
Product Requirements Document

Submission 3

TeamWZ

SWEN90007 SM2 2023 Project

Student Name	Student ID	UniMelb Username	Github Username	Email
HanXiang Wang	1343151	HANXIANGW	Hanxiang-WANG	hanxiangw@student.unimelb.edu.au
Kehan Zhang	1331979	KEHZHANG	KehanZhang0712	kehzhang@student.unimelb.edu.au
Xiaoran Zhang	1253830	XIAORANZ3	xiaoranz1	xiaoranz3@student.unimelb.edu.au
Zhiyao Wang	1290370	ZHIYAOW1	ZhiyaoW	zhiyaow1@student.unimelb.edu.au

Contents

1. Introduction	3
1.1 Proposal	3
1.2 Target Users	3
1.3 Conventions, terms, and abbreviations	3
2. Actors	3
3. Class diagram	4
4. Concurrency issues by Use Case	7
4.1 Use case 1: User buy tickets	7
4.1.1 Concurrency issue 1.1	7
4.1.2 Concurrency issue 1.1.1	8
4.2 Use case 2: User register	8
4.2.1 Concurrency issue 2.1	8
4.3 Use case 3: Planner edit event	10
4.3.1 Concurrency issue 3.1	10
4.4 Use case 4: Admin manage event	11
4.4.1 Concurrency issue 4.1	11
4.5 Use case 5: Planner create event	12
4.5.1 Concurrency issue 5.1	12
4.6 Use case 6: Admin delete user	14
4.6.1 Concurrency issue 6.1	14
4.6.2 Concurrency issue 6.2	15
5. Test Strategy	17
5.1 Why Jmeter	17
5.2 Manual Test	17
5.3 Test Cases	17
5.4 Test Outcomes	19
5.4.1 1.1 issue & 1.1.1 issue	19
5.4.2 2.1 issue	20
5.4.3 3.1 issue	21
5.4.4 4.1 issue	21
5.4.5 5.1 Issue	23
5.4.6 6.1 issue	24
5.4.8 6.2 issue	25

1. Introduction

1.1 Proposal

This document has updated the use cases based on feedback from Part 1. Also, this document also specifies a series of patterns, including Data mapper, Unit of work, Lazy load, Identity field, Foreign key mapping, Association table mapping, Embedded value, inheritance patterns, Authentication and Authorization.

1.2 Target Users

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

1.3 Conventions, terms, and abbreviations

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

Term	Description
Administrator	Special user who responsible for managing the music event system
Customer	Users who use the website to book tickets
Event Planner	Users who create music events (price of ticket and venue,...)
Tickets	The corresponding performance tickets for each music event.
Venue	The corresponding performance location for each music event.
Bookings	Customers book concert tickets on the music event platform.
Events	Music events that can be found on this platform.

2. Actors

Actor	Description
Customer	Customers who use the app to book tickets, etc
Event Planner	Event planners who create events
Administrator	Administrators who manage the system

3. Class diagram

This section decomposes the system into multiple different class diagrams, including the overview class diagram, controller layer class diagram, domain layer class diagram, and DataSource layer class diagram. Committed to displaying the relationships between classes and layers of the system in a simplified manner. The main focus of class diagram update is to redraw the correct domain layer class diagram, add the controller layer class diagram, datasource class diagram, and overview class diagram.

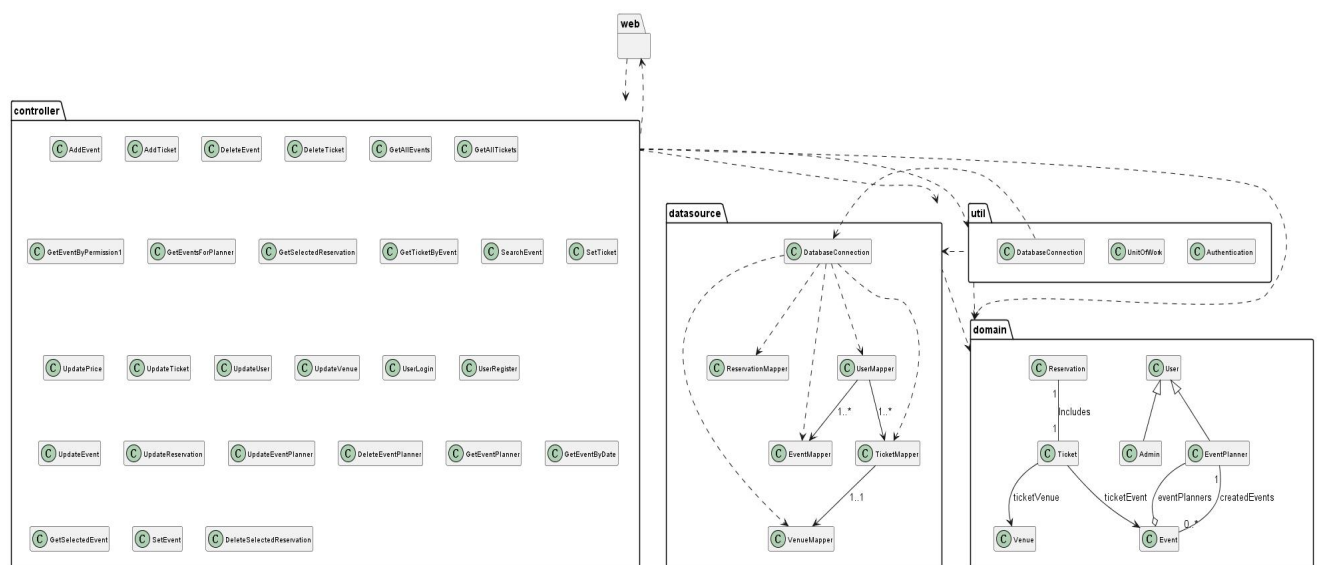


Figure 1 Overview class diagram

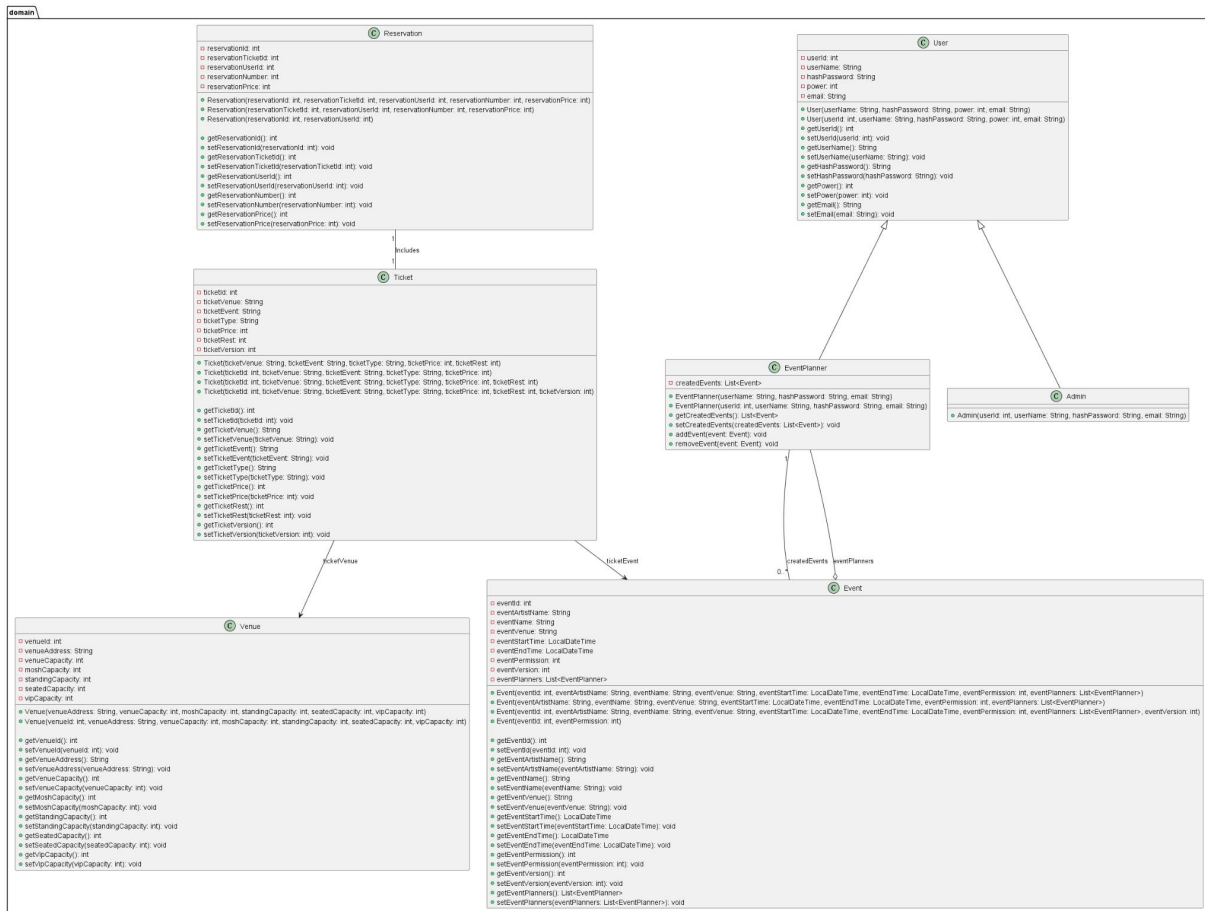


Figure 2 domain layer class diagram

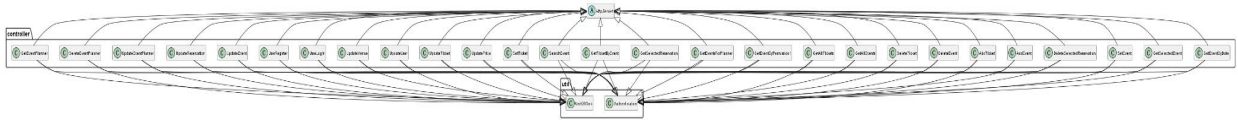


Figure 3 controller layer class diagram

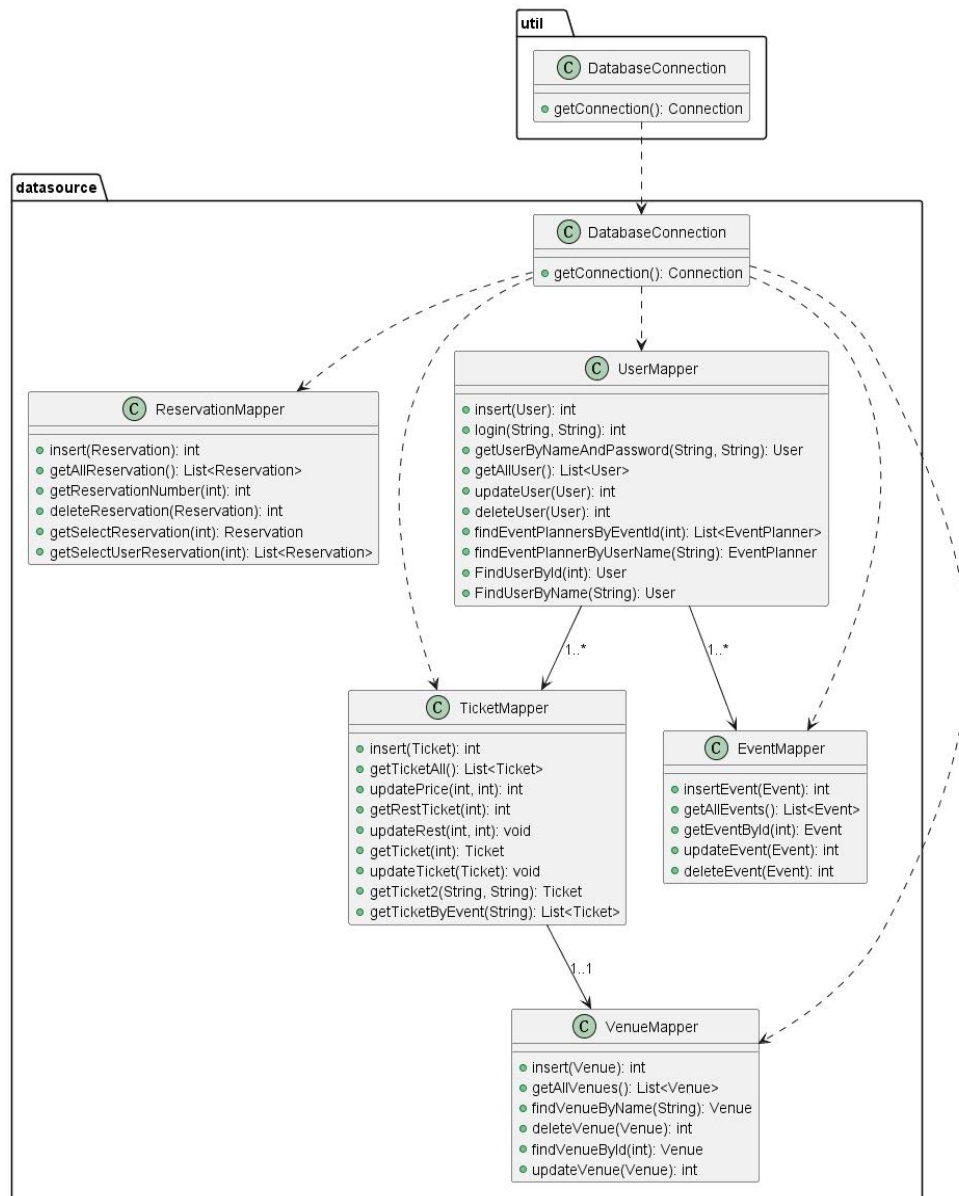


Figure 4 Datasource class diagram

4. Concurrency issues by Use Case

4.1 Use case 1: User buy tickets

Since the number of tickets in different areas of each event is limited, concurrency problems will occur if a large number of users buy tickets at the same time.

4.1.1 Concurrency issue 1.1

Description: Multiple users purchase the same type of tickets at the same time. The quantity purchased by each user is less than the remaining quantity, but the total quantity purchased by these users exceeds the remaining quantity.

Sequence Diagram:

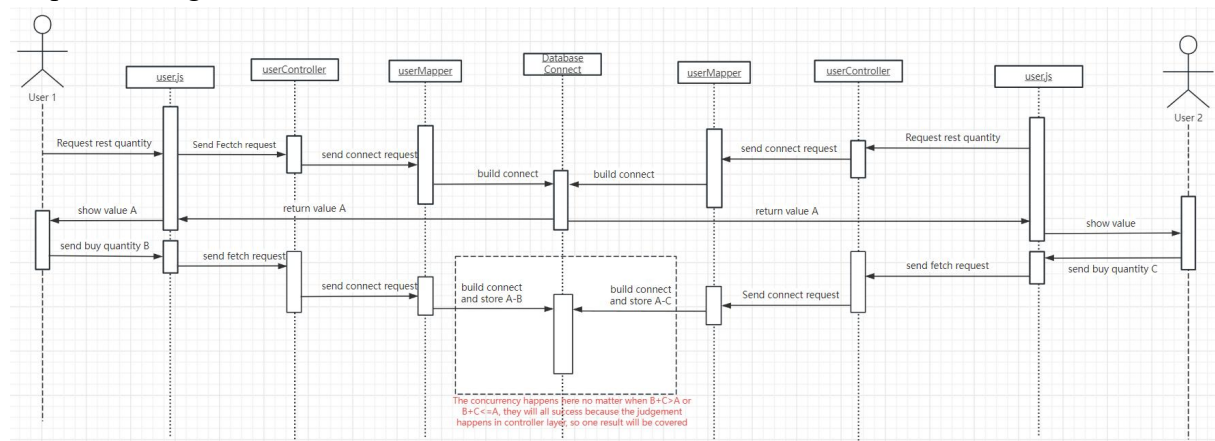


Figure5 Sequence diagram of Concurrency issue 1.1

Solution Pattern and description: We use Optimistic lock + version control to solve the issue. Before we add the Optimistic lock, when two users try to buy tickets at the same time, they both require the same number A from the database, and they input B and C respectively. The AddTicketController will process the data and send $(A-B) / (A-C)$ to the Mapper at the same time. Finally the result stored into the database will become A-B or A-C (one is covered by another), but the correct result should be A-B-C. To solve this problem, we use the Optimistic lock and set an exception for version control. When judging the purchasing, we add one more condition for the version. We add a version number for each ticket, their initial values are all 0, when an user A tries to buy the ticket, he will firstly acquire the version number in the database, we will compare the version of the object with which in database, if they are all zero, it means no user is buying this now, user A will successfully buy the tickets and update the version number into 1 (version+1). But if at the same time user B also want to buy tickets, he will also get a version of 0 because he acquire this number at the same time with A, but when he try to process the request, the version in database has been changed by A to 1, so we compare the versions and find that they were different, so user B will fail to buy tickets and throw the exception.

4.1.2 Concurrency issue 1.1.1

Description : When optimistic locking is used to successfully handle concurrency problems, purchases will be protected and over-purchasing will not occur even when traffic is large. However, if there are still enough tickets, two people happen to purchase them at the same time. For the same ticket, one of the requests will be rejected, which is not very humane.

Proposed Solution : After we add the lock, the concurrency issue is solved. But in that situation, only one request will be processed, others will fail due to the judgment. So we add a for loop when processing concurrent request information so that requests at the same time are processed asynchronously. If the rest of the tickets are enough, they will successfully buy the tickets and change the rest until the quantity is lower than the request number.

Design rationale:

We choose Optimistic Lock because Optimistic locking does not lock data immediately when performing data operations. Only when the data is actually updated, will it be checked whether the data has been modified by other transactions after reading the data in the previous period. By doing this, we can:

1.Improved user experience: Since resources do not need to be locked frequently, users are not easily blocked by other transactions when operating data, providing a smoother user experience.

2.Improved responsiveness: There is no need to wait for a long time for the lock to be released, which is especially important for systems that require high concurrent processing. By adding a loop work with optimistic locking, we can allow more users to buy tickets when there are sufficient tickets.

Furthermore, we don't have to worry about the loss of user data. Because the operation of purchasing tickets is very simple, just select the quantity and purchase, so even the loss of data loss is very small and within the acceptable range.

4.2 Use case 2: User register

Our system performs login verification based on the user name when registering, so each user name should be unique.

4.2.1 Concurrency issue 2.1

Description : When two or more users register with the same username at the same time, concurrency problems will occur because the same username is not allowed.

Sequence Diagram:

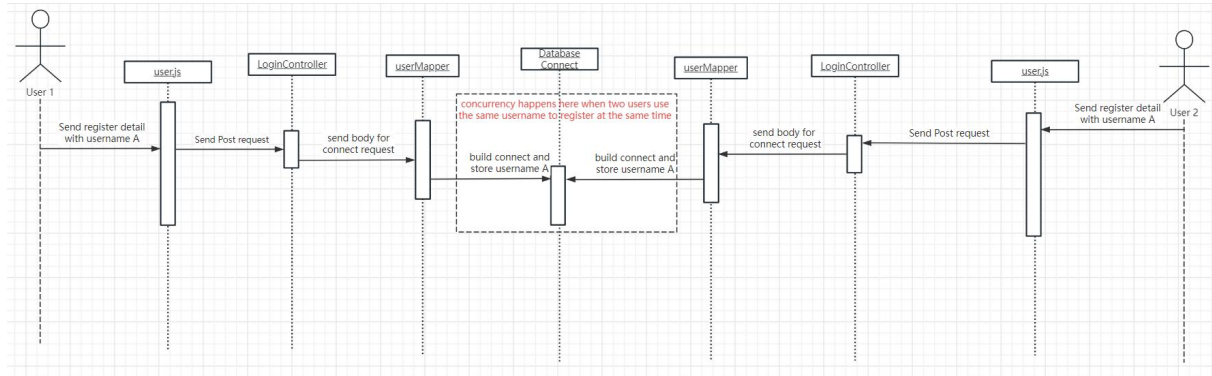


Figure6 Sequence diagram of Concurrency issue 2.1

Proposed Solution Pattern: We use **Optimistic Lock** to solve the issue. Before we add the lock, when two users use the same username to register, it will violate our design principles in that we use username as the unique login verification. To avoid this, when a user successfully registers, which means the information is inserted into the database, we will add an associated version number as 1. We define an int version = 0 in the usermapper, when trying to insert a new user, if the insertion is successful, inserted will be 1, then version will increase the value of inserted, that is, become 1. If the insert fails, version remains 0, the code will call the `throwConcurrencyException` method. So if two users try to register as the same username, one user will successfully insert data and set version = 1, another user will use his version 0 to compare with 1, he will fail and throw the concurrency exception.

Design rationale: We chose Optimistic Lock for the reasons:

- 1.Data consistency: Using optimistic locking ensures that even if multiple users try to register with the same username at the same time, the system only allows one user to succeed. This ensures the uniqueness of usernames and data consistency in the database.
- 2.Performance advantages: The optimistic locking mechanism does not require locking the entire table or row before data is inserted, which means that the throughput and response time of the registration process are optimized.
- 3.Simplify system design: Compared with complex pessimistic locking mechanisms, optimistic locking provides a simpler and more direct way to deal with concurrency issues, thus simplifying system design and implementation.
4. Frequency of conflicts: The probability of conflict between register concurrent transactions is low so it's good to use Optimistic Lock.

4.3 Use case 3: Planner edit event

After the Planner creates the event and before it is allowed or denied by the Admin, the Planner can modify the content of the event, including venue information, event information, and related planners.

4.3.1 Concurrency issue 3.1

Description : Two or more planners may modify the same event at the same time, which may cause conflicts in modified content and failure to save data.

Sequence Diagram:

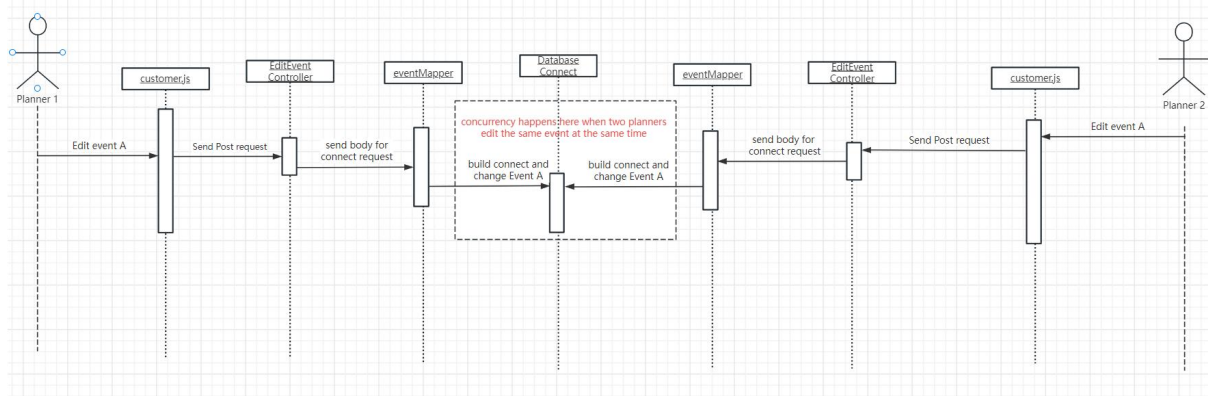


Figure7 Sequence diagram of Concurrency issue 3.1

Proposed Solution Pattern: We use **Optimistic Lock** to solve the issue.

Before we add the lock, When two or more planners try to edit the same event simultaneously, the results will be covered and only one planner's edit will be saved. That will cause conflicts and misunderstanding of the event information, which may lead to some problems between users and admins. So we add a version number for each event, when a planner requests the update of event information, he will firstly get the version number as 0, and we do the comparison. If it is the same as the version in the database, he will successfully update the data and add version to 1. Now if another user brings the initial version 0 to request for the update comparison, he will fail to process and rollback.

Design rationale:

1.Reduce data conflicts: Optimistic locking ensures that even if multiple planners edit the same event at the same time, only one planner's edits will be saved. This prevents data overwrites and inconsistencies. Although users will lose the saves they made, at least they won't mistakenly believe that they successfully modified the event and caused a misunderstanding.

2.Improved data integrity: Optimistic locking helps maintain data integrity and accuracy by ensuring that only users with the correct version number can update data.

3.Improve concurrency: Because it does not always lock resources, optimistic locking usually provides better performance in high-concurrency scenarios.

4.Reduce the risk of deadlock: Using optimistic locking reduces the risk of deadlock due to multiple threads or processes waiting for resources.

4.4 Use case 4: Admin manage event

Admin can approve or reject all events on the event management page. However, the Admin page will not be refreshed in real time, and you may not be able to see the updated events.

4.4.1 Concurrency issue 4.1

Description : When a planner has modified the content in an event or is modifying the event, and the Admin approves or rejects the event simultaneously.

Sequence Diagram:

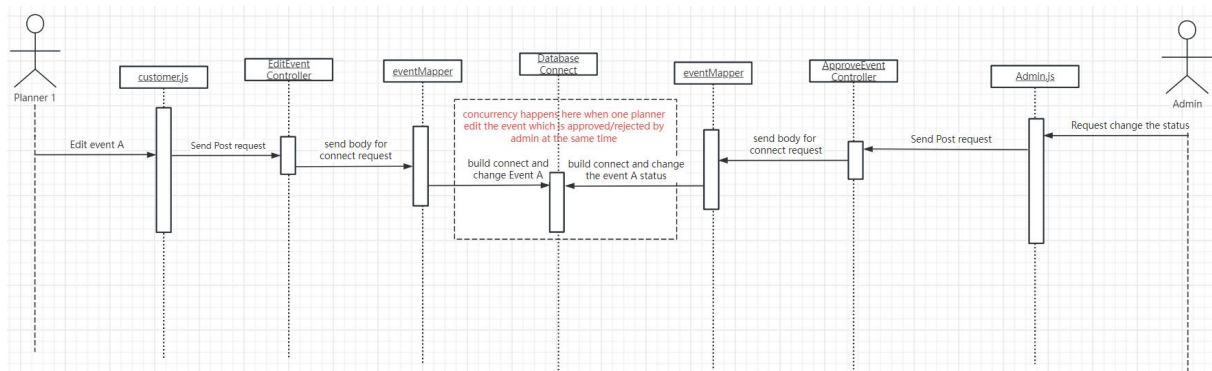


Figure8 Sequence diagram of Concurrency issue 4.1

Proposed Solution Pattern and Description: We use **Exclusive Read lock** to solve the issue. Before we add the lock, if the planner tries to edit the event and the admin approves or rejects it simultaneously, the data updated in the database will not be the same as what the Admin saw when processing. It will violate the design of our system. So we set a table to record the lockable state of events. When the admin tries to process the event, the event's name will be added to the database with the lockable state, so if any planners try to edit the event, we will firstly check if its locked by the admin, if so the planners will get the wrong message and prevent them from doing the edit.

Design rationale: By using an exclusive read lock, we give Admin the highest authority. This design is in line with the design of our system. Although we sacrifice the liveness of the system, we ensure the correctness and consistency of the data :

1.Ensure data correctness: When an administrator is approving an activity, an exclusive read lock ensures that planners cannot modify the activity at the same time, ensuring that the data the administrator sees and is based on is consistent with the data stored in the database.

2. Avoid data obfuscation: Not using locks could result in planner edits being overwritten after an administrator approves or denies an activity, or administrators could make decisions based on outdated or changing information. Exclusive read locks prevent this confusion.

3. Process simplification: Locks provide a clear mechanism to determine who can edit an activity at a specific moment. This streamlines workflow and reduces confusion caused by multiple people editing at the same time.

4.5 Use case 5: Planner create event

Planner needs to select a venue and time when creating an event. The same venue can only allow one event to exist within a period of time.

4.5.1 Concurrency issue 5.1

Description : When multiple Event Planners selected venues, they tried to select the same venue and the time periods overlapped simultaneously.

Sequence Diagram:

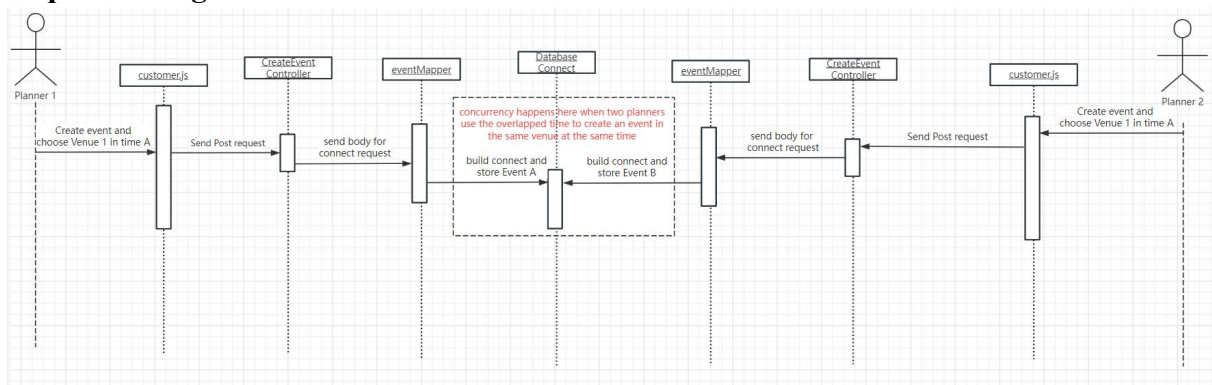


Figure9 Sequence diagram of Concurrency issue 5.1

Proposed Solution Pattern and Description: We use **Read/Write lock** to solve the issue. Before we add the lock, if multiple event planners select a venue at the same time and they try to select the same venue and time slot it may result in the venue being "double booked". This will not only cause trouble for the venue, but may also cause planners' plans to be disrupted. In addition, since multiple requests are written to the database almost simultaneously, it may cause inconsistency in the venue reservation information in the database. For example, two planners may both believe they have successfully booked a venue, but in the database, only one planner's reservation is recorded.

The `CreateEventController` uses a concurrency control mechanism known as a `ReentrantReadWriteLock`, specifically designed to handle scenarios where many threads need to read data simultaneously, but only a single thread should modify the data at a given time.

1. The method `eventLock.readLock().lock();` acquires the read lock. This allows multiple threads to check if a time slot is available concurrently.
2. If a time slot is available, the read lock is released with `eventLock.readLock().unlock();`.
3. Immediately after that, the write lock is acquired with `eventLock.writeLock().lock();` to ensure exclusive access. This means that no other thread can write (or modify) the data at the same time.
4. The time slot availability is checked again as a safety measure, ensuring that the status hasn't changed between releasing the read lock and acquiring the write lock.
5. If the time slot is still available, the event is created and registered.
6. The write lock is then released, ensuring other threads can now access the data.

Design rationale:

The use of `ReentrantReadWriteLock` in this context effectively addresses potential concurrency issues related to simultaneous event creation. Here's the rationale behind its effectiveness:

1. **Separation of Reading and Writing:** By differentiating between read and write locks, this approach ensures that multiple threads can simultaneously read (check for availability) without blocking each other. This boosts the performance as checking the availability is likely a frequent operation.
2. **Exclusive Writes:** The write lock ensures that only one thread can create an event (or modify data) at any given time. This ensures that once a thread determines a time slot is available, it can safely proceed to create an event without worrying about other threads creating conflicting events.
3. **Safety Check Post Locking:** The second check for time slot availability after acquiring the write lock ensures that the scenario where the time slot becomes unavailable between releasing the read lock and acquiring the write lock is handled. This double-checking pattern provides an extra layer of safety.
4. **Minimizing Lock Contention:** By releasing the read lock before acquiring the write lock, the system ensures that the period where no other operations (read or write) can occur is as short as possible, reducing potential contention and improving overall throughput.

In summary, the ReentrantReadWriteLock, combined with the double-checking pattern, ensures that simultaneous event creation by multiple planners does not result in conflicting events, thus effectively addressing the concurrency challenge.

4.6 Use case 6: Admin delete user

In order to maintain system stability and user experience or help users delete their accounts, the admin can delete some inactive user accounts. Admin cannot delete users who have purchased tickets or planners who have created events. However, the deletion process may affect the normal use of users.

4.6.1 Concurrency issue 6.1

Description : When the user wants to login the system, his account is deleted by an admin at the same time.

Sequence Diagram:

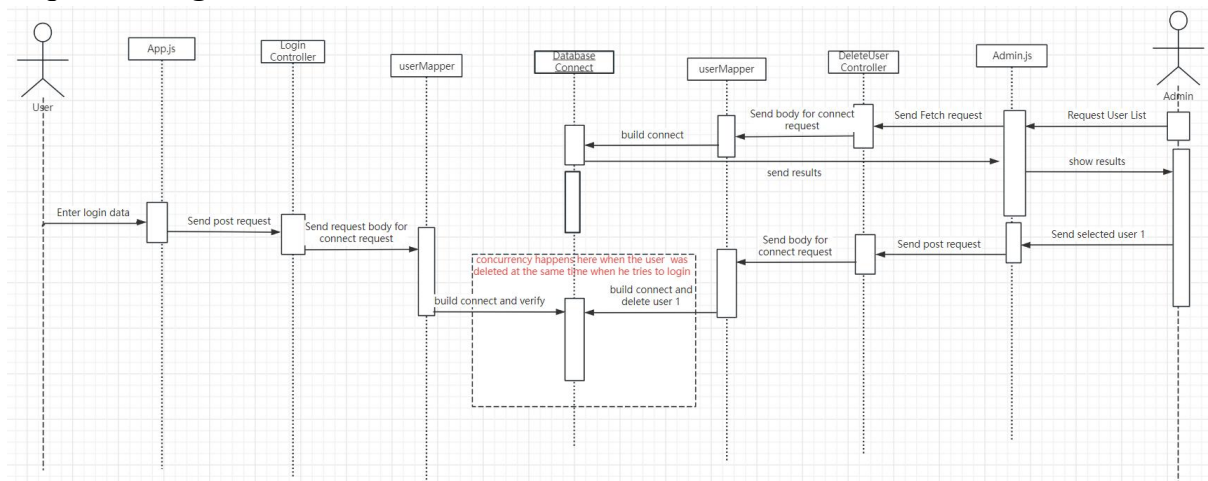


Figure10 Sequence diagram of Concurrency issue 6.1

Proposed Solution Pattern and Description: We use **Exclusive Read lock** to solve the issue. Before we add the lock, if the user logs in at the same time when he is deleted by the admin, he can still get into the next page outside the authority until do some operations. So we add a table to manage the lockable status of users in our database to give the Admin the highest priority. We add the lock function for the Admin, before the admin does any operations on someone, he will acquire the lock, add the username and lockstatus in the database then that user can not do anything until the Admin finishes the operation and releases the lock. By doing this, when the user tries to login, we will firstly check if there's a lock on that username, if so the user will get the response "user information changed!" and fail to login.

Design rationale: We use an exclusive read lock to give Admin the highest authority. By giving Admin the highest authority, the security of the database is effectively ensured and

deleted users are prevented from inserting data by mistake. This design is in line with the design of our system :

1.Ensure data consistency: Using an exclusive read lock can ensure that users will not log in to the system concurrently when the administrator performs a deletion operation. This way, when an administrator deletes a user, the system does not create data inconsistencies caused by users trying to log in.

2.Avoid system errors: Without locks, users may still be logged in and try to perform some operations after an administrator deletes their account, which may cause database or system errors. Exclusive read locks can effectively prevent this situation from happening.

3.Enhanced user experience: If users try to log in while being deleted, they may experience some unexpected errors or system behavior. However, by using a lock, the user receives a clear response message telling them "User information has changed!", which is much friendlier than having the user face unexpected errors or behavior.

4.Enhanced data security: The exclusive read lock ensures that when administrators perform sensitive operations such as deleting user accounts, no other concurrent operations will affect the process, thereby reducing the risk of data leakage or misoperation.

4.6.2 Concurrency issue 6.2

Description : The user's account is deleted by the Admin while the user is purchasing tickets simultaneously.

Sequence Diagram:

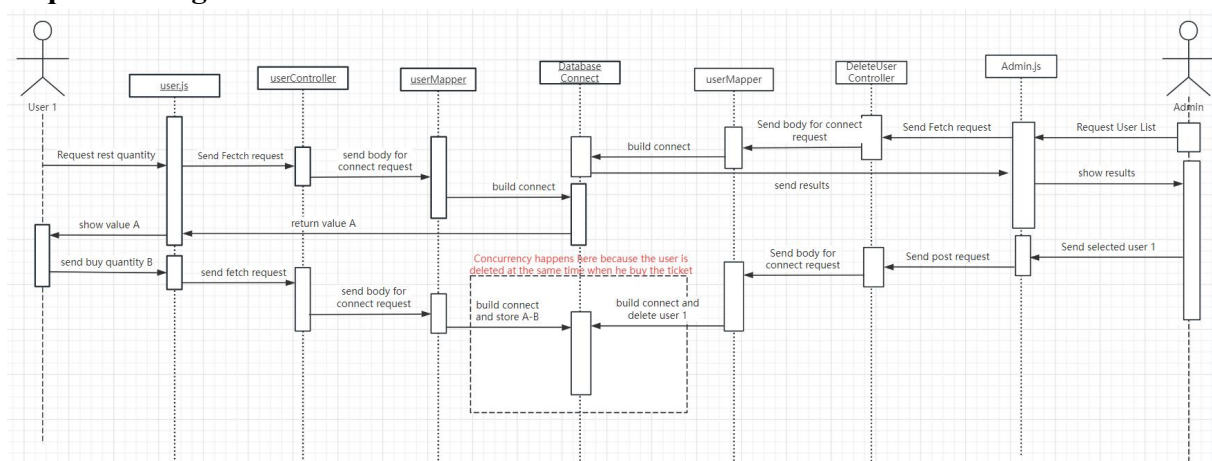


Figure11 Sequence diagram of Concurrency issue 6.2

Proposed Solution Pattern and Description: Same with issue 6.1, We use **Exclusive Read lock** to solve the issue. Before we add the lock, if the user buys tickets at the same time when he is deleted by the admin, he can still save the reservation information outside the authority. So we add a table to manage the lockable status of users in our database to give the Admin the

highest priority. We add the lock function for the Admin, when the admin does any operations on someone, he will acquire the lock, add the username and lockstatus in the database then that user can not do anything until the Admin finishes the operation and releases the lock. By doing this, when the user tries to buy tickets, we will firstly check if there's a lock, if so the user will get the response "user information changed!" and nothing will be added to the reservation database.

Design rationale: Same with issue 6.1, we use an exclusive read lock to give Admin the highest authority. By giving Admin the highest authority, the security of the database is effectively ensured and deleted users are prevented from inserting data by mistake. This design is in line with the design of our system :

1.Ensure data consistency: Using an exclusive read lock can ensure that users will not log in to the system concurrently when the administrator performs a deletion operation. This way, when an administrator deletes a user, the system does not create data inconsistencies caused by users trying to log in.

2.Avoid system errors: Without locks, users may still be logged in and try to perform some operations after an administrator deletes their account, which may cause database or system errors. Exclusive read locks can effectively prevent this situation from happening.

3.Enhanced user experience: If users try to log in while being deleted, they may experience some unexpected errors or system behavior. However, by using a lock, the user receives a clear response message telling them "User information has changed!", which is much friendlier than having the user face unexpected errors or behavior.

4.Enhanced data security: The exclusive read lock ensures that when administrators perform sensitive operations such as deleting user accounts, no other concurrent operations will affect the process, thereby reducing the risk of data leakage or misoperation.

5. Test Strategy

Concurrency testing is an important part of ensuring software performance and reliability, especially for web applications and services that are accessed by multiple users simultaneously. To create a sound and robust concurrency testing strategy, our group considered manual testing using tools like JMeter. All test cases are two or more concurrent requests. The specific test cases and test methods can be viewed below.

5.1 Why Jmeter

JMeter is a popular open source tool for load testing and performance testing. It is very efficient for simulating concurrent users and can generate valuable data quickly.

Create JMeter test plans to simulate different levels of concurrency by configuring the number of threads, startup time, and thread-specific requests. It is very easy to use and flexible.

You can use JMeter to measure and analyze various performance indicators under different concurrency levels, such as response time, return information, error rate and resource consumption (CPU, memory, etc.)

5.2 Manual Test

Due to system design factors and server factors, it is difficult to send messages synchronously during testing, so Jmeter is used for testing. In addition, because it involves the addition, deletion, modification and query of data, repeated operations cannot be performed on the same piece of data. Before each operation, you need to ensure that there is relevant data in the database that can be requested, so manual testing is used.

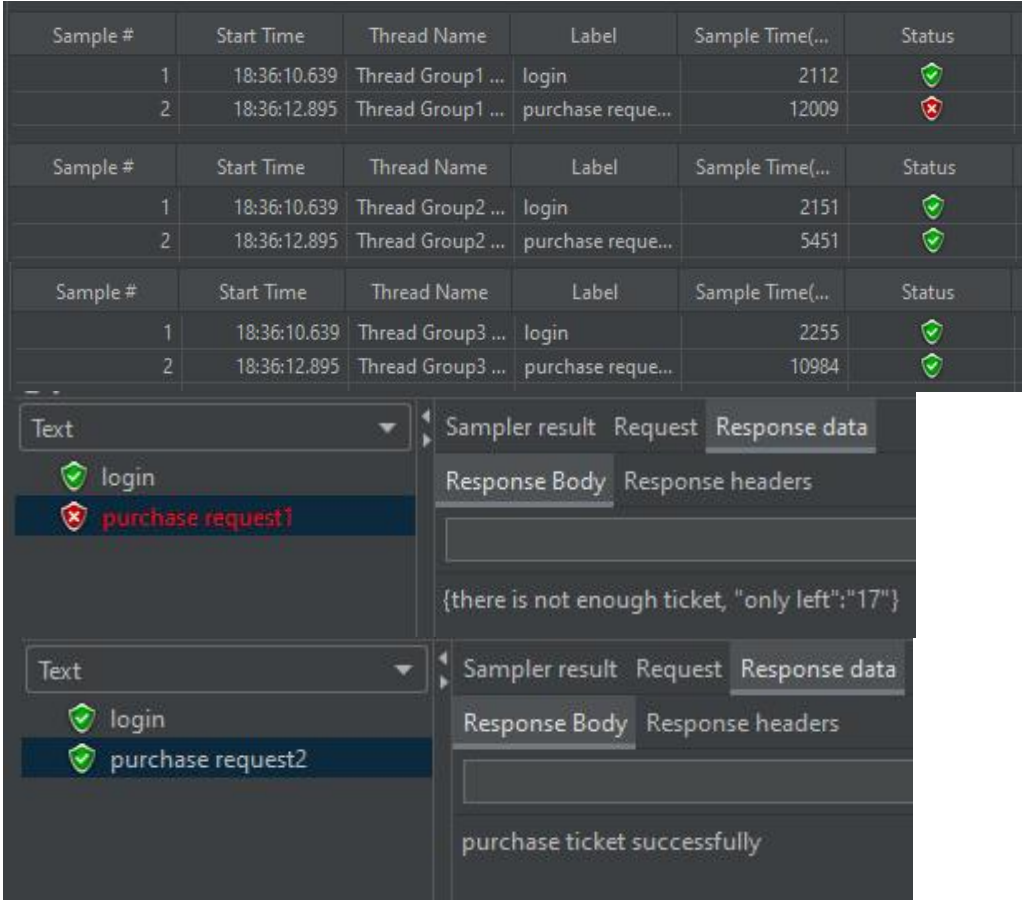
5.3 Test Cases

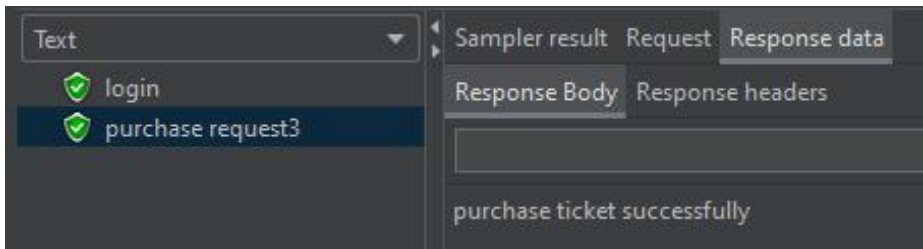
Use Case	Issue	Test Case
Use case 1: User buy tickets	1.1 Multiple users purchase the same type of tickets at the same time. The quantity purchased by each user is less than the remaining	Send 3 requests which 2 users' purchase accounts are enough but the last one's purchase amount is not enough.

	quantity, but the total quantity purchased by these users exceeds the remaining quantity.	
Use case 2: User register	2.1 When two or more users register with the same username at the same time, concurrency problems will occur because the same username is not allowed.	Send 2 user registers with the same user name at the same time.
Use case 3: Planners Edit Event	3.1 When two or more event planners edit the same event simultaneously, the consistency of the data will be compromised.	Two event planners send two requests about editing the same event simultaneously, and only one of them can edit it successfully.
Use case 4: Admin manages events	4.1 When a planner has modified the content in an event or is modifying the event, and the admin approves or rejects the event simultaneously.	Admin approves the event while an event planner is editing the event.
		Admin rejects(deletes) the event while an event planner is editing the event.
Use case 5: Planner creates event	5.1 When multiple Event Planners selected venues, they tried to select the same venue, and the time periods overlapped simultaneously.	Multiple planners create their events at the same venue and overlapped time periods.
Use case 6: Admin delete user	6.1 When the user wants to login the system, his account is deleted by an admin at the same time.	Send the user login and delete the user who is logging requests at the same time.
Use case 6: Admin delete user	6.2 The user's account is deleted by the Admin while the user is purchasing tickets simultaneously.	Send the user purchase ticket and delete the user who is purchasing tickets at the same time.

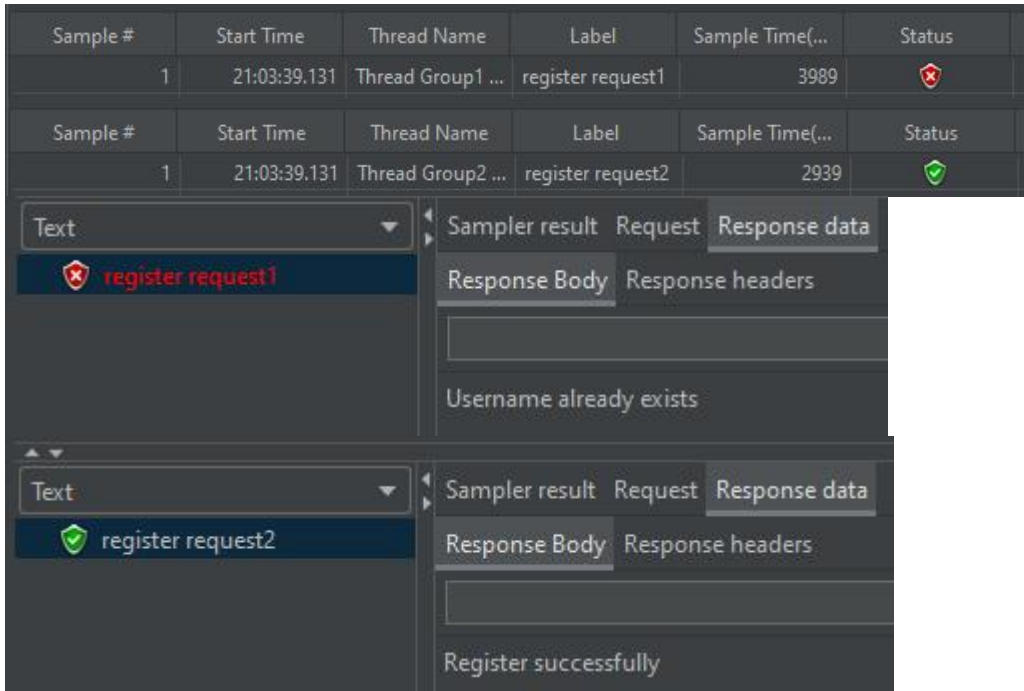
5.4 Test Outcomes

5.4.1 1.1 issue & 1.1.1 issue

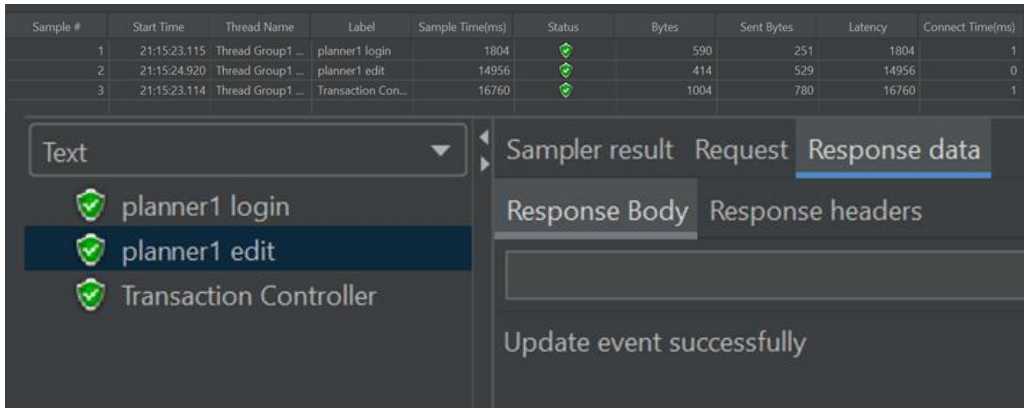
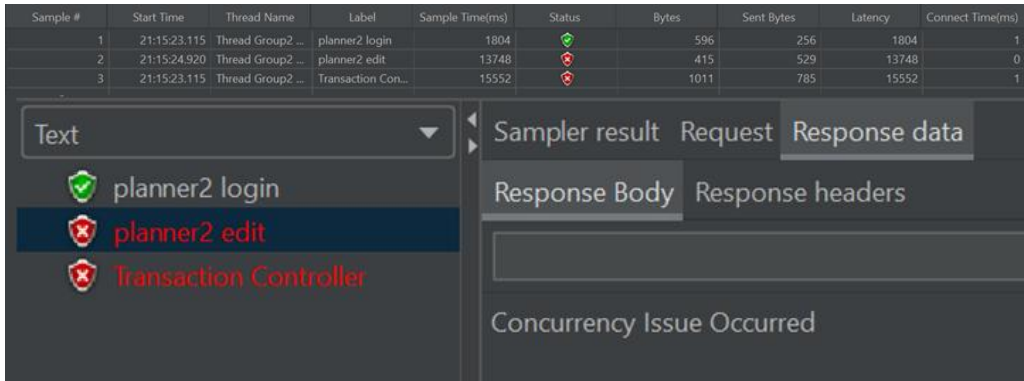
Pre-condition	Three users are trying to purchase tickets at the same time. Database contains the users information and ticket information(not enough for them to purchase).
Expected outcome	Only some of them will succeed in purchasing tickets because there are not enough tickets.
Detail test case	Three different users purchase the same ticket. 3*number will be not enough but 2*number are enough for purchase.
Test file	1.1.jmx
Result	Work as expected
Note	 <p>The screenshot displays JMeter results for three thread groups. Thread Group 1 shows a failed 'purchase request' (red X icon). Thread Group 2 and Thread Group 3 show successful 'purchase request' (green checkmark icons). Below the table, the 'Response data' for the failed request shows an error message: '{there is not enough ticket, "only left":"17"}'. The 'Response data' for the successful requests shows 'purchase ticket successfully'.</p>

	
--	--

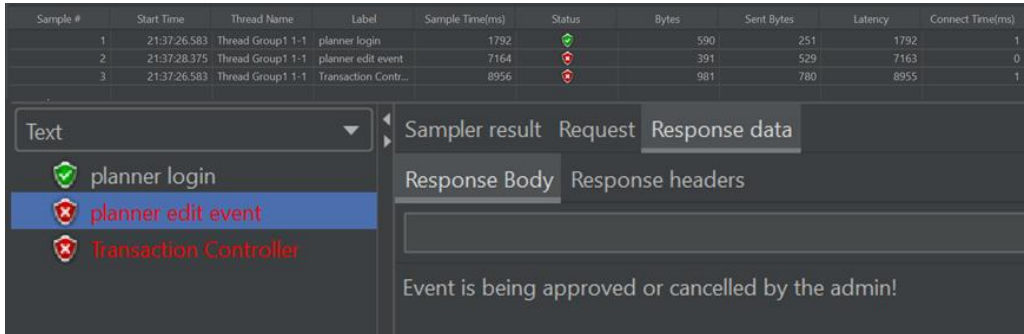
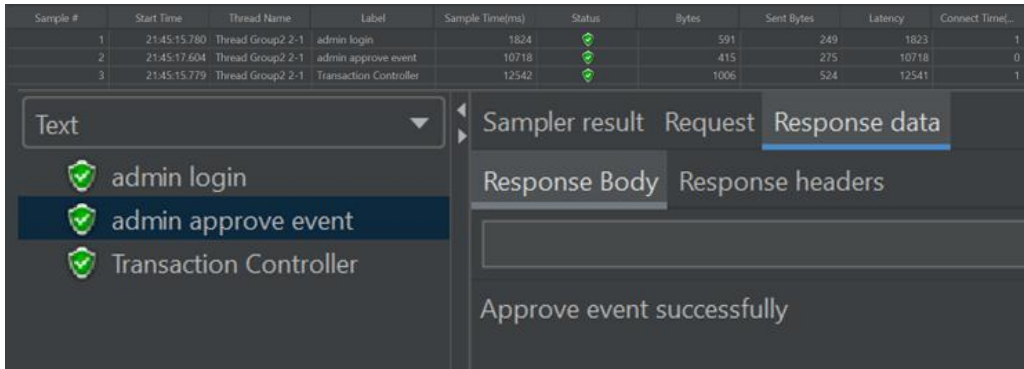
5.4.2 2.1 issue

Pre-condition	Two users want to register two accounts with the same user name but different information (like password). The user's name had not been registered.
Expected outcome	One user will succeed but the other will fail
Detail test case	Send 2 request with same user name but different information at the same time
Test file	2.1.jmx
Result	Work as expected
Note	

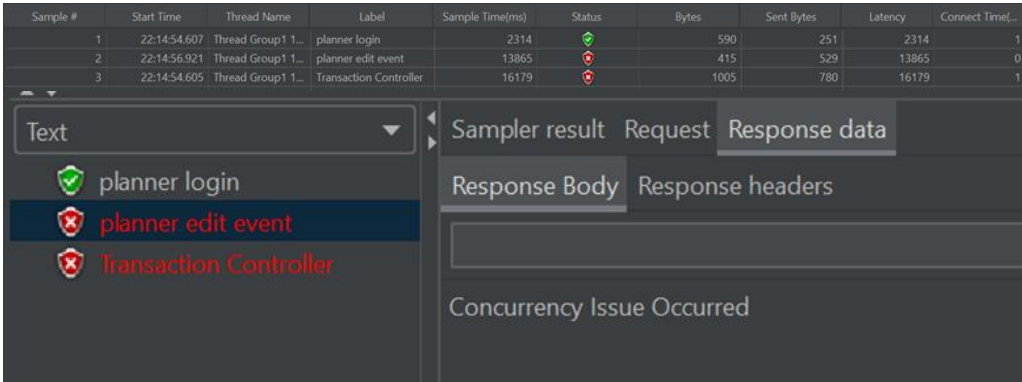
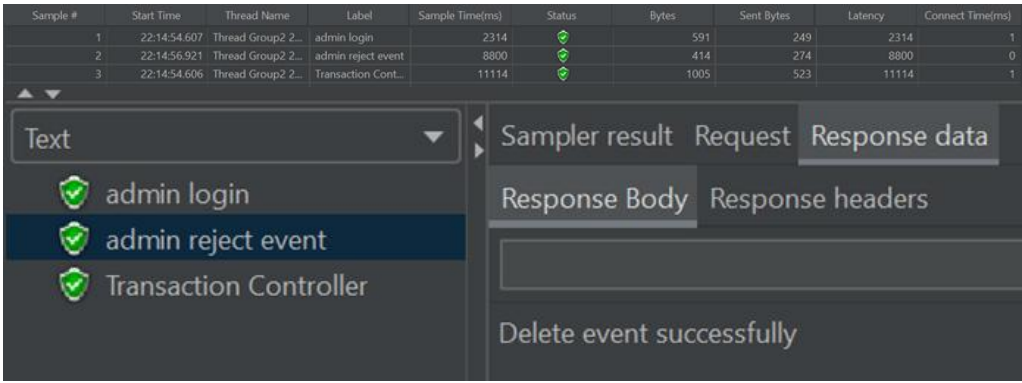
5.4.3 3.1 issue

Pre-condition	Two or more planners may modify the same event at the same time, which may cause conflicts in modified content and failure to save data.
Expected outcome	Only one planner can edit the event successfully, on the other hand, another planner will fail to edit and receive the notice that the other planner is currently editing the event.
Detail test case	Planner1 tries to edit the start time of “event011” to “2024-03-07”, while planner2 is trying to edit the start time of “event111” to “2024-03-08”.
Test file	3.1.jmx
Result	Work as expected
Note	 <p>Planner1 edited the event successfully.</p>  <p>Planner2 failed to edit the event.</p>

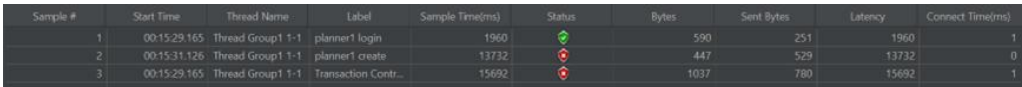
5.4.4 4.1 issue

Pre-condition	When a planner has modified the content in an event or is modifying the event, and the Admin approves the event simultaneously.
Expected outcome	Admin's operation will be permitted and the modification by the planner will be rejected.
Detail test case	Admin approves "event014", at the same time, the planner tries to edit "event014".
Test file	4.1.1.jmx
Result	Work as expected
Note	 <p>Planner failed to edit the event.</p>  <p>Admin approved the event successfully.</p>

Pre-condition	When a planner has modified the content in an event or is modifying the event, and the Admin rejects the event simultaneously.
Expected outcome	Admin's operation will be permitted and the modification by the planner will be rejected.
Detail test case	Admin deletes "eventDelete01", at the same time, the planner tries to edit "eventDelete01".
Test file	4.1.2.jmx

Result	Work as expected
Note	 <p>Planner failed to edit the event.</p>  <p>Admin deleted the event successfully.</p>

5.4.5 5.1 Issue

Pre-condition	When 2 Event Planners selected venues, they tried to select the same venue, and the time periods overlapped simultaneously.
Expected outcome	Only one planner can create the event, while another planner will be rejected by the system.
Detail test case	Planner1 creates “event020” at “carlton venue” from 2024-02-14T07:00:00 to 2024-02-14T08:00:00, at the same time, planner2 creates “event021” at “carlton venue” from 2024-02-14T07:00:00 to 2024-02-14T08:00:00(overlapped period).
Test file	5.1.jmx
Result	Work as expected
Note	

Text

- planner1 login
- planner1 create
- Transaction Controller

Sampler result
Request
Response data

Response Body
Response headers

Concurrency issue happens! The time slot is not available!

Planner1 failed to create the event.

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	00:15:29.165	Thread Group2 2-1	planner2 login	1784	✓	596	256	1784	1
2	00:15:30.950	Thread Group2 2-1	planner2 create	14983	✓	414	529	14983	0
3	00:15:29.165	Thread Group2 2-1	Transaction Contr...	16767	✓	1010	785	16767	1

Text

- planner2 login
- planner2 create
- Transaction Controller

Sampler result
Request
Response data

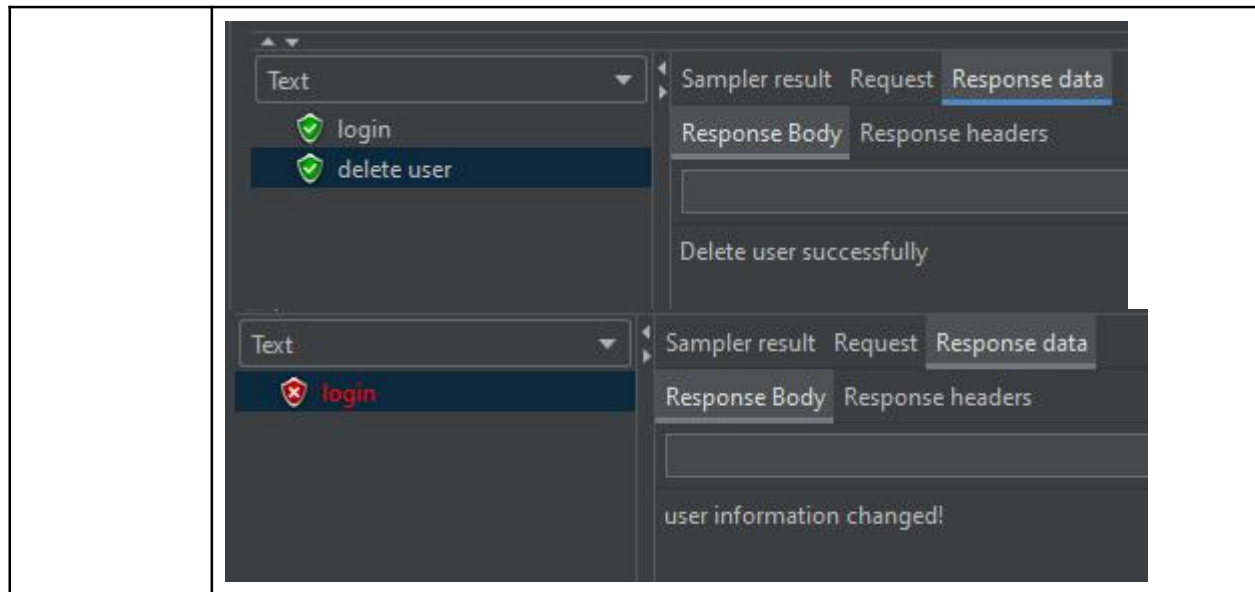
Response Body
Response headers

Create event successfully

Planner2 created the event successfully.

5.4.6 6.1 issue

Pre-condition	The user is trying to login and the admin is trying to delete him. Database contains the user information. Admin has already logged in.																																																
Expected outcome	The admin succeeds and the user can not login.																																																
Detail test case	Admin login first, then select one user to send login information and delete the user with user information.																																																
Test file	6.1.jmx																																																
Result	Work as expected																																																
Note	<table><tr><td>Sample #</td><td>Start Time</td><td>Thread Name</td><td>Label</td><td>Sample Time(...</td><td>Status</td><td></td></tr><tr><td>1</td><td>16:36:21.739</td><td>Thread Group1 ...</td><td>login</td><td>2121</td><td></td><td></td></tr><tr><td>2</td><td>16:36:23.861</td><td>Thread Group1 ...</td><td>delete user</td><td>6364</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Sample #</td><td>Start Time</td><td>Thread Name</td><td>Label</td><td>Sample Time(...</td><td>Status</td><td></td></tr><tr><td>1</td><td>16:36:23.893</td><td>Thread Group2 ...</td><td>login</td><td>2135</td><td></td><td></td></tr></table>							Sample #	Start Time	Thread Name	Label	Sample Time(...	Status		1	16:36:21.739	Thread Group1 ...	login	2121			2	16:36:23.861	Thread Group1 ...	delete user	6364										Sample #	Start Time	Thread Name	Label	Sample Time(...	Status		1	16:36:23.893	Thread Group2 ...	login	2135		
	Sample #	Start Time	Thread Name	Label	Sample Time(...	Status																																											
	1	16:36:21.739	Thread Group1 ...	login	2121																																												
	2	16:36:23.861	Thread Group1 ...	delete user	6364																																												
	Sample #	Start Time	Thread Name	Label	Sample Time(...	Status																																											
1	16:36:23.893	Thread Group2 ...	login	2135																																													



5.4.8 6.2 issue

Pre-condition	The user is trying to purchase tickets and the admin is trying to delete him. Database contains the user information and ticket information. They both login in advance.																							
Expected outcome	The admin succeeds and the user can not purchase tickets.																							
Detail test case	Login first and select one user and one kind of ticket to send a purchase request and delete the user with user information.																							
Test file	6.2.jmx																							
Result	Work as expected																							
Note	<table><tr><td>Sample #</td><td>Start Time</td><td>Thread Name</td><td>Label</td><td>Sample Time(...)</td><td>Status</td></tr><tr><td>1</td><td>17:33:24.488</td><td>Thread Group1 ...</td><td>login</td><td>2656</td><td></td></tr><tr><td>2</td><td>17:33:27.152</td><td>Thread Group1 ...</td><td>delete user</td><td>6409</td><td></td></tr></table>						Sample #	Start Time	Thread Name	Label	Sample Time(...)	Status	1	17:33:24.488	Thread Group1 ...	login	2656		2	17:33:27.152	Thread Group1 ...	delete user	6409	
	Sample #	Start Time	Thread Name	Label	Sample Time(...)	Status																		
	1	17:33:24.488	Thread Group1 ...	login	2656																			
	2	17:33:27.152	Thread Group1 ...	delete user	6409																			
	<table><tr><td>Sample #</td><td>Start Time</td><td>Thread Name</td><td>Label</td><td>Sample Time(...)</td><td>Status</td></tr><tr><td>1</td><td>17:33:24.488</td><td>Thread Group2 ...</td><td>login</td><td>2664</td><td></td></tr><tr><td>2</td><td>17:33:27.152</td><td>Thread Group2 ...</td><td>purchase ticket</td><td>2061</td><td></td></tr></table>						Sample #	Start Time	Thread Name	Label	Sample Time(...)	Status	1	17:33:24.488	Thread Group2 ...	login	2664		2	17:33:27.152	Thread Group2 ...	purchase ticket	2061	
	Sample #	Start Time	Thread Name	Label	Sample Time(...)	Status																		
	1	17:33:24.488	Thread Group2 ...	login	2664																			
2	17:33:27.152	Thread Group2 ...	purchase ticket	2061																				

	<div><div><div>Text</div><div><div>login</div><div>delete user</div></div></div><div><div>Text</div><div><div>login</div><div>purchase ticket</div></div></div></div> <div><div><div>Sampler resultRequestResponse data</div><div>Response BodyResponse headers</div><div></div><div>Delete user successfully</div></div><div><div><div>Sampler resultRequestResponse data</div><div>Response BodyResponse headers</div><div></div><div>user information changed!</div></div></div></div>
--	---