



SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Testing Document

SWEN90007

Music Events System

Team: GGBond

In Charge of

Lijing Bi (1369370)

Jie Zhou (1442449)

Baorui Chen (1320469)

Liuming Teng (1292608)



SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
21/9/2022	03.00-D01	Initial draft	Linjing Bi Jie Zhou Baorui Chen Liuming Teng
27/9/2022	03.00-D02	Added draft concurrency issues	Linjing Bi
28/9/2022	03.00-D03	Adjusted draft concurrency issues	Jie Zhou
05/10/2022	03.00-D04	Added draft of concurrency patterns	Baorui Chen
09/10/2022	03.00-D05	Draft of pattern implementations	Jie Zhou Baorui Chen
11/10/2022	03.00-D06	Refined concurrency patterns	Linjing Bi
12/10/2022	03.00-D07	Refined issue descriptions	Linjing Bi Jie Zhou Baorui Chen
13/10/2022	03.00-D08	Added test results	Linjing Bi Liuming Teng
14/10/2022	03.00-D09	Added class diagram	Linjing Bi
15/10/2022	03.00-D10	Added sequence diagrams	Jie Zhou Baorui Chen
17/10/2022	03.00-D11	Finalised test results	Linjing Bi Jie Zhou Baorui Chen Liuming Teng
17/10/2022	03.00-D12	Added appendix	Linjing Bi Jie Zhou Baorui Chen Liuming Teng
17/10/2022	03.00-D13	Finalised document content	Linjing Bi Jie Zhou Baorui Chen Liuming Teng

Contents

1. Introduction	4
1.1 Proposal	4
1.2 Target Users	4
1.3 Conventions, terms and abbreviations	4
2. Class Diagram	5
2.1 Component Diagram	5
2.2 Class Diagram	5
3. Covered Requirements	10
3.1 Functional or Product Requirements	10
4. Concurrency Issues	10
4.1 Use case 18 Modify Event	10
4.1.1 Issue 18.1:Concurrency Conflict on Ticket Pricing	10
4.2 Use case 23 Booking tickets	10
4.2.1 Issue 23.1:Duplicate Ticket Orders	11
4.2.2 Issue 23.2:Overbooking Due to Concurrent Purchases	13
5. Implemented concurrency patterns	15
5.1 Pessimistic Lock	15
5.1.1 Lock and Release	17
5.1.2 Lock Timeout Problem	17
5.1.3 False Lock Removal Problem	17
5.1.4 Synchronized Problem	17
5.2 Optimistic Lock	18
6. Functional Test Cases	19
7. Entry Data	29

1. Introduction

1.1 Proposal

The purpose of this document is to define and present the test cases for project Music Event System and GGBond, covering the test cases for the system use cases.

1.2 Target Users

This document is mainly designed for those responsible for executing the test cases in this project [team members and SWEN90007 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 described in the following table, presented in alphabetical order.

Term	Description
Redis	Redis is an open-source, in-memory data storage system that is highly suitable for handling high concurrency and low-latency applications.
Booking Confirmation	A digital or electronic receipt provided to the user upon successful reservation of tickets for a music event.

2. Class Diagram

Given the intricacy of system-wide class diagrams, we've segmented the entire system into several class diagrams, each organized by package. This approach is designed to methodically present the relationships between classes and layers within the Music Event System.

2.1 Component Diagram

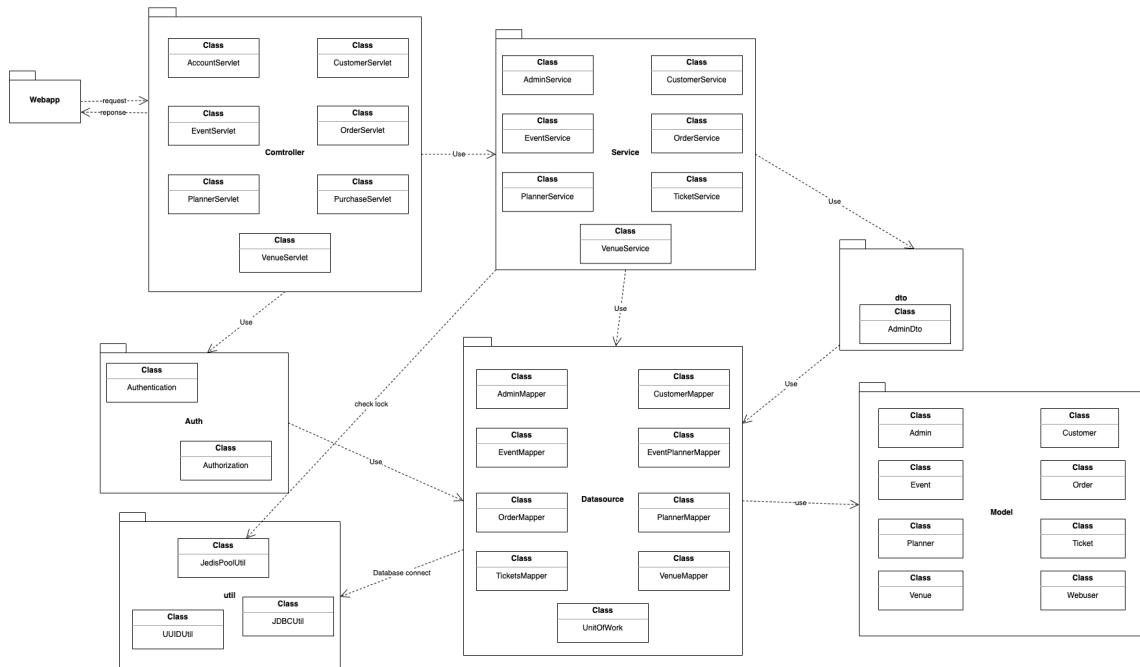


figure 1: High-level class diagram of the packages

2.2 Class Diagram

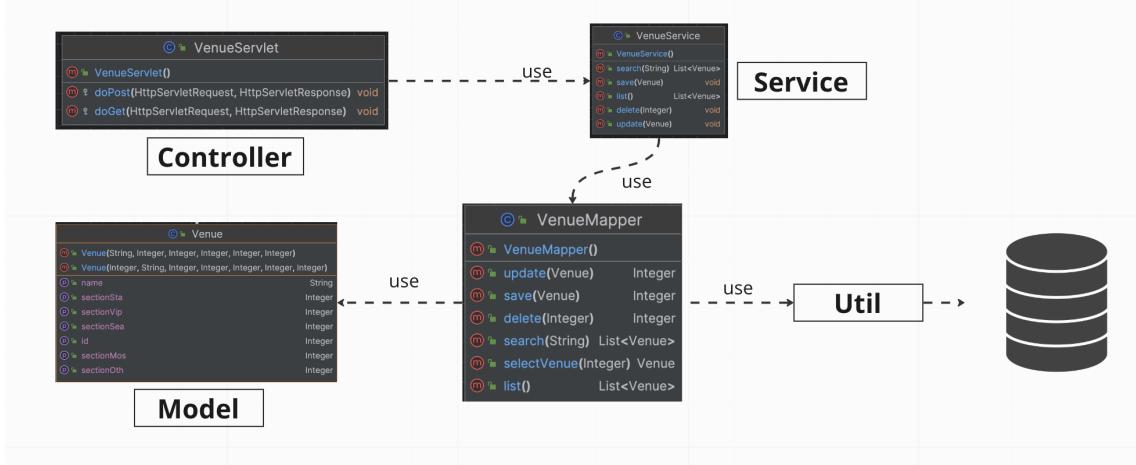


figure 2 :Main association between these class diagrams

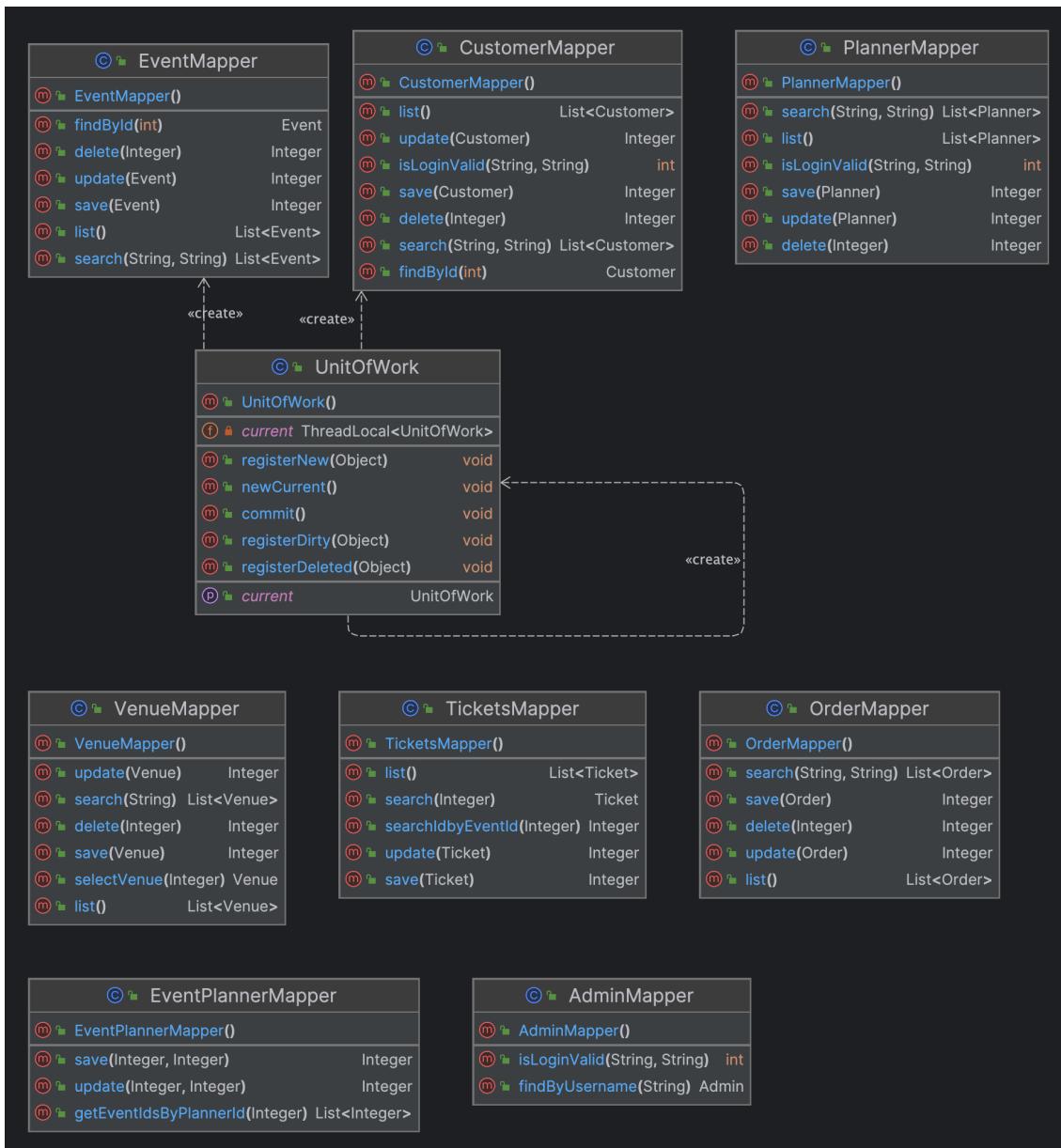


Figure 3 : Class diagram for **datasource** package

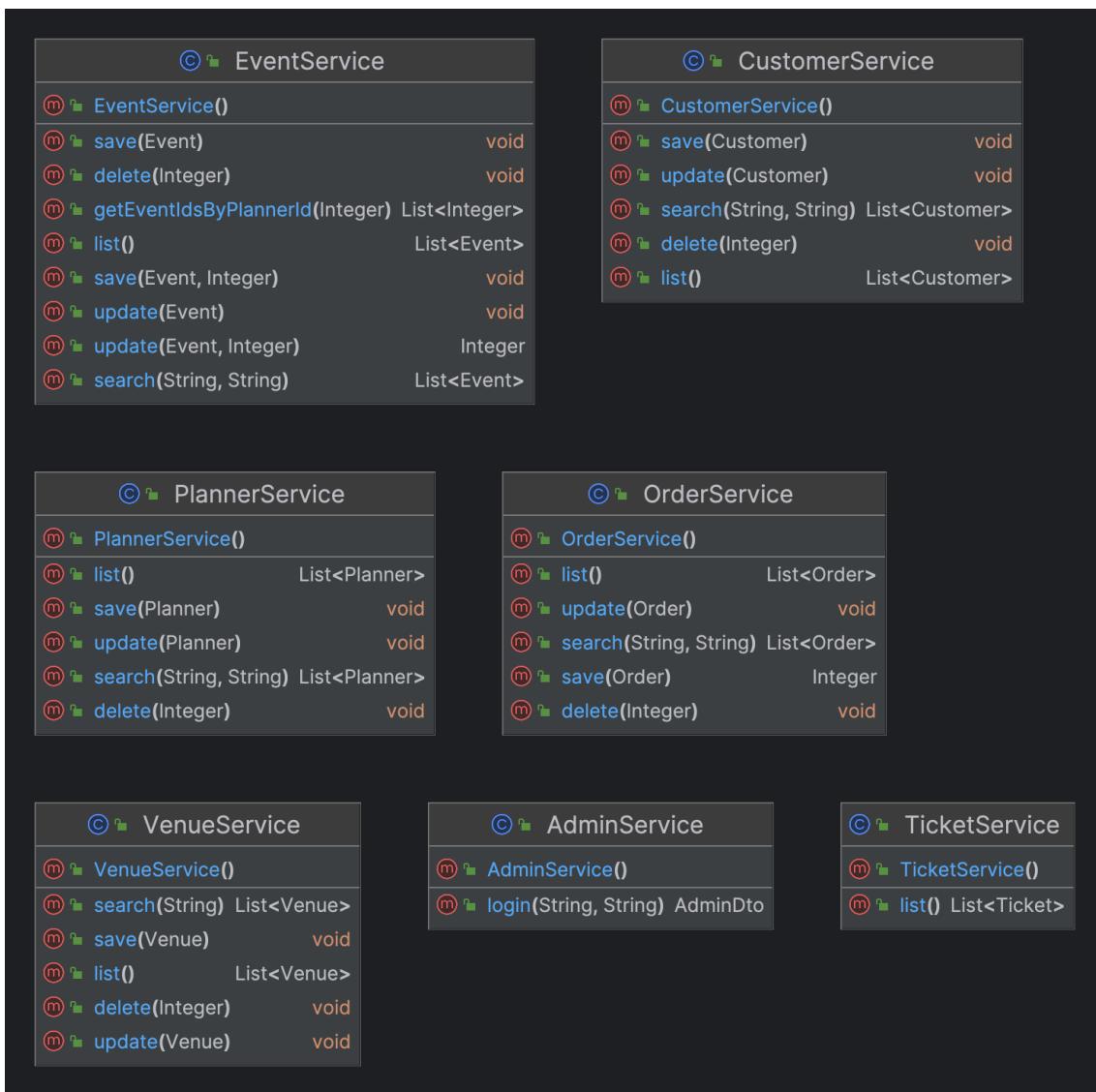


Figure 4 : Class diagram for **service** package

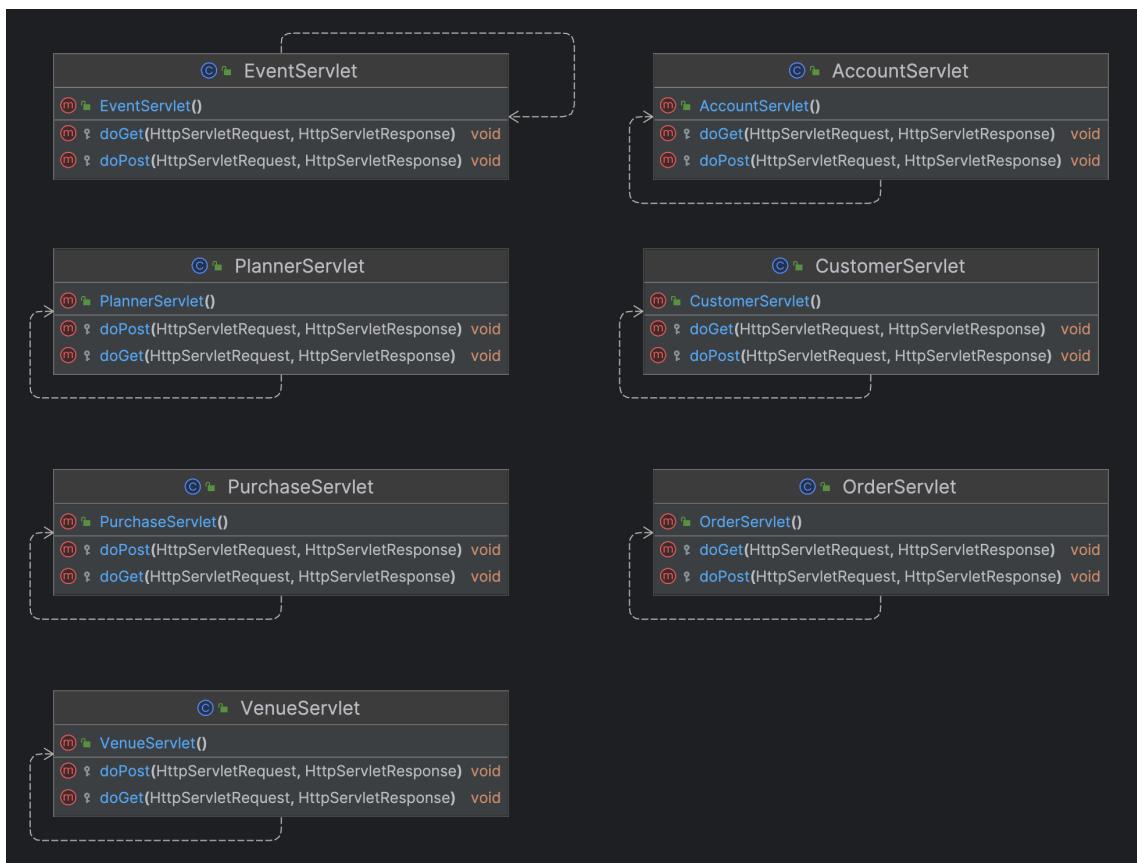


Figure 5 : Class diagram for **controller** package

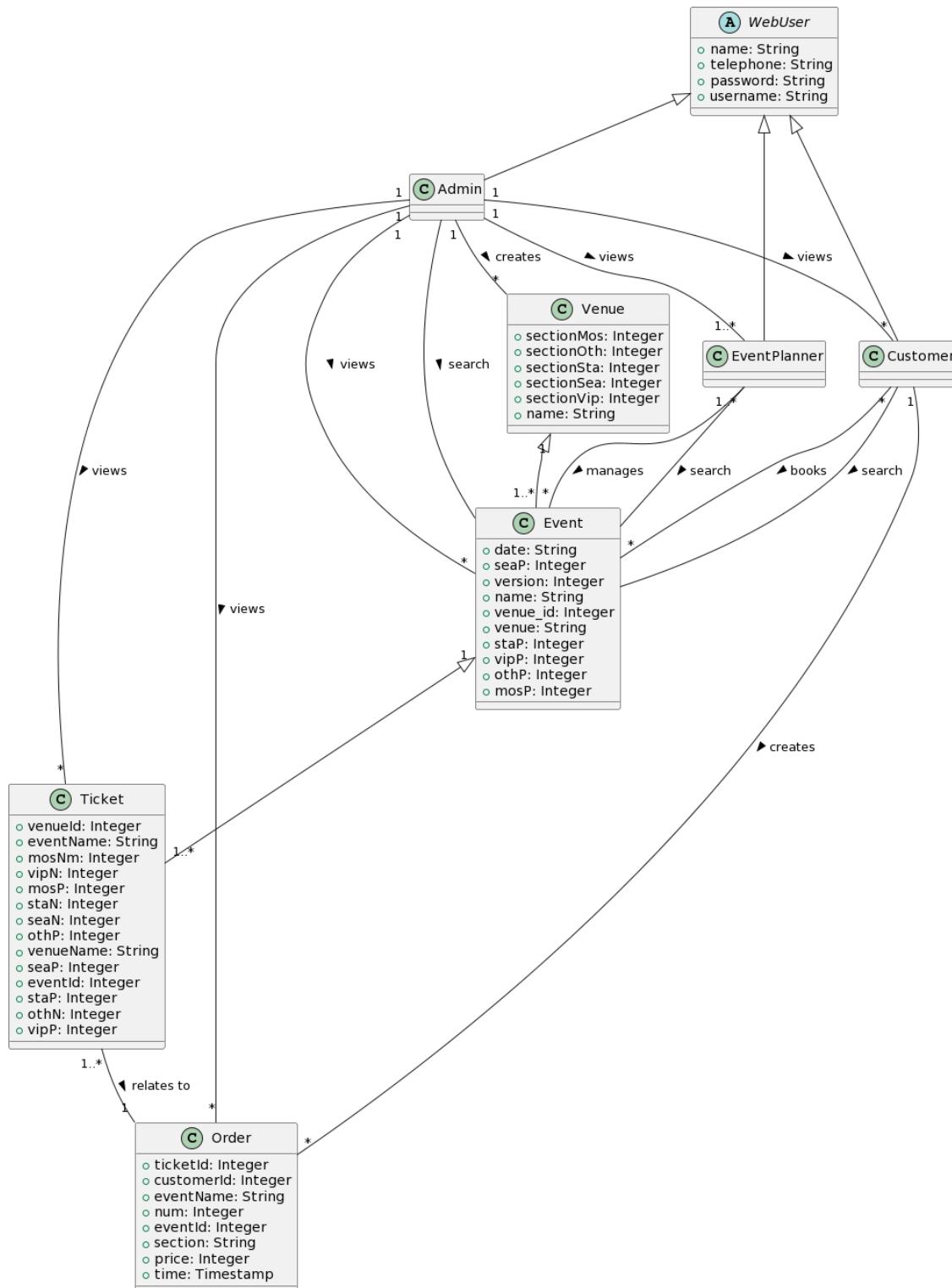


Figure 6 : Class diagram for **Domain Model** package

3. Covered Requirements

This section lists the system requirements covered in the test cases.

3.1 Functional or Product Requirements

Requirement Identifier	Requirement Name
UC018	Modify Event
UC024	Book Tickets

4. Concurrency Issues

4.1 Use case 18 Modify Event

4.1.1 Issue 18.1 :Concurrency Conflict on Ticket Pricing

Description: When several Event Planners simultaneously attempt to adjust the ticket prices of the same event, there's a risk of causing pricing overwrites or introducing pricing inconsistencies.

Proposed Solution Pattern: Implement an Optimistic Locking strategy paired with a Real-time Notification system specifically for ticket pricing changes that come into conflict.

Rationale: Utilizing Optimistic Locking, the system will allow concurrent ticket pricing adjustments but will scrutinize for conflicts prior to confirming those changes. If one Event Planner's price modification conflicts with another's, the Real-time Notification system would promptly alert the involved Event Planners about the conflicting price adjustments. They can then collaboratively review, discuss, and decide on the most appropriate ticket price. This approach ensures that ticket pricing updates are meticulously coordinated, preventing unintended overwrites and maintaining the accuracy of event ticket pricing.

4.2 Use case 23 Booking tickets

Based on the project requirement analysis, the ticket booking feature can be accessed by a single customer, and also supports multiple customers accessing it simultaneously. Therefore, when multiple customers are booking tickets at the same time, concurrency issues may arise.

4.2.1 Issue 23.1 : Duplicate Ticket Orders

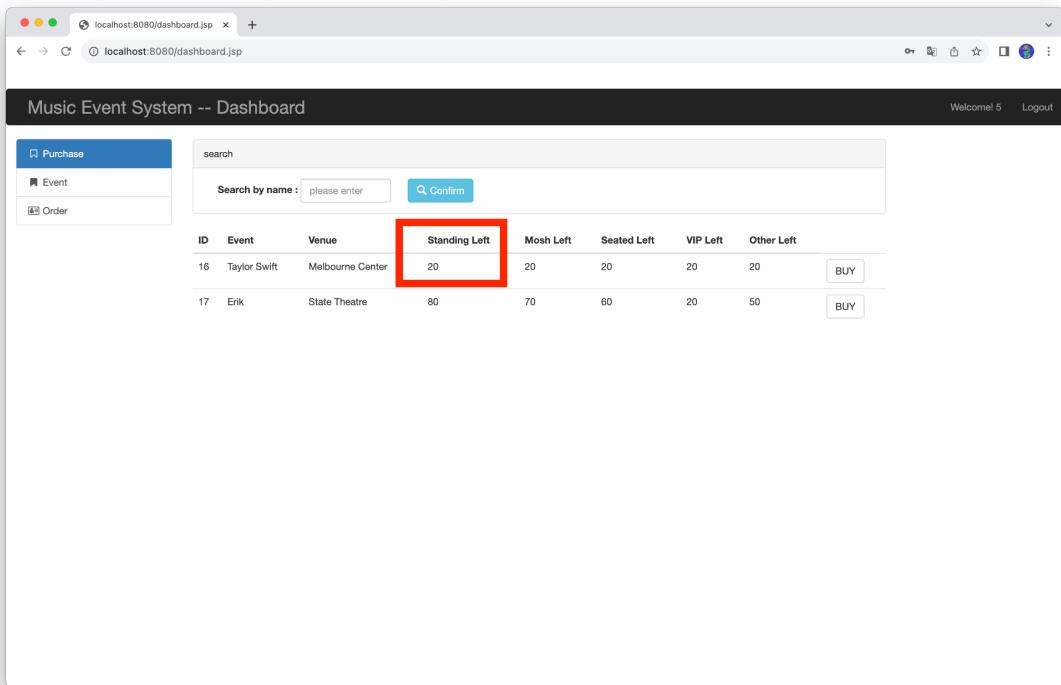
Description: When a customer is making a purchase, operational issues could lead to the same ticket order being placed multiple times, which is essentially the problem of duplicating orders.

Proposed Solution Pattern: pessimistic lock on the order object

Rationale: During the purchasing process, once a customer initiates a ticket booking request, the control handles the front-end request, and then calls the save function provided by the service to submit the newly created form to the database. However, if the user makes multiple clicks due to a delay, multiple forms would be created. If the form hasn't been fully submitted, then the duplicate forms will obtain the same data and submit it to the database, thereby causing errors. To address this issue, a pessimistic lock can be added when processing the order form to ensure the correctness of the ticket booking process.

Issues test:

Condition: Multiple users are ready to buy Taylor Swift standing tickets with 20 remaining in stock.



ID	Event	Venue	Standing Left	Mosh Left	Seated Left	VIP Left	Other Left	
16	Taylor Swift	Melbourne Center	20	20	20	20	20	<button>BUY</button>
17	Erik	State Theatre	80	70	60	20	50	<button>BUY</button>

Figure 7 : Inventory situation

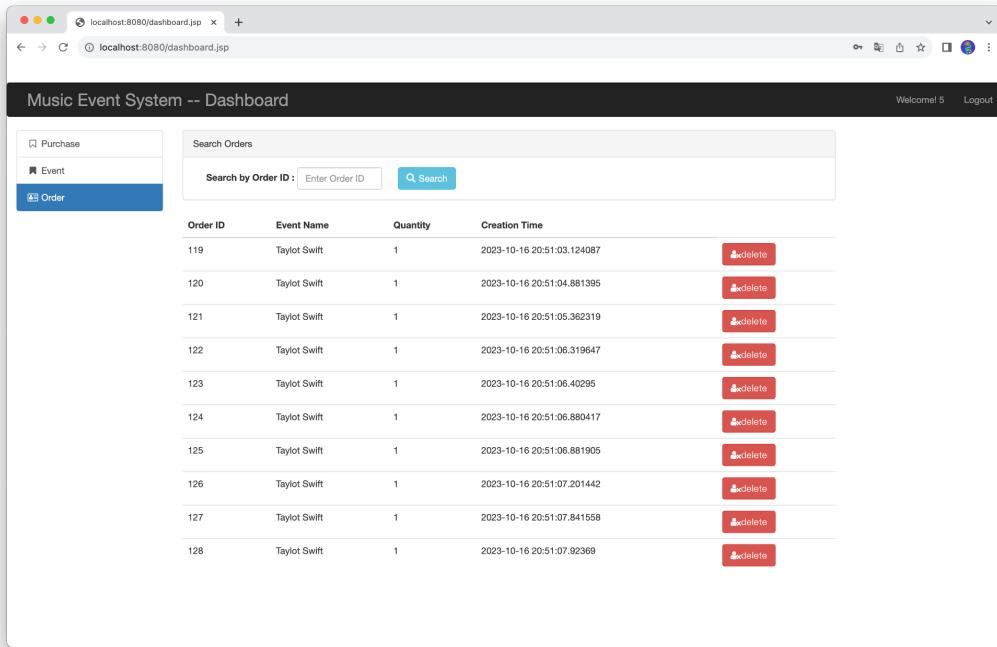
Test: Ten purchases occur in one second.

Thread Properties

Number of Threads (users):	<input type="text" value="10"/>
Ramp-up period (seconds):	<input type="text" value="1"/>
Loop Count:	<input checked="" type="checkbox"/> Infinite <input type="text" value="1"/>

Figure 8 : Concurrent request configuration

Outcome: Ten orders were created successfully.



The screenshot shows a web browser window for 'localhost:8080/dashboard.jsp'. The title bar says 'localhost:8080/dashboard.jsp'. The page header includes 'Welcome 5' and 'Logout'. On the left, there's a sidebar with 'Purchase', 'Event', and 'Order' buttons, where 'Order' is highlighted. The main content area is titled 'Search Orders' with a search bar labeled 'Search by Order ID: Enter Order ID' and a 'Search' button. Below is a table with 10 rows of order data:

Order ID	Event Name	Quantity	Creation Time	Action
119	Taylor Swift	1	2023-10-16 20:51:03.124087	
120	Taylor Swift	1	2023-10-16 20:51:04.881935	
121	Taylor Swift	1	2023-10-16 20:51:05.362319	
122	Taylor Swift	1	2023-10-16 20:51:06.319647	
123	Taylor Swift	1	2023-10-16 20:51:06.40295	
124	Taylor Swift	1	2023-10-16 20:51:06.880417	
125	Taylor Swift	1	2023-10-16 20:51:06.881905	
126	Taylor Swift	1	2023-10-16 20:51:07.201442	
127	Taylor Swift	1	2023-10-16 20:51:07.841558	
128	Taylor Swift	1	2023-10-16 20:51:07.92369	

Figure 9 : Snap up orders

But the inventory is down by two tickets, so the two tickets were purchased repeatedly.

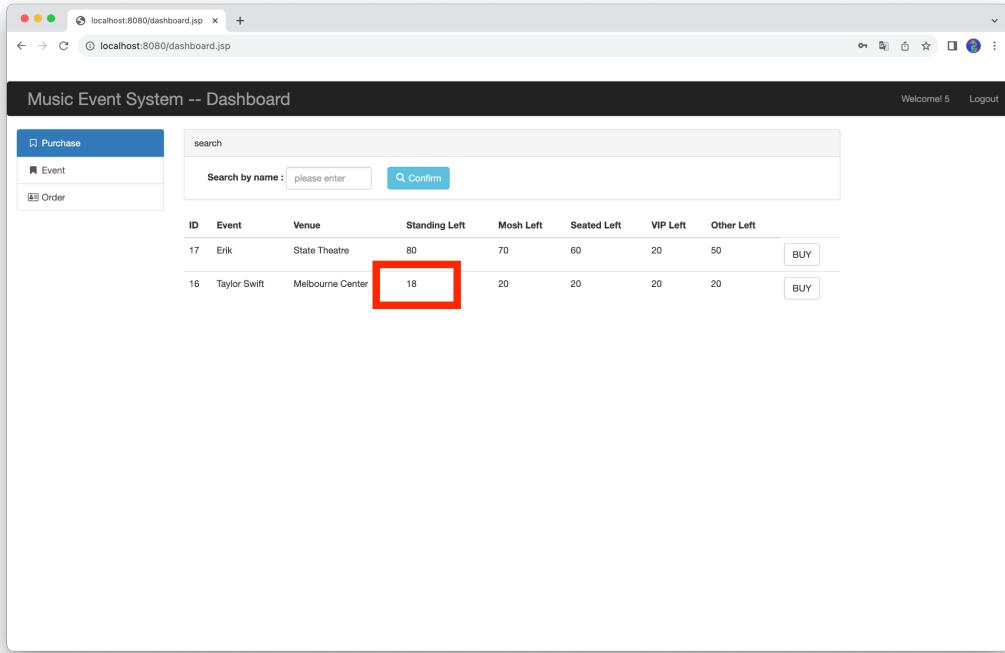


Figure 10 : Stock situation after buying

4.2.2 Issue 23.2: Overbooking Due to Concurrent Purchases

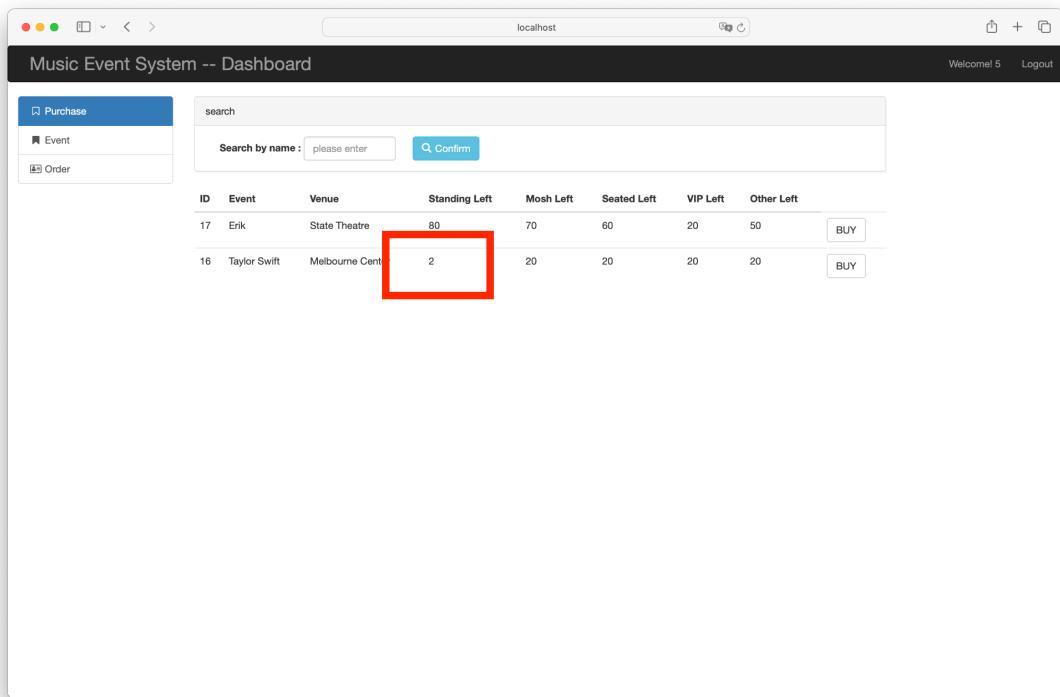
Description: When multiple customers are making purchases simultaneously, it's possible to encounter situations where the number of orders exceeds the current remaining ticket quantity, thus going beyond the scope of ticket purchases.

Proposed Solution Pattern: pessimistic lock on the order object

Rationale: During the ticket purchasing process, multiple consumers are allowed to carry out ticket purchasing actions simultaneously. The determination of whether a ticket purchase can be successfully completed, i.e., whether there are remaining tickets, is made prior to the purchase. Therefore, when multiple users are booking tickets at the same time, if one user's ticket purchase hasn't been committed to the database while other users have also initiated ticket purchasing actions, the remaining ticket count wouldn't have been updated at this point, leading to the scenario of ticket sales exceeding the remaining tickets. To resolve this issue, a pessimistic lock can be added to the order object to ensure correctness.

Issues test:

Condition: Multiple users are ready to buy Taylor Swift standing tickets with 2 remaining in stock.



ID	Event	Venue	Standing Left	Mosh Left	Seated Left	VIP Left	Other Left	
17	Erik	State Theatre	80	70	60	20	50	<button>BUY</button>
16	Taylor Swift	Melbourne Cent	2	20	20	20	20	<button>BUY</button>

Figure 11 : Inventory situation

Test: Thirty purchases occur in three seconds.



- Thread Properties

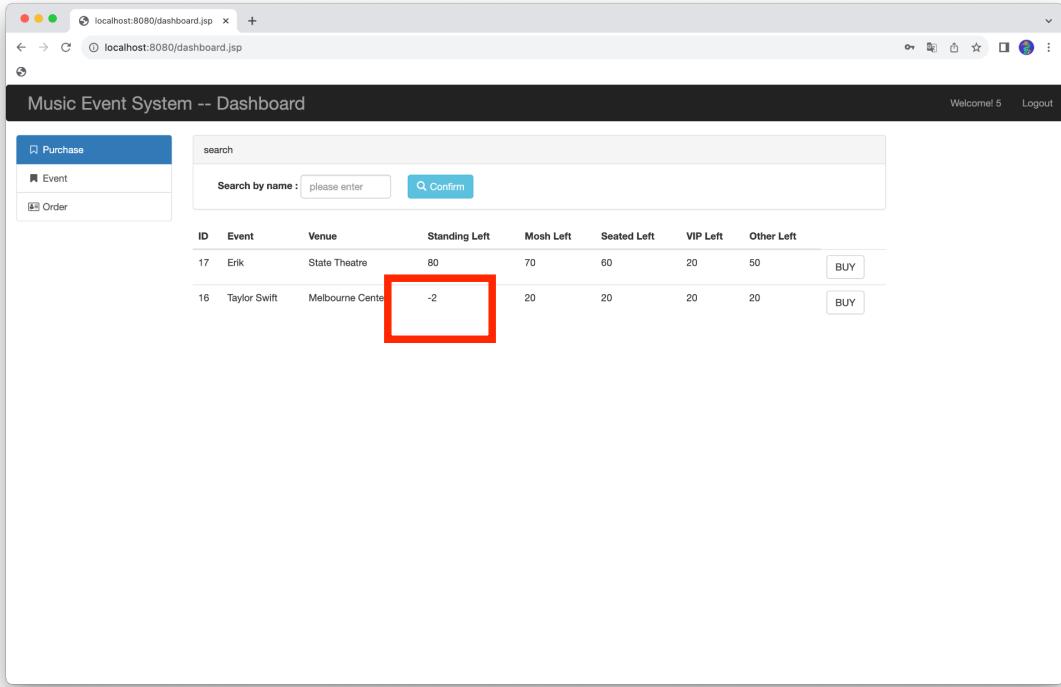
Number of Threads (users):

Ramp-up period (seconds):

Loop Count: Infinite

Figure 12 : Concurrent request configuration

Outcome: Inventory is negative.



ID	Event	Venue	Standing Left	Mosh Left	Seated Left	VIP Left	Other Left	
17	Erik	State Theatre	80	70	60	20	50	<button>BUY</button>
16	Taylor Swift	Melbourne Centre	-2	20	20	20	20	<button>BUY</button>

Figure 13 : Stock situation after buying

5. Implemented concurrency patterns

5.1 Pessimistic Lock

In music event systems, we classify the situations in which concurrency can occur into two categories. One is high concurrency and requires strong consistency, such as ticket purchasing. Since the ticketing operation is not atomic, when users snap up tickets for popular music events, it may happen that the same ticket is purchased repeatedly, or the number of purchases may exceed the number in stock. This type of situation involves monetary transactions, so strong consistency is required. So for the ticket-buying operation, we use pessimistic locking.

As the name suggests, the strategy of pessimistic locks is extremely pessimistic. Once a thread is ready to make a change to some data, it is assumed that concurrency issues must arise. Therefore, the thread applies a pessimistic lock to the data at the beginning of the operation and releases the lock when the operation is complete. When the lock acquisition fails, it waits or exits. Pessimistic locking achieves strong consistency by making operations serial, while also degrading performance.

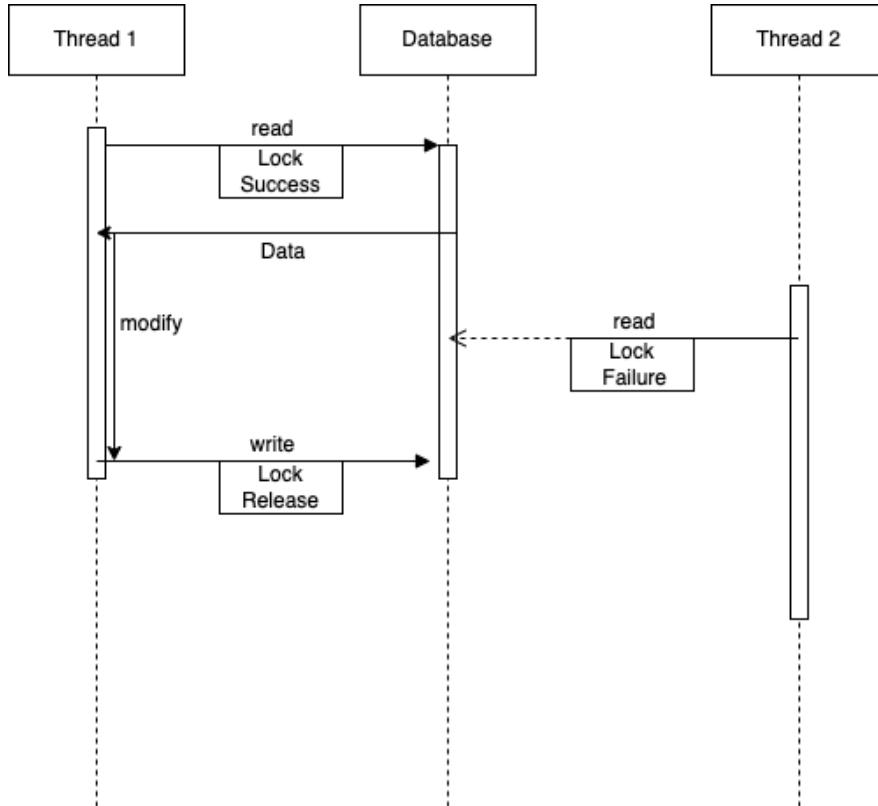


Figure 14 : Pessimistic lock sequence diagram

Because Redis performs well and is easy to implement distributed locking. So we adopt jedis, the java client of redis, to implement pessimistic locking. During the implementation I ran into the following issues, which have been resolved. The solution is in the code. Here's the key code:

Acquire lock

```

JedisPool jedisPool = JedisPoolUtil.getInstance();
Jedis jedis = jedisPool.getResource();
String value = UUID.randomUUID().toString();
int lock = ((int) jedis.setnx("ticketlock", value));
jedis.expire("ticketlock", 30);

```

Release lock

```

String lua = "if redis.call(\"get\",KEYS[1])==ARGV[1] then\n" +
    "return redis.call(\"del\",KEYS[1])\n" +
    "else\n" +
    "return 0\n" +
    "end";

```

```
String scriptLoad = jedis.scriptLoad(lua);
jedis.evalsha(scriptLoad, Collections.singletonList("ticketlock"),
Collections.singletonList(value));
```

5.1.1 Lock and Release

Redis provides a command to implement distributed locking, SETNX, which sets the key to value only if it does not already exist and does nothing if the key exists. Based on this feature, we can implement a distributed lock using Redis.

5.1.2 Lock Timeout Problem

If the service that acquired the lock goes down before releasing the lock, then the lock-stock in Redis will exist forever, so the lock cannot be released, and other services will not be able to acquire the lock, resulting in a deadlock. In order to solve this problem, we need to set an automatic timeout of the lock, which is the timeout of the Key. Even if the service fails without calling del to release the lock, the lock itself has a timeout, which can be automatically removed and acquired by another service, and the expiration time of the Key in Redis can be implemented using command “expire (lock, 30)”.

5.1.3 False Lock Removal Problem

When DEL releases the lock, it may accidentally delete the lock added by others. For example, A acquires the lock lock_stock with an expiration time of 10s. If the lock expires automatically and another service obtains the lock lock_stock during the period during which service A executes the service logic, then the del(lock_stock) of service A will remove the lock of others.

The solution is to first determine if the lock you want to remove is a lock you already own. For example, the value of the lock can be used as a UUID. When releasing the lock, it first checks whether the value of the lock is the same as the UUID created in the current service, and if so, it can execute del to remove the lock.

5.1.4 Synchronized Problem

In 4.1.3, the lock removal process is performed in three steps: query, judge, and delete, which is not atomic. If service A decides to remove the lock and the lock expires automatically, another service B acquires the lock, and then service A executes DEL, it may remove the lock from service B. Therefore, lock lookup, lock judgment, and lock removal must be done atomically. The solution can use Redis and Lua scripts to solve the consistency problem. In Redis, Lua scripts can guarantee atomicity for multiple commands. Besides, using the scriptLoad(lua) function to preload scripts to the redis server can improve performance.

5.2 Optimistic Lock

Concurrency also occurs when there are multiple planners working on the same event at the same time in the event management interface. However, the amount of concurrency is not high, and there are more reads than writes. That's why optimistic locking is appropriate.

Optimistic lock is very optimistic when operating data, and it is believed that others will not modify the data at the same time, so optimistic lock will not lock the data before updating the data. Only when the data is updated, judge whether the data is modified. If the data is modified, it will give up the current modification operation.

The optimistic lock was implemented to combat any conflicts in event details that can arise when the same event is being modified by a different planner. The system allows for multiple planners to be assigned as co-planner for a single event, and there can be situations where both co-planners are making changes to the event at the same time. In order to ensure that the co-planner is modifying the most recent event details, the optimistic lock was chosen so that when there are inconsistencies between event versions on commit, the co-planner is forced to fetch the latest version and start again.

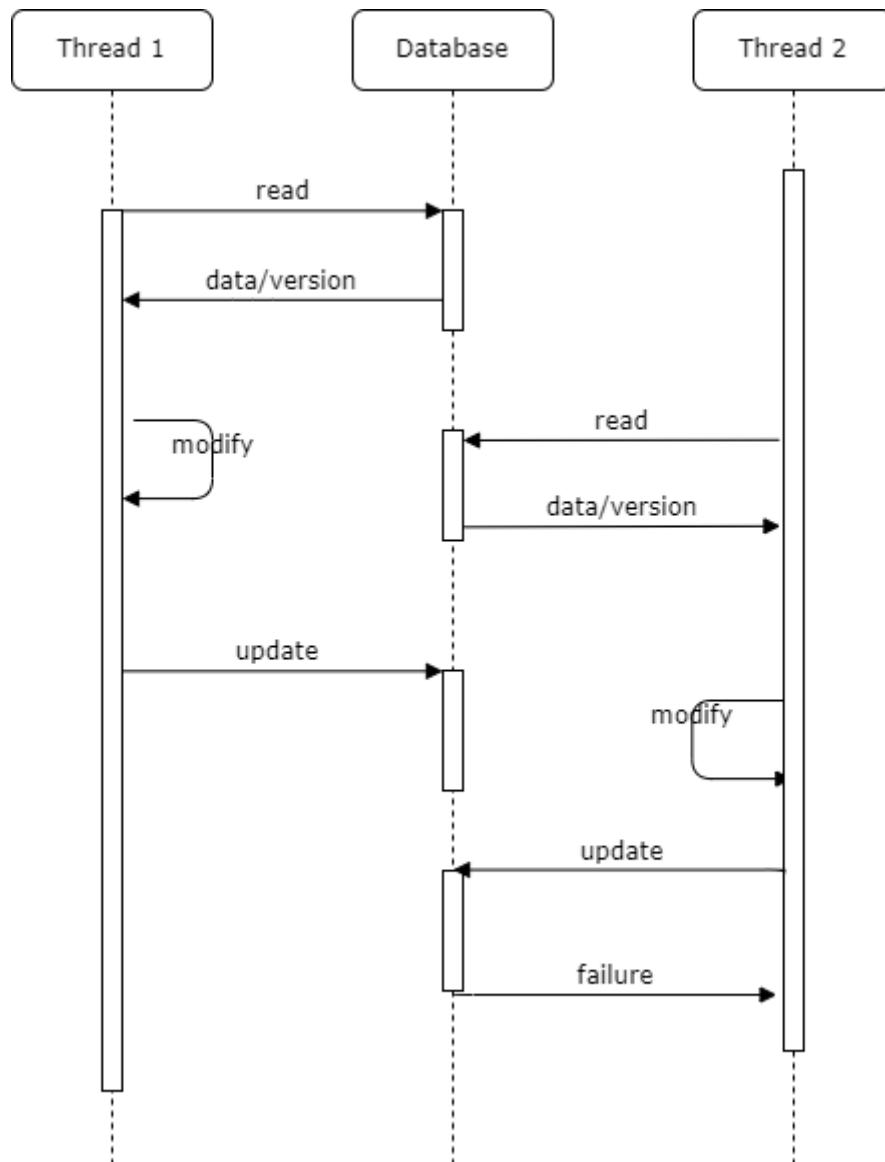


Figure 14 : Optimistic lock sequence diagram

6. Functional Test Cases

This section describes the test cases that cover the product requirements of the system.

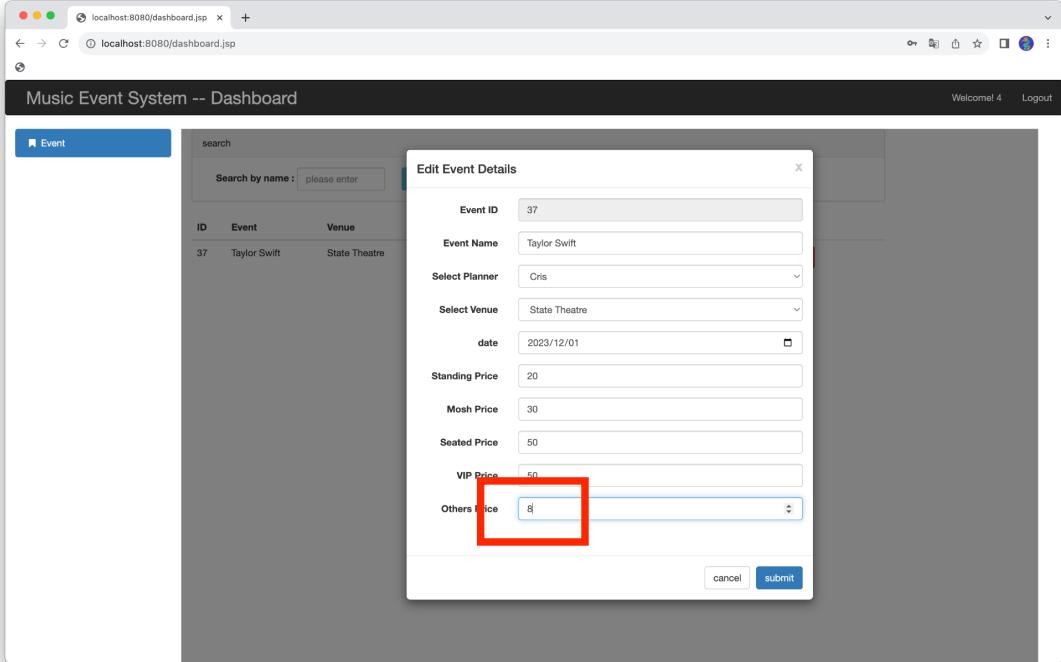
6.1 UC018: Modify Event

6.1.1 TC18.1: Event Planners Modify Event Details

This test is designed to validate that Event Planners can smoothly adjust the ticket prices for an event without encountering any system glitches. The privilege to modify ticket prices should be limited exclusively to Event Planners associated with the specific event, ensuring the accuracy and reliability of the pricing information. Additionally, when adjusting ticket prices, the system should ensure that the modification does not create conflicts or discrepancies with ticket prices or promotions of other events scheduled at the same venue and time, thereby preventing potential pricing differences or confusions.

Test Type:	Execution Type:
Functional	Manual testing
Objective:	
<ul style="list-style-type: none"> ➤ Ensure event details are consistently updated without system errors. ➤ Confirm only the associated Event Planners can modify event details. ➤ Ensure no modification is allowed if it conflicts with another committed modification. 	
Setup:	
<ol style="list-style-type: none"> 1. Event Planner “joker” login into account 2. Event Planner “planner1” login into account 3. “joker” adjust the “Other Price” to 8 4. “planner1” adjust the “Other Price” to 9 simultaneously 	
Pre-Conditions:	
<ul style="list-style-type: none"> ➤ Event Planner is logged into the system. ➤ Event to be modified exists in the system. ➤ The Event details can be modified. 	
Expected Outcomes:	
<ul style="list-style-type: none"> ➤ Event details are successfully updated by the Event Planner. ➤ System prevents one of the Event Planners from making changes to the event. ➤ System prevents modifications that would cause scheduling conflicts. 	

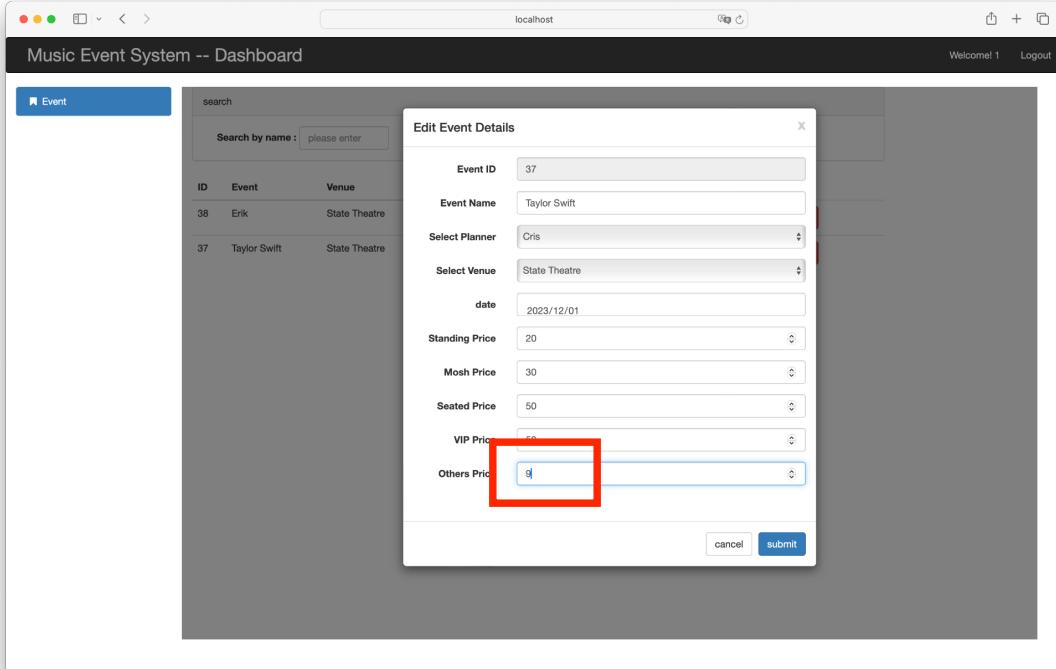
Notes:



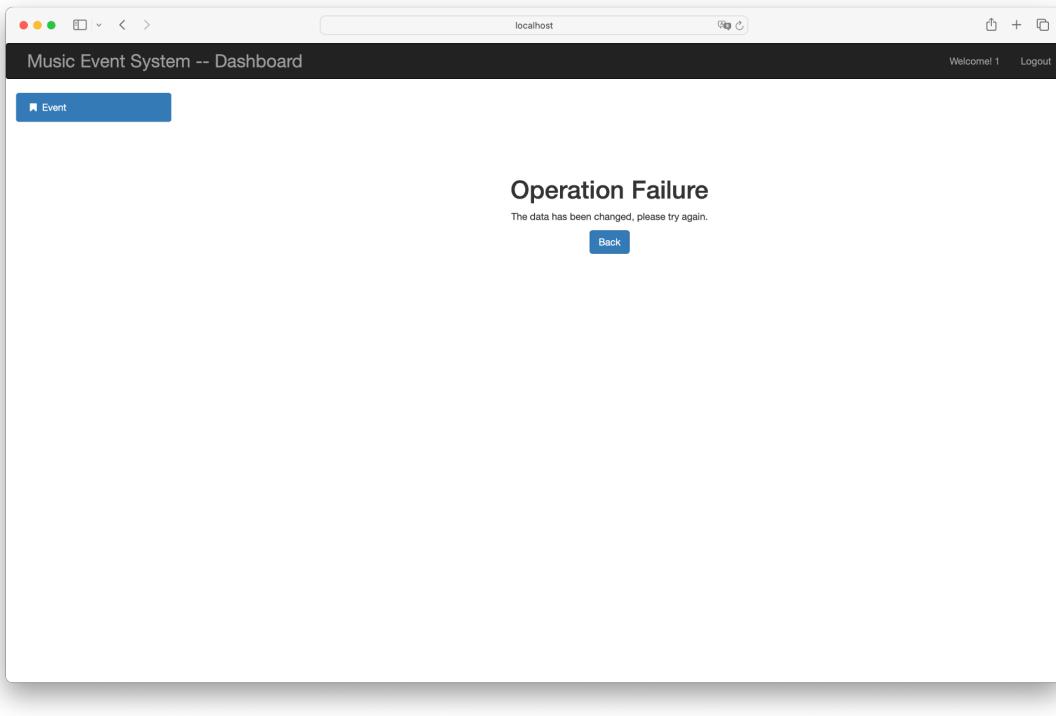
The screenshot shows a web application for managing music events. On the left, there's a sidebar with a 'Event' button highlighted in blue. The main area is titled 'Music Event System -- Dashboard'. A search bar says 'Search by name : please enter'. Below it is a table with columns 'ID', 'Event', and 'Venue'. One row shows ID 37, Event 'Taylor Swift', and Venue 'State Theatre'. To the right, a modal window titled 'Edit Event Details' is open. It contains fields for 'Event ID' (37), 'Event Name' (Taylor Swift), 'Select Planner' (Cris), 'Select Venue' (State Theatre), 'date' (2023/12/01), 'Standing Price' (20), 'Mosh Price' (30), 'Seated Price' (50), 'VIP Price' (50), and 'Others Price' (8). The 'Others Price' field is highlighted with a red box. At the bottom of the modal are 'cancel' and 'submit' buttons.

Planner B submitted the changes during Planner A's editing

Result:



The screenshot shows a web browser window for the 'Music Event System -- Dashboard'. The main area displays a table of events with columns for ID, Event, and Venue. Two rows are visible: one for 'Erik' at 'State Theatre' and another for 'Taylor Swift' at 'State Theatre'. To the right, a modal window titled 'Edit Event Details' is open, showing form fields for Event ID (37), Event Name (Taylor Swift), Select Planner (Cris), Select Venue (State Theatre), date (2023/12/01), Standing Price (20), Mosh Price (30), Seated Price (50), VIP Price (50), and Others Price (9). The 'Others Price' field is highlighted with a red border.



The screenshot shows a web browser window for the 'Music Event System -- Dashboard'. The main area displays a table of events. A central message box titled 'Operation Failure' contains the text 'The data has been changed, please try again.' and a 'Back' button.

Time constraint:

Minimum: 20 min

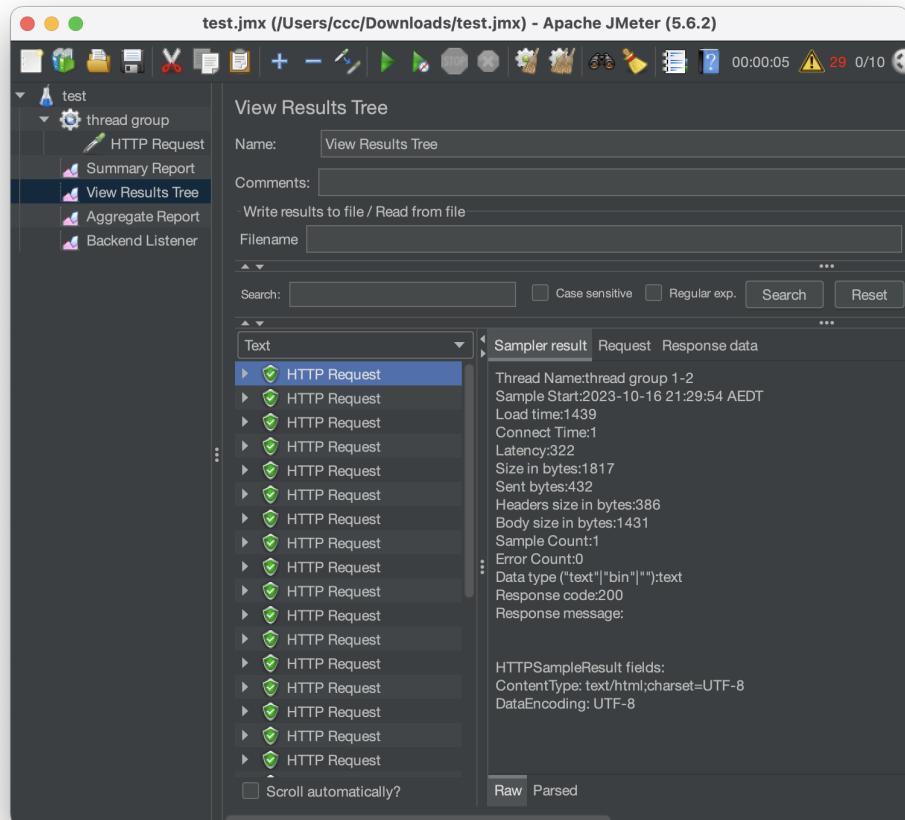
Maximum: 25 min

6.2 UC024: Book Tickets

6.2.1 TC24.1: Multiple Customers Booking Simultaneously

In scenarios where multiple customers attempt to book tickets simultaneously for the same event, section, and seat, the system should be capable of handling such concurrency. The objective of this test is to verify that the system efficiently manages these concurrent booking attempts, ensuring that only one customer successfully secures the seat while promptly marking it as unavailable for others. The test also aims to confirm that subsequent customers are immediately notified of the seat's unavailability, preventing potential double-bookings and ensuring a smooth user experience.

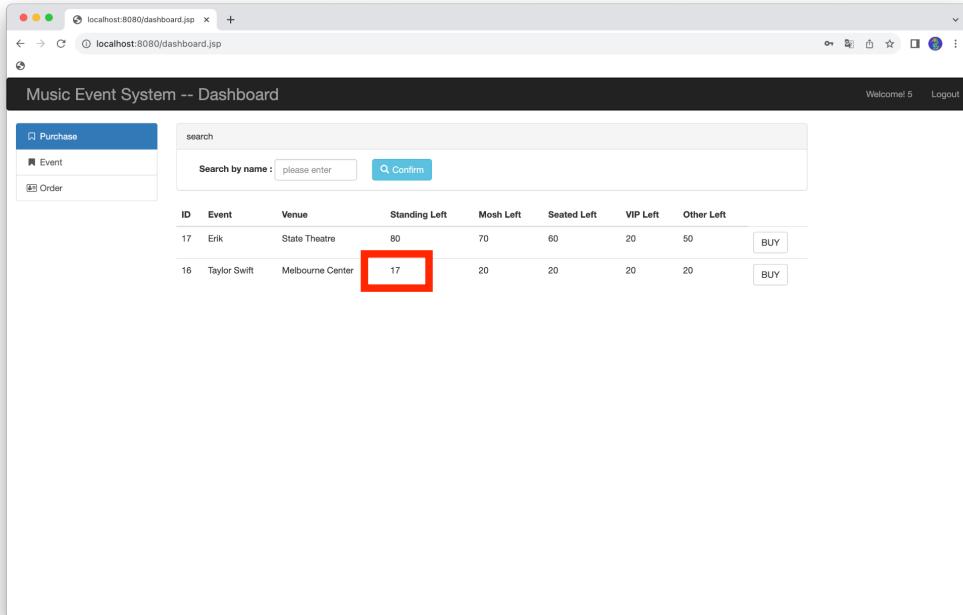
Test Type: Functional	Execution Type: Automated testing
Objective: <ul style="list-style-type: none">➤ Ensure system consistency and liveness during simultaneous bookings.➤ Confirm that once a seat is booked by a customer, it is immediately marked as unavailable to others.	
Setup: <ol style="list-style-type: none">5. Open TC24.1.jmx6. Run test plan	
Pre-Conditions: <ul style="list-style-type: none">➤ There are 20 tickets left.➤ Two customer accounts are active in the system.➤ Both customers have selected the same event, the same section, and the same seat for booking.➤ The selected seat is currently available.	



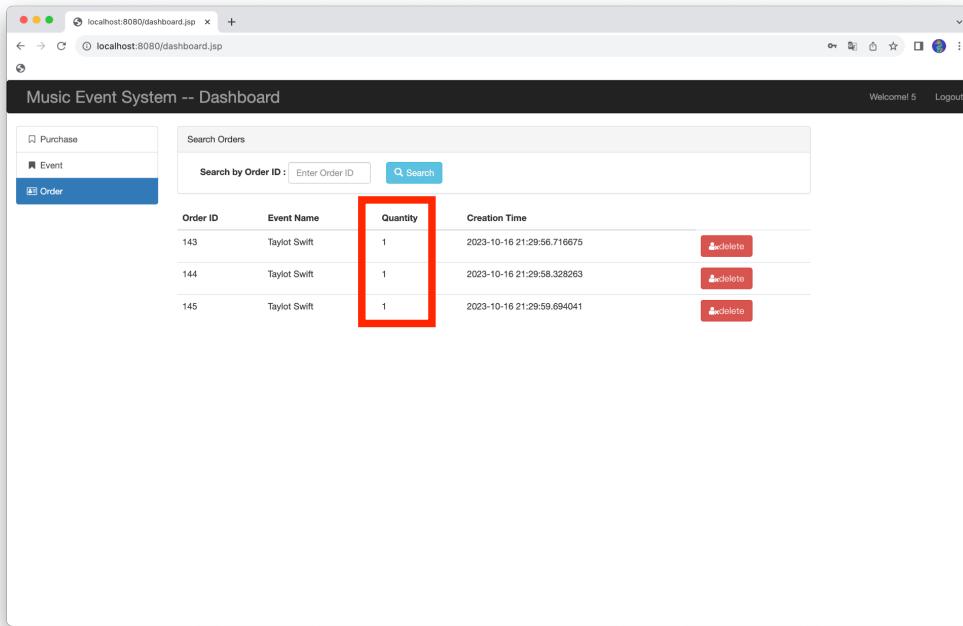
Notes:

Expected Outcomes :

- Inventory reduction is equal to the number of orders.
- Some requests failed.

Result:


ID	Event	Venue	Standing Left	Mosh Left	Seated Left	VIP Left	Other Left	Buy
17	Erik	State Theatre	80	70	60	20	50	<button>BUY</button>
16	Taylor Swift	Melbourne Center	17	20	20	20	20	<button>BUY</button>



Order ID	Event Name	Quantity	Creation Time	Action
143	Taylor Swift	1	2023-10-16 21:29:56.716675	<button>Delete</button>
144	Taylor Swift	1	2023-10-16 21:29:58.328263	<button>Delete</button>
145	Taylor Swift	1	2023-10-16 21:29:59.694041	<button>Delete</button>

Time constraint:

Minimum: 20 min

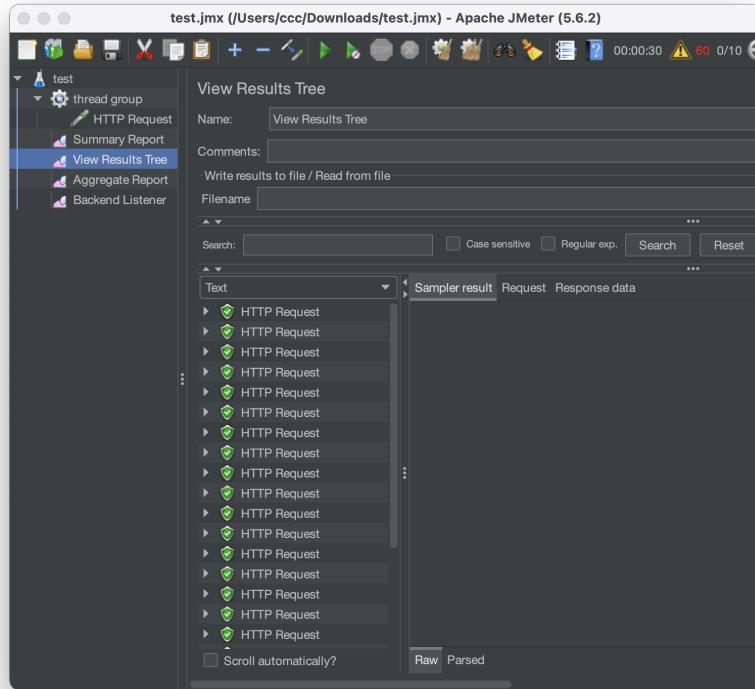
Maximum: 25 min

6.2.2 TC24.2: Last-Ticket Scenario

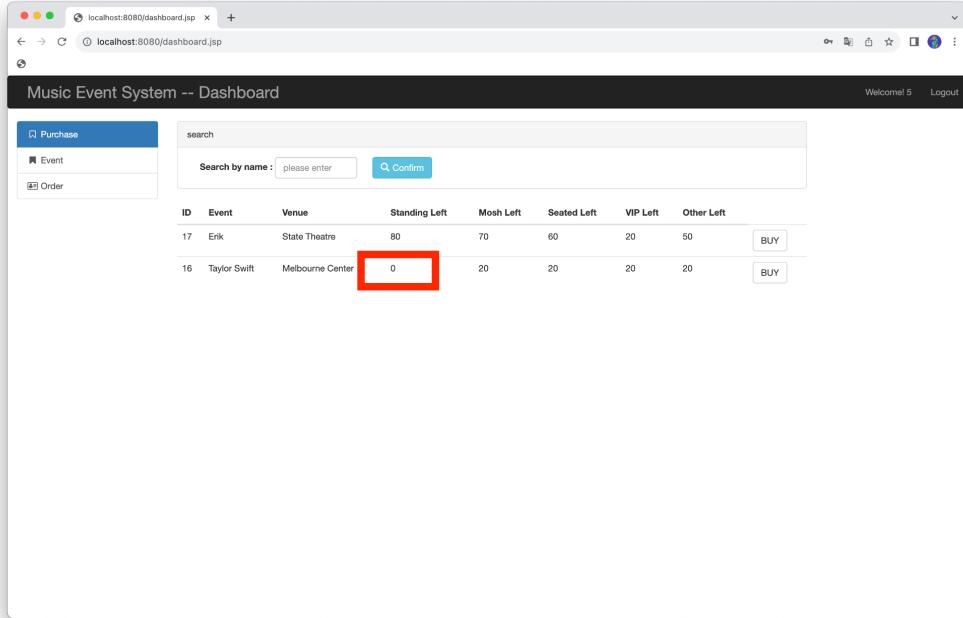
In high-demand ticket scenarios, especially when only a few tickets remain, it's critical for the booking system to handle simultaneous booking attempts with precision and fairness. This test is designed to simulate and evaluate the system's performance in such a "Last-Ticket Scenario". The objective is to ensure that only the initial successful transaction secures the remaining ticket(s), and all subsequent attempts are swiftly and clearly informed of the unavailability, preventing over-booking and ensuring user satisfaction.

Test Type:	Execution Type:
Functional	Automated testing
Objective:	
<ul style="list-style-type: none"> ➤ Ensure data consistency and liveness for the system during high-demand ticket scenarios. ➤ Verify that when multiple customers attempt to book the last few tickets simultaneously, only the first transaction is successful. 	
Setup:	
<ol style="list-style-type: none"> 7. Open TC24.2.jmx 8. Run test plan 	
Pre-Conditions:	
<ul style="list-style-type: none"> ➤ Only one ticket is left for a particular event. ➤ Multiple users are active and ready to book these last tickets. 	
Expected Outcomes:	
<ul style="list-style-type: none"> ➤ Only the first user to finalize the transaction successfully books the last ticket(s). ➤ Other users fail when trying to create an order, indicating that the ticket is not available. 	

Notes:



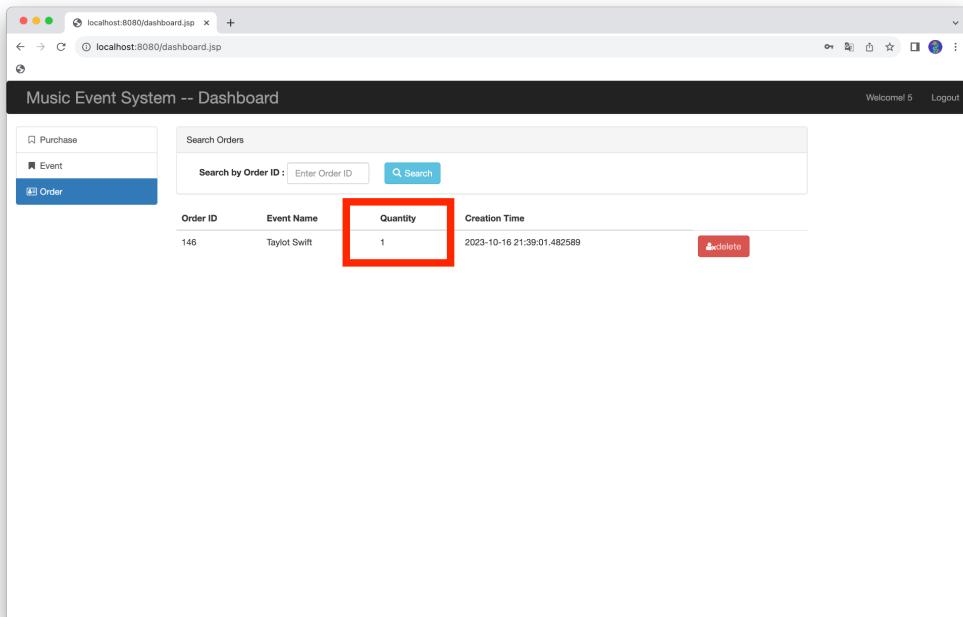
Result:



The screenshot shows the Music Event System -- Dashboard. On the left, there is a sidebar with three buttons: Purchase (selected), Event, and Order. In the center, there is a search bar with the placeholder "Search by name : please enter" and a "Confirm" button. Below the search bar is a table with the following data:

ID	Event	Venue	Standing Left	Mosh Left	Seated Left	VIP Left	Other Left
17	Erik	State Theatre	80	70	60	20	50
16	Taylor Swift	Melbourne Center	0	20	20	20	20

A red box highlights the value "0" in the "Standing Left" column for Taylor Swift's event.



The screenshot shows the Music Event System -- Dashboard. On the left, there is a sidebar with three buttons: Purchase, Event, and Order (selected). In the center, there is a search bar with the placeholder "Search Orders" and a "Search" button. Below the search bar is a table with the following data:

Order ID	Event Name	Quantity	Creation Time
146	Taylor Swift	1	2023-10-16 21:39:01.482589

A red box highlights the value "1" in the "Quantity" column for the order.

Time constraint:

Minimum: 20 min

Maximum: 25 min

7. Entry Data

<Define the entry data that will be used by more than one test case. Generally, these data represent form fields and are used to test field validation. It is a good practice to group them by the entities that they represent. Create a table for every group you define. Complete and / or adapt the following text.>

This section describes the entry data that will be used by more than one test case, avoiding data replication. These data are referenced by the test cases.

7.1 DATA...: <Data Set Name>

Description: admin	
username	password
admin1	123123

Description: planner	
username	password
joker	123
plannerrrr	123123
planner1	123321

Description: customer	
username	password
superman	123



Jenny	520
ccus	123456
