Architecture Document

SWEN90007

Music Events System

Team: SDA1

In charge of:

Name	Student id	Email	Unimelb username	Github username
Chengyi Huang	1173994	chengyih@student.unimelb.edu.a	chengyih	Kind0fblue
Ziqiang Li	1173898	ziqiangl1@student.unimelb.edu.au	ziqiangl1	johnnylild
Yuxin Liu	1408760	yuxliu20@student.unimelb.edu.au	YUXLIU20	YuxinLiu-unimelb
Hanming Mao	1257522	hanmingm@student.unimelb.edu.au	HanmingM	KkkellyM

THE UNIVERSITY OF MELBOURNE

SCHOOL OF COMPUTING & INFORMATION SYSTEMS

Revision History

Date	Version	Description	Author
09/10/2023	01.00	Initial draft with one issues	Hanming Mao
17/10/2023	02.00	Add class diagram	Yuxin Liu
17/10/2023	Final	Fix issues	Chengyi Huang

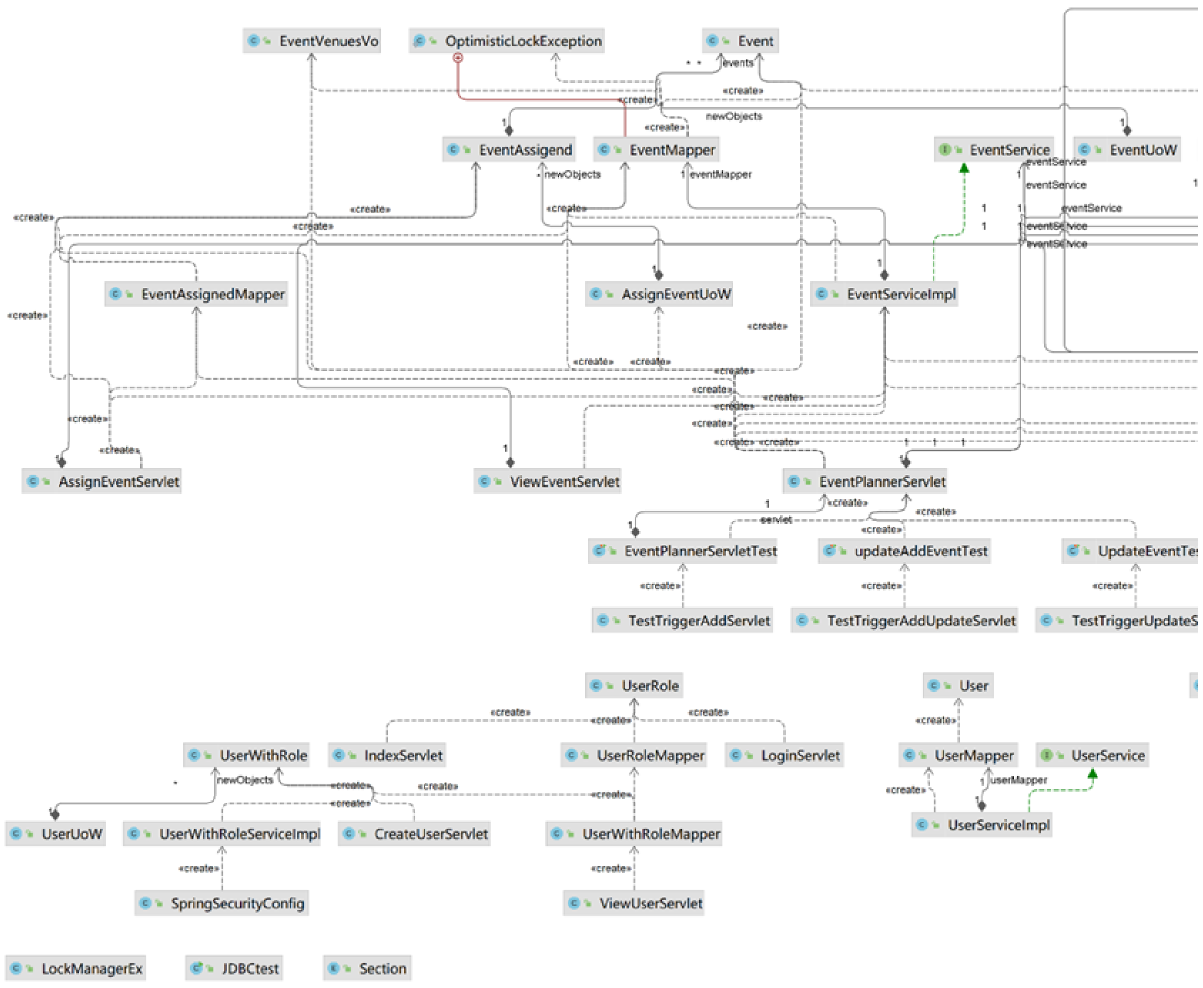
Link to deployed app:
https://nine0007-1b.onrender.com/musicsystem1017/login

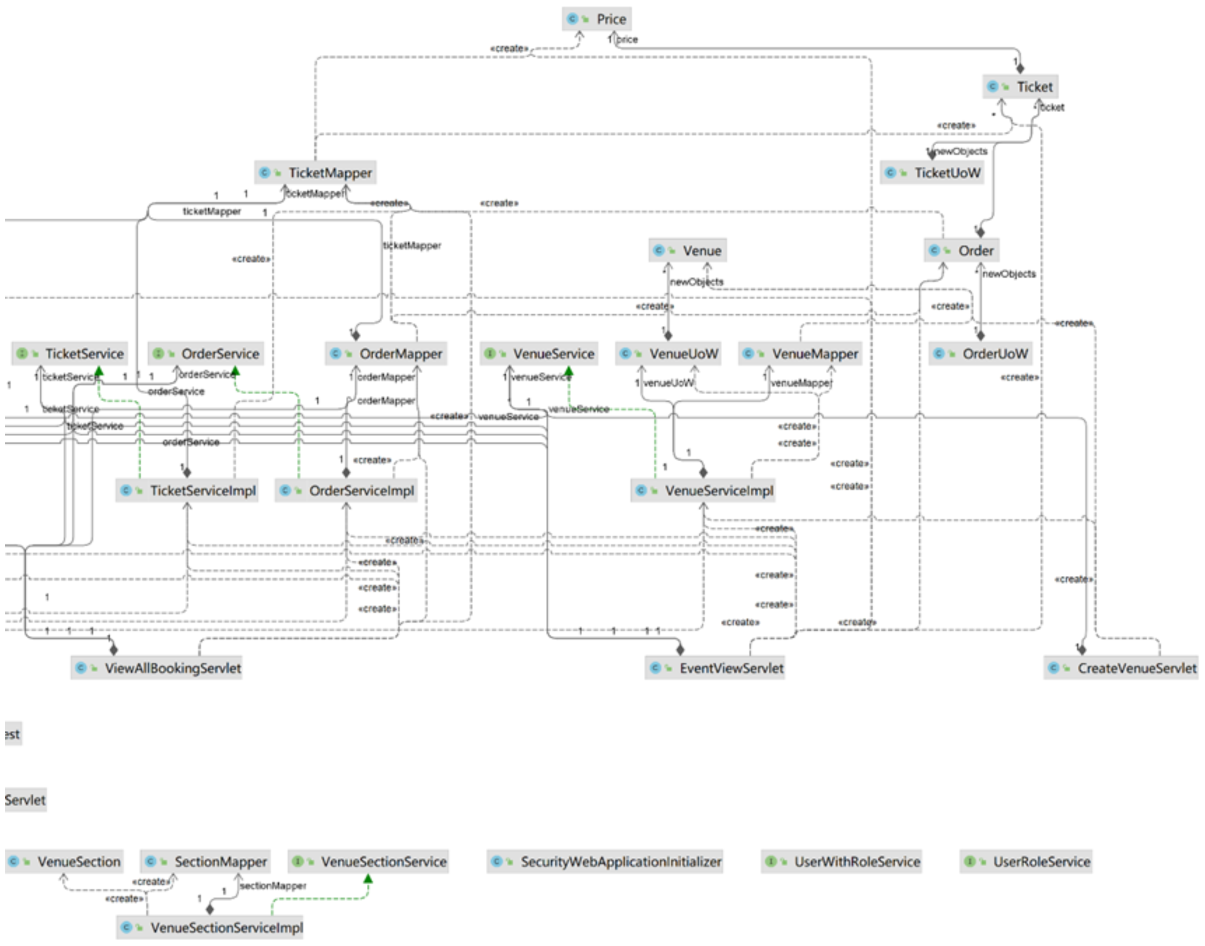
Class diagram

Since the project's classes contain a large number of properties, constructors, methods, and dependencies, a traditional class diagram is too large and unreadable in reports. So in this report, we split the class diagram into a high-level dependency diagram and a detailed implementation diagram for each class. This improves the readability of class diagrams.

Note: Updated classes are highlighted with red wireframe.

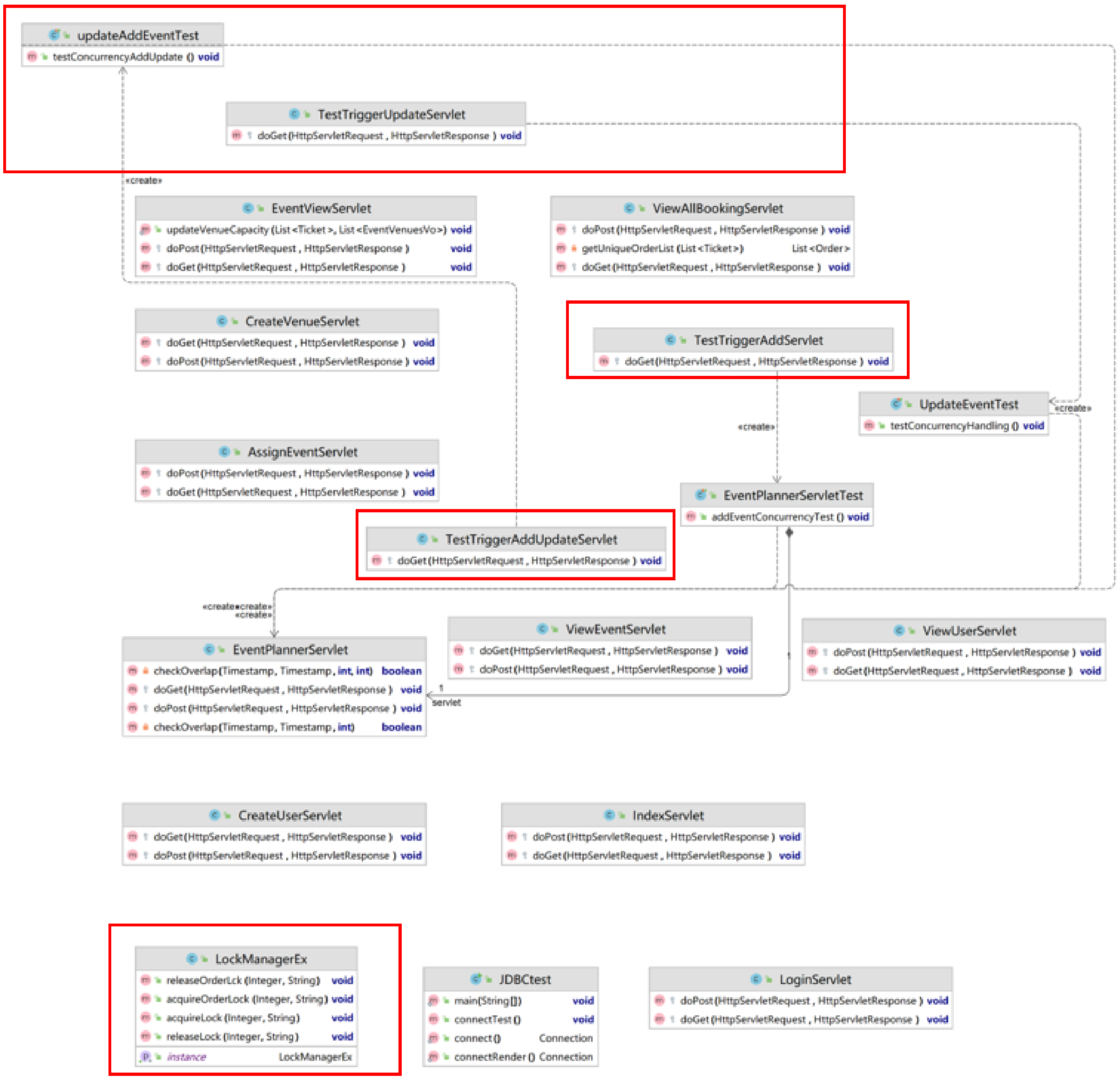
1.1 High-level dependency diagram


EventVenuesVo
OptimisticLockException
Event
events
«create»
newObjects
EventAssigend
EventMapper
eventMapper
EventService
EventUoW
eventService
EventAssignedMapper
AssignEventUoW
EventServiceImpl
AssignEventServlet
ViewEventServlet
EventPlannerServlet
servlet
EventPlannerServletTest
updateAddEventTest
UpdateEventTe
TestTriggerAddServlet
TestTriggerAddUpdateServlet
TestTriggerUpdateS
UserRole
User
UserWithRole
IndexServlet
UserRoleMapper
LoginServlet
UserMapper
UserService
userMapper
UserUoW
UserWithRoleServiceImpl
CreateUserServlet
UserWithRoleMapper
UserServiceImpl
SpringSecurityConfig
ViewUserServlet
LockManagerEx
JDBCtest
Section

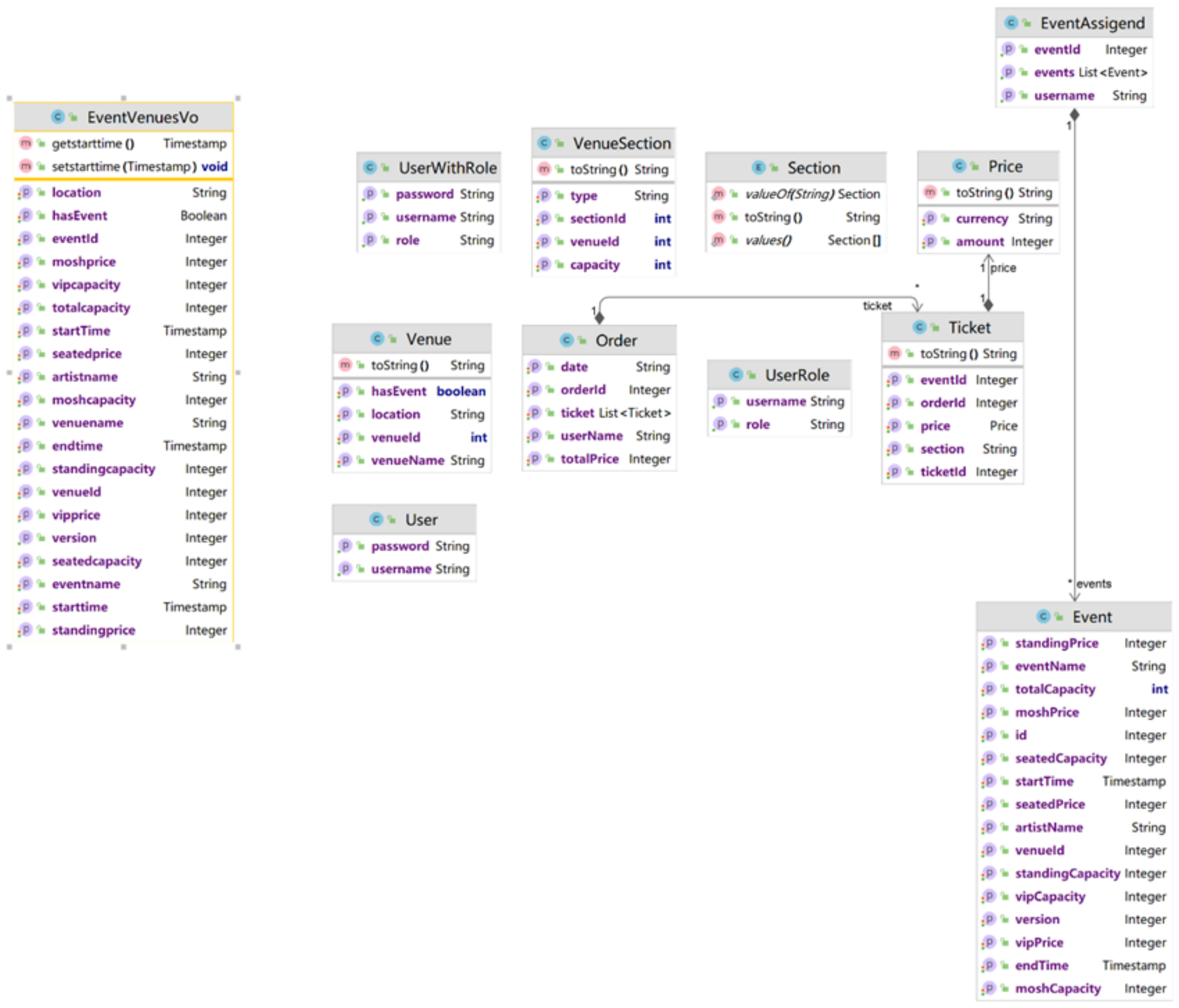


1.2 Implementation diagram for each class

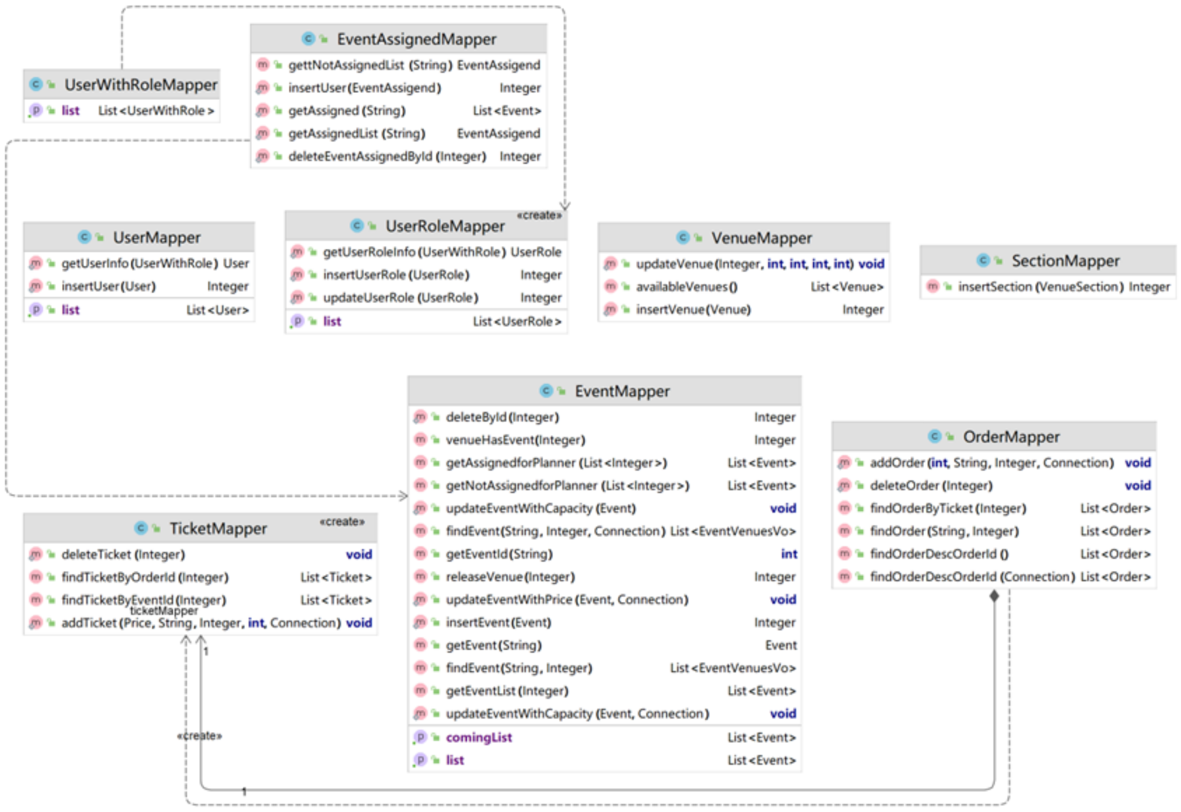
1.2.1 Class in servlet package


updateAddEventTest
testConcurrencyAddUpdate () void
TestTriggerUpdateServlet
doGet (HttpServletRequest , HttpServletResponse) void
«create»
EventViewServlet
updateVenueCapacity (List<Ticket>, List<EventVenuesVo>) void
doPost (HttpServletRequest , HttpServletResponse) void
doGet (HttpServletRequest , HttpServletResponse) void
ViewAllBookingServlet
doPost (HttpServletRequest , HttpServletResponse) void
getUniqueOrderList (List<Ticket>) List<Order>
doGet (HttpServletRequest , HttpServletResponse) void
CreateVenueServlet
doGet (HttpServletRequest , HttpServletResponse) void
doPost (HttpServletRequest , HttpServletResponse) void
TestTriggerAddServlet
doGet (HttpServletRequest , HttpServletResponse) void
UpdateEventTest
testConcurrencyHandling () void
«create»
«create»
AssignEventServlet
doPost (HttpServletRequest , HttpServletResponse) void
doGet (HttpServletRequest , HttpServletResponse) void
EventPlannerServletTest
addEventConcurrencyTest () void
TestTriggerAddUpdateServlet
doGet (HttpServletRequest , HttpServletResponse) void
«create»«create»
«create»
EventPlannerServlet
checkOverlap (Timestamp, Timestamp, int, int) boolean
doGet (HttpServletRequest , HttpServletResponse) void
doPost (HttpServletRequest , HttpServletResponse) void
checkOverlap (Timestamp, Timestamp, int) boolean
ViewEventServlet
doGet (HttpServletRequest , HttpServletResponse) void
doPost (HttpServletRequest , HttpServletResponse) void
1
servlet
ViewUserServlet
doPost (HttpServletRequest , HttpServletResponse) void
doGet (HttpServletRequest , HttpServletResponse) void
CreateUserServlet
doGet (HttpServletRequest , HttpServletResponse) void
doPost (HttpServletRequest , HttpServletResponse) void
IndexServlet
doPost (HttpServletRequest , HttpServletResponse) void
doGet (HttpServletRequest , HttpServletResponse) void
LockManagerEx
releaseOrderLck (Integer, String) void
acquireOrderLock (Integer, String) void
acquireLock (Integer, String) void
releaseLock (Integer, String) void
instance LockManagerEx
JDBCtest
main (String[]) void
connectTest () void
connect () Connection
connectRender () Connection
LoginServlet
doPost (HttpServletRequest , HttpServletResponse) void
doGet (HttpServletRequest , HttpServletResponse) void

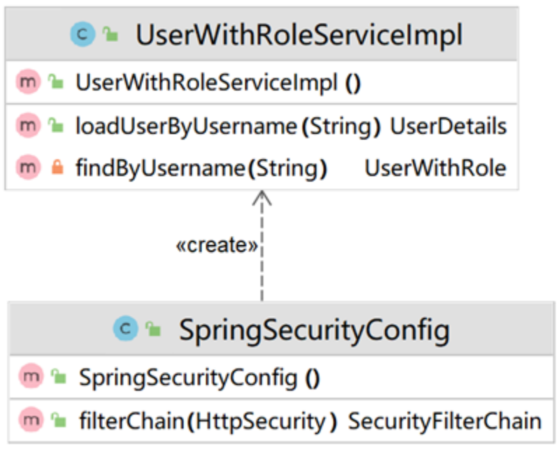
1.2.2 Class in domain package

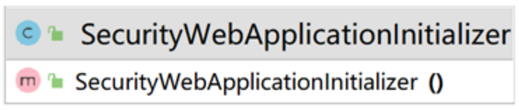


1.2.3 Class in Mapper package

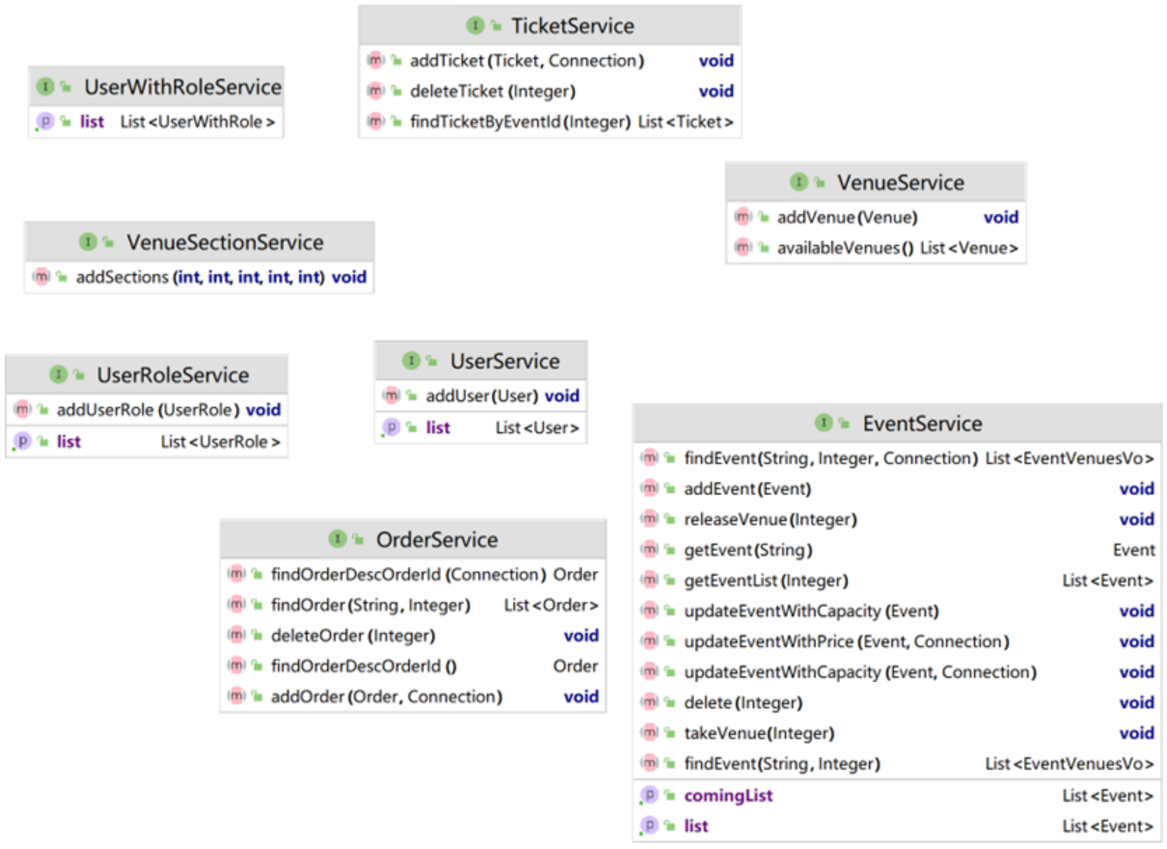

EventAssignedMapper
gettNotAssignedList (String) EventAssigend
insertUser(EventAssigend) Integer
getAssigned (String) List<Event>
getAssignedList (String) EventAssigend
deleteEventAssignedById (Integer) Integer
UserWithRoleMapper
list List<UserWithRole>
UserMapper
getUserInfo (UserWithRole) User
insertUser(User) Integer
list List<User>
«create»
UserRoleMapper
getUserRoleInfo (UserWithRole) UserRole
insertUserRole (UserRole) Integer
updateUserRole (UserRole) Integer
list List<UserRole>
VenueMapper
updateVenue(Integer, int, int, int, int) void
availableVenues() List<Venue>
insertVenue(Venue) Integer
SectionMapper
insertSection (VenueSection) Integer
EventMapper
deleteById (Integer) Integer
venueHasEvent(Integer) Integer
getAssignedforPlanner (List<Integer>) List<Event>
getNotAssignedforPlanner (List<Integer>) List<Event>
updateEventWithCapacity (Event) void
findEvent(String, Integer, Connection) List<EventVenuesVo>
getEventId(String) int
releaseVenue(Integer) Integer
updateEventWithPrice (Event, Connection) void
insertEvent (Event) Integer
getEvent (String) Event
findEvent(String, Integer) List<EventVenuesVo>
getEventList (Integer) List<Event>
updateEventWithCapacity (Event, Connection) void
comingList List<Event>
list List<Event>
OrderMapper
addOrder (int, String, Integer, Connection) void
deleteOrder (Integer) void
findOrderByTicket (Integer) List<Order>
findOrder (String, Integer) List<Order>
findOrderDescOrderId () List<Order>
findOrderDescOrderId (Connection) List<Order>
«create»
TicketMapper
deleteTicket (Integer) void
findTicketByOrderId (Integer) List<Ticket>
findTicketByEventId (Integer) List<Ticket>
ticketMapper
addTicket (Price, String, Integer, int, Connection) void
1
«create»
1

1.2.4 Class in SecurityCheck package

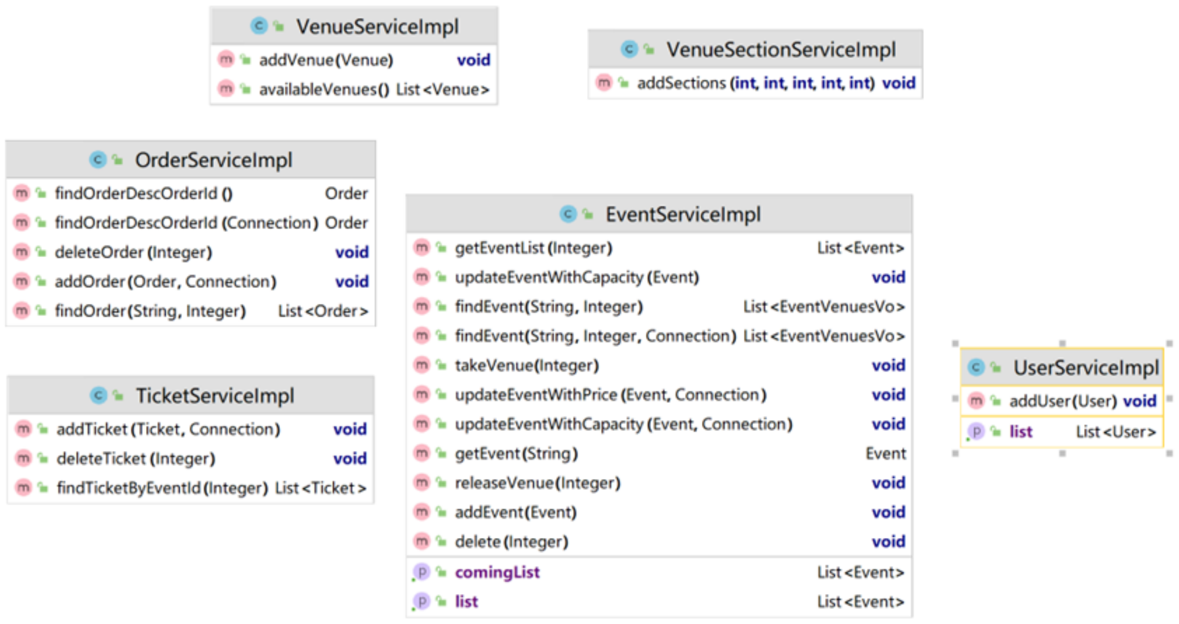

UserWithRoleServiceImpl
UserWithRoleServiceImpl ()
loadUserByUsername (String) UserDetails
findByUsername (String) UserWithRole
«create»
SpringSecurityConfig
SpringSecurityConfig ()
filterChain (HttpSecurity) SecurityFilterChain


SecurityWebApplicationInitializer
SecurityWebApplicationInitializer ()

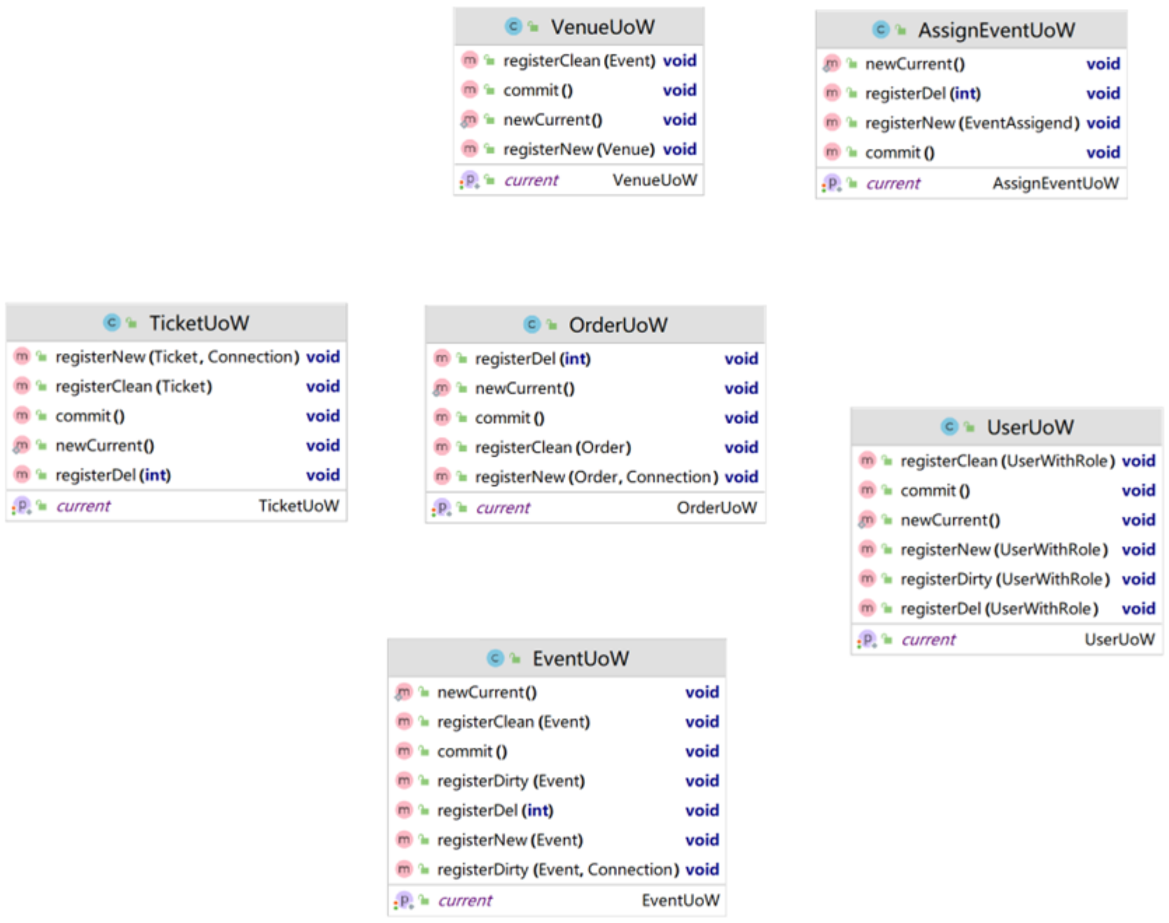
1.2.5 Class in Service package


TicketService
addTicket(Ticket, Connection) void
deleteTicket(Integer) void
findTicketByEventId(Integer) List<Ticket>
UserWithRoleService
list List<UserWithRole>
VenueService
addVenue(Venue) void
availableVenues() List<Venue>
VenueSectionService
addSections(int, int, int, int, int) void
UserRoleService
addUserRole(UserRole) void
list List<UserRole>
UserService
addUser(User) void
list List<User>
EventService
findEvent(String, Integer, Connection) List<EventVenuesVo>
addEvent(Event) void
releaseVenue(Integer) void
getEvent(String) Event
getEventList(Integer) List<Event>
updateEventWithCapacity(Event) void
updateEventWithPrice(Event, Connection) void
updateEventWithCapacity(Event, Connection) void
delete(Integer) void
takeVenue(Integer) void
findEvent(String, Integer) List<EventVenuesVo>
comingList List<Event>
list List<Event>
OrderService
findOrderDescOrderId(Connection) Order
findOrder(String, Integer) List<Order>
deleteOrder(Integer) void
findOrderDescOrderId() Order
addOrder(Order, Connection) void

1.2.6 Class in ServiceImpl package


VenueServiceImpl
addVenue(Venue) void
availableVenues() List<Venue>
VenueSectionServiceImpl
addSections(int, int, int, int, int) void
OrderServiceImpl
findOrderDescOrderId() Order
findOrderDescOrderId(Connection) Order
deleteOrder(Integer) void
addOrder(Order, Connection) void
findOrder(String, Integer) List<Order>
EventServiceImpl
getEventList(Integer) List<Event>
updateEventWithCapacity(Event) void
findEvent(String, Integer) List<EventVenuesVo>
findEvent(String, Integer, Connection) List<EventVenuesVo>
takeVenue(Integer) void
updateEventWithPrice(Event, Connection) void
updateEventWithCapacity(Event, Connection) void
getEvent(String) Event
releaseVenue(Integer) void
addEvent(Event) void
delete(Integer) void
comingList List<Event>
list List<Event>
TicketServiceImpl
addTicket(Ticket, Connection) void
deleteTicket(Integer) void
findTicketByEventId(Integer) List<Ticket>
UserServiceImpl
addUser(User) void
list List<User>

1.2.7 Class in UoW package



2.0 Concurrency Use Cases

2.1 Two client book limited tickets at the same time

1. Description

   The system allows multiple clients to book tickets for different events at the same time but the ticket of each event and different sections of the event is limited. From a business transaction point of view, we should prevent two or more users from booking the last several tickets when they click book at the same time.

   This issue occurs because the system first gets the event's remaining tickets information from the database then adds the order and new tickets to the database and updates the event's capacity. However, when multiple users use the system, situations like they read the remaining tickets information at the same time. For example, at this time event A has one remaining ticket for the vip section, and user A and B both read this record from the database. Then the system checks where the remaining tickets are greater than they want to buy. This is if both user A and B want to buy one VIP ticket of event A and their system already runs czech check then they can all add order to the database which will lead to two tickets being sold even though there is only one available.

2. Choice of pattern

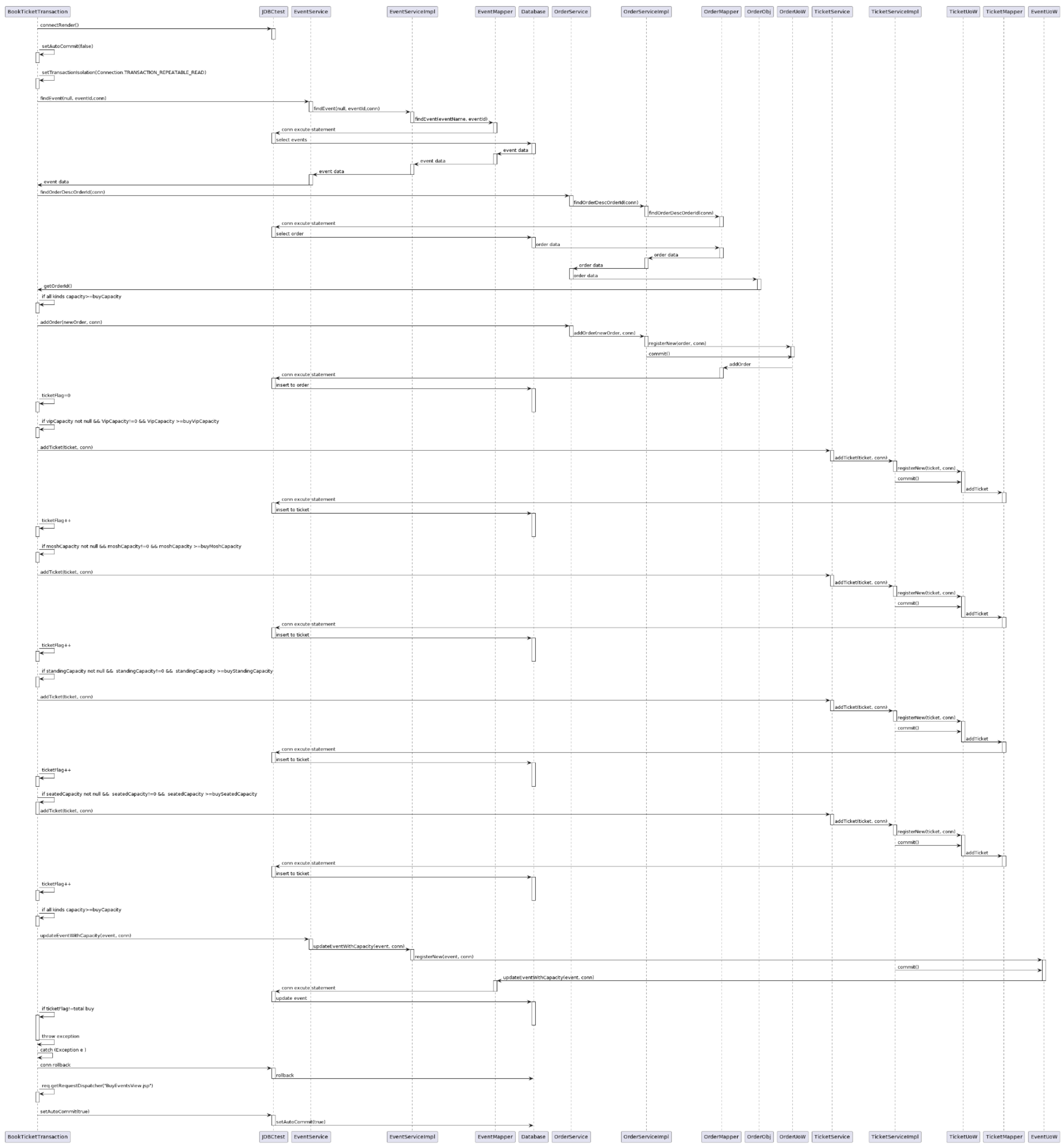
   We choose to use the ACID structure to let the database deal with the whole process.

3. Rationale for the choice of pattern

   For this function, even though it involves multiple database interaction which means multiple system transactions, such as read event data, add order data, add tickets data and update event data, all these interactions with the database are within one method type under one request so it can be organized to a long transaction. Based on these, it is more direct and more reliable to use the ACID structure as we can hand the rollback and the error finding process over to the database which works better than our own development logic.

   Meanwhile, if the conditions permit, the ACID structure actually needs less change to our original codes which helps better maintain the systems' other functions and make it easier for all teammates to get the concurrency dealing code for this function without any modification to their process.

4. Implementation detail



As shown in the sequence diagram, the system first set the connection to not auto commit and TRANSACTION_REPEATABLE_READ.

the whole transaction includes first get event information from the database then checking if the buyCapacity is less than remaining capacity.

If so, the system gets the newest orderId in the database then uses this orderId+1 as the new orderId. As the orderId is the primary key of table order. If more than one user get the same orderId to add SQLException throws by the addOrder methods and if the system catch this exception the change for this user will rollback.

If no conflict orderId occurs, the system will then try to add tickets for the user. In this process, we set a ticketFlag to record how many tickets are added to the database. If the tickets added to the database do not equal to the total number of tickets clients buy, the system will throw a RuntimeException and the changes to the database will rollback.

After roll back, the system redirects the user to the buy ticket page and on this page the user can see the updated remaining capacity.

If none of the exceptions occurs, the transaction is successful and the database will save the changes.

At last, the system set the connection back to

5. Testing strategy

   To test this function, we use Jmeter with one request in one thread group. The request simulates the actions of several different users (at least two) first login to the system, navigate to the buy ticket page and then book tickets for the same event with limited tickets.

   Test Data:

   User1:

   username: test0918

   password: test0918

   User2:

   username: client

   password: client

   User3:

   username: client2

   password: client2

   Event:

   eventName:test1

   eventId:73

   vipcapacity:1

   moshcapacity:2

   standingcapacity:3

   seatedcapacity:4

   We choose to use 24 threads and run only 1 loops. Each user tries to buy 1 vip ticket and 1 mosh ticket at a time.

   The expected result should be that only one user can succeed in buying the ticket one time. Only one order was added to the system. The event's vipcapacity and moshcapacity should be 0 and 1 respectively.

6. Summary of the outcomes of the concurrency tests

   The result in the Jmeter shows that only one order is added to the system and other transactions are rolled back. When we get into the system, we can see the vipcapacity and moshcapacity are 0 and 1.

When cancel the order for the specific user, the vipcapacity and moshcapacity return to 1 and 2.

2.2 Multiple Event Planners add events with the same venue and overlapping time (clash) simultaneously.

1. Description

In the Music Event System, race conditions can arise when multiple event planners try to add events with the same venue for overlapping times. Consider two planners, A and B. A begins booking a venue, and while he's entering details, B checks the same venue's availability. Without proper concurrency controls, both might successfully book the venue for overlapping periods. This can lead to scheduling conflicts and customer dissatisfaction. To prevent such issues, the system needs mechanisms like locking to ensure sequential and non-conflicting bookings.

In addition, Checking a venue's availability and then booking it should be an atomic operation. If these operations are separate and not atomic, it can lead to race conditions where the venue is double booked.

2. Choice of concurrency pattern and rationale

The concurrency pattern used to handle this issue is Pessimistic offline lock, specifically exclusive write lock. A lock is acquired on the Venue that the event planner is performing adding operation on until a full adding event transaction is completed. The rationale for using this pattern is based these standpoints:

Correctness: The main strength of using a pessimistic lock compared to an optimistic lock is its consistency in correctness. Once a thread acquires a lock on a resource (in this case, a venue via venueId), no other thread can access it until the lock is released. This ensures that two event planners cannot simultaneously book the same venue, thus maintaining data integrity and consistency. The provided acquireLock method solidifies this by preventing the acquisition of a lock if one already exists. This is especially important Given the high stakes involved in event management — including venue bookings, artist contracts, ticket sales, and marketing expenses, any error due to concurrency issues can result in significant financial loss.

Liveness: It's recognized that pessimistic locks offer less liveness compared to optimistic locks, potentially leading to reduced system throughput due to waiting for locks. However, in our system, the number of event planners operating within the system is limited (expected to have somewhere around 10-20 event planners). This means the probability of simultaneous access and contention for the same resources is relatively low. Thus, our priority is correctness over liveness.

3. Implementation detail

“LockManagerEx” Class:

The LockManagerEx class implements a Singleton pattern, ensuring there's only one instance of the LockManagerEx throughout the application. The purpose of this class is to manage exclusive locks on resources (in this case, venues identified by venue IDs). Its key features include:

1. Uses a ConcurrentMap to keep track of which resource (venue) is locked and by whom (owner).
2. Provides a synchronized getInstance method to ensure only a single instance is created.
3. Offers acquireLock and releaseLock methods to handle the locking mechanism.

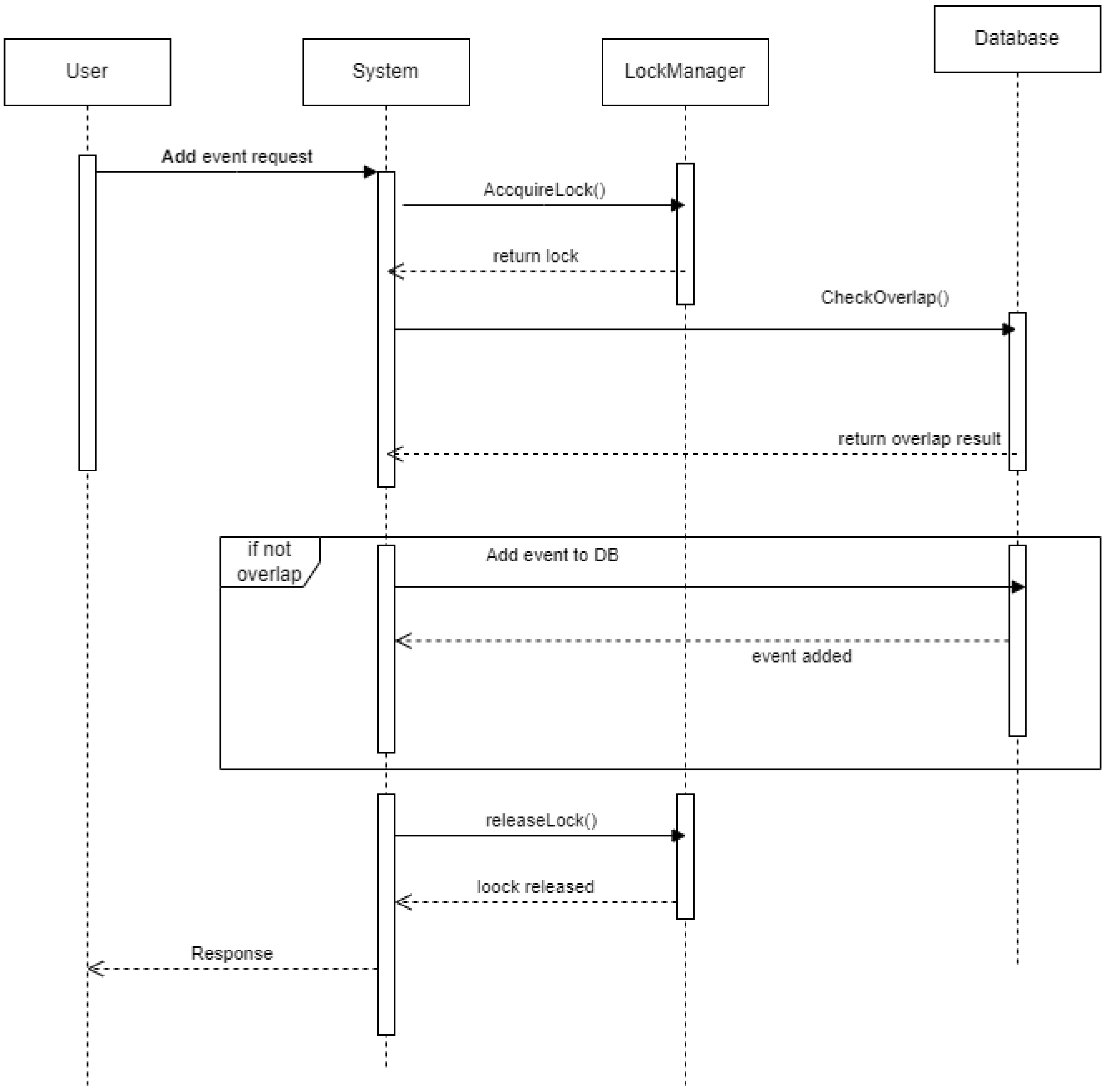
"add" Case Sequence in “EvenetPlannerServlet”:

When trying to add an event:

1. Fetches various parameters related to the event (user, event details, timing, prices, etc.).
2. Checks if the requested timing for the event overlaps with existing events using the checkOverlap method.
3. If there's an overlap, the user is redirected back with an error messages
4. If there's no overlap, it tries to acquire a lock for the venue using the LockManagerEx's acquireLock method.
5. If the lock is successfully acquired, the event is added to the system.
6. If the lock cannot be acquired (venue is already locked by another session), the user receives an error message about failing to acquire the lock.
7. After processing, regardless of success or exceptions, the lock on the venue is released.

Sequence Diagram:

A visualization of the steps above:



Testing strategy:

The strategy is to employ java multithreading to simulate real-world scenarios where multiple event planners might be trying to add events to the system concurrently. Through the use of mocking and junit, the test focuses purely on the servlet's logic and behavior under these conditions, abstracting away external dependencies and interactions. Actual test code can be found in the "EventPlannerServletTest" file, below is a rundown of the test:

1. Mocking and setup

For each simulated adding action, several objects related to HTTP servlet requests are mocked using the Mockito framework, including HttpServletRequest, HttpServletResponse, HttpSession, and RequestDispatcher.

Return values for methods of these mocked objects are defined using the when method. This helps in controlling the behavior of these objects during testing.

2. Defining Request Data

Two separate sets of event data are defined for the two adding tasks. These datasets represent different events with distinct details such as venueId, eventName, startTime, and endTime to simulate actual event planners submitting a new event through a web page. Notice that the only data attributes that we care about are “venueId” and “startTime” and “endTime”, as those are the only data needed to simulate clashing of events, other attributes can be arbitrary as they do not affect test results.

By using these distinct datasets, the test checks the servlet's handling of concurrent requests for different events.

After the mock request is set up, it can be sent and run by the actual servlet, as always, Any exceptions thrown during the execution of the doPost method of the servlet are caught and printed for troubleshooting.

3. Concurrency simulation

An arbitrary number of events (set to 10 in this example) is used to determine how many threads will be spawned to simulate concurrent event-adding requests. A loop is used to create and start threads. Threads alternate between addEventTask and addEventTask2 to ensure variety in the concurrent requests being made. Once all threads are started, the main thread waits for each of them to complete using the join method. This ensures that all concurrent operations are finished before the test concludes.

Test cases:

1. Successful addition
    a. expected outcome:

       For each of the concurrent requests made by the threads, the system should successfully add the events with the provided data to the system. The two events would not have clashing venueId and overlapping time, thus both events should be added to the system.

After all threads are done executing, there should be a total of 2 new events in the system

b. Sample data:

addEventTask:

venueId: 5

eventName: NewTestEvent01

startTime: 2023-10-20 03:00:00

startTime: 2023-10-20 04:00:00

addEventTask2:

venueId: 5

eventName: NewTestEvent02

startTime: 2023-10-20 05:00:00

startTime: 2023-10-20 06:00:00

Other attributes of the sample data do not have any effect on the test results thus are not explicitly listed here.

Testing outcome:

Both “NewTestEvent01” and “NewTestEvent02” are added to the database with the correct attributes, no exceptions are thrown.

2. Clashing events
    a. expected outcome:

The two sample events have the same venueId and same start and end time, thus a clash happened. It was expected that only one event would successfully be added to the system after running the test

b. Sample data:

   addEventTask:

   venueId: 6

   eventName: NewTestEvent03

   startTime: 2023-10-20 07:00:00

   endTime: 2023-10-20 08:00:00

   addEventTask2:

   venueId: 6

   eventName: NewTestEvent04

   startTime: 2023-10-20 07:00:00

   endTime: 2023-10-20 08:00:00

   Other attributes of the sample data do not have any effect on the test results thus are not explicitly listed here.

   Testing outcome:

   Only "NewTestEvent03" is added to the database, it also does not overlap with existing events in the system.

2.3 Two event planners update a event and add a event to same venue and overlapping time simultaneously

4. **Description**

   In the Music Event System, a concurrency issue arises when an event planner adds a new event while another updates an existing one at the same venue with overlapping times simultaneously. Without robust controls, both operations might proceed concurrently, leading to double bookings at the same venue.

5. Choice of concurrency pattern and rationale

The concurrency pattern used to handle this issue is Pessimistic offline lock. The lock being used in the “add” operation is the same as the one in use case “1.2 Multiple Event Planners add events with the same venue and overlapping time (clash) simultaneously.”, and the same lock manager is also used to handle the concurrency in “updateEvent” operation. Integrating pessimistic locking for both "add" and "update" operations ensures seamless consistency. When updating, it's imperative that the system doesn't just look at current states but also considers potential incoming events that might be in the process of being added. Using the same locking strategy across both operations mitigates the risk of potential race conditions that could arise from using different concurrency controls. The advantage of using a pessimistic lock over an optimistic lock is discussed in detail in “1.2 Multiple Event Planners add events with the same venue and overlapping time (clash) simultaneously.”, in short, with a low number of event planners, the system does not need to handle much liveness and the consistency of correctness in pessimistic lock is preferred.

6. Implementation detail

“LockManagerEx” Class:

The LockManagerEx class implements a Singleton pattern, ensuring there's only one instance of the LockManagerEx throughout the application. The purpose of this class is to manage exclusive locks on resources (in this case, venues identified by venue IDs). Its key features include:

4. Uses a ConcurrentMap to keep track of which resource (venue) is locked and by whom (owner).
5. Provides a synchronized getInstance method to ensure only a single instance is created.
6. Offers acquireLock and releaseLock methods to handle the locking mechanism.

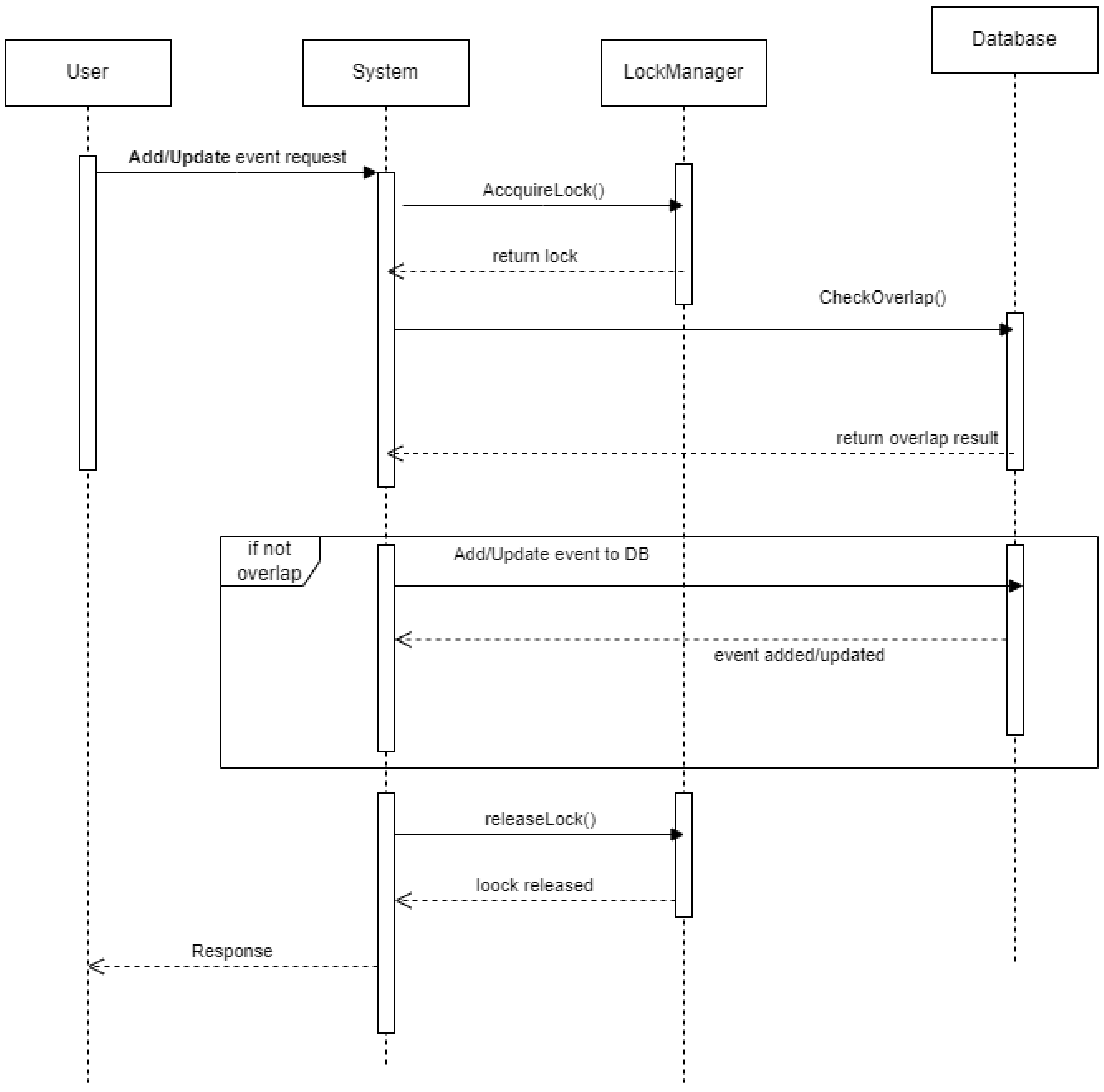
"add" and “updateEvent” Case Sequence in “EvenetPlannerServlet”:

acquireLock(): Before executing any operation that could potentially modify the data (either adding a new event or updating an existing one), the system tries to acquire a lock on the venue. If the venue is already locked by another operation, acquireLock() throws a RuntimeException.

Error Handling and Releasing Locks: If any exception occurs during the operation (e.g., a RuntimeException due to the lock, a database exception, etc.), the system handles it by forwarding the request to a JSP page with an error message. The lock on the venue is always released in the finally block, ensuring it is released regardless of whether the operation was successful or not.

Sequence Diagram:

A visualization of the pattern:



Testing strategy:

The strategy is to employ java multithreading to simulate real-world scenarios where two event planners might be trying to add events and updating events to the system concurrently. Tools like junit and mockito is used. Actual test code can be found in the "updateAddEventTest" file, below is a rundown of the test:

4. Mocking and setup

For each simulated updating action, several objects related to HTTP servlet requests are mocked using the Mockito framework, including HttpServletRequest, HttpServletResponse, HttpSession, and RequestDispatcher.

Similarly, another runnable is also set up to simulate the adding actions

5. Defining Request Data

Two separate sets of event data are defined for the adding and updating task. These datasets represent different events with distinct details such as venueId, eventName, startTime, and endTime to simulate actual event planners submitting a new event through a web page. Notice that the only data attributes that we care about are "venueId" and "startTime" and "endTime", as those are the only data needed to simulate clashing of events, other attributes can be arbitrary as they do not affect test results.

Notice that this time, the eventId of an existing event also need to be predefined so it can be successfully update

After the mock request is set up, it can be sent and run by the actual servlet, as always, Any exceptions thrown during the execution of the doPost method of the servlet are caught and printed for troubleshooting.

6. Concurrency simulation

Two threads, thread1 and thread2, are created. thread1 is assigned the addEventTask and thread2 is assigned the updateEventTask.

Both threads are started.

The latch.countDown() method is called, releasing both threads to execute their tasks simultaneously, ensuring concurrent execution.

The test waits for both threads to complete their execution using thread1.join() and thread2.join().

Test cases:

3. Clashing “update” and ”add”
   a. expected outcome:

      To validate the system's behavior when there is a concurrent attempt to add an event with the same VenueId and start and end time as an existing event that is simultaneously being updated.

      The system should successfully update the existing event with the new startTime and endTime

      The system should reject or raise an error for the add operation due to the venue and time clash.

   b. Sample data:

      addEventTask:

      venueId: 23

      eventName: NewTestEvent05

      startTime: 2023-10-20 03:00:00

      endTime: 2023-10-20 04:00:00

      updateEventTask:

      eventId: 96

      venueId: 23

      eventName: OldEvent

      startTime: 2023-10-20 03:00:00

      endTime: 2023-10-20 04:00:00

      Other attributes of the sample data do not have any effect on the test results thus are not explicitly listed here.

Testing outcome:

The system successfully updated the existing event with new time while new event were not added to database.

2.4 Two event planners update the same event at the same time

1. Description:

In the music event booking system, the different event planners can be assigned with the same event by the administrator and update the same event. Therefore, there is a possibility that the same event gets modified by different event planners at the same time. This can cause some concurrency problems such as lost updates which means that the update made by the first event planner to the event is overwritten by the second event planner's update and the first update is lost.

This issue occurs as the two event planners read the same event with the same information from the database. However, as they update the event concurrently, the transaction that gets run first will update the information of the event from the database, the event read by the second event planner does not get updated. When the event gets updated by the second event planner, the first update to the event has been lost.

2. Choice of pattern,

In order to resolve the problem, we decided to use Optimistic offline lock which allows multiple updates to the same event concurrently.

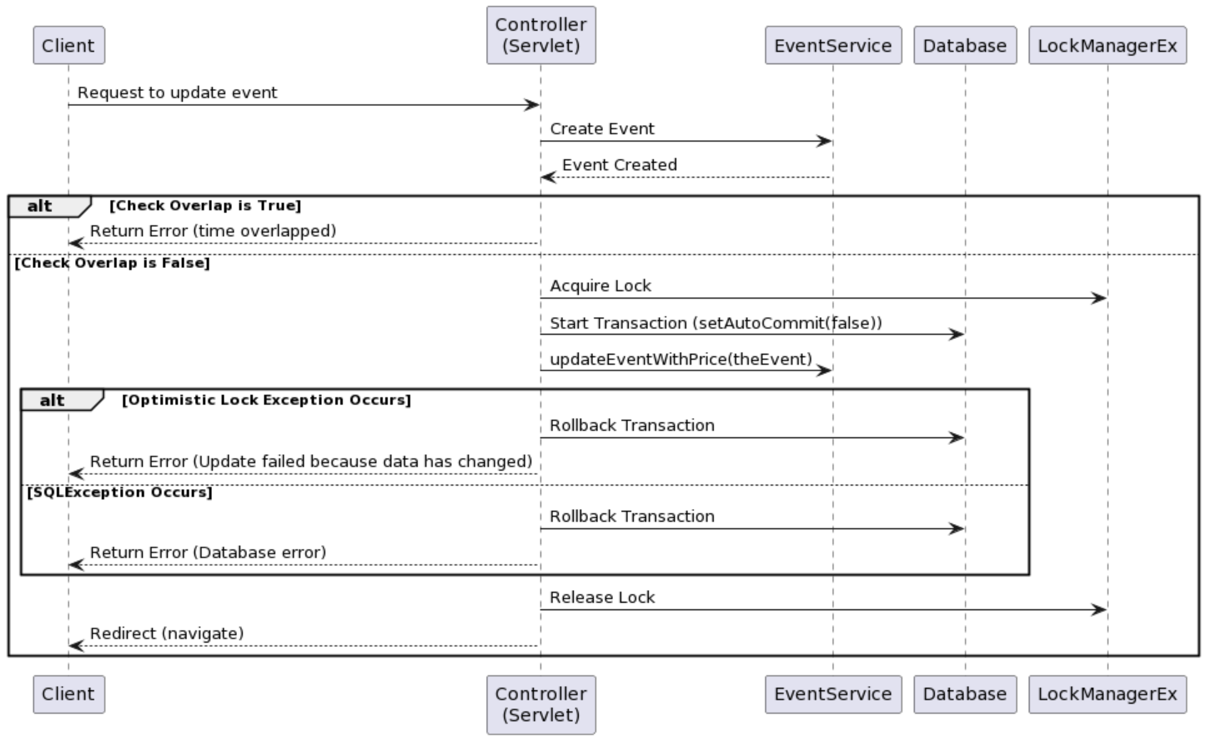
3. Rationale for the pattern.

Different from the pessimistic lock which locks the resources whenever the user is performing some actions, optimistic lock will allow all users to view and work on the shared resources concurrently, the lock will only be added when the data is committed. This can significantly reduce the time of the resources being locked which therefore ensure good liveness of the system especially for a large enterprise system.

Moreover, the optimistic offline lock is easy to implement and is more scalable in a large system which requires numerous simultaneous users. The issue with the lock is that the conflict is detected at the commit time, which means that any update made by the second event planner to the event will be lost as this transaction is rolled back. However, because there is not much information needed to be updated for a single event, the lost update problem is acceptable.

4. Implementation:

Concurrency Handling Sequence Diagram



Client
Controller (Servlet)
EventService
Database
LockManagerEx
Request to update event
Create Event
Event Created
alt [Check Overlap is True]
Return Error (time overlapped)
[Check Overlap is False]
Acquire Lock
Start Transaction (setAutoCommit(false))
updateEventWithPrice(theEvent)
alt [Optimistic Lock Exception Occurs]
Rollback Transaction
Return Error (Update failed because data has changed)
[SQLException Occurs]
Rollback Transaction
Return Error (Database error)
Release Lock
Redirect (navigate)

To implement this lock, we need to add an additional version number for each event in the database to keep track of the number of times each event has been modified by the event planner. Once the event needs to be updated, it first gets read by the event planner, before the update gets committed, the version of this record is compared with the current version of this event in the database. If the versions are the same which indicates that the event has not been modified by another event planner before the event planner commits the change. Therefore the event planner can safely commit the update. If the version number read by the event planner does not match the version number of the same event in the database, this implies that the event has been updated by another event planner, in this case, the current transaction needs to be rolled back.

5. Testing strategy:

We use Jmeter to test the update event functionality. The http request simulates the action of two event planners login to the system and then modify the same event.

Test Data:

Event planner1:

Username: eventplanner

Password: eventplanner

Event planner2:

Username: EventPlaner1

Password: EventPlanner1

Event gets modified:

Event name: event20

Event id: 20

Modified by the event planner 1:

Event name: event_updated 20

Artist name: artist_968

Modified by the event planner 2:

Artist name: artist_007

6. Summary of test outcome

We run the test for 20 threads with only one loop. After the test, the expected result should be that only one update is successfully applied to the event and others are rejected by the optimistic lock. The unsuccessful transaction will be rolled back.

2.5 Event planners delete the same event at the same time.

1. Description:

The system allows the two event planners to have access to the same event, therefore similarly to the last issue, two event planners can delete the same event at the same time which can cause concurrency issues. We should prevent two or more event planners from deleting the same event at the same time.

To do so, the idea is to block other event planners from editing events while one event planner is deleting the event. This issue occurs as event planners all read the same data from the event database, while one of the event planners delete the event, other event planners still have the last read event list. The inconsistent read causes the issue.

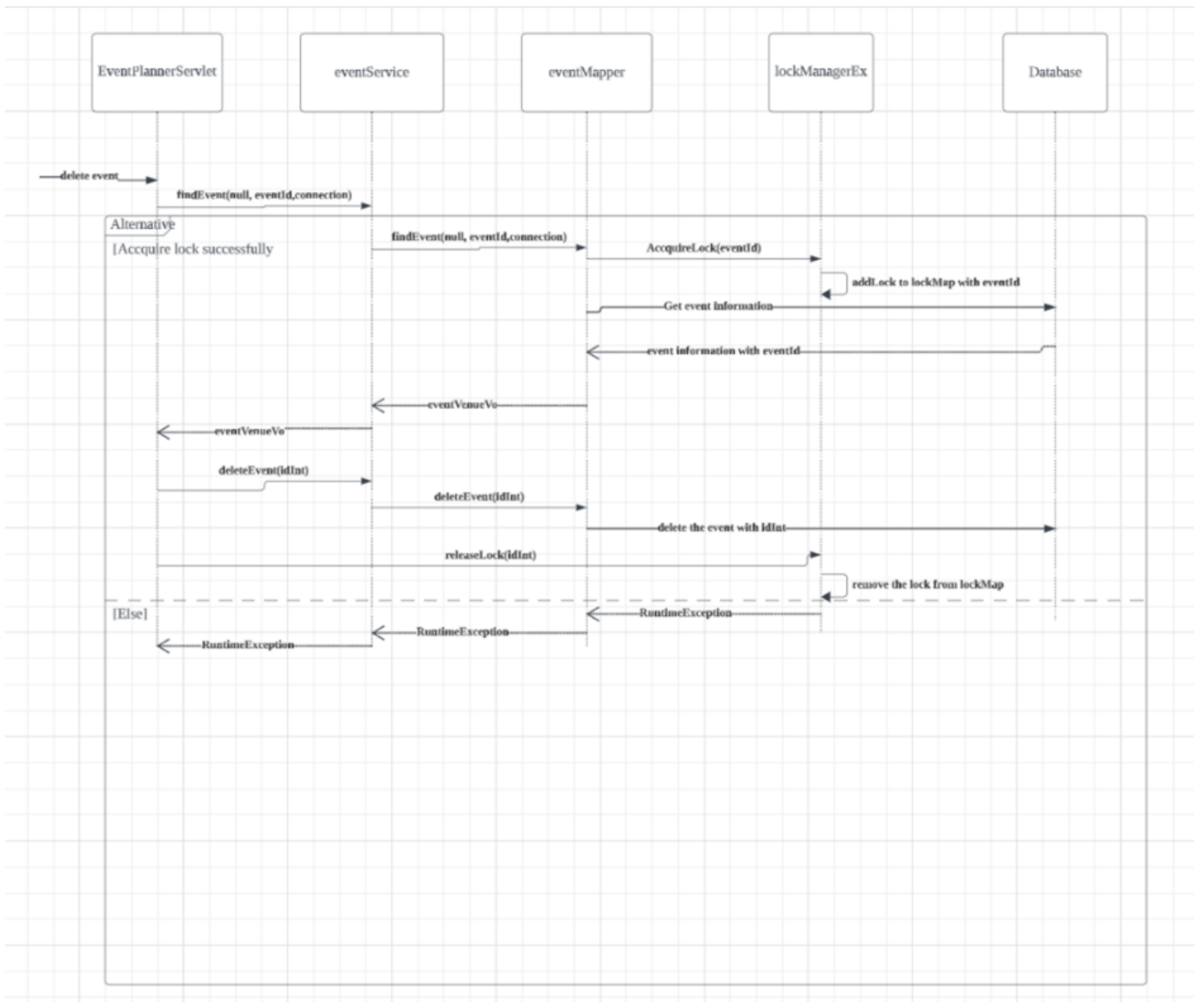
2. Choice of pattern,

We decide to use pessimistic offline lock to deal with this issue.

3. Rationale for the pattern,

We decide to use the pessimistic lock as it can totally prevent the conflict of deleting the same event from happening. Unlike the optimistic lock which only detects the conflict while the transaction is committed. Once the conflict is detected, then one of the transactions is rolled back. In our case, once one of the event planners decides to delete the event, we put the pessimistic read lock which rejects other event planners to access the same event. The pessimistic lock helps to ensure the consistency of the system as a consequence of reducing some liveness.

4. Implementation detail:



To implement the lock, we need to first set up the lock manager which grants or denies any lock requests, the lock manager contains a lock map which maps the locks to owners of corresponding locks. The lock manager uses a singleton pattern to ensure only one manager is accessed. Then, we put a pessimistic lock to the method findEvent in the event mapper class, the system acquires the lock when accessing this method and releases the lock when the delete of the event finishes. By adding the lock, other transaction which need to delete the event will be first checked by the lock manager, if the second event planner tries to lock the same event (distinguish by the event id), the manager checks if the lock map contains the event id of the second event already, if so, then the lock manager throws the runtime exception to indicate the second event can not be locked. Therefore, the second deletion is blocked.

5. Testing strategy:

To test this function, we use Jmeter with two requests in two different thread groups which simulate the actions of two event planners deleting the same event. Two event planners login to the system and then view the event list. Finally, deleting the event with the same event id.

Test Data:

Event Planner 1:

Username: eventplanner

Password: eventplanner

Event Planner 2:

Username: EventPlanner1

Password: EventPlanner1

Event:

eventide: 28

venueId: 9

6. Summary of test outcome

We choose to use 5 threads for each thread group. Each event planner tries to delete the event with id of 28.

The expected result should be that there is only one request of deleting the event is successful.

The outcome of the Jmeter test implies that only one event is deleted by the event planner, other deleting requests are blocked by the pessimistic read lock.

2.6 Client books tickets while the event planner deletes the event.

1. Description:

In the music event system, the client can book a ticket which is related to the booking of the event. The event can have many bookings which corresponds to many tickets booked by the client. Therefore, there is one possible issue that the client books the ticket on one event while the event planner deletes the event concurrently. This can cause problems when booking tickets to the event which does not exist anymore. To resolve the issue, we need to prevent the event planner from deleting the event while the client is purchasing a ticket on that event.

This issue is happening because the system first gets the event information from the database before the event is deleted. After the event is deleted, the system still has the same information of the event and displays it to the client, the client gets the event information and proceeds the booking of this event.

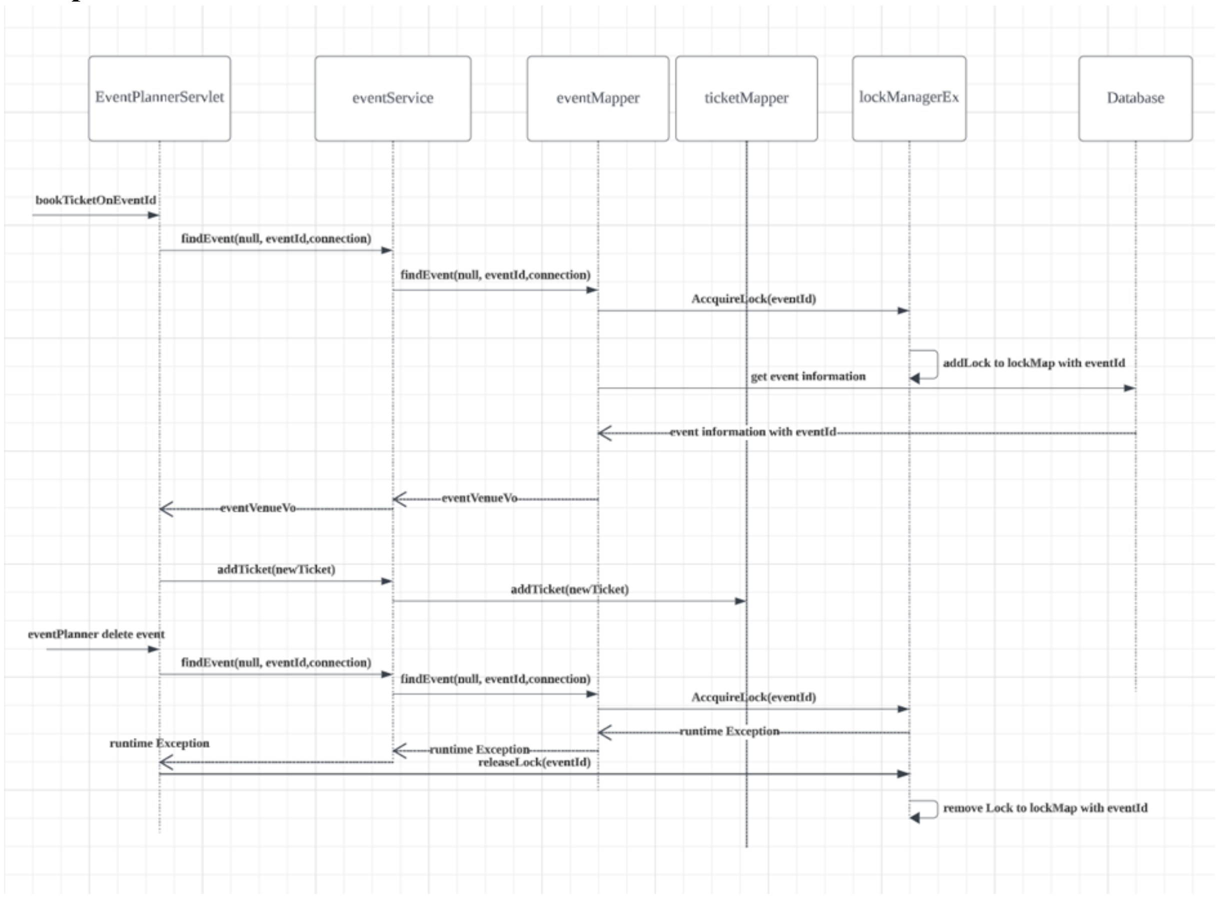
2. Choice of pattern:

We decide to use pessimistic offline lock to deal with this problem.

3. Rationale for the choice of pattern:

For this function, because both event planner and client read events from the database first, we decide to put a pessimistic read lock which fully rejects the event planner while the client is booking a ticket. Moreover, because the probability of conflict between a client adding a ticket and the event planner deleting the same event is high, we also do not want to roll back the ticket transaction which is painful for the client. Overall, the pessimistic lock is a suitable choice.

4. Implementation detail:



To implement this lock, we first acquire the lock from the lock manager in method findEvent in Event Mapper class. This lock is to block the event planner from reading the event in the database while the client is booking a ticket. After the client finishes booking, the client releases the lock and the event planner can continue to delete the event.

Testing strategy:

To test this feature, we use Jmeter with one request in two different thread group which simulate the action of one event planner delete the event while one client books the ticket on that event.

5. Test Data:

Client:

Username: client

Password: client

Event Planner 1:

Username: eventplanner

Password: eventplanner

Event Planner 2:

Username: EventPlanner1

Password: EventPlanner1

Ticket to buy:

eventide: 26

buyVipcapacity: 1

buyMoshcapacity: 1

buyStandingcapacity: 0

buySeatedcapacity: 0

totalAmoumt 7

Event to delete:

Id: 26

venueId: 9

6. Summary of test outcome

We decide to use 5 threads and run one loop for both thread groups. Each event planner deletes the event for one time and the client tries to buy one vip and one mosh ticket at a time.

Summary of the result of test:

The result indicates that the event planner can not delete the event from the database while the client is purchasing a ticket on the event. The delete event action violates the pessimistic lock and the lock manager generates an error message.

2.7 Event Planners and clients cancel the same booking at the same time.

1. Description:

In the system, event planners and clients both have access to the booking, the event planner has access to all the booking corresponding to each event, the client can also view and cancel his own booking. Therefore, one concurrency issue may arise as both event planner and client cancel the same booking at the same time.

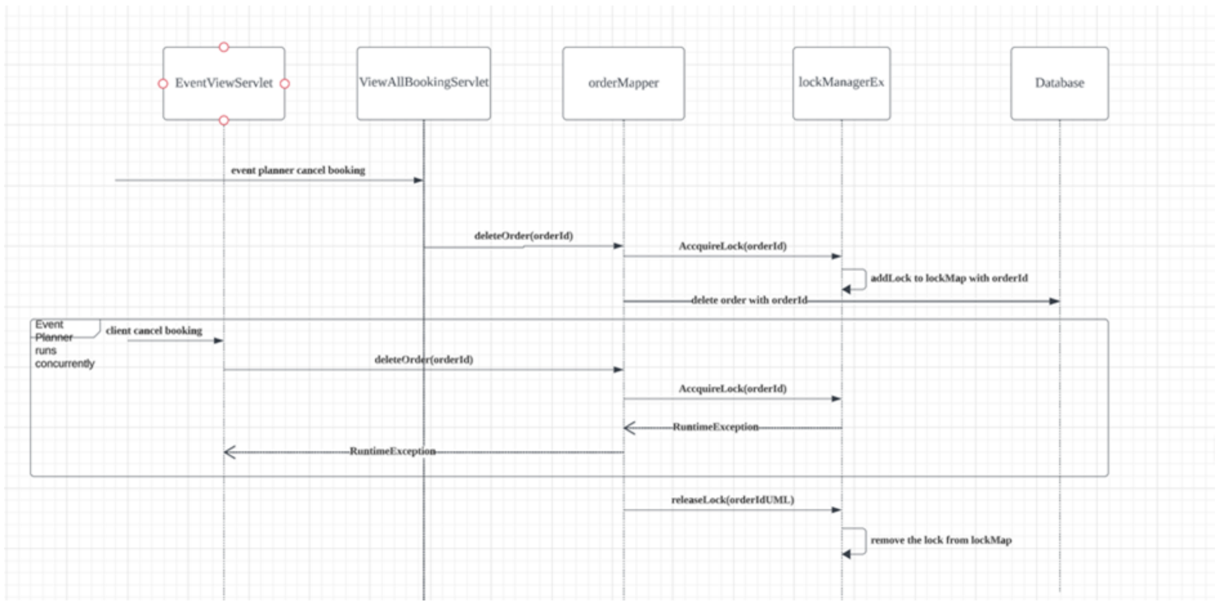
2. Choice of pattern:

We decide to use a pessimistic lock to handle the issue.

3. Rationale for the pattern:

Unlike optimistic locks where conflicts are detected at the end of the transaction which require additional logic to resolve such as adding version number to the database. The pessimistic locking requires less change to the original code and avoids any rollbacks and retries by ensuring that conflicts do not occur in the first place. Moreover, pessimistic locks also ensure that multiple transactions do not interfere with each other, thereby maintaining the integrity of the data.

4. Implementation:



By adding the pessimistic lock in the deleteOrder method in the order Mapper class, and release the lock after deleting the order. As shown in the sequence diagram above, the event planner first deletes the order which acquires the order locking to this specific order via lockManagerEx. As the client also tries to delete the same order, the system acquires another lock on the order which is rejected by lockManagerEx and generates a Runtime Exception. Once the order is deleted successfully by the event planner, the lock is released and removed from the lock map.

5. Testing strategy:

Using Jmeter with two different thread groups which simulate the event planner and the client. Both event planner and client cancel the same event.

Client:

Username: client

Password: client

eventide: 79

orderId: 2

Event Planner:

Username: eventplanner

Password: eventplanner

EventId: 79

orderId: 2

6. Summary of test outcome

We decide to use 5 threads and run one loop for two thread groups. The test outcome is expected as there is only one thread group eg: event planner can successfully cancel the booking where the other one cannot. There is no error message generated when the delete is blocked by the lock.