3.2.3.1 *Example of Deadlock*

Solution:

One possible way to handle this problem is to make use of the owner id field of the lock. The pessimistic lock will use the pair of resource id and owner id to match the lock ownership such that the lock owner can reenter the form, submit it and release the lock. Also, having the owner id field can prevent accidental release of other user's locks.

However, this strategy may not be enough in the real world. For example, if a failure of electricity occurs while the user is editing a form, the user can not immediately recover his working flow, which can cause a delay. Also, the user may not realize the importance of submission behavior that could release a lock, he may forget about it and never go back to the editing page, so that the pessimistic lock is never released.

To solve this potential problem, we have attached an expiration time for the exclusive lock. When acquiring the lock, the system will check the expiration time of the lock record. If the lock is outdated, the lock will be released. Since editing room info may not take a long time, having an expiration time such as 2 minutes can prevent deadlock if the editing user is crashed.

However, it can still cause troubles if the lock expires while the user is editing, then other users will interfere with the editing transaction. We could implement a watchdog mechanism to renew the lock if the user is still editing. In other words, we can let the frontend automatically request the acquire lock api to renew the existing lock in case that the user is still in the editing status of the web page. Once the frontend is shutdown and unable to renew the lock, the lock will be expired by the backend logic. The workflow for the frontend is shown below:

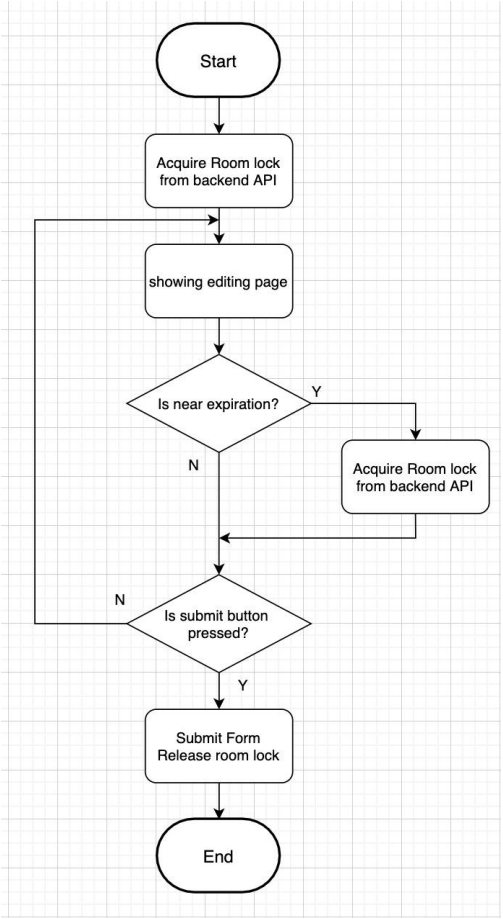


Figure 3.2.3.2 *Example of renewing lock*

With this solution, the pessimistic lock can make the editing business transaction not only isolated, but also robust to crushes.

As for implementation, several fields are defined in the lock database table, as is shown below:

Table Field	Type	Description
resource_id	varchar	Locked data record id.
user_id	varchar	Id for the owner of the lock. Can be used to reenter the lock for

		lock owners, or prevent releasing other's locks by accident.
create_time	timestamps	Time when the lock is acquired. Can be used in expiration logic.

In the *ExclusiveResourceUserLockManager* 's acquire method, the create_time of the lock record from the database is checked. If the create_time exceeds expiration period, the existing lock will be released, and a new lock is acquired.

The lock expiration logic is shown below. If the lock is expired, the original one will be removed, and a new lock with new create_time is created and inserted in the lock table.

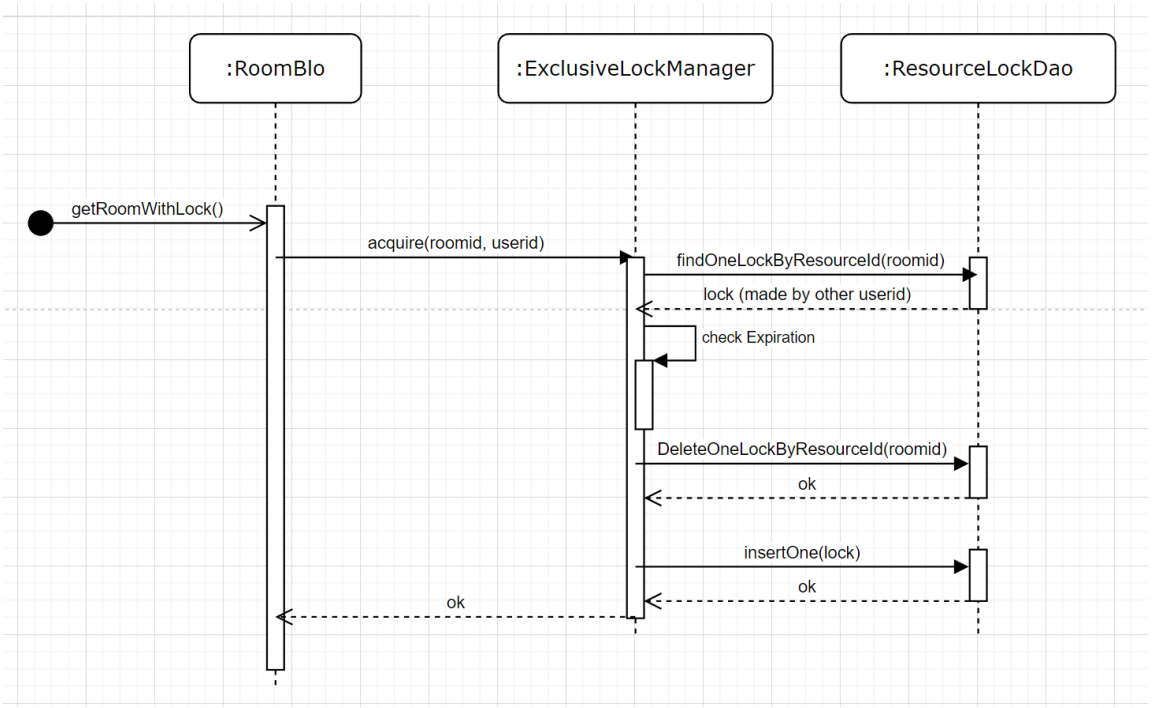


Figure 3.2.3.3 *Sequence Diagram of Backend*

Please refer to Section 4.3 for the corresponding testing and outcome.

3.2.4 [Isolation Issue] Customers Booking Hotels

The other concurrency issue is that while booking hotels, concurrency bookings can cause hotels to be over quota, as is shown in the figure below.

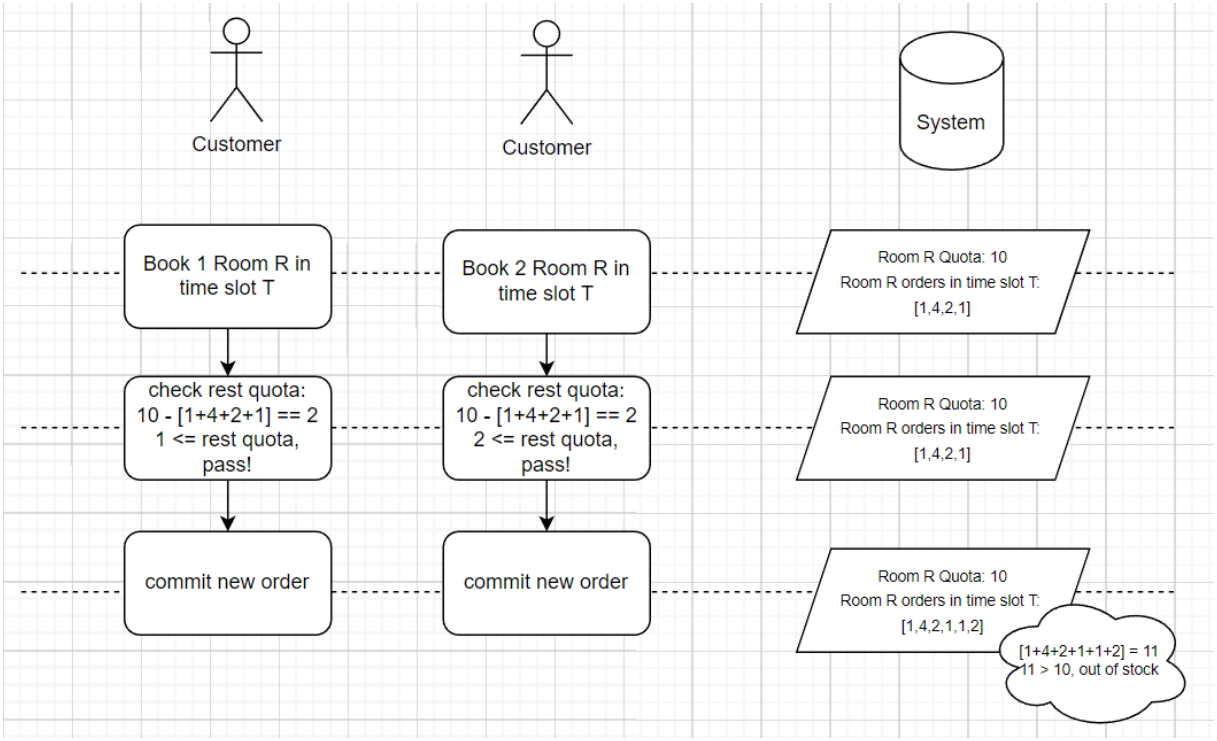


Figure 3.2.4.1 *Concurrency Issue in Hotel Booking*

As is shown in the figure above, multiple users may simultaneously make room R booking in step 1. Customer A wants to purchase 1 Room R in the time slot T, and customer B wants to purchase 2 Room R in the time slot T.

At step 2, the system will check the rest quota of the room in the time slot. The total valid quota for Room R is 10, and there are 4 room orders in the time slot T: [1, 4, 2, 1], so the remaining number of Room Rs is *10 - (1+4+2+1) = 2*. Since each request from customers only needs no more than 2 rooms, they can pass the check and make the order.

At step 3, transactions from 2 customers are done. Out of stock issues occur: In the time slot T, there are room orders for 1+4+2+1+1+2 = 11, which exceeds Room R's total quota 10.

This issue shows that we need to isolate concurrent requests in the procedure of checking rest quota and make room orders/bookings.

Solution:

Pessimistic offline lock can be used here to handle the scenario.

As for reasons:

- Frequency of conflicts is high: There are many customers in the system, and may concurrently book the room. We need to use locks to isolate transactions, make them serialized.
- Losing work/data is not painful for users, customers just click on the book button again.
- Low liveness can be controlled. Pessimistic offline lock brings low liveness to the system, but it can be improved, for example:

Firstly, we don't lock the hotel rooms booking page, as we did in the room editing page. We only lock the object when the transaction is started, so that users can still visit the hotel rooms page at the same time. Room only acquires the lock when the user clicks on the book button;

Secondly, we can make the frontend not pop up conflict alert, but retry the request silently. Or create a queue in the backend to buffer the request, and retry it later until it acquires the lock.

- We have limited time for development, we can reuse the exclusive read lock component created before. The liveness could be improved by changing the lock types in the future.
- Optimized Lock is not suitable: Optimized lock handles data record's updating conflicts by comparing versions. But in this scenario, only new room order records are inserted in the database, with nothing else updated. Room's quota is an unchanged value, because the room domain is similar to a room type/set, e.g. Standard Room has 100 total quota in the hotel for booking.

  This room domain design is made from the consideration that the app is a public hotel booking system, but not a hotel management system for a single hotel. The system doesn't need to track millions of single room unit data from all hotels, so as to reduce records in the room table, and frees hoteliers from inserting bulk of real room unit info one by one in the system.

  One possible way to make the optimized lock suitable for this scenario is that we can make the quota number changeable in the room record, so we can lock on the room. However, the room quota number is also related to time slots, which brings challenges for database design.

The workflow of the pessimistic offline lock is shown in the figure below:

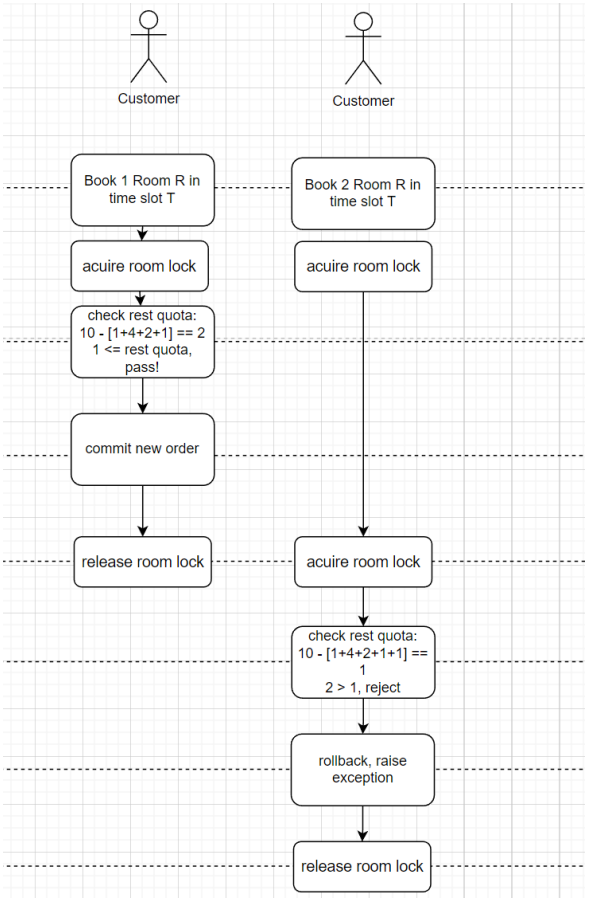


Figure 3.2.4.2 *Workflow of the Pessimistic Offline Lock*

From the diagram above, we can see that room lock isolated 2 booking transactions, so that conflicts are prevented.

As for system implementation, the sequence diagram is shown below:

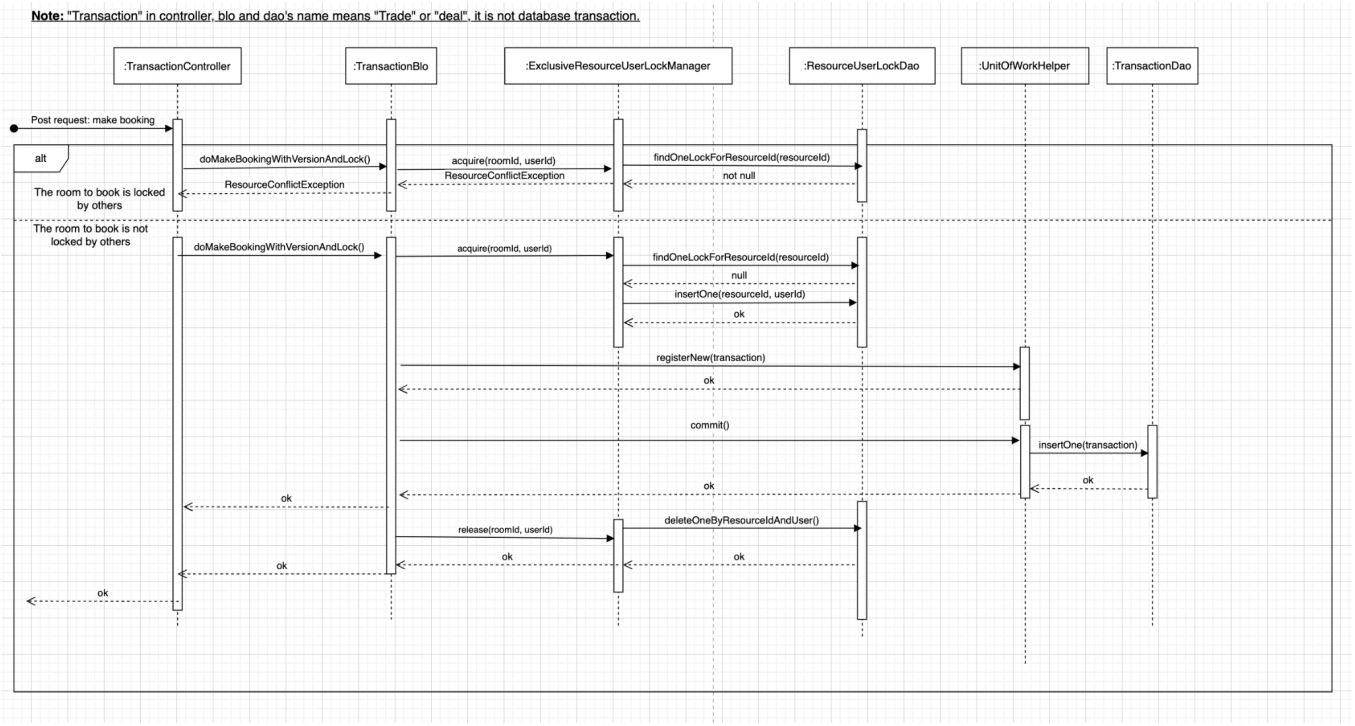


Figure 3.2.4.3 *Sequence Diagram for Backend*

The sequence diagram here shows two alternatives. The first alternative happens when the room the customer is trying to book is already locked by another customer. In this case, after receiving the request from the frontend app, the TransactionController calls doMakeBookingWithVersionAndLock() with relevant parameters to pass the request to the business logic layer (TransactionBlo). In the corresponding method, the TransactionBlo calls acquire() to the ExclusiveResourceUserLockManager, trying to acquire the lock. Since the room is already locked by others, the resourceLockDao will return a row after findOneLockByResourceId(resourceId) called by ExclusiveResourceUserLockManager, which triggers the ResourceConflictException. The exception then sends back to the frontend app, blocking the customer from accessing the locked resource.

The second alternative differs after the acquire() phase, after findOneLockByResourceId(resourceId) called by ExclusiveResourceUserLockManager, there is no lock on that room, so the lock can be acquired by the customer. The TransactionBlo then called the TransactionDao to insert the Transaction with that room if allowed, and commit the changes in the UnitOfWorkHelper. Lastly, the TransactionBlo calls ExclusiveResourceUserLockManager to release the lock.

Please refers to Section 4.4 for the corresponding testing and outcome.

3.2.5 [Outdated Data] Customers Booking Hotels (Continue from Section 3.2.4)

For liveness consideration, we didn't lock the hotel rooms booking page, but only locked the booking button in 3.2.4, outdated data on the hotel rooms booking page can bring troubles. While customers are

viewing the hotel rooms page, the hotelier can finish an isolated room updating transaction and change the room price without exclusive lock. After a customer clicks the booking button, he can also finish the isolated room booking transaction, but the price is different with outdated view data.

For example, when a customer goes to a room page, and sees the price of the room is $100 at that moment. He clicked the booking button, and the system started to perform a booking transaction.

However, before the booking transaction started, a hotelier changed the room price from $100 to $300, and committed the change. During the client's booking transaction, the system will query the database for the price (backend won't receive price from frontend for security consideration), and get $300 for the room. Then, the new room order is inserted in the database.

As a result, the customer is misled by the outdated price in the view data, and the booking price exceeds the customer's expectation.

The procedure is shown in the diagram below:

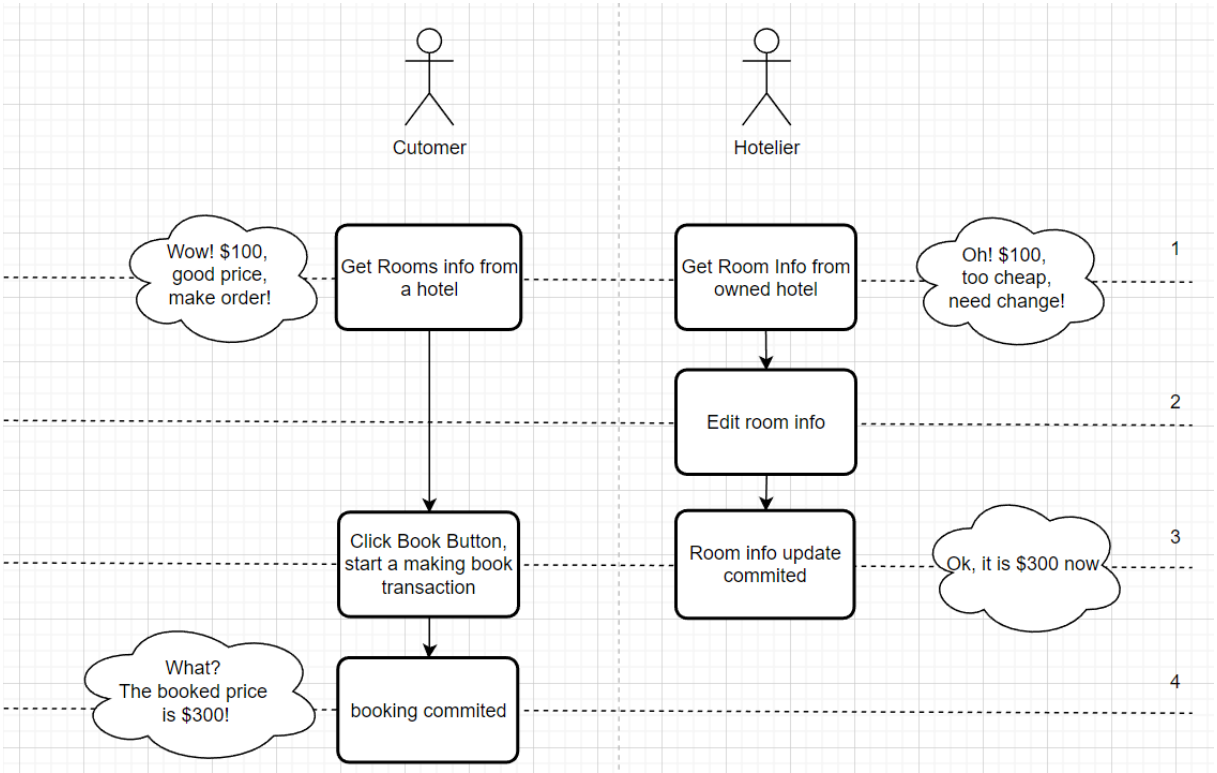


Figure 3.2.5.1 *Concurrency Issue in Customer Booking Hotel*

As we can see from the *Figure 3.2.5.1*, a user experience problem will occur. Because the user believes he made a room order for $100, but the system allocates $300 to the order. This problem is caused by the dirty view data in the frontend, even though these two system transactions are isolated.

Solutions:

This issue can not simply use the pessimistic lock on the room as *3.2.4 [Isolation Issue] Customers Booking Hotels* did. As we mentioned before, since the hotel rooms page is not locked for liveness, booking rooms by customers and updating rooms by hoteliers can be executed as 2 isolated transactions.

We have an option to use pessimistic offline lock on hotel lock to make the hotel rooms page exclusively, but it can further reduce the system's liveness. We already have a pessimistic lock on the room data. So we try to avoid using pessimistic offline locks again.

Finally, we tried to implement optimistic lock's version logic to the room's updating, and create the version field to the room record. As for incoming submission requests, one tiny extension from the original optimistic offline lock is that we perform an extra version check for incoming view data at the start of the transaction to prevent outdated form. In this way, when a customer clicks on the booking button, the request will bring the target room version information to backend, then a version comparison check is performed in the room booking system transaction. As is shown in the diagram below:

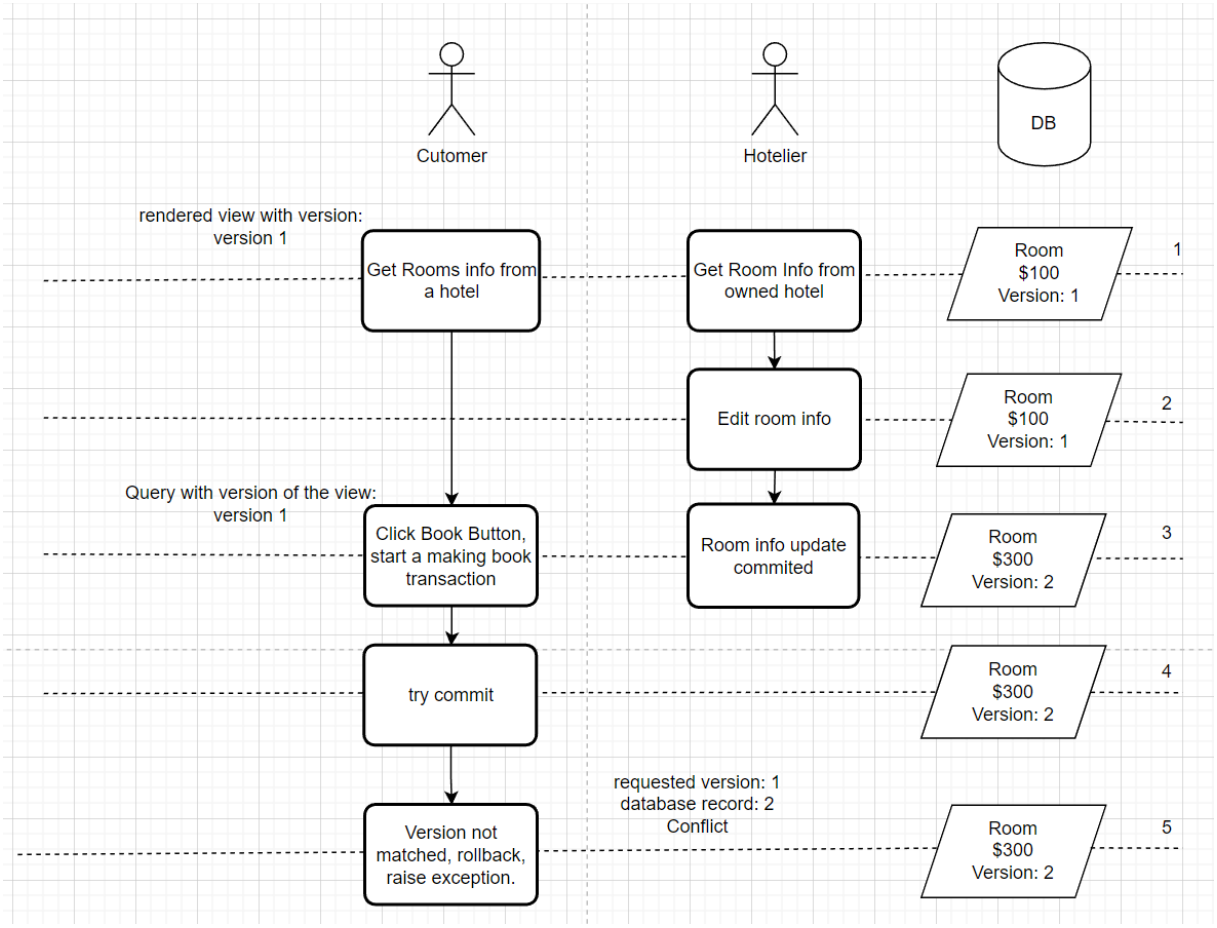


Figure 3.2.5.2 *Sequence Diagram of Customer Booking Hotel*

As is shown in *Figure 3.2.5.2*, if a customer books a room with the outdated view data, the system will reject the booking.

The implementation sequence diagram is shown below:

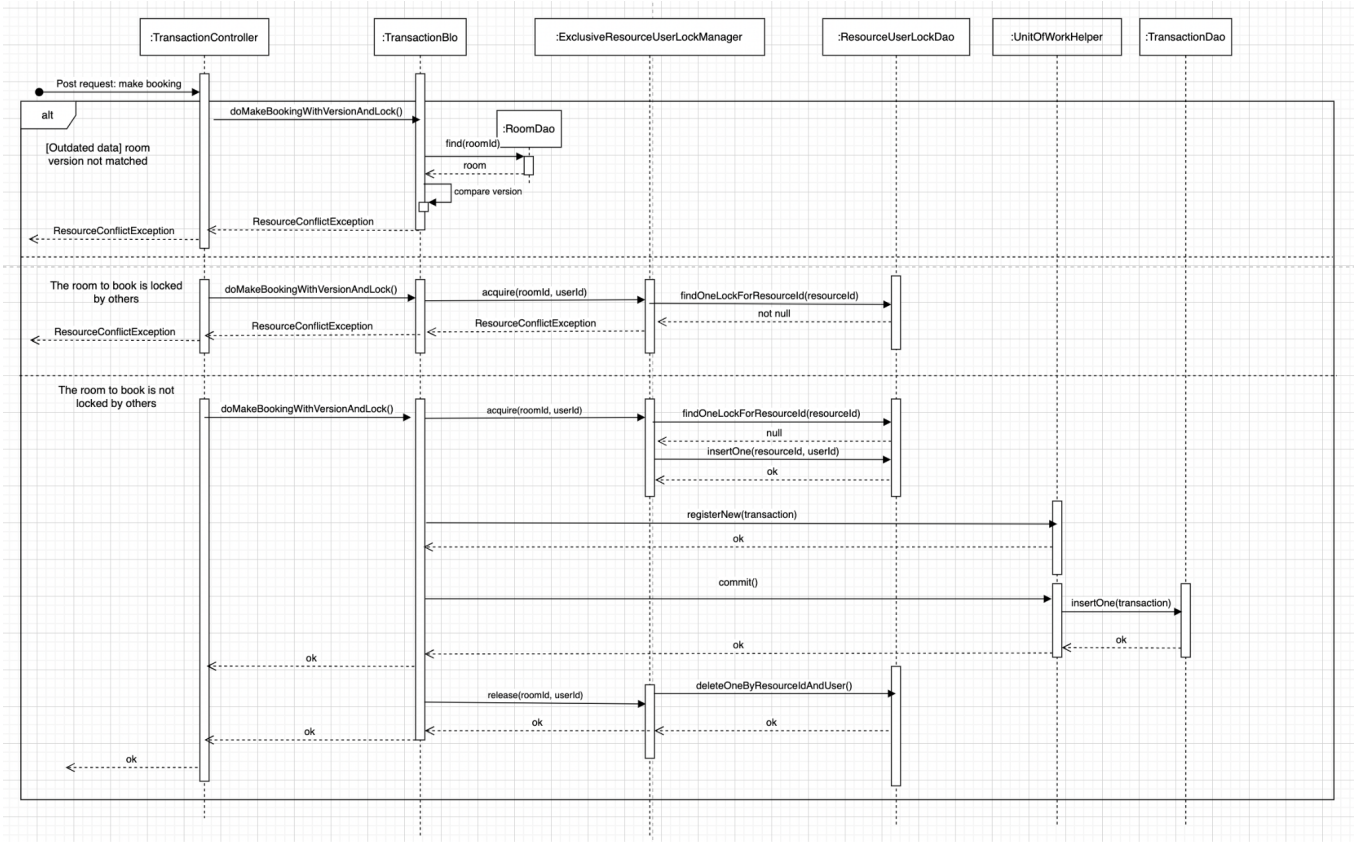


Figure 3.2.5.3 *Sequence Diagram for Backend*

The solution is implemented as the first alternative in the sequence diagram, with the other two alternatives mentioned in Section 3.2.4. Before acquiring the lock, the TransactionBlo performed a version check, using the current version from the database, and the version passed in as a parameter that was obtained by when the customer acquired all rooms in previous requests. In this alternative where the two version numbers do not match, the TransationBlo returns ResourceConflictException to the frontend app. The other alternatives assume two versions are the same, the relevant sequences are omitted for simplicity and readability.

Please refer to Section 4.5 for the corresponding testing and outcome.

3.2.6 [Isolation] Customers update room orders

This issue happens when multiple customers try to update their transaction that orders the same room type. Since the backend app checks if the number of nights of a room type can be changed for the customer order, if the checks are performed at the same time, there could be overbooking, or lost updates.

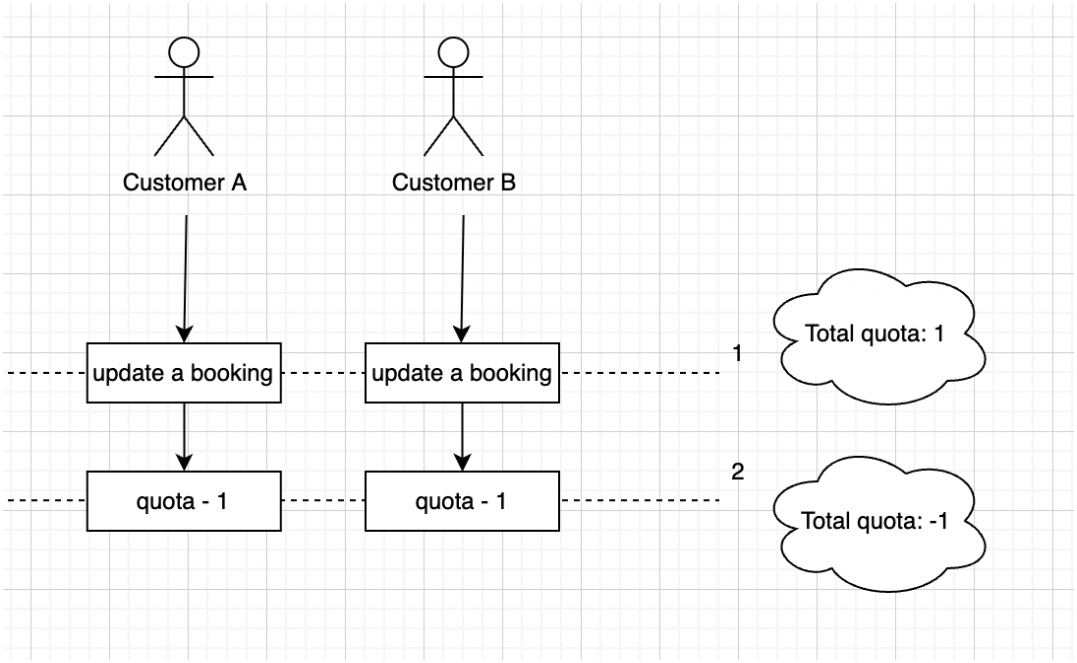


Figure 3.2.6.1 *Concurrency Issue in Customers Update Room Orders*

As shown in *Figure 3.2.6.1*, if two customers are trying to increase their booking quantity by one, but there is only 1 quota left, performing a quota check at the same time could enable them both to increase their booking quantity successfully by the system. However, there is actually overbooking, and one of the customers will find that this is going to affect his/her travel plan.

Solution:

We implemented the JDK synchronized lock to tackle the problem.

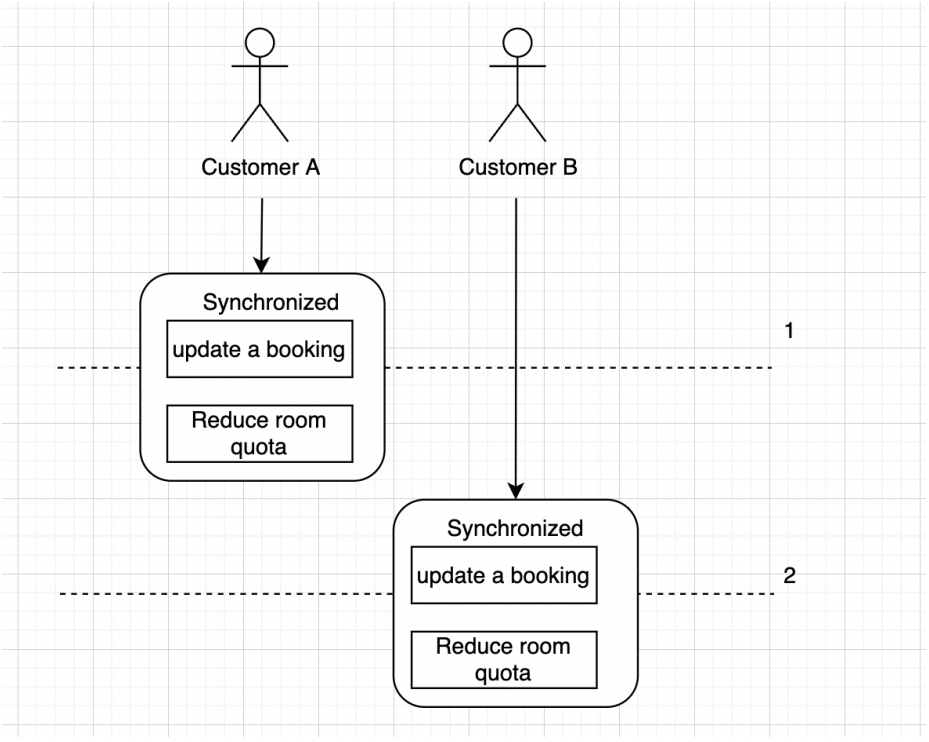


Figure 3.2.6.2 *JDK Synchronized Lock Illustration*

Rationale:

- Frequency of conflicts is low: It is rare that customers are trying to update their transactions that ordered the same room type at the same time.
- Severity of conflicts is low: The JDK synchronized lock could serialize tasks, so customers just need to wait until the system handles the request from the customer.
- Simplicity: The JDK synchronized lock is sample and easy to use. We could use pessimistic offline lock, but due to the limiting timeframe and workloads of team members, pessimistic offline lock can be used in the future.

There are disadvantages with JDK synchronized lock. The lock can only work in a non-distributed system, so if the hotel booking system needs to be distributed in the future, this lock needs to be modified.

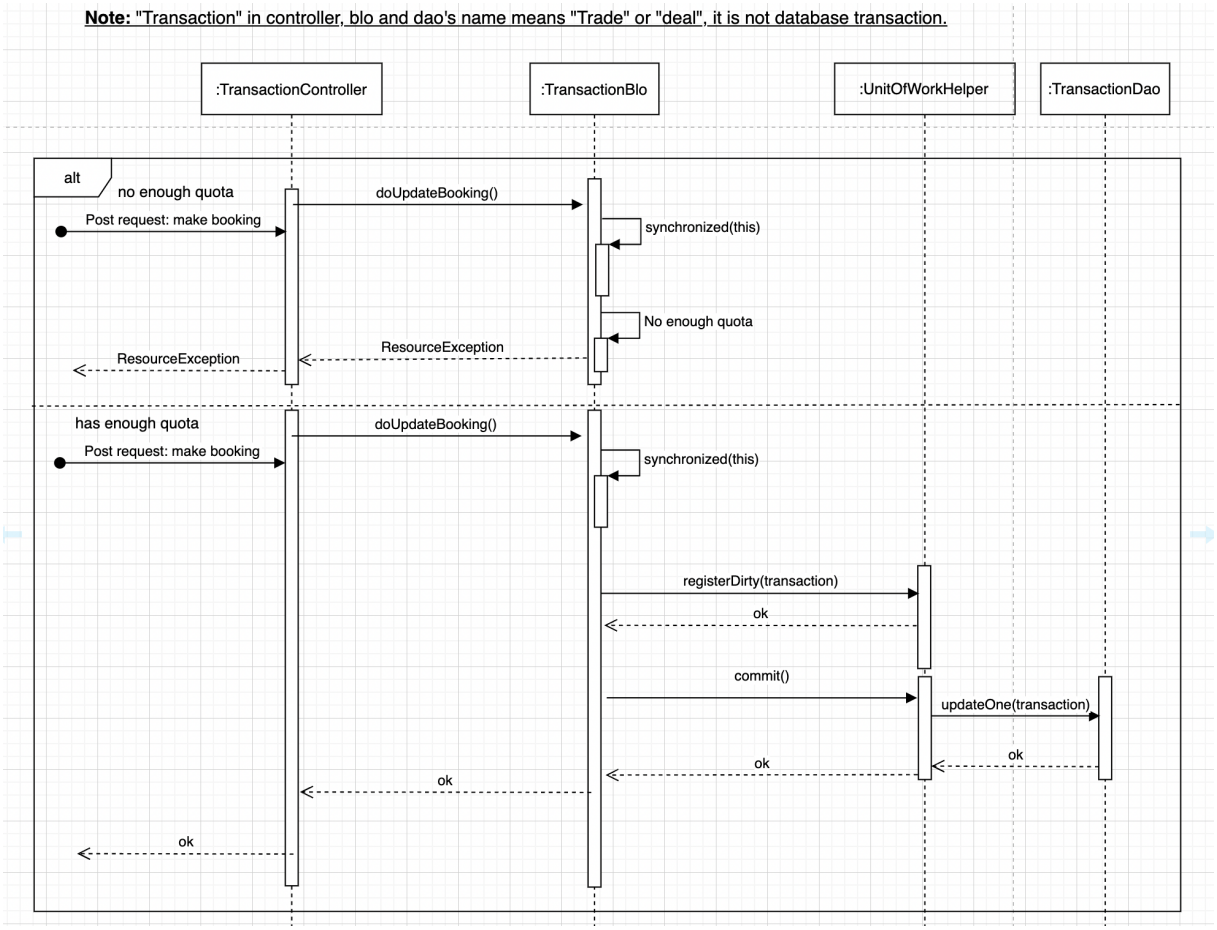


Figure 3.2.6.3 *Sequence Diagram of Backend*

The sequence diagram above shows the synchronized lock, it synchronizes the method to solve the issue, so that multiple tasks are in a queue.

Please refer to Section 4.6 for the corresponding testing and outcome.

4. Tests and Outcome

In this section, we have carried out testings for the scenarios mentioned above.

4.1 [lose updates] Hoteliers Update shared Hotel information

4.1.1 Execution Type

JMeter API test.

Reason: We can perform the concurrency test for this API easily by utilizing the JMeter tool. The reason we use the JMeter is that to simulate concurrent updates is troublesome, for example, setting breakpoints in the source code, running in the debug mode and editing the database record to simulate the concurrent updation. Also, simulating boundary conditions is difficult, such as 10+ users performing the concurrent operation.

As a result, we decided to use the JMeter tool to test the optimistic Lock. JMeter is a common tool to test functional behavior and measure performance, and we can get help from the open source community easily. JMeter can easily initiate concurrent requests.

4.1.2 Testing Objective

If all Hoteliers start the hotel updating transaction at the same time, only one hotelier's transaction will succeed, the rest of transactions are rejected.

4.1.3 Set Up

Firstly, we should simplify authentication logic of the hotelier edit owned hotel API. So that we don't need to use the login API to generate tokens for parsing, which is troublesome and irrelatent to the concurrency test. Also, we just put the user id into the request header in the test.

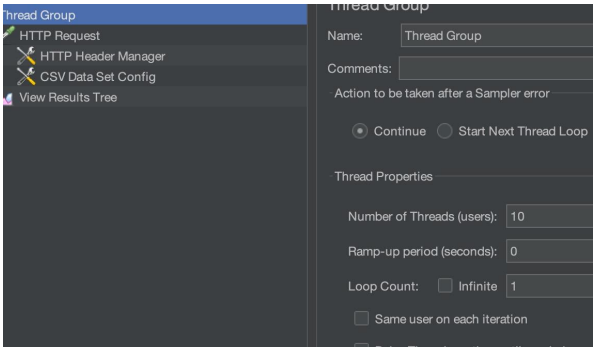
```
/**
 * edit an existing hotel
 * the hotel is the one that this hotelier is managing
 */
996Worker *
@HandlesRequest(path = "/", method = RequestMethod.PUT)
//    @AppliesFilter(filterNames = {SecurityConstant.HOTELIER_ROLE_NAME})
public R editOwnedHotel(HttpServletRequest request, @RequestJsonBody @Valid HotelParam param)
//        String token = request.getHeader(SecurityConstant.JWT_HEADER_NAME);
//        AuthToken authToken = TokenHelper.parseAuthTokenString(token);
//        String userId = authToken.getUserId();
    String userId = request.getHeader( name: "UserID");
    hotelBlo.editOwnedHotel(userId, param);

    return R.ok();
}
```

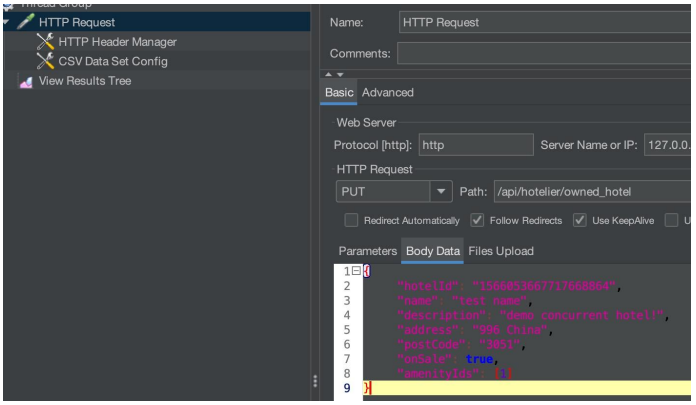
And, then prepare 10 hotelier accounts in the db, they all manage the same hotel.

4.1.4 Steps

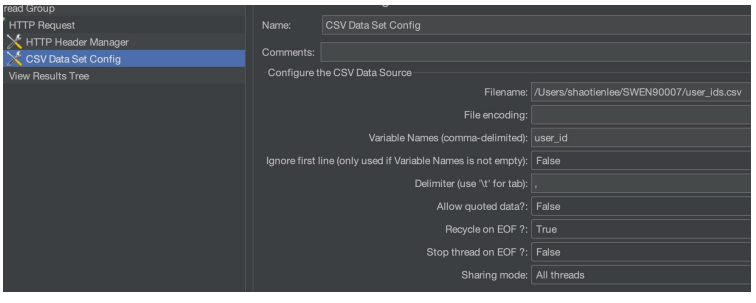
- Run JMeter tool. configure the thread group. We start with 10 threads and 0 Ramp-up period:

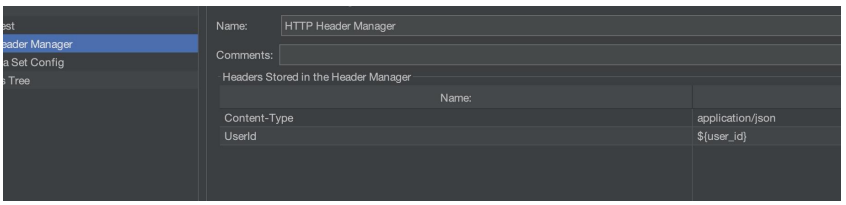


- Then, we configure http request as follows:

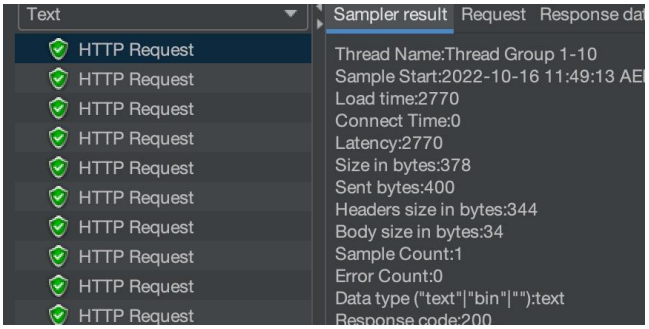


- Then, we use a .csv file to hold all user ids for header manager to loop, and configure http header as follows. Also we need to configure Content-Type as application/json. As is shown in the figure:





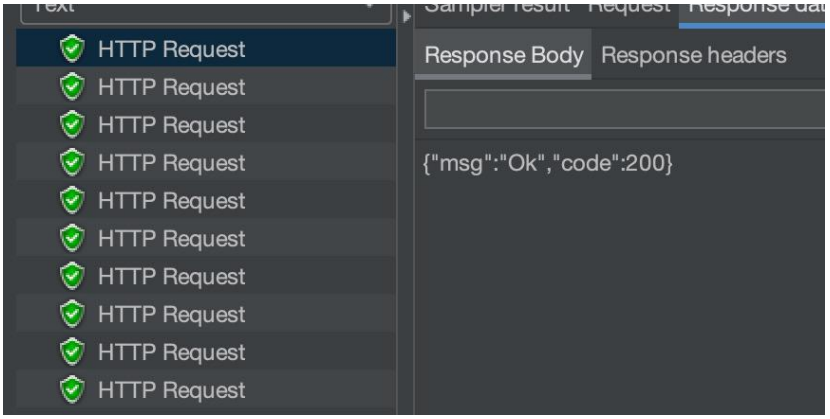
- Start the backend end application, then run the 10-thread-task group in the JMeter, and analyze the test result:



We confirmed that all 10 requests started from “2022-10-16 11:49:13 AEDT”, which means they are concurrent requests. Transactions will start simultaneously.

We can see all responses seem to be “successful”, however, we put the error information and error code in the Response Body, not in the http response’s default status code.

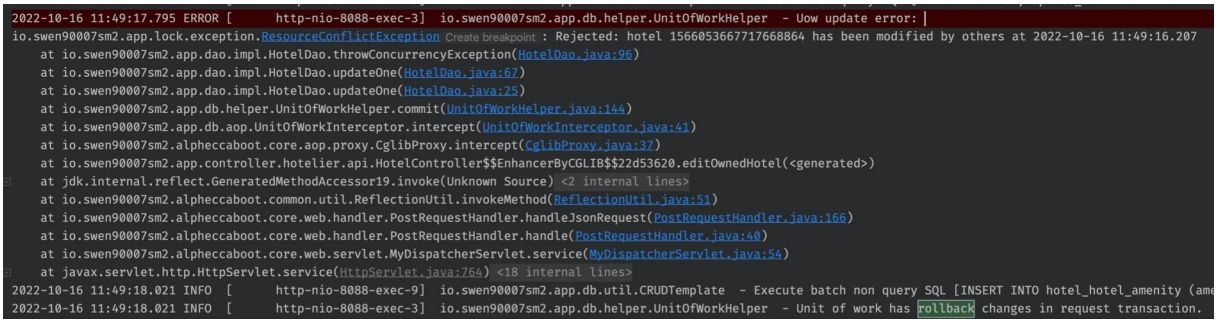
From response body, we can see only 1 of all 10 requests have the response below, it means that the data record is updated:



The rest of 9 requests were all rejected by backend, as is shown in the figure:

If the request is rejected, an exception with message “Rejected: hotel 1566053667717668864 has been modified by others at 2022-10-16 11:49:16.207","code":20001” is raised. The frontend can render the message to inform the user.

- Check backend log to track system behavior: We need to track the system behavior, so we need to check the runtime log. We implemented a log system to track system behavior by slf4j. As for the concurrency exception raised by optimistic lock, the system not only rejects one single SQL updating, but rolls back the whole transaction in the Unit of Work for the request thread. As is shown in the backend log:



As is shown in the backend log, we can see the request thread *8088-exec-3*’s updating is rejected, and the whole transaction of the request thread is rolled back by the unit of work helper.

- check the database: We need to check the data consistency of the system, so we need to check the database. If we go to the database with a visualization tool, we can see the version number of the hotel record has increased by 1.

- Further tests: In order to confirm the robustness, we did extra test rounds and found that requests with the same sample start time can always be handled properly.

4.1.5 Results

As for the test outcome, we can see that Optimistic Offline Lock uses version to prevent concurrency issues. During the commit time, if the version conflict is detected, the change made by this transaction will not perform, and the whole transaction is rolled back. If there is no conflict, the transaction is committed, and the version number of the target data record will be increased by 1.

In the test, we tested the backend API for the hotelier group to edit shared hotel info. We performed 10 concurrent threads based on the JMeter tool. Only one of concurrent requests successed to commit the updating transaction, while others were rejected by version conflict.

This test result shows that the optimistic offline lock is properly implemented with the system, and the concurrency issues are resolved.

4.2 [Lose Updates] Hotelier Group Edit the Room Information

4.2.1 Execution Type

Manual

Reason: For this test we chose to use manual testing because we think that we have a more intuitive sense of how a piece of software feels to the user. In addition, manual testers can sometimes find coding errors that automated tests cannot detect.

4.2.2 Testing Objective

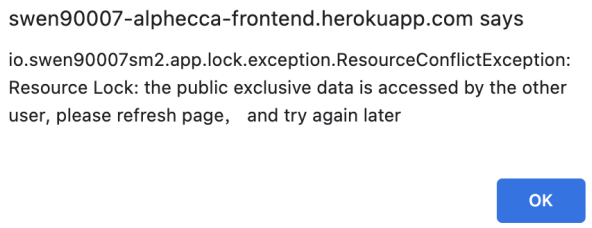
Multiple hoteliers are able to update the room details at the same time.

4.2.3 Setup

- Make sure both of the deployed front-end and back-end are up-to-date in Heroku to ensure the locks have been implemented.
- Open the website in two separate browsers to make sure each of the hotelier is separated
- Log into the hotelier role correspondingly, name Hotelier A and Hotelier B.
- Log into the admin role to make sure that Hotelier A and Hotelier B are in the same hotelier group.

4.2.4 Steps

- As Hotelier A and Hotelier B are in the same hotel group, they have access to the same hotel and the corresponding hotel rooms. Both Hotelier A and Hotelier B are trying to edit the information of the same room.
- Now Hotelier A is putting new information for a specific room, and at the same time, Hotelier B also wants to make changes to that specific room. However, Hotelier B is unable to edit the shared hotel information while Hotelier A is doing it.
- An error message will be arised in Hotelier B site as shown below:



- When Hotelier A finishes editing, Hotelier B is able to see the new updates if he refreshes his page, and he is able to make a new edit if he wants to.
- If Hotelier A is editing Room 1, although Hotelier B is not able to edit the information of Room 1, he is still able to edit the information of Room 2 as these two rooms are not shared data.

4.2.5 Results

Pessimistic offline lock has been implemented successfully and it avoids lost updates while two hoteliers are trying to edit the information of the same room at the same time.

4.3 [deadlock] Hotelier Group Edit the Room Information

4.3.1 Execution Type

Type: Manual

Reason: Since this feature can only be used with a logged in account, we applied manual testing on the frontend to avoid any changes to backend code on authentication checking. Moreover, our design

of the frontend will have a pop-up window to alert users if there are any errors returned from the backend.

4.3.2 Testing Objective

This testing is setup to examine 2 scenarios:

1. A hotelier's network crushes down during the editing room process, but he/she can still edit the room type he/she opened before, when he/she re-enter the edit page with his/her account.
2. A hotelier hasn't finished editing a room type and can keep editing without any error alert.
3. A hotelier never goes back to the edit room page/submit changed details after they have requested to edit one room, other hoteliers can not access the editing function until the lock is released after 2 mins.

4.3.3 Testing Setup

- Make sure both of the deployed front-end and back-end are up-to-date in Heroku to ensure the locks have been implemented.
- Use Admin account to assign multiple hoteliers to a same hotelier group
- Open Alphecca heroku app website in two separate browsers to make sure different hoteliers are separated.
- Log into two hoteliers accounts , named Hotelier A and Hotelier B. (Here can use admin account to make sure those hoteliers manage same hotel)

4.3.4 Steps and Outcome

Scenario 1

- **Steps:**
  - Hotelier A starts editing one room type by clicking the edit button.
  - Hotelier A closes the website or edit window accidentally without clicking the submit button on the edit window.
  - Hotelier A opens up the hotelier portal again and tries to edit the same room type.
- **Outcome:**

  Hotelier A can still open up and start editing the room type he/she left before because we applied a pair of resource id and owner id for pessimistic lock to match the lock ownership.

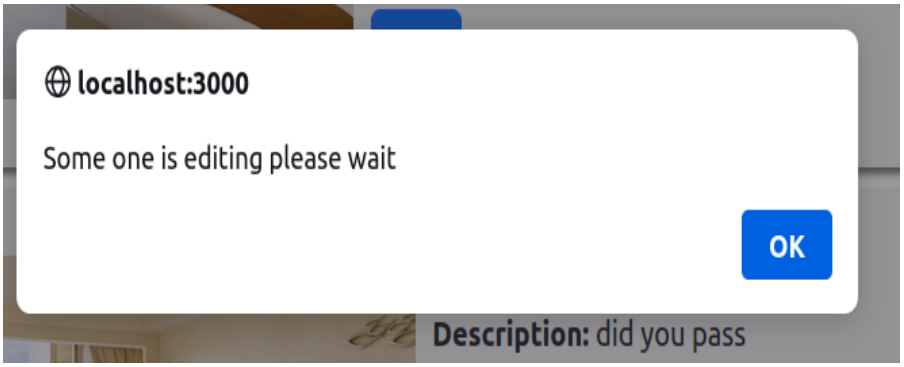
Scenario 2

- **Steps:**
  - Hotelier A starts to edit a room type.
  - Hotelier A keeps editing for more than 2 minutes.(just keep the edit window open without clicking submit button for more than 2 minutes)
- **Outcome**:

  The frontend keeps sending a request for room detail with an API that can lock this room every minute to extend the time usage of this resource before the lock expires and someone interrupts. The result is that no error message was logged on the frontend console.

Scenario 3

- **Steps**
  - Hotelier A starts to edit a room type.
  - Hotelier A left the edit page and never go back to the page and click the submit button to release the lock
  - Hotelier B clicking the edit button on that room type and receiving an alert message.



Alert message when Hotelier B tries to access a locked room resource

- Hotelier B attempts to edit the room after 2 min.

- Outcome:

  Hotelier B cannot use the edit function when the lock has not expired, but after 2 minutes, Hotelier B can successfully open the edit window.

4.3.5 Results

Pessimistic lock was successfully implemented with a paired resource id and user id on editing room function. One hotelier can always return to the editing page before the lock expiration time, and extend the editing time with continuous requests on this room resource from the frontend. Moreover, the lock on one room data can automatically be released after 2 minutes without any issues.

4.4 [Isolation Issue] Customers Booking Hotels

4.4.1 Execution Type

JMeter API test

Reason: It is troublesome to perform concurrent requests manually, so we use Jmeter to request booking API concurrently.

4.4.2 Testing Objective

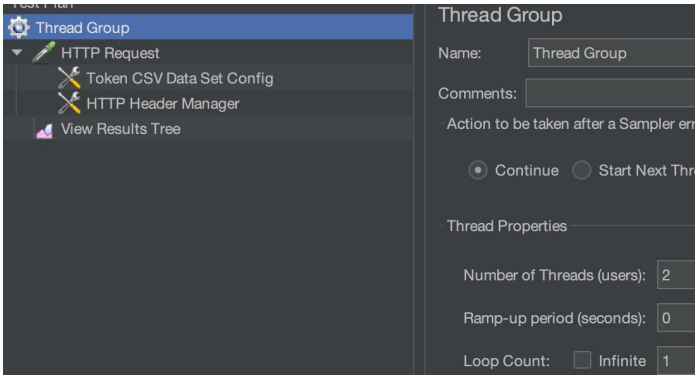
Multiple customers request to book the same room at the same time. Since we used Pessimistic lock to isolate transactions, only one customer's booking request can complete.

4.4.3 Set Up

- Prepare 2 customer accounts;
- Call login API, get 3 customer tokens, put in user_tokens.csv;

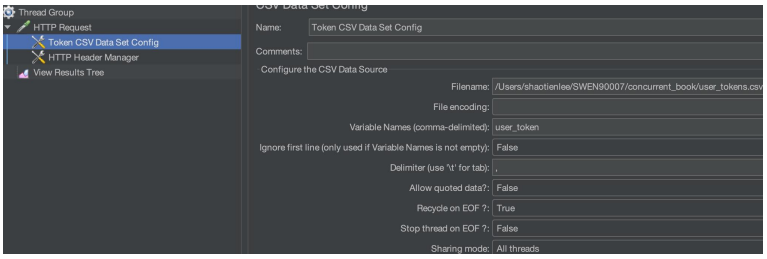
4.4.4 Steps

- Setup Jmeter as is shown below:

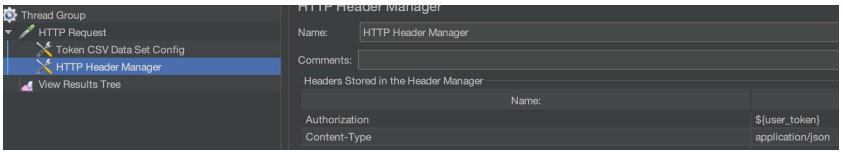


- Configure test metadata:

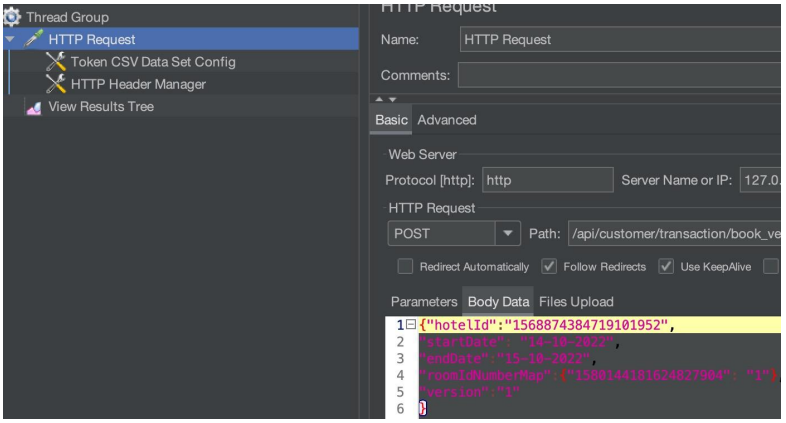
  Configure the auth token csv input:



  Configure request headers:



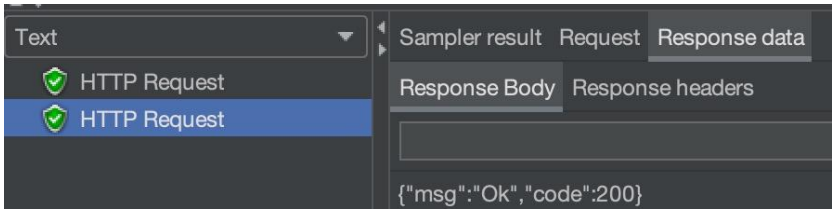
  Configure request body:



- Run the Jmeter test;
- confirm the result:

All 2 requests are initialized at the same time, they are concurrent.

Only one request succeeded to make the order:



While the other request is rejected by pessimistic lock with exception “Resource Lock: the public exclusive data is accessed by the other user, please refresh page， and try again later”:

- Repeat the test:

We repeat the test several times, and all results are successful. We confirmed that the lock implementation is robust and correctly applied.

4.4.5 Results

Pessimistic Offline Lock is correctly implemented to isolate customer booking transactions. At the same time, only one customer’s booking transaction will be successful. In this way, the over-quote issue will not occur.

4.5 [Outdated Data] Customers Booking Hotels

4.5.1 Execution Type

Manual

Reason: This feature can be tested with a logged in hotelier account and a logged in customer account, then we can use the manual test to avoid complex steps in Jmeter.

4.5.2 Testing Objective

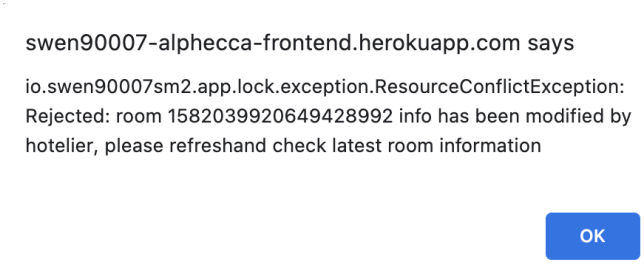
While a customer is browsing a hotel and about to make a reservation of a specific room type, if the hotelier changes the price of that room type, the customer should not be able to make the reservation.

4.5.3 Set Up

- Make sure both of the deployed front-end and back-end are up-to-date in Heroku to ensure the locks have been implemented.
- Open the website in two separate browsers to make sure the customer and the hotelier are separated.
- Customer login and search for Hotelier’s hotel.
- Hotelier login and check for the room type. In this testing, there is only one room named Room 1.
- The customer books for Room 1 and puts in the needed information details but doesn’t proceed with the transaction.
- Hotelier changes the price of Room 1 right before Customer makes the transaction.

4.5.4 Notes

- The customer presses the Reserve Now button, an error message appears as shown below:



- When Customer refreshes the webpage, he is now able to see the updated price and decide if he still wants to make the booking.
- Customer is able to make the reservation for the updated room type.

4.5.5 Results

Pessimistic offline lock has been implemented to avoid outdated data. The customer is not able to make the reservation if the price of his preferred room has changed. An error message will be arised in the customer's webpage notifying him the price has been changed. After the customer refreshes his webpage, he is able to see the new updated price.

4.6 [Isolation] Customers update room orders

4.6.1 Execution Type

Jmeter

Reason: Same as the first test case. Using Jmeter is convenient for testing concurrent requests

4.6.2 Testing Objective

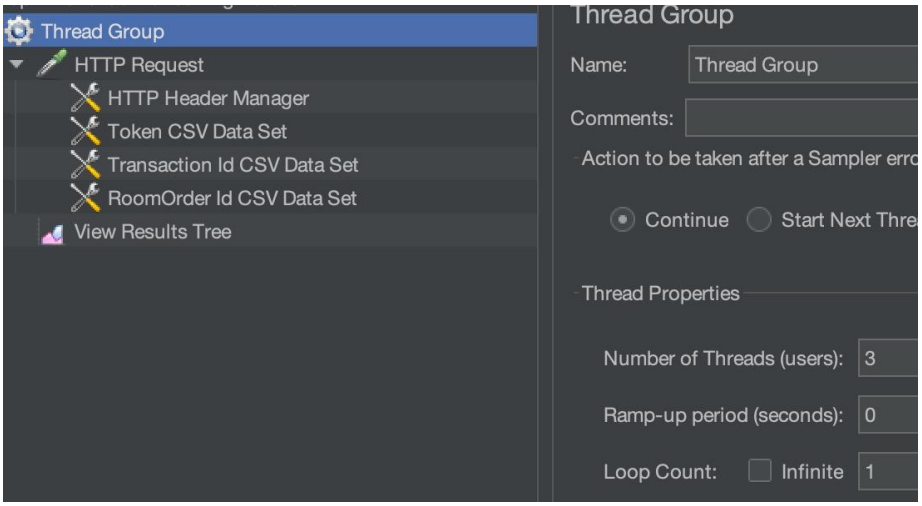
Multiple customers change one room order at the same time, but no over quota issue will occur. The troublesome updating transaction should be rejected by system

4.6.3 Setup

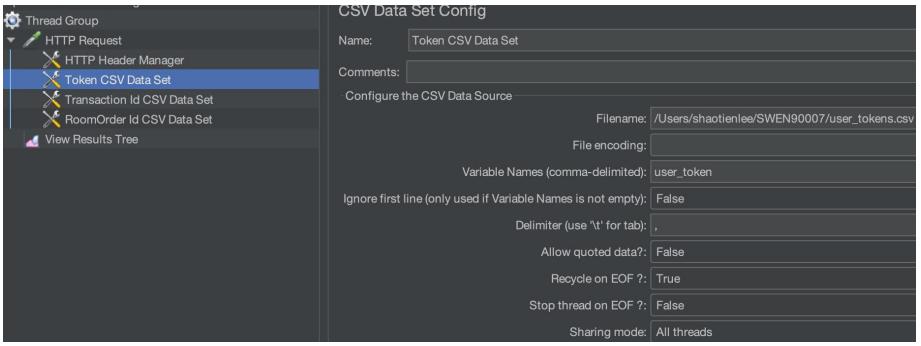
- Prepare 3 customer accounts, and book the same room for 1. The room's vacant num(total quota) is 10;
- Call login API, get 3 customer tokens, put in user_tokens.csv;
- Go to the database, get 3 customer's transaction ids, and room_order ids. Put these ids in transaction_ids.csv and roomorder_ids.csv.

4.6.4 Steps

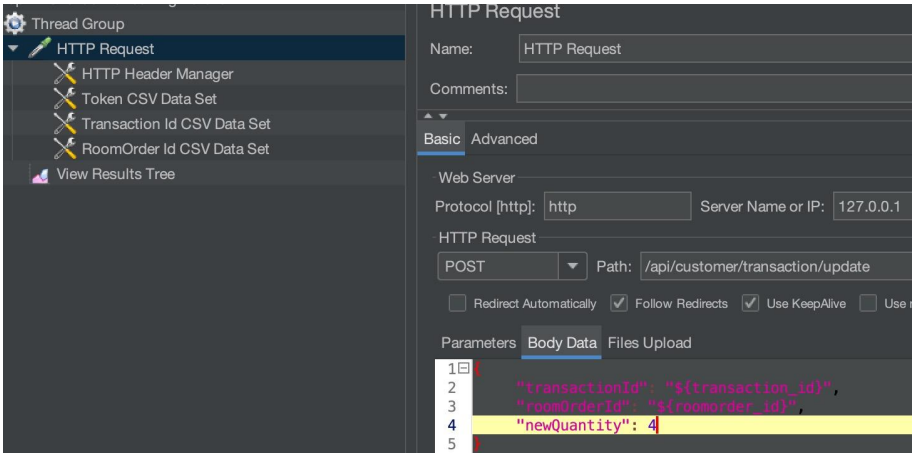
- Setup Jmeter, as is shown in the screenshot below:



Configure metadata:

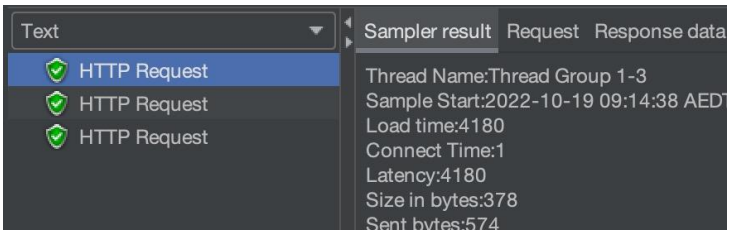


In Request JSON body, we make each request change the room order number to 4, so that quota needed will be 3 * 4 == 12 > 10. One of the updating requests should be blocked.

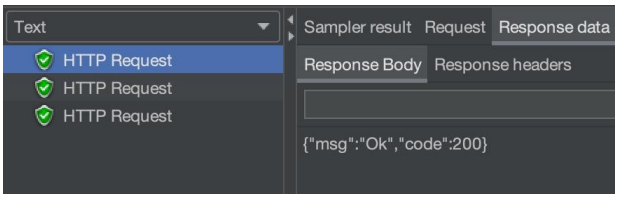


- Run the Jmeter test;
- Confirm the test result:

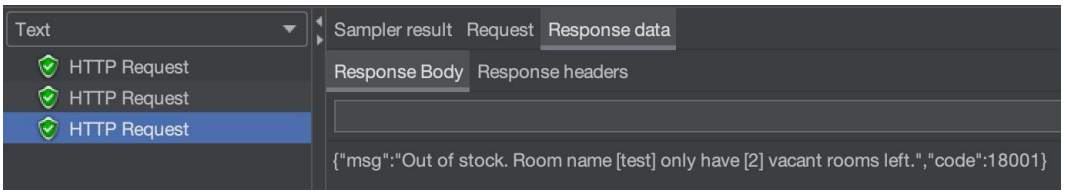
All 3 book updating requests are started at the same time “2022-10-19 09:14:38 AEDT”. They are concurrent requests:



2 of 3 requests successfully updated the room booking:



1 of 3 request is blocked, because of over quote:



The total number of rooms is 10, the first 2 requests update the room order, and take 8 quota, there are only 2 rooms left, which is not enough for the third request to update the order for 4 rooms.

This test result shows that, although 3 updating room booking transactions are started at the same time, the system isolated them, so that no over quota issue will occur.

4.6.5 Results

JDK Synchronized lock has been implemented successfully and it isolated concurrent updating room booking transactions to prevent over quota issues. Synchronized lock is easier to implement for one-node backend systems than the pessimistic offline lock is.