Part 3 Concurrency and Testing

SWEN90007 SM2 2021 Project

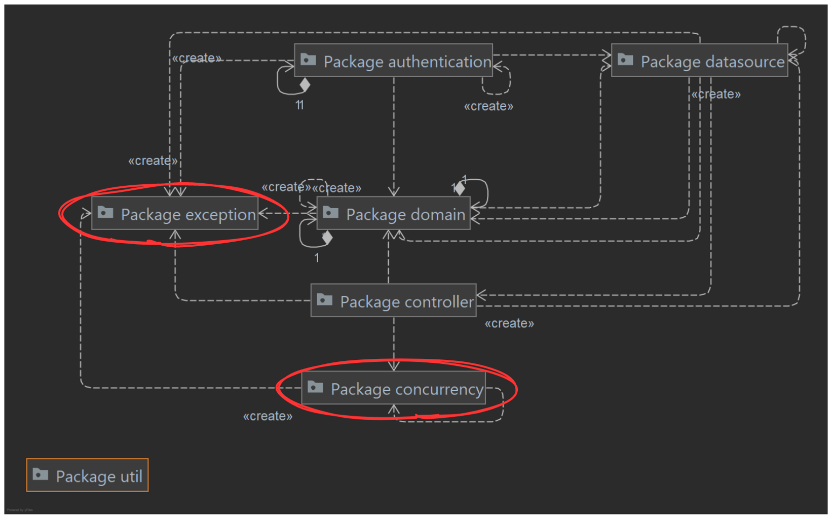
Team: Four Aces

My Tien Hinh - 923427 - mhinh@student.unimelb.edu.au
Xueqi Guan - 1098403 - xueqig@student.unimelb.edu.au
Yiyuan Wei - 793213 - yiyuanw1@student.unimelb.edu.au
Yuxiang Wu - 1006014 - yuxiang2@student.unimelb.edu.au

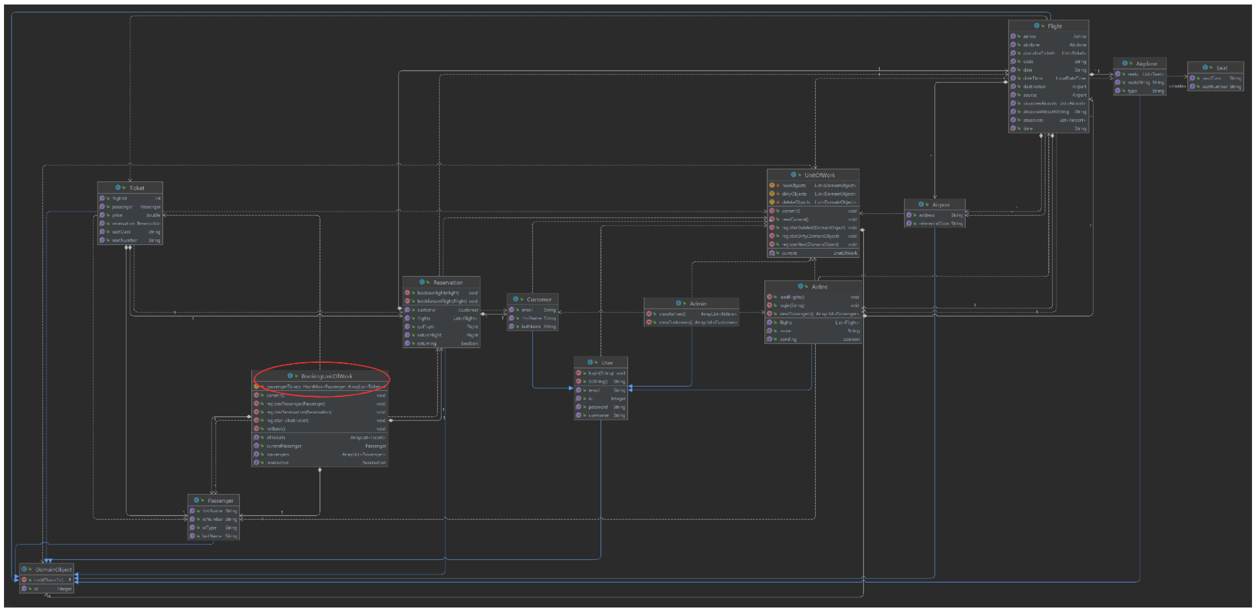
Class Diagram

NOTE: The class diagram for each package is attached in the code under the same package name, if the following diagram is not clear enough, please check through Github. The class diagram in the git repository does not highlight updates.

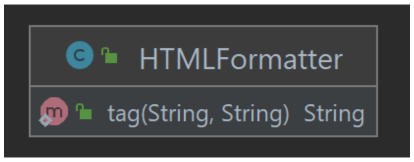
Package Overview



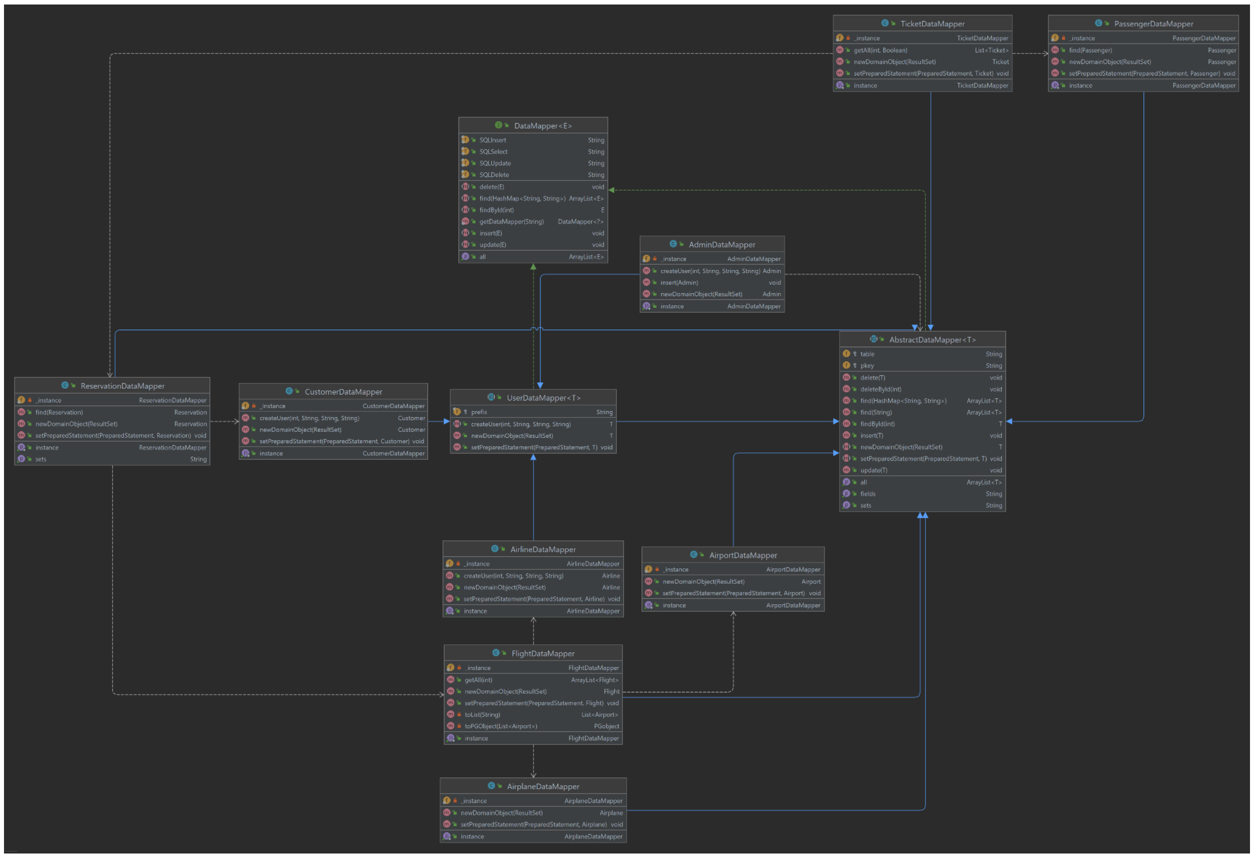
Package domain



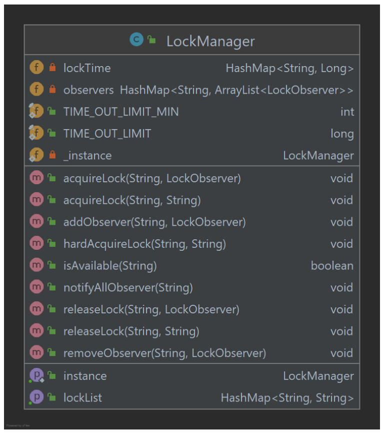
Package Util



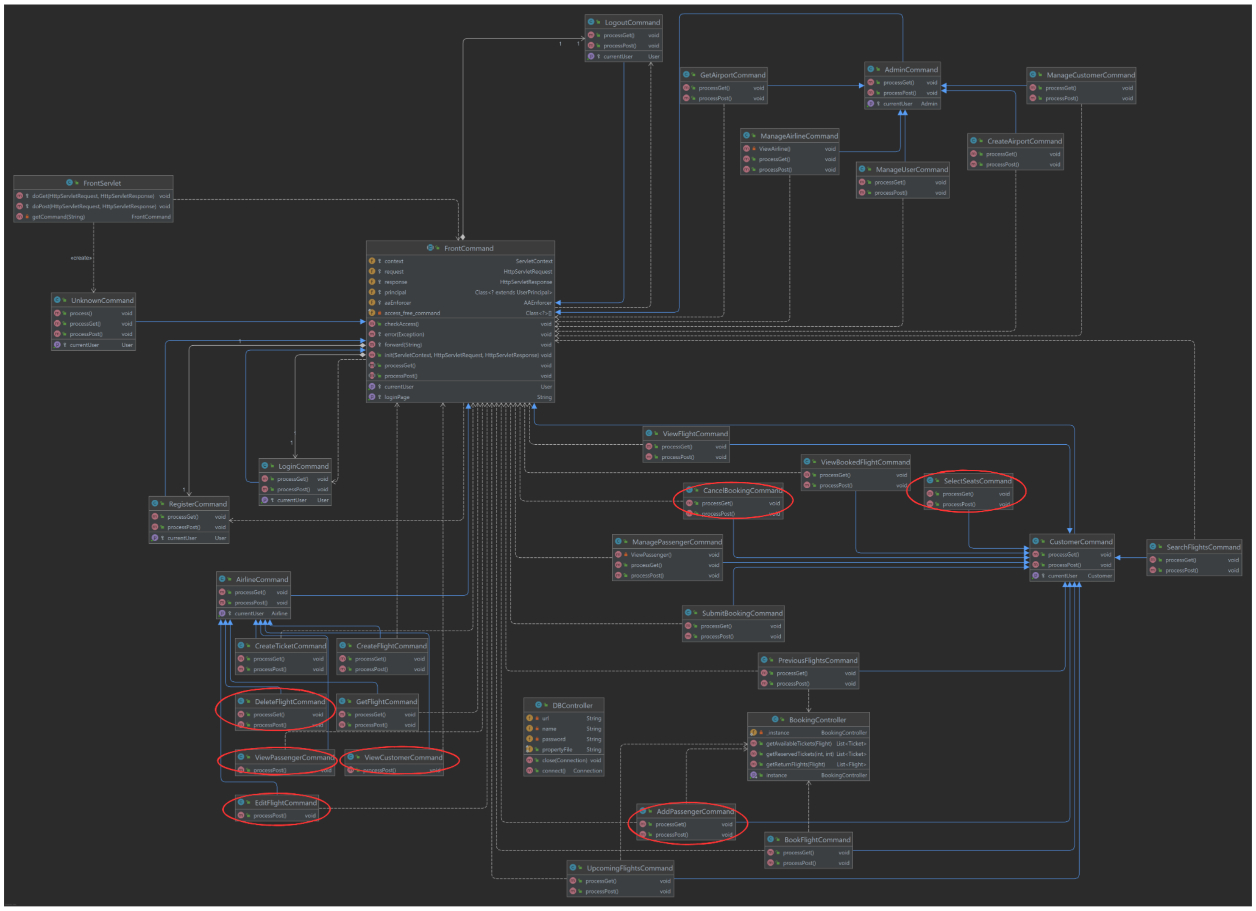
Package datasource



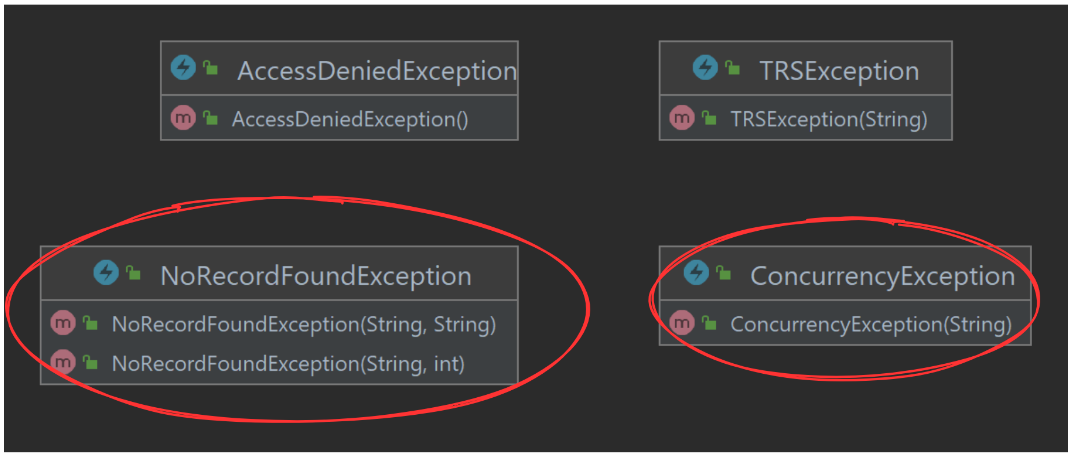
Package Concurrency (New package)


LockManager
lockTime HashMap<String, Long>
observers HashMap<String, ArrayList<LockObserver>>
TIME_OUT_LIMIT_MIN int
TIME_OUT_LIMIT long
_instance LockManager
acquireLock(String, LockObserver) void
acquireLock(String, String) void
addObserver(String, LockObserver) void
hardAcquireLock(String, String) void
isAvailable(String) boolean
notifyAllObserver(String) void
releaseLock(String, LockObserver) void
releaseLock(String, String) void
removeObserver(String, LockObserver) void
instance LockManager
lockList HashMap<String, String>

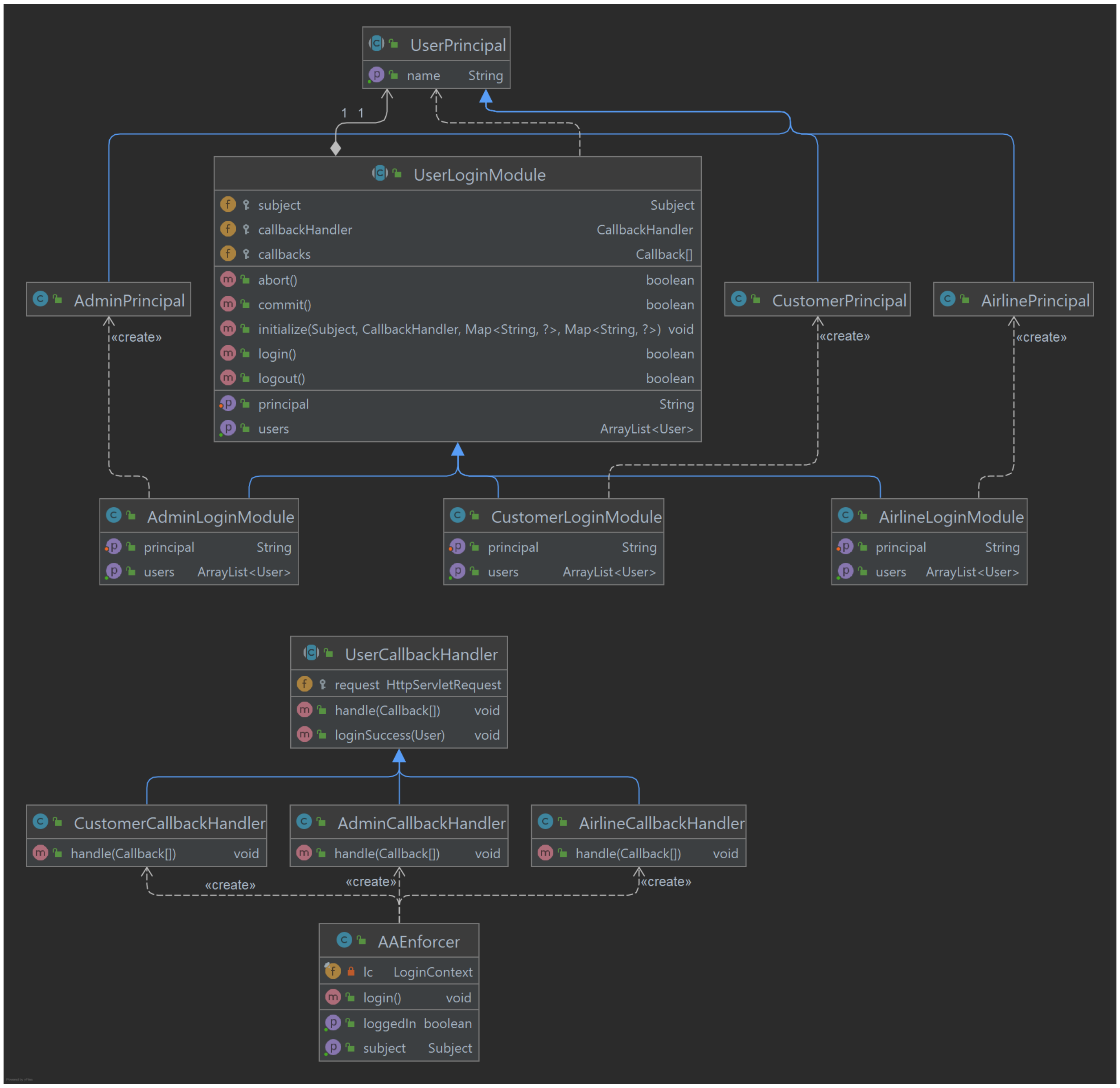
Package controller



Package Exception



Package authentication



Concurrency

Concurrency Patterns

Concurrency

When it comes to our TRS system's concurrency issues, our system needs to support concurrent access to the data in the data source layer, which means that multiple processes within the system will be trying to access the same data at the same time. If all of the processes are trying to read the data only, then this is not such an issue, but if some are trying to write, then problems can occur. [1]

Offline Concurrency

When it comes to the offline concurrency, system transaction and business transaction need to be discussed.

- System transaction – transactions supported by transactional resources. E.g., a database transaction (a group of SQL commands delimited by instructions to begin and end it).
- Business transaction – those at the enterprise level that may bring together several system transactions.

System transactions can be categorised based on their relationship to the business transactions that they implement.

- Request transaction – a transaction that spans exactly one request to the system.
- Long transaction – a transaction that spans two or more requests to the system.
- Late transaction – a transaction that does as much reading as possible, calculates what changes are required, and then does an update as late as possible.

Business transactions usually need to take multiple requests to complete, which means we need to take a single system transaction to implement one result in a long transaction. Using a long transaction means that we can avoid a lot of annoying problems. However, most transaction systems don't work efficiently with long transactions. The application won't be extensible becauselong transactions will turn the database into a major bottleneck. In addition, the refactoring from long to short transactions is both complex and not well understood.[2]

Because of this reason, many enterprise applications don't want to take the risk of using long transactions. In this case, we need to break the business transaction down into a series of short transactions. This means that we control concurrency for business transactions that span over multiple system transactions[1], which raises a problem called offline concurrency. Whenever the business transaction interacts with a transactional resource, such as a database, that interaction will execute within a system transaction in order to maintain the integrity of that resource. [2]

In the next part, the offline concurrency will be highlighted and discussed on the server side of our TRS system, which means we need to handle multiple requests from many users including customers, airlines that want to access the same data on the data layer.

There are three different offline concurrency patterns including Optimistic Offline Lock, Pessimistic Offline Lock and Implicit Offline Lock. The following table shows the basic differences between them.

Pattern	Description
Optimistic Offline Lock	Allows multiple transactions to access shared data simultaneously.
Pessimistic Offline Lock	Allows only one business transaction at a time to operate on the data.
Implicit Lock	Allows a framework layer supertype code to acquire offline locks.

Table 1

Optimistic Offline Lock
The Optimistic Offline Lock is a way to ensure data integrity and avoid the option of different clients submitting conflicting changes.
As the name suggests, it assumes that the probability of conflict is low. In fact, in this case, optimistic locking will not slightly slow down user interaction.
The goal of this pattern is to detect conflicting changes, not to apply it, but to roll back the business transaction and present the error to the user. It achieves this goal by verifying that no one else has tampered with the records in the data source before allowing changes to be submitted. [2]

Pessimistic Offline Lock
Pessimistic Offline Lock prevents conflicts by avoiding them altogether. It forces a business transaction to acquire a lock on a piece of data before it starts to use it, so that, most of the time, once you begin a business transaction you can be sure you'll complete it without being bounced by concurrency control.[2]

Implicit Lock
Implicit locks are automatically acquired by the server to ensure data integrity among multiple users. To allow greatest concurrency, the server acquires locks only when needed and frees them as soon as it can. The lock duration is the length of a server unit of work, which is, for example, from the time a file is opened until it is closed and any changes are committed. Maintaining the implicit lock eliminates the need to obtain an explicit lock each time a file is updated. [2]

However we only used the Pessimistic Offline Lock pattern, the following table shows it’s difference to show why we use Pessimistic Offline Lock.

Aspect	Optimistic Offline Lock	Pessimistic Offline Lock
Conflict	Conflict are detected at a commit time	Conflicts are avoided
Implementation	● Associate a version number with each record in the database. ● Compare the version of the record when it was read with the current version in the database.	● Lock the ID, or primary key. Obtain the lock before the object is loaded from the database ● Release the lock when the business transaction finishes.
Pros	● Good liveness ● Simplicity	No work/data loss.
Cons	Lost work/data	● Low liveness ● Harder to program
When to use	● The possibility of conflict between concurrent transactions is low. ● Losing work/data is not a big deal.	● The possibility of conflict between concurrent transactions is high. ● Losing work/data is painful for users.

Table 2

In addition, Pessimistic Offline Lock has three types of “locks” which will be compared in the following table.

Types of Lock	Property	Liveness	Correctness	Complexity
Exclusive write Lock	Lock to edit data	Good	Inconsistent Reads	Easy
Exclusive read lock	Lock to load (either to read or edit)	Bad	Yes	Easy
Read/write lock	● Lock to read, lock to write. ● Multiple read locks & single write lock	Good	Yes	Hard

	● Read locks and write lock are mutually exclusive			

Table 3

It is important to choose the most appropriate type of Pessimistic Offline Lock according to the actual situation. We need to follow some basic instructions shown below. [1]

- Maximize liveness
- Meeting business needs
- Consider correctness
- Minimize code complexity

Concurrency Issue 1

Description

Multiple people within an airline log into the same account.

- Situation 1: Multiple people create flights simultaneously.
- Situation 2: Multiple people edit the same flight simultaneously.
- Situation 3: Multiple people delete the flights simultaneously.

Choice of pattern and rationale

With respect to Situation 1, all people can create a flight by clicking the "Click Flight" button in the airline home page and there will be no conflict involved. If multiple people create exactly the same flights, they can view all the flights and decide which flight(s) should be retained and delete the redundant one.

With respect to Situation 2 and 3, pessimistic offline lock is used to prevent conflicts. This is because:

- The probability of conflict between concurrent transactions is high. An airline is usually a large company that contains hundreds of employees. There will always be more than one person who is responsible for managing all its flights, tickets and passengers on the website. Hence, it is highly likely that more than one person is editing the same flight at a time.
- Losing work is painful for users. While editing a flight, a user might need to re-enter all the required information about the flight, including date, time, source, destination, stopovers, etc. It will be painful for the user if the editing process is unsuccessful and all the works are lost.

Pessimistic offline lock has three types and exclusive write lock is chosen. When some is writing to the database, exclusive write lock will only prevent others from writing. This will provide great liveness, but have the risk of inconsistent reads. In this situation, we want to prioritise liveness rather than consistency. This is because:

- There are various reasons that can lead to changing or cancelling a flight, for example, bad weather or security issues. Hence, editing and cancelling a flight are highly likely to occur many times during a day. If we always prevent others from viewing the flight data when someone is editing or deleting a flight, this will significantly increase the system latency and reduce liveness.
- Besides, since people know that there is a certain probability that a flight is changed or cancelled, they always check the flight details multiple times. As a result, one inconsistent read will not become an issue.

Implementation

In a package named "concurrency", there is a LockManage class which is responsible for managing the lock. When an airline is editing or deleting a flight (e.g. flight A), it should follow the steps below:

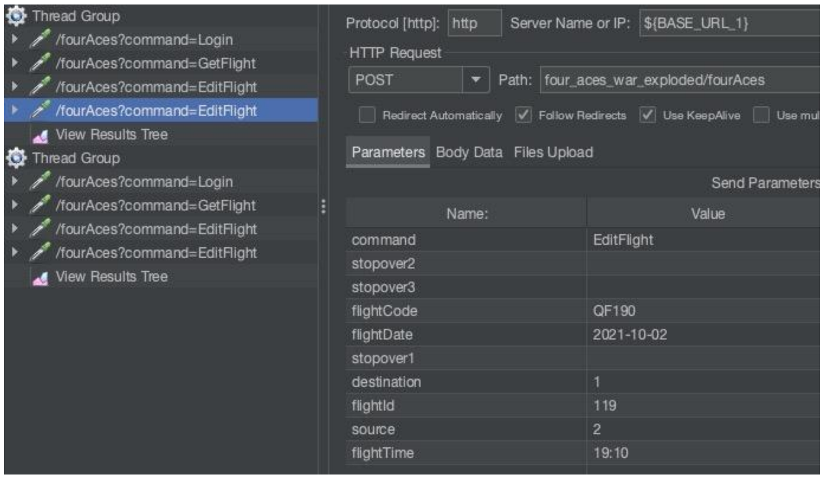
1. Airline acquire lock
2. Airline edit or delete flight A
3. Airline release lock

In step 1, if another person from the same airline is editing or deleting flight A, the lock cannot be acquired, and an error message will be displayed on the page.

Testing Strategy

Situation 2 Editing Flight Testing

Use JMeter to send 2 concurrent requests from the same airline account to edit the flight with ID equals 119. The origin flight time is 19:00. One request tried to change flight time to 19:10, and another request tried to change flight time to 19:20.



- Thread Group
  - /fourAces?command=Login
  - /fourAces?command=GetFlight
  - /fourAces?command=EditFlight
  - /fourAces?command=EditFlight
  - View Results Tree
- Thread Group
  - /fourAces?command=Login
  - /fourAces?command=GetFlight
  - /fourAces?command=EditFlight
  - /fourAces?command=EditFlight
  - View Results Tree

Protocol [http]: http Server Name or IP: ${BASE_URL_1}

HTTP Request

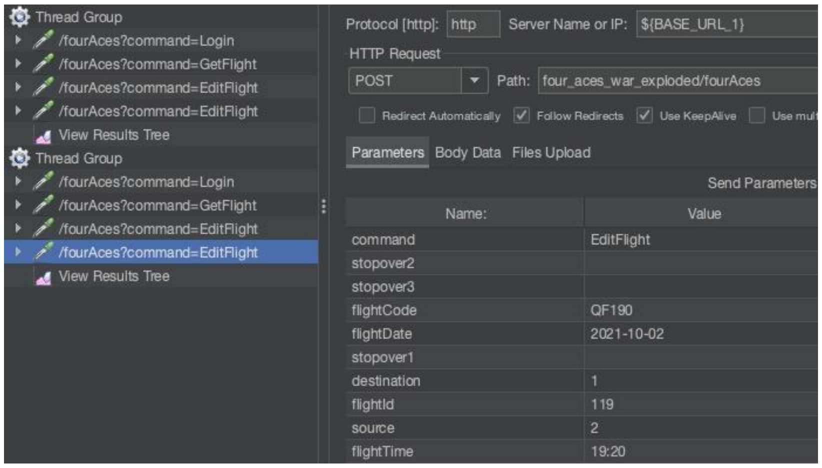
POST Path: four_aces_war_exploded/fourAces

Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use mul

Parameters Body Data Files Upload

Send Parameters

Name:	Value
command	EditFlight
stopover2	
stopover3	
flightCode	QF190
flightDate	2021-10-02
stopover1	
destination	1
flightId	119
source	2
flightTime	19:10



- Thread Group
  - /fourAces?command=Login
  - /fourAces?command=GetFlight
  - /fourAces?command=EditFlight
  - /fourAces?command=EditFlight
  - View Results Tree
- Thread Group
  - /fourAces?command=Login
  - /fourAces?command=GetFlight
  - /fourAces?command=EditFlight
  - /fourAces?command=EditFlight
  - View Results Tree

Protocol [http]: http Server Name or IP: ${BASE_URL_1}

HTTP Request

POST Path: four_aces_war_exploded/fourAces

Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use mul

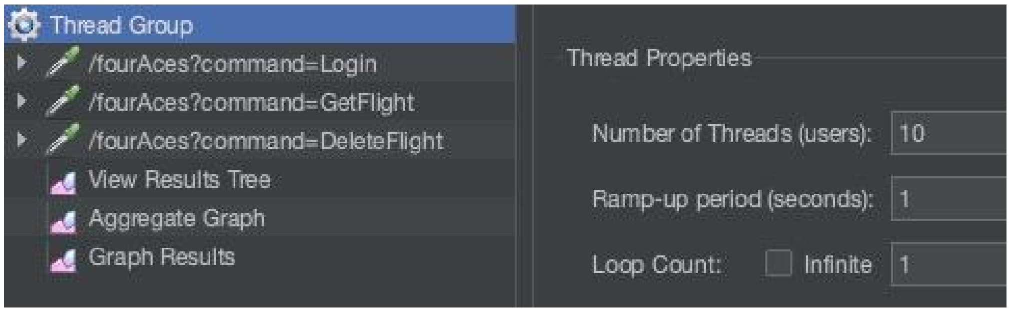
Parameters Body Data Files Upload

Send Parameters

Name:	Value
command	EditFlight
stopover2	
stopover3	
flightCode	QF190
flightDate	2021-10-02
stopover1	
destination	1
flightId	119
source	2
flightTime	19:20

Situation 3 Deleting Flight Testing

Use JMeter to send 10 concurrent requests from the same airline account to delete the flight with ID equals 118.



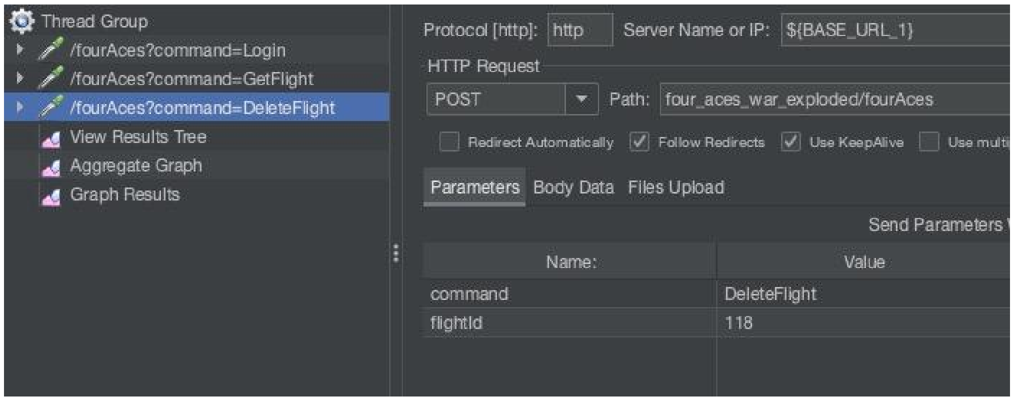
- Thread Group
  - /fourAces?command=Login
  - /fourAces?command=GetFlight
  - /fourAces?command=DeleteFlight
  - View Results Tree
  - Aggregate Graph
  - Graph Results

Thread Properties

Number of Threads (users): 10

Ramp-up period (seconds): 1

Loop Count: Infinite 1

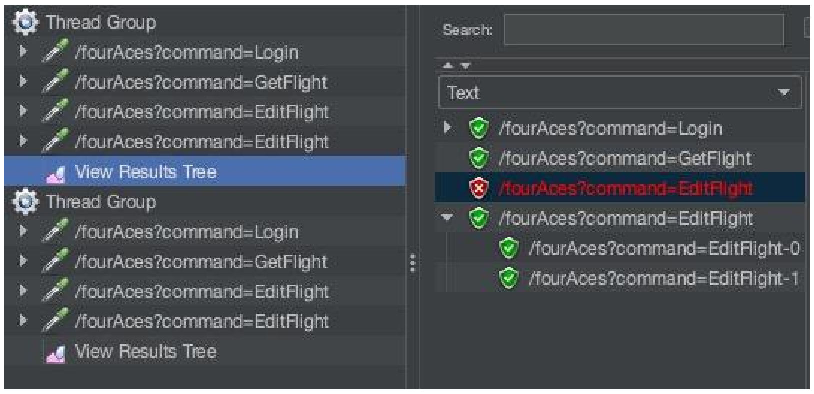


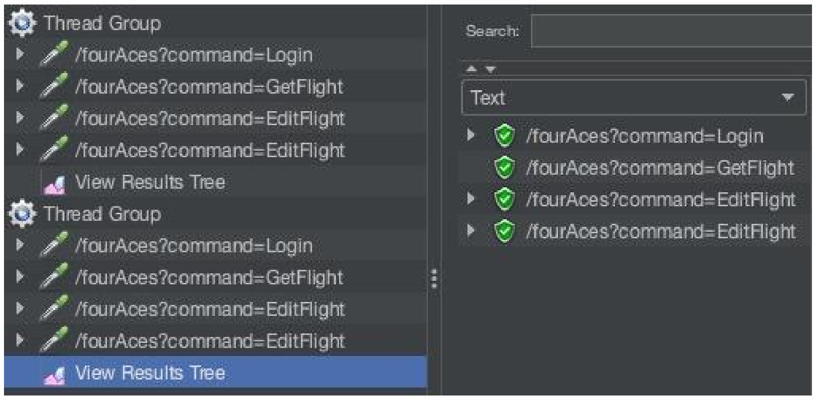
Name:	Value
command	DeleteFlight
flightId	118

Testing Outcomes

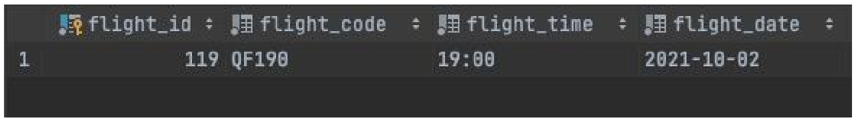
Situation 2 Editing Flight Testing Outcomes

Results Tree


Thread Group
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight
View Results Tree
Thread Group
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight
View Results Tree
Search:
Text
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight-0
/fourAces?command=EditFlight-1

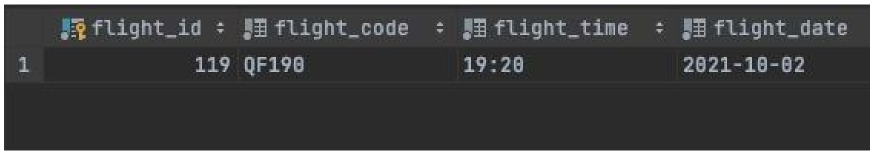

Thread Group
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight
View Results Tree
Thread Group
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight
View Results Tree
Search:
Text
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=EditFlight
/fourAces?command=EditFlight

Database before concurrent request



	flight_id	flight_code	flight_time	flight_date
1	119	QF190	19:00	2021-10-02

Database after concurrent request

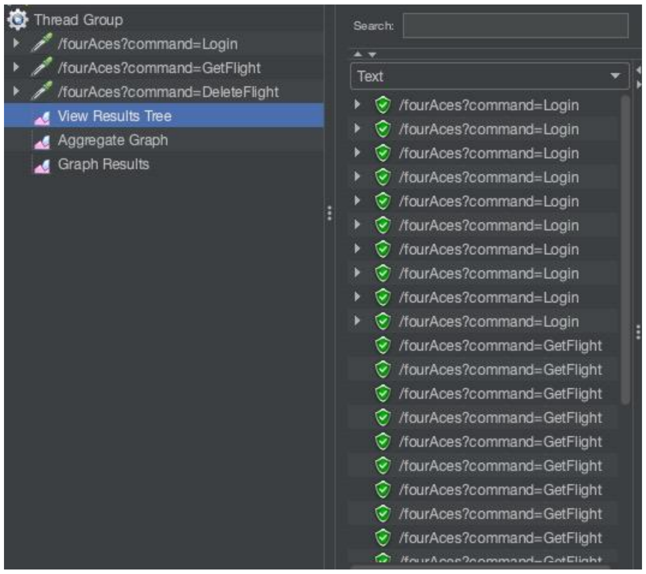


	flight_id	flight_code	flight_time	flight_date
1	119	QF190	19:20	2021-10-02

As the above pictures show that, only the second request succeeded. Since the second request tries to change the flight time to 19:20, the flight time shown in the database becomes 19:20. The database is still consistent.

Situation 3 Deleting Flight Testing Outcomes

Results tree

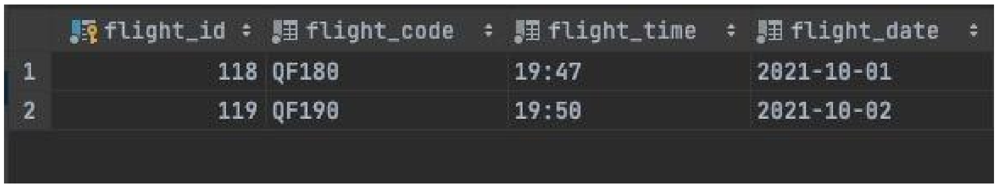


Thread Group
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=DeleteFlight
View Results Tree
Aggregate Graph
Graph Results

Search:
Text

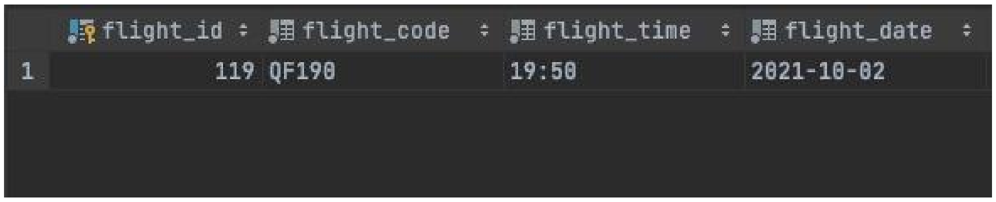
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=Login
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight
/fourAces?command=GetFlight

Database before concurrent request



	flight_id	flight_code	flight_time	flight_date
1	118	QF180	19:47	2021-10-01
2	119	QF190	19:50	2021-10-02

Database after concurrent request



	flight_id	flight_code	flight_time	flight_date
1	119	QF190	19:50	2021-10-02

The result shows that only one flight is deleted and the database is still consistent.

Concurrency Issue 2

Description

Situation 1: Customer books a flight that is being edited by an airline
Situation 2: Customer books a flight that is being deleted by an airline
Situation 3: Multiple customers booking the same seat.

Choice of pattern and rationale

Situation 1: The customer clicks "View Flight" and then "Book Flight" if they wish to book the flight. This will add a flight to a reservation for easy keeping track of which flight the customer books for and whether or not they are returning. This action simply reads the Flight object from the database without writing to the table. A concurrency issue might happen when an airline is editing the same flight. In this case, we do not implement a Read-Write lock and only use an exclusive Write lock to prevent multiple people from the same airline editing the flights, as mentioned in Concurrency Issue 1.

- This may result in the customer reading and choosing to book an outdated flight. However, when the airline edits a flight, it does not just affect those who are viewing and booking the flight, but also those who have already finished booking that flight. There will have to be notifications or announcements to make sure the customers are aware of the change. We should treat those who are in the middle of the booking process this way, because they are booking for an outdated flight, similar to those who have finished their booking.
- Chances that an airline would want to edit a flight that has already been published is very low, because it would cause a lot of trouble when dealing with unhappy customers. Therefore, customers are not likely to book an outdated flight.
- We would like to preserve the liveness of the system. We do not want to block all customers from viewing the flight information whenever the airline is editing it. Therefore, reading the flight is still allowed without causing too much concurrency issue.
- After the "Book Flight" step, the customer is asked to add passengers and selects the seats for the passengers. This action writes to the Ticket table, but airlines are forbidden from changing the tickets when editing a flight, so no conflict will occur on the Ticket objects.

Situation 2: This is similar to Situation 1. We apply a Write exclusive lock on the Flight when an airline is editing it, but customers are still allowed to read it. The difference is that deleting a flight affects the tickets by deleting them as well, whereas editing a flight does not. In this case, both the customer and the airline might try to write to the Ticket object at the same time. We have implemented a Write lock, so that when a thread wants to write to the object, it has to acquire the lock. As a result, the customer and the airline will not be able to write to the ticket table at the same time. If the flight has been deleted in the middle of the booking process when a command is made from the customer, then they will be redirected to the error page and allow the customer to cancel the booking.

- Although the probability that this situation happens is very low, for the same reason as in Situation 1, we still implement the lock because the repetition of the whole editing/booking process when a concurrency error occurs is very painful in case it happens.

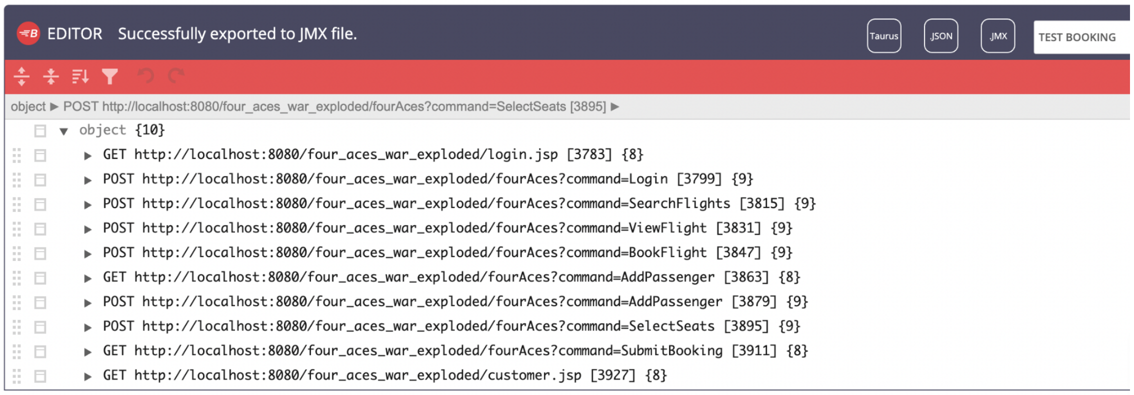
Situation 3: The customer chooses the specific seats for the passengers. This action writes to the Ticket table in the database. We have decided to use a write lock for the Ticket objects. The lock is acquired when the customer selects the seats and is released after the customer submits the booking or cancels the whole process. The reasons are:

- We use the pessimistic approach because chances that more than one customer selects the same seat are very high, especially when the flight is selling fast.
- It is better to lock the seat right when it has been selected until the customer is done with it, than to allow conflicts to happen and customers have to go back and change their options, especially when the process is made longer by adding many passengers.. Customers can choose from plenty of other seats so the locking of the ticket would not cause too much disruption to the liveness of the system.

Testing Strategy

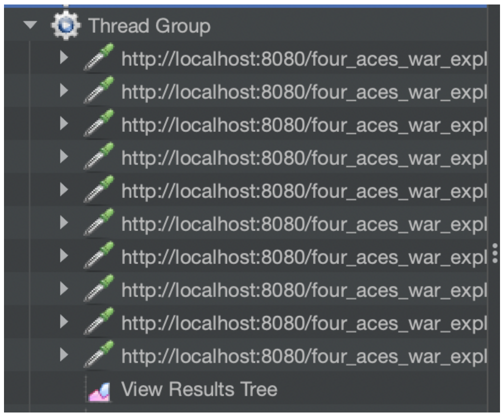
Situation 3: Two passengers booking the same seat

Step 1, we used BlazeMeter to record the process from customer login to customer submitting tickets.

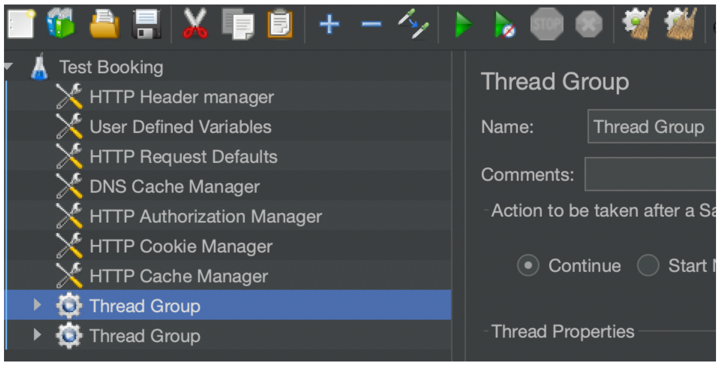


```
Successfully exported to JMX file.
object ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=SelectSeats [3895] ▶
▼ object {10}
  ▶ GET http://localhost:8080/four_aces_war_exploded/login.jsp [3783] {8}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=Login [3799] {9}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=SearchFlights [3815] {9}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=ViewFlight [3831] {9}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=BookFlight [3847] {9}
  ▶ GET http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger [3863] {8}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger [3879] {9}
  ▶ POST http://localhost:8080/four_aces_war_exploded/fourAces?command=SelectSeats [3895] {9}
  ▶ GET http://localhost:8080/four_aces_war_exploded/fourAces?command=SubmitBooking [3911] {8}
  ▶ GET http://localhost:8080/four_aces_war_exploded/customer.jsp [3927] {8}
```

Step 2, We added a listener “View Results Tree”.


Thread Group
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
http://localhost:8080/four_aces_war_expl
View Results Tree

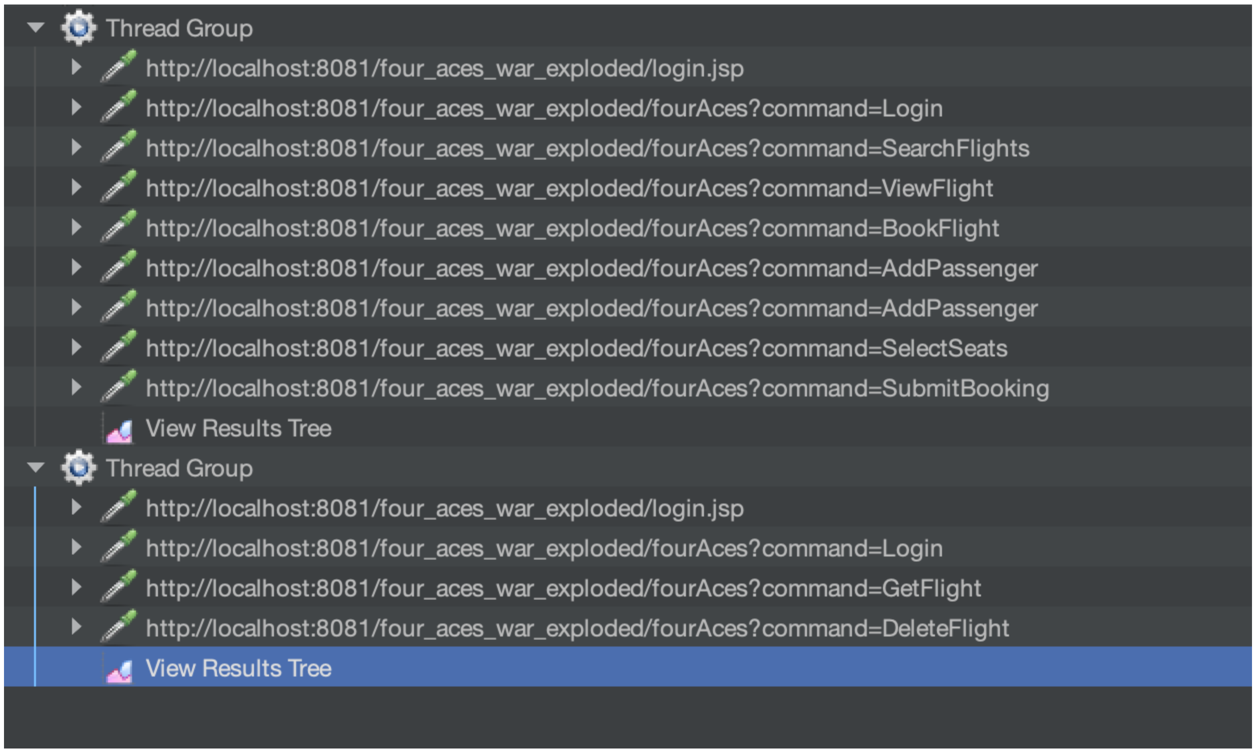
Step 3, for multiple customers booking the same seat, we sent two booking tickets thread group requests (one is from the BlazeMeter JMX file, the other is a duplicate one) to test.


Test Booking
HTTP Header manager
User Defined Variables
HTTP Request Defaults
DNS Cache Manager
HTTP Authorization Manager
HTTP Cookie Manager
HTTP Cache Manager
Thread Group
Thread Group

Thread Group
Name: Thread Group
Comments:
Action to be taken after a Sa
[✓ Continue] Start
Thread Properties

Situation 2: Booking a flight that is being deleted

Below are 2 threads performing 2 series of actions. These series of actions are recorded using the BlazeMeter plugin, as demonstrated above. The first thread is the customer booking a ticket for a flight. The second thread is the airline deleting that flight.



```
Thread Group
  http://localhost:8081/four_aces_war_exploded/login.jsp
  http://localhost:8081/four_aces_war_exploded/fourAces?command=Login
  http://localhost:8081/four_aces_war_exploded/fourAces?command=SearchFlights
  http://localhost:8081/four_aces_war_exploded/fourAces?command=ViewFlight
  http://localhost:8081/four_aces_war_exploded/fourAces?command=BookFlight
  http://localhost:8081/four_aces_war_exploded/fourAces?command=AddPassenger
  http://localhost:8081/four_aces_war_exploded/fourAces?command=AddPassenger
  http://localhost:8081/four_aces_war_exploded/fourAces?command=SelectSeats
  http://localhost:8081/four_aces_war_exploded/fourAces?command=SubmitBooking
  View Results Tree
Thread Group
  http://localhost:8081/four_aces_war_exploded/login.jsp
  http://localhost:8081/four_aces_war_exploded/fourAces?command=Login
  http://localhost:8081/four_aces_war_exploded/fourAces?command=GetFlight
  http://localhost:8081/four_aces_war_exploded/fourAces?command=DeleteFlight
  View Results Tree
```

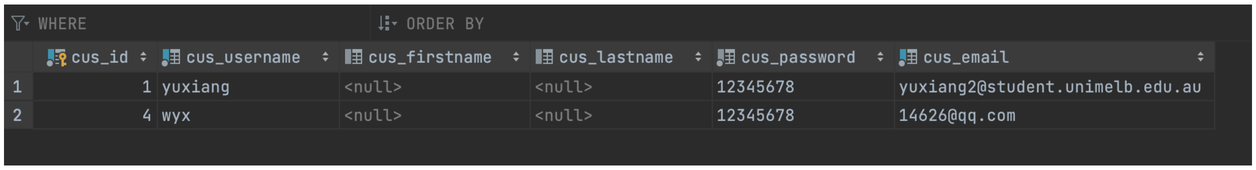
Situation 1: Booking a flight that is being edited.

As discussed above, there is no concurrent write access to a shared resource in this situation so it is not tested.

Testing Outcomes

Situation 3: Two passengers booking the same seats

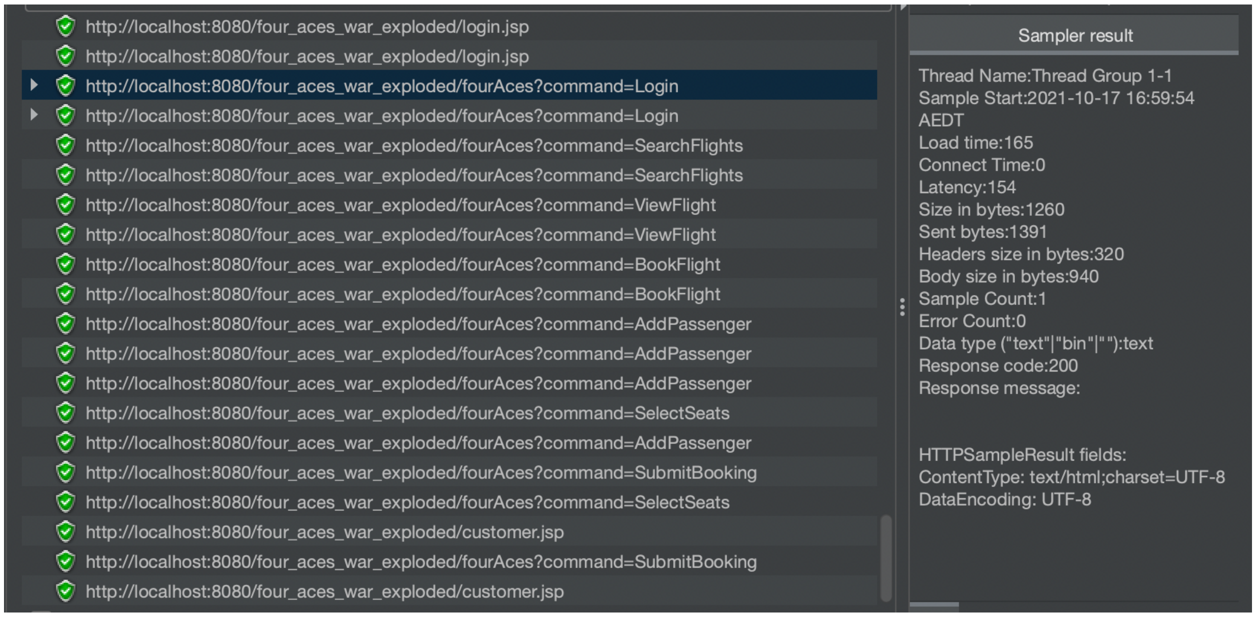
There are two different customers: cus_id = 1 and cus_id = 4 login our TRS systems to book the same seat, which is shown in our customer database table.



WHERE ORDER BY

	cus_id	cus_username	cus_firstname	cus_lastname	cus_password	cus_email
1	1	yuxiang	<null>	<null>	12345678	yuxiang2@student.unimelb.edu.au
2	4	wyx	<null>	<null>	12345678	14626@qq.com

After we input two thread groups (one is from our BlazeMeter the other is a duplicate to JMeter and choose "view results tree" listener, we start the test and the testing outcomings as the following screenshot.



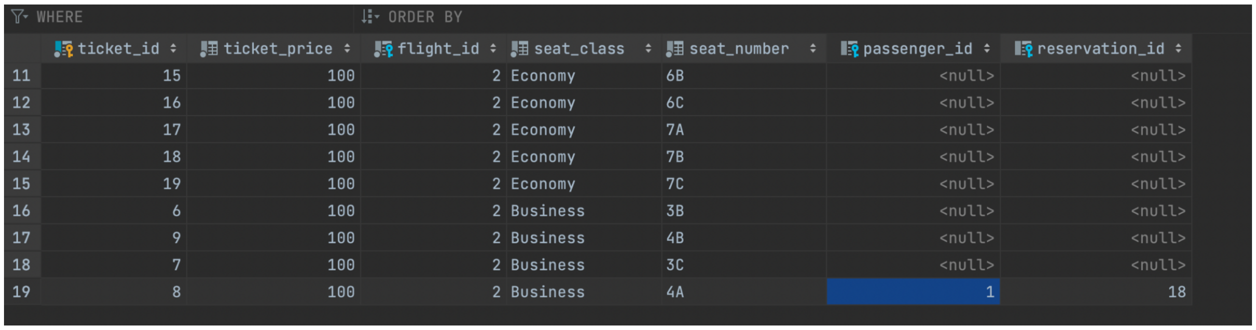
```
http://localhost:8080/four_aces_war_exploded/login.jsp
http://localhost:8080/four_aces_war_exploded/login.jsp
http://localhost:8080/four_aces_war_exploded/fourAces?command=Login
http://localhost:8080/four_aces_war_exploded/fourAces?command=Login
http://localhost:8080/four_aces_war_exploded/fourAces?command=SearchFlights
http://localhost:8080/four_aces_war_exploded/fourAces?command=SearchFlights
http://localhost:8080/four_aces_war_exploded/fourAces?command=ViewFlight
http://localhost:8080/four_aces_war_exploded/fourAces?command=ViewFlight
http://localhost:8080/four_aces_war_exploded/fourAces?command=BookFlight
http://localhost:8080/four_aces_war_exploded/fourAces?command=BookFlight
http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger
http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger
http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger
http://localhost:8080/four_aces_war_exploded/fourAces?command=SelectSeats
http://localhost:8080/four_aces_war_exploded/fourAces?command=AddPassenger
http://localhost:8080/four_aces_war_exploded/fourAces?command=SubmitBooking
http://localhost:8080/four_aces_war_exploded/fourAces?command=SelectSeats
http://localhost:8080/four_aces_war_exploded/customer.jsp
http://localhost:8080/four_aces_war_exploded/fourAces?command=SubmitBooking
http://localhost:8080/four_aces_war_exploded/customer.jsp
```

```
Sampler result

Thread Name:Thread Group 1-1
Sample Start:2021-10-17 16:59:54
AEDT
Load time:165
Connect Time:0
Latency:154
Size in bytes:1260
Sent bytes:1391
Headers size in bytes:320
Body size in bytes:940
Sample Count:1
Error Count:0
Data type ("text"|"bin"|""):text
Response code:200
Response message:


HTTPSampleResult fields:
ContentType: text/html;charset=UTF-8
DataEncoding: UTF-8
```

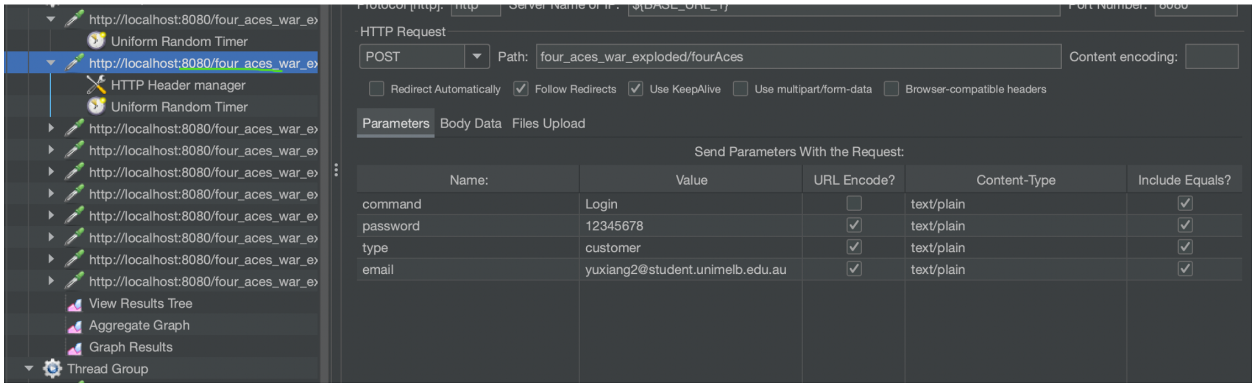
And we can see our ticket database table shows there is only one ticket generated and reserved by passenger 1.



WHERE ORDER BY

	ticket_id	ticket_price	flight_id	seat_class	seat_number	passenger_id	reservation_id
11	15	100	2	Economy	6B	<null>	<null>
12	16	100	2	Economy	6C	<null>	<null>
13	17	100	2	Economy	7A	<null>	<null>
14	18	100	2	Economy	7B	<null>	<null>
15	19	100	2	Economy	7C	<null>	<null>
16	6	100	2	Business	3B	<null>	<null>
17	9	100	2	Business	4B	<null>	<null>
18	7	100	2	Business	3C	<null>	<null>
19	8	100	2	Business	4A	1	18

After clicking the "http://localhost:8080/four_aces_war_exploded/fourAces?command=Login" request , we can see the left table showing the first customer login information whose customer id is 1, and is corresponding to the ticket information.



```
http://localhost:8080/four_aces_war_e›
    Uniform Random Timer
http://localhost:8080/four_aces_war_e›
    HTTP Header manager
    Uniform Random Timer
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
http://localhost:8080/four_aces_war_e›
View Results Tree
Aggregate Graph
Graph Results
Thread Group
```

HTTP Request

POST Path: four_aces_war_exploded/fourAces Content encoding:

Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use multipart/form-data Browser-compatible headers

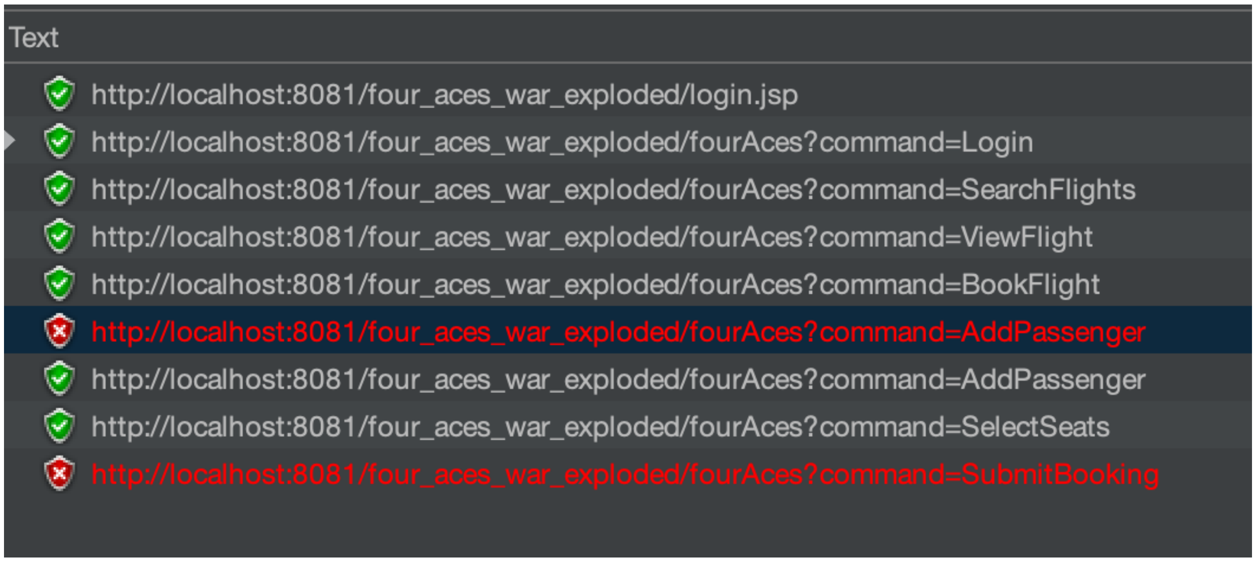
Parameters Body Data Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?
command	Login		text/plain	✓
password	12345678	✓	text/plain	✓
type	customer	✓	text/plain	✓
email	yuxiang2@student.unimelb.edu.au	✓	text/plain	✓

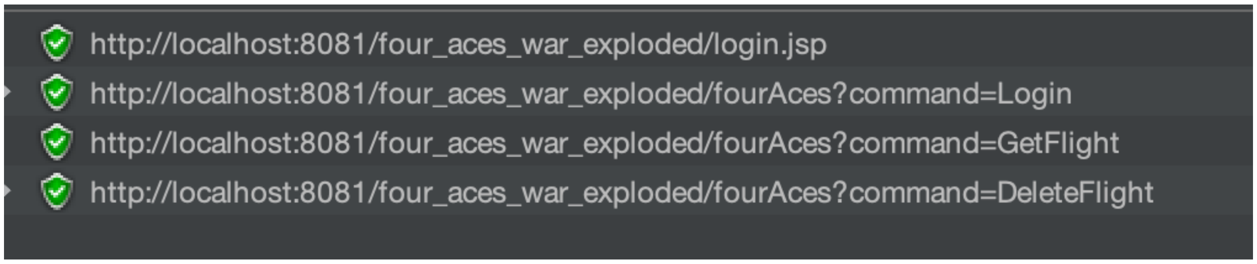
Situation 2: Booking a flight that is being deleted

As a result, after running two threads concurrently, only one of them succeeds. The customer thread fails because the flight has already been deleted by the customer. The result persists after several trials.



```
Text
✓ http://localhost:8081/four_aces_war_exploded/login.jsp
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=Login
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=SearchFlights
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=ViewFlight
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=BookFlight
✗ http://localhost:8081/four_aces_war_exploded/fourAces?command=AddPassenger
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=AddPassenger
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=SelectSeats
✗ http://localhost:8081/four_aces_war_exploded/fourAces?command=SubmitBooking
```

The airline's thread successfully deleles the flight.



```
✓ http://localhost:8081/four_aces_war_exploded/login.jsp
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=Login
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=GetFlight
✓ http://localhost:8081/four_aces_war_exploded/fourAces?command=DeleteFlight
```

Heroku App Link

Heroku deployed app link

https://travel-reservation.herokuapp.com/

Testing Guide

Administrator
Username: admin
Password: admin

Airline
Email: qantas@gmail.com
Password: qantas#

Customer 1
Email: xueqi.guan@gmail.com
Password: guanxueqi

Customer 2
Email: yiyuanw1@student.unimelb.edu.au
Password: jiuk7o98l

Flight details
Date: 2021-12-01
Source: Melbourne
Destination: Sydney

To test concurrency issues, please use two browsers to open the app link.
Test concurrency issue 1
Situation 1 Edit flight

- In browser 1, login with airline email and password. In browser 2, login with the same email and password.
- In browser 1, click "View Flight", then choose one flight to edit.
- In browser 2, click "View Flight", then choose the same flight to edit.
- An error message will be displayed on browser 2, indicating that it is not able to edit flight.
- After browser 1 finishes editing the flight and save, browser 2 will be able to edit the flight.

Situation 2 Delete flight

- In browser 1, login with airline email and password. In browser 2, login with the same email and password.
- In browser 1, click "View Flight", then choose one flight to delete.
- In browser 2, click "View Flight", then choose the same flight to delete.
- An error message will be displayed on browser 2, indicating that it is not able to delete the flight or the flight has been deleted.

including data/information that we need in order to be able to test your application.

Test concurrency issue 2

Situation 1 Airline edit flight while customer booking this flight
Since we are allowing the customer to book an outdated flight, the airline edits the flight details will not affect the process of booking a ticket. Therefore, it is not necessary to run some particular steps for this test. Following the normal steps will just safices, and there should be no any error messages related to concurrency issues.

Situation 2 Delete flight while customer booking this flight

- In browser 1, login with the customer credentials, and airline credentials in browser 2.
- In browser 1, customer book flight as usual (search -> view -> return (optional) -> book), until reaching addPassenger
- In browser 2, click “View Flight”
- In browser 1, customer fill in the passenger information and click “Add Passenger”
- Then in browser 1, customer choose a seat and click “Select Seats”
- In browser 2, airline clicks “Delete Flight”
- If there is no error displayed in browser 2 and the flight the airline chose to delete is the one customer is booking, an error message will be displayed in browser 1 when clicking on either “Add Passenger” or “Submit Booking”

Situation 3 Two passenger books same seat

- In browser 1, login with customer 1 email and password. In browser 2, login with the customer 2 email and password.
- In browser 1, customer 1 book a flight as usual (search -> view -> return (optional) -> book), until reaching select seat
- In browser 2, customer 2 book a flight as usual (search -> view -> return (optional) -> book), until reaching select seat
- Both customers select the same seat and click the “Select Seat” button. Only the first one who clicks the button will be able to select the seat and the other should select another seat to continue booking.

Reference

[1] The University of Melbourne, “Software Design and Architecture Subject Notes for SWEN90007”, 2021.

[2] M. Fowler, Patterns of enterprise application architecture. Boston, Mass.: Addison-Wesley, 2015.