SWEN90007 Software Design and Architecture
Part 3 Report

Team: Green Day

GitHub Release Tag: SWEN90007_2022_Part3_GreenDay

Quanchi Chen	quanchic@student.unimelb.edu.au	quanchic
Yijie Xie	yijxie@student.unimelb.edu.au	yijxie
Wenxuan Xie	wexie2@student.unimelb.edu.au	wexie2
Jingning Qian	jingningq@student.unimelb.edu.au	jingningq

This document specifies the updated design and architecture of the music events system developed by the team Green Day, explaining its latest logical view, the underlying database schema, recognised concurrency issues and implemented concurrency patterns. This document also serves as the common understanding among all the team members and should help new developers quickly understand the design rationale behind the concurrency issues and patterns.

Updated Database Schema and Logical View

This section describes the updated physical entity-relationship diagram (ERD) and the class diagrams.

Figure 1 is the latest physical ERD demonstrating the database schema. To implement optimistic concurrency patterns, we add one more column named ***version_number to*** the tables Events and Sections, respectively.

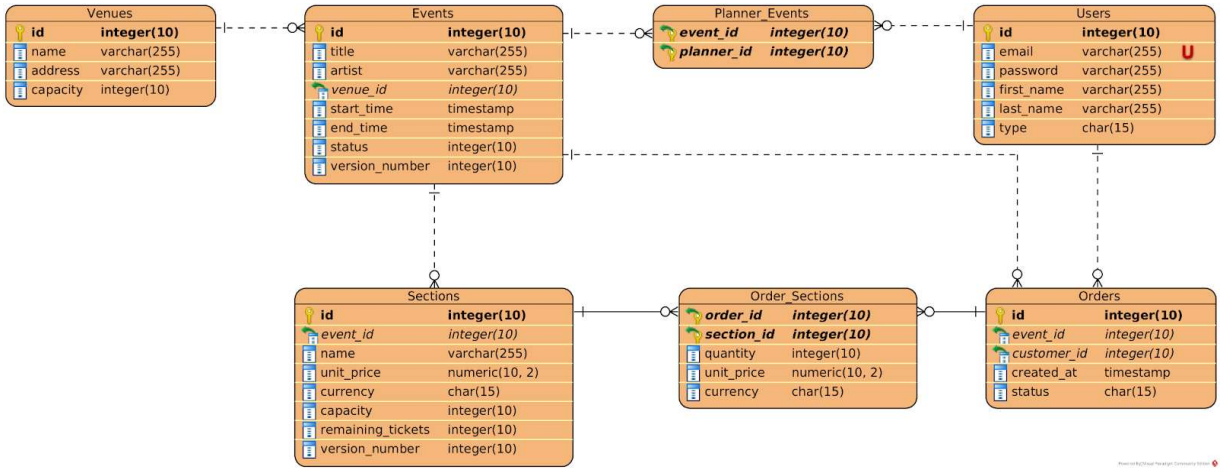


Figure 1 - Latest Physical ERD

Figure 2 is the latest class diagram for Servlets/Controllers. Based on the tutor's feedback, we specify multiplicities to the associations.

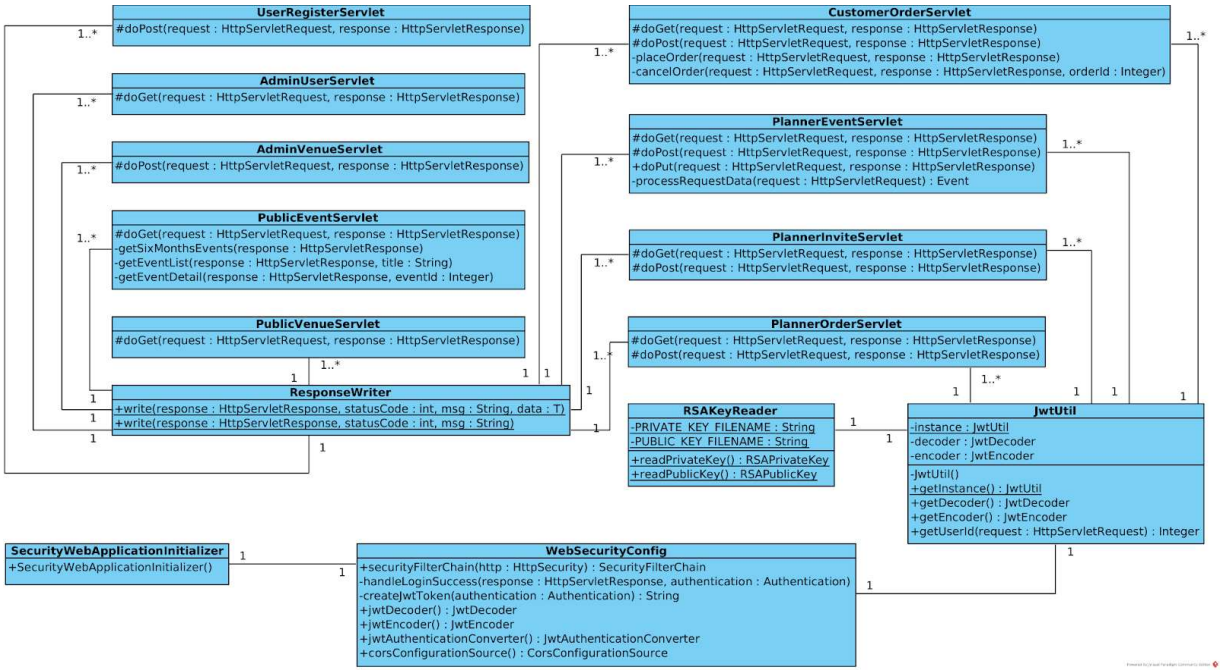


Figure 2 - Latest Class Diagram (Servlets)

Figure 3 is the latest class diagram for services. We provide one more class, LockManager, used by the pessimistic offline locking pattern to this diagram and add multiplicities to the associations. We also remove the redundant overridden methods from the concrete classes to increase readability.

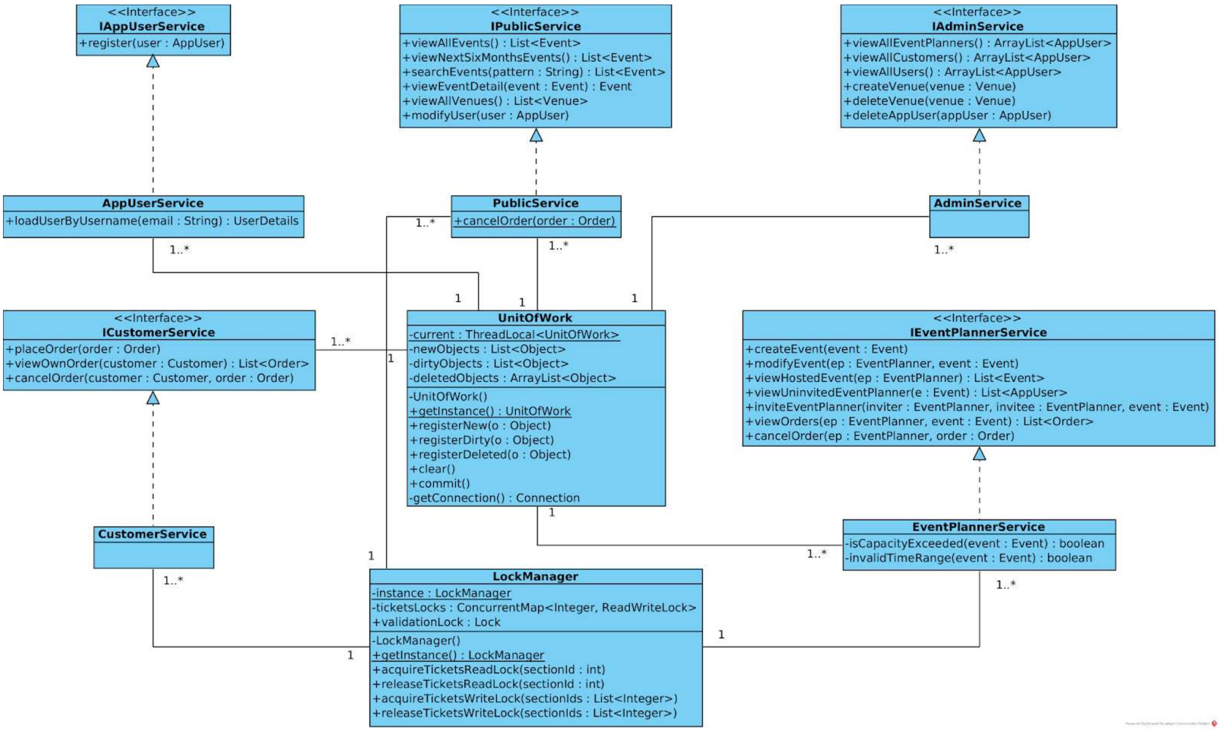


Figure 3 - Latest Class Diagram (Services)

Figure 4 is the latest class diagram for domain objects. We remove the getters and setters to increase the readability and add multiplicities to the associations.

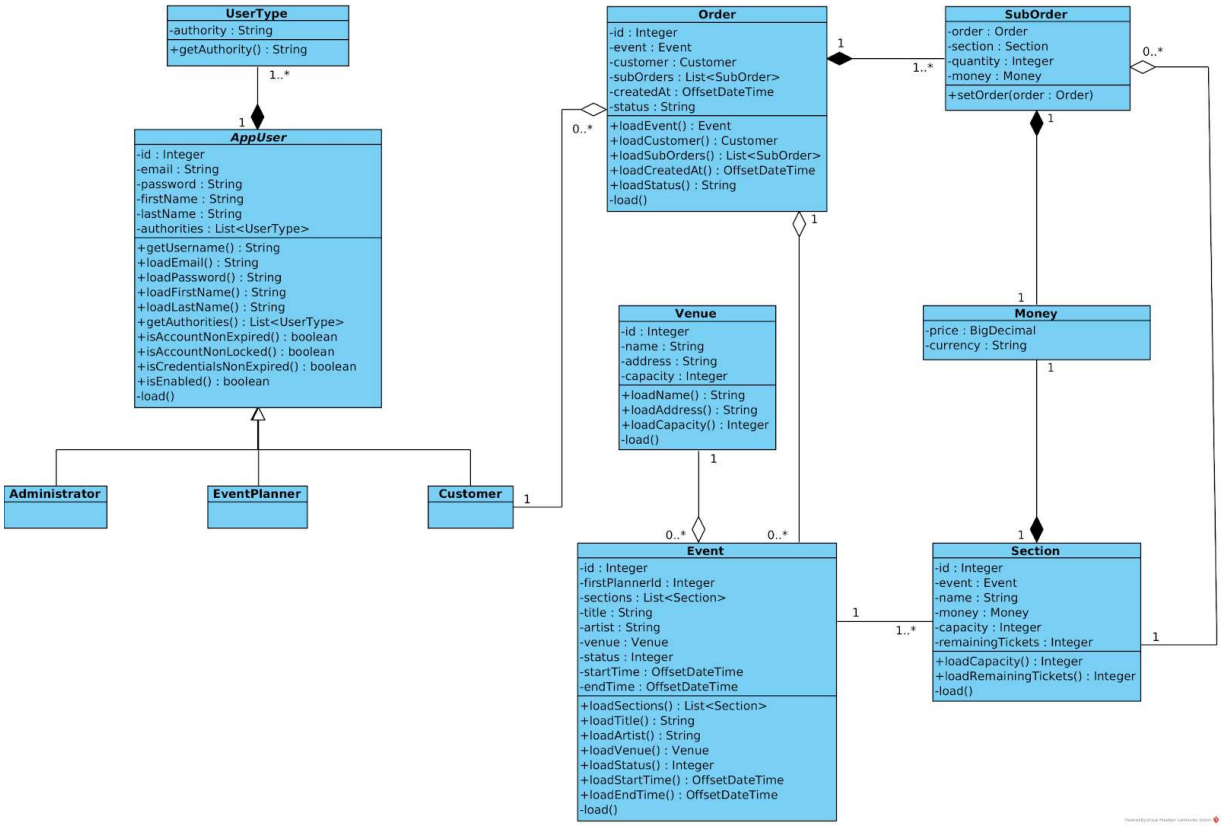


Figure 4 - Latest Class Diagram (Domain Objects)

Figure 5 is the latest class diagram for data mappers. We provide one more class related to concurrency control, LockManager, and add multiplicities to the associations.

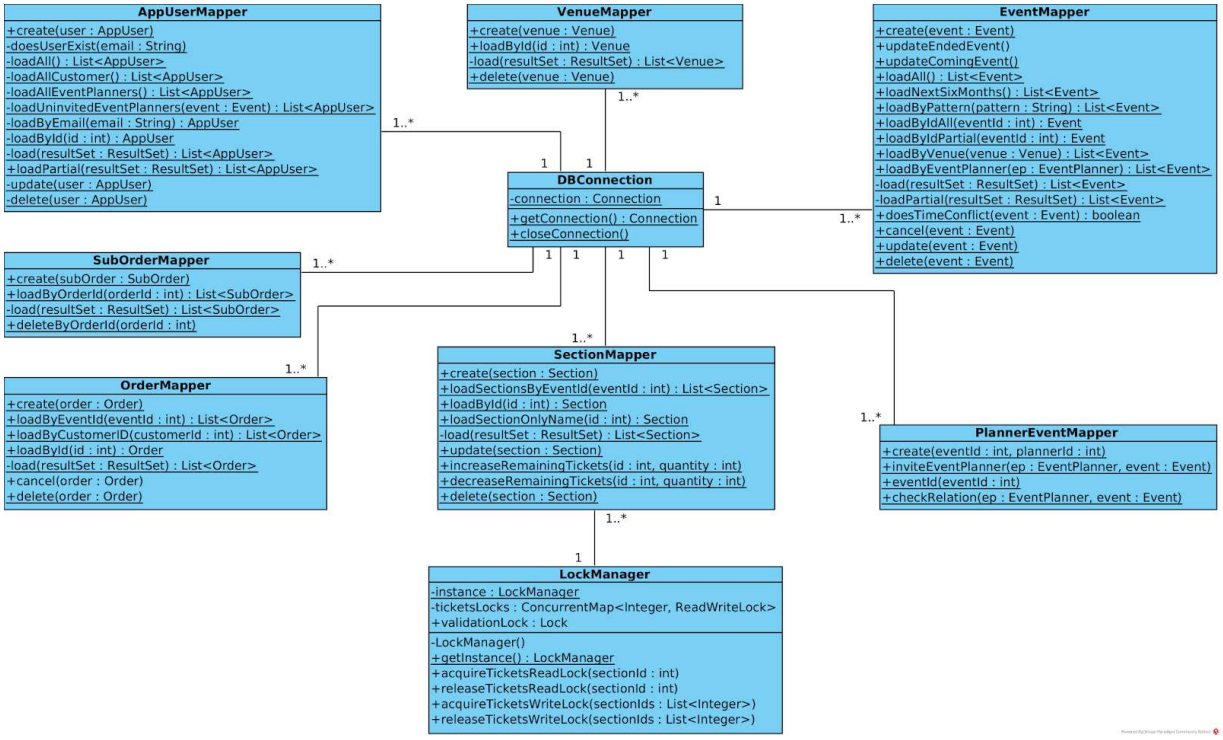


Figure 5 - Latest Class Diagram (Data Mappers)

Concurrency Issues, Patterns and Design Rationale

We identify four concurrency issues in the system, and we use a combination of **optimistic lock** and **pessimistic lock** to tackle them. Specifically, we add a `version_number` for the `events` and `sections` tables for version management to aid our optimistic lock, and we use **read-write lock** for pessimistic lock implementation. To make the code logic clear and well-structured, we implement the optimistic lock in our `UnitOfWork` class and apply pessimistic locks where necessary in the service layer. To enable the rollback feature of JDBC and SQL for our optimistic lock, we wrap all SQL queries for each of our business logics into a single transaction, with the isolation level set to the default value READ COMMITTED which prevents dirty reads under any circumstances.

Issue 1

Multiple customers may concurrently purchase tickets for the same section of the same event. Without proper concurrency management, lost update of the `remainingTickets` field of the `sections` table may occur, leading to inconsistency of data. Addressing this concurrency issue, we use **read-write lock** to maintain the integrity of the number of remaining tickets in the database, with which the process of ticket purchase of a customer will be blocked while another transaction is in progress simultaneously until it is committed. Nonetheless, read operations of the remaining tickets from other processes remain non-blocked as long as no update on the corresponding sections is in progress. The implementation detail of the lock is reflected in the

following sequence diagram, illustrating how the concurrency issue is handled in scenario where two customers is buying tickets for the same section of the same event:

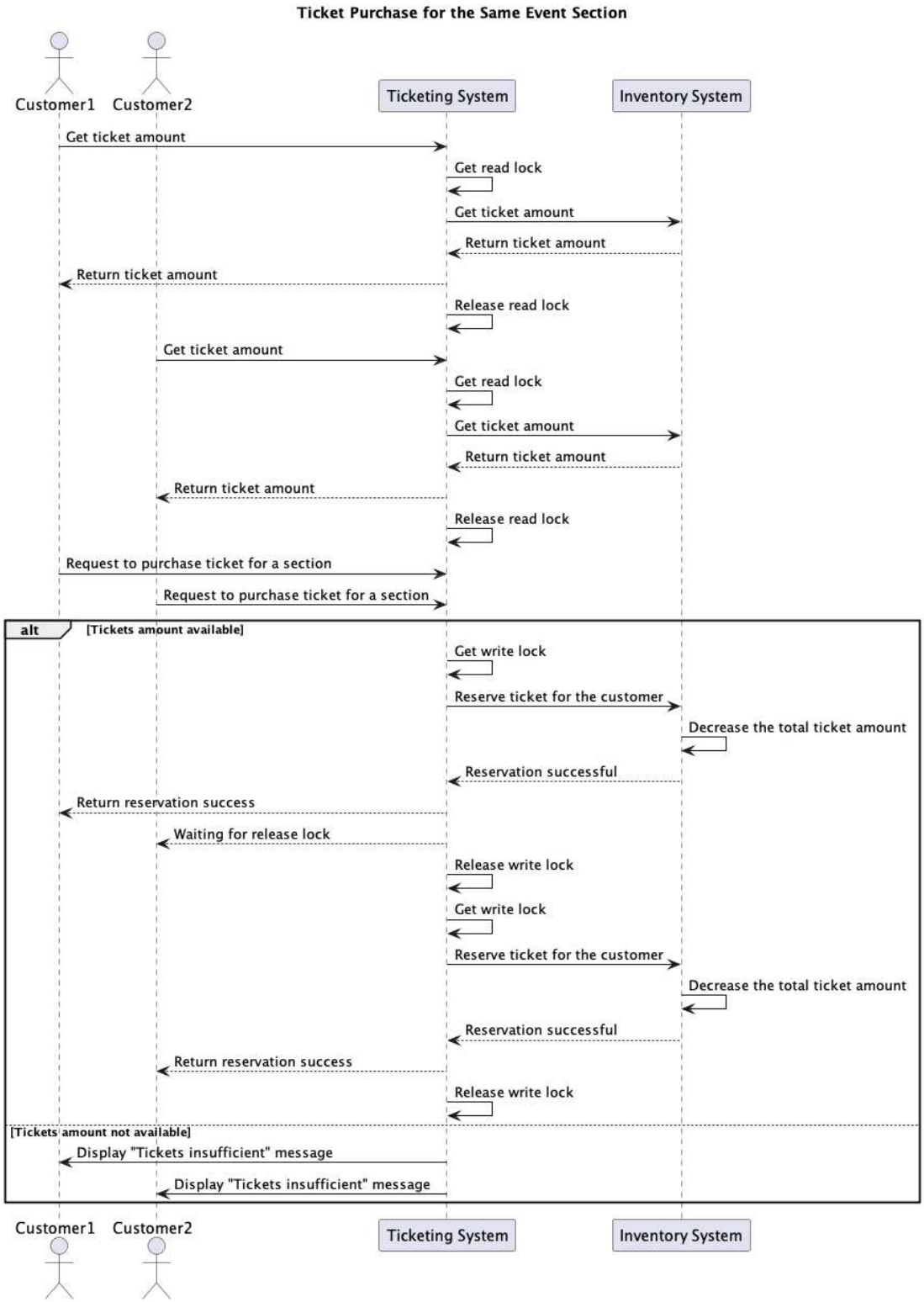


Figure 6 - Ticket purchase for the same event section

To justify our choice of read-write lock pattern, we expect high concurrency and frequent change in the number of remaining tickets for the ticket purchase use case. Besides, data consistency is of top priority for this use case, as selling more tickets than the available quantity is undesirable in our system. Apparently, optimistic lock does not work in this scenario, since customers would never expect frequent failure responses of ticket purchase attempts due to unmatched versions. On the other hand, read-write lock preserves customers' work through the user interface maximally while guaranteeing data consistency. Meanwhile, read-write lock enhances read throughput by read locks and thus improves user experience greatly. Therefore, we adopt read-write locks to gracefully handle this concurrent ticket purchase issue.

Issue 2

Multiple event planners may simultaneously select the same venue and time to hold their events. Concurrency issues arise if two event planners concurrently pick the same venue and create the different event occupying overlapping time slots, resulting in time clashes in our database, which is undesirable by our design. We employ **optimistic locks** for the create event use case. Despite the chance of the concurrency issue described, the operation will only be successful for the first committed request, while the other request ends up failing due to the clash of event venue and time, triggering the rollback of its transaction. This scenario is illustrated in the sequence diagram below:

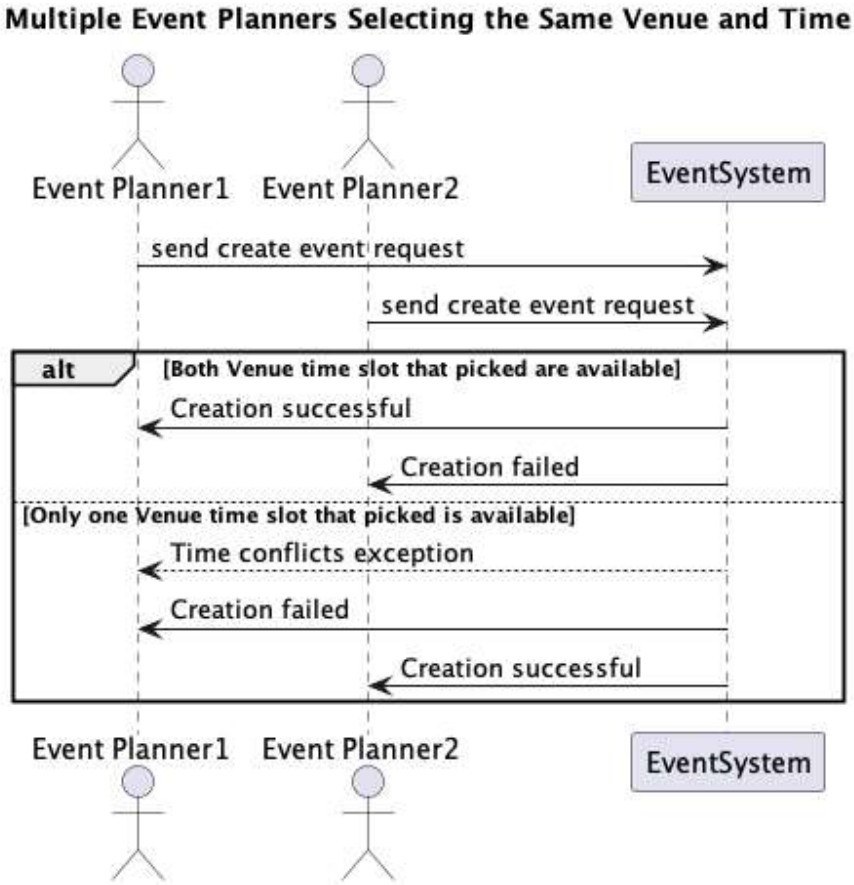


Figure 7 - Event planners selecting the same venue and time

Creating a new event involves querying a number of rows in the `events` table. As our business logic centres around music events, applying pessimistic locks when creating new events or elevating the database isolation level may greatly compromise the liveness of our application. Moreover, the number of event planners is significantly smaller as compared to customers in our system, which further mitigates potential concurrency issues of event planner use cases. Considering the likelihood of two event planners simultaneously creating events that clash on venues and times is relatively low, it is not a bad idea to insert a new event into the table anyway and then check if any clash exists. The cost of rolling back is negligible since the chance of reaching this point is fairly low. Such an optimistic lock mechanism provides our application a simple and lightweight solution which addresses both liveness and data consistency.

Issue 3

Multiple event planners of the same event may modify that event simultaneously. This use case involves the update of tables `events` and `sections`. Considering two event planners trying to modify the same event in parallel, if one of the requests ends up committed first, the other request will simply overwrite the previous update, causing a lost update issue. We adopt

optimistic locks to address this concurrency issue. By applying an optimistic lock, an event update request will perform the update anyway before validating the version numbers. If the version does not match for a specific event or section record, the whole transaction will be rolled back to guarantee data consistency. The relevant the sequence diagram is attached as follows:

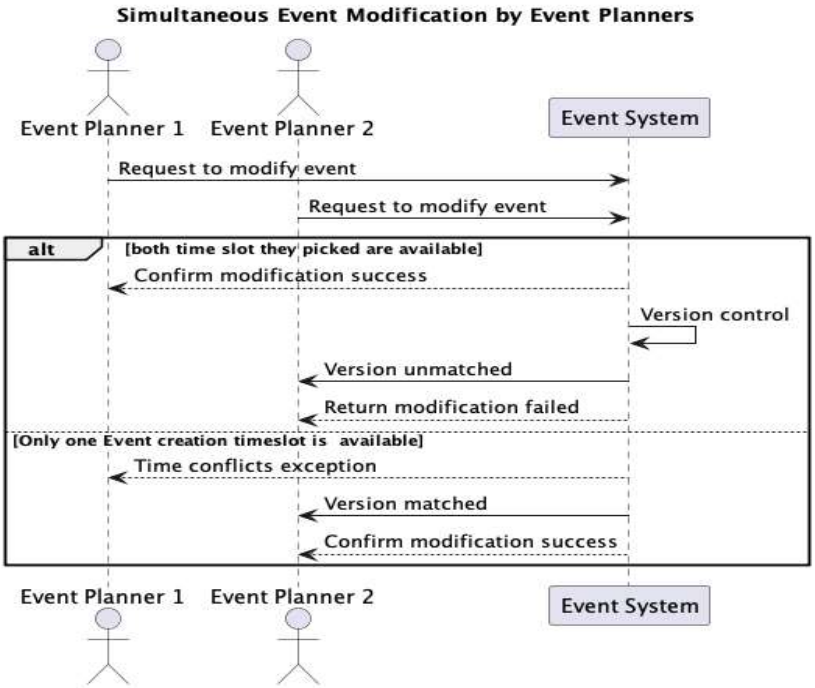


Figure 8 - Simultaneous Event modification by event planners

Although event planners in our system are given the privilege to modify events and associated sections, they are not encouraged to do so as customers are easily frustrated by frequent rearrangements. Instead, we expect only slight changes of posted events for this use case, such as adjusting prices or rescheduling an event. Nonetheless, we still want to maintain the data integrity of these two tables while maximally preserving liveness of the whole system. In this scenario, an optimistic lock with version management is sufficient for protecting the data integrity. On the other hand, pessimistic locks will only create unnecessary implementation overhead while compromising overall system responsiveness in this case. Moreover, similar to creating an event, this use case also involves altering the venue or datetime of an event which may lead to clash with another event created or updated simultaneously. An optimistic lock addresses such concurrency problems well as discussed in Issue 2.

Issue 4

An event planner and a customer may cancel the same order simultaneously. Since our system will return the exact quantities of tickets in the order back to the `sections` table in the database when an order is cancelled, altering the `remaining_tickets` field of the `orders` table, the consistency of the `remainingTickets` field needs to be maintained similar to the Issue 1 discussed. Therefore, in consistency with the ticket purchase use case, we adopt **read-write**

locks to safeguard the data integrity. The Figure 9 reflects how read-write locks deal with the described concurrency issue.

As discussed previously, cancelling an order triggers alteration of the `remaining_tickets` field of the `orders` table in the database. Despite that these actions themselves are rarely parallelled, the risk of data inconsistency is still greatly increased considering their potential concurrent executions with the heavy traffic of customers buying tickets. Same as the Issue 1, we again prefer consistency over liveness in this scenario. Therefore, we adopt pessimistic locks implemented with read-write locks in accordance with the create order use case, with the read lock mitigating the read availability of the `remaining_tickets` column.

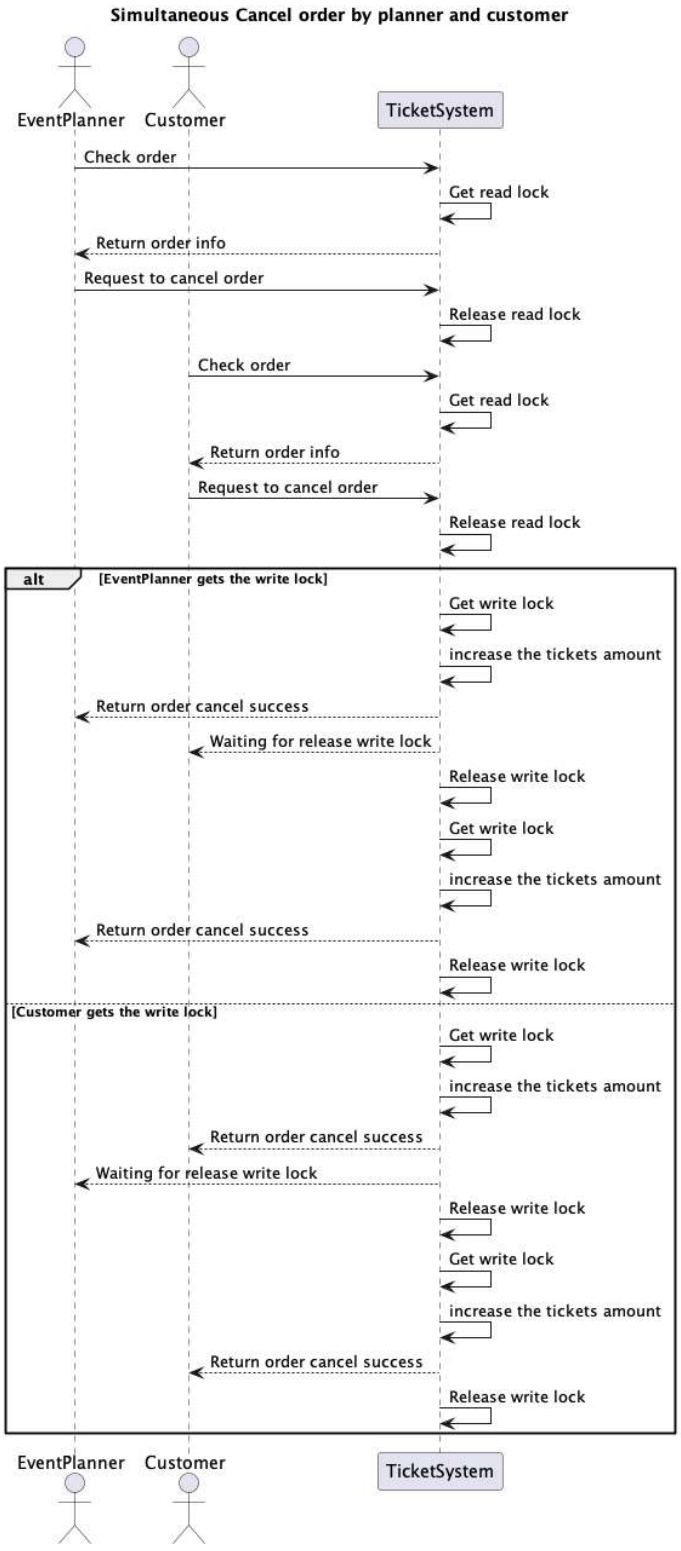


Figure 9 - Simultaneous cancel order by planner and customer

Testing Strategy and Outcomes

This section describes the concurrency testing strategy. The testing code with detailed comments resides in the *edu.unimelb.swen90007.mes.service.test* package.

Below are brief descriptions of employed classes.

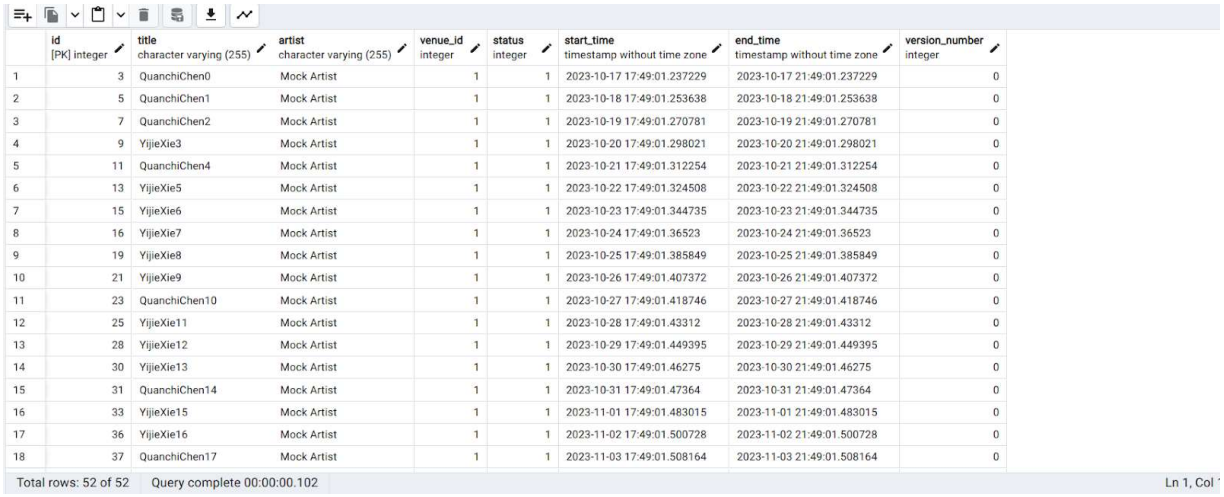
1. EventPlannerThread extends Thread and simulates a general event planner who can create events, invite other planners, and update hosted events.
2. CreateEventThread extends EventPlannerThread and simulates an event planner who only creates events and invites other planners in the concurrency test.
3. ModifyEventThread extends EventPlannerThread and simulates an event planner who can also update hosted events in the concurrency test.
4. CustomerThread extends Thread and simulates a customer who places orders.
5. CancelOrderThread extends Thread and simulates a customer to cancel all placed orders of a particular customer.

First, the testing program spawns and waits for two CreateEvent threads, each tending to create fifty events in the same venue. The aim is to check if the problem of interference happens when multiple event planners simultaneously select the same event venue and time to hold their events. The correct result should be that the table Events only contains fifty records since the software system fails to create another fifty events due to time conflict.

Second, the testing program instantiates two ModifyEvent instances and lets them each create a valid event and invite each other. The testing program then starts the two ModifyEvent threads and spawns two Customer threads that place orders for existing events. We observed the message “ERROR - UoW commit error: Version Unmatched” in the server log.

Finally, spawn two CancelOrder threads to cancel the orders of the same customers. The aim is to check if the remaining tickets remain consistent when multiple cancel operations happen simultaneously.

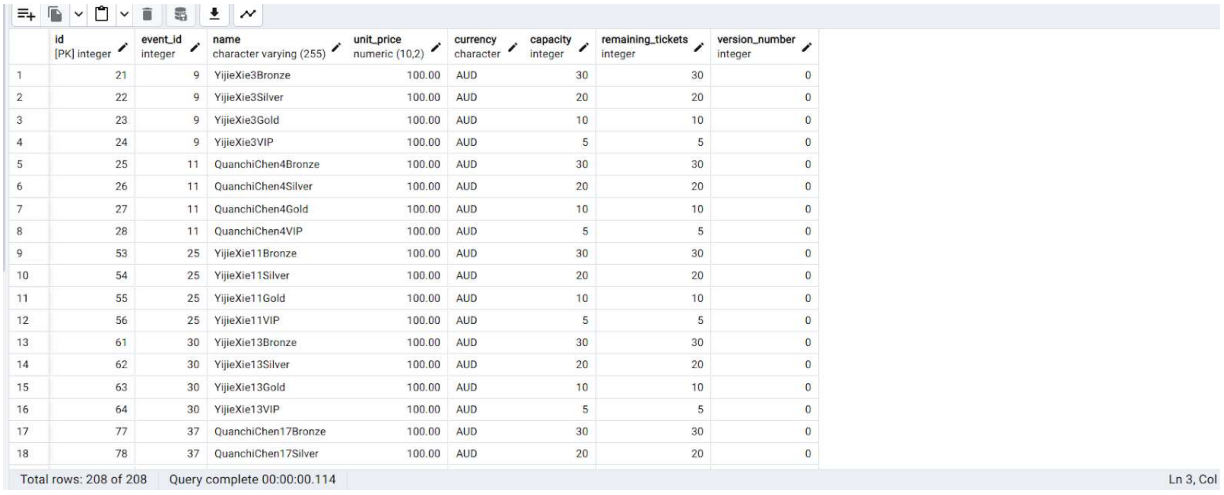
After inspecting the database, we believe the software system passes the concurrency test. Below are the screenshots of the two crucial tables after the above operations.



	id [PK] integer	title character varying (255)	artist character varying (255)	venue_id integer	status integer	start_time timestamp without time zone	end_time timestamp without time zone	version_number integer
1	3	QuanchiChen0	Mock Artist	1	1	2023-10-17 17:49:01.237229	2023-10-17 21:49:01.237229	0
2	5	QuanchiChen1	Mock Artist	1	1	2023-10-18 17:49:01.253638	2023-10-18 21:49:01.253638	0
3	7	QuanchiChen2	Mock Artist	1	1	2023-10-19 17:49:01.270781	2023-10-19 21:49:01.270781	0
4	9	YijieXie3	Mock Artist	1	1	2023-10-20 17:49:01.298021	2023-10-20 21:49:01.298021	0
5	11	QuanchiChen4	Mock Artist	1	1	2023-10-21 17:49:01.312254	2023-10-21 21:49:01.312254	0
6	13	YijieXie5	Mock Artist	1	1	2023-10-22 17:49:01.324508	2023-10-22 21:49:01.324508	0
7	15	YijieXie6	Mock Artist	1	1	2023-10-23 17:49:01.344735	2023-10-23 21:49:01.344735	0
8	16	YijieXie7	Mock Artist	1	1	2023-10-24 17:49:01.36523	2023-10-24 21:49:01.36523	0
9	19	YijieXie8	Mock Artist	1	1	2023-10-25 17:49:01.385849	2023-10-25 21:49:01.385849	0
10	21	YijieXie9	Mock Artist	1	1	2023-10-26 17:49:01.407372	2023-10-26 21:49:01.407372	0
11	23	QuanchiChen10	Mock Artist	1	1	2023-10-27 17:49:01.418746	2023-10-27 21:49:01.418746	0
12	25	YijieXie11	Mock Artist	1	1	2023-10-28 17:49:01.43312	2023-10-28 21:49:01.43312	0
13	28	YijieXie12	Mock Artist	1	1	2023-10-29 17:49:01.449395	2023-10-29 21:49:01.449395	0
14	30	YijieXie13	Mock Artist	1	1	2023-10-30 17:49:01.46275	2023-10-30 21:49:01.46275	0
15	31	QuanchiChen14	Mock Artist	1	1	2023-10-31 17:49:01.47364	2023-10-31 21:49:01.47364	0
16	33	YijieXie15	Mock Artist	1	1	2023-11-01 17:49:01.483015	2023-11-01 21:49:01.483015	0
17	36	YijieXie16	Mock Artist	1	1	2023-11-02 17:49:01.500728	2023-11-02 21:49:01.500728	0
18	37	QuanchiChen17	Mock Artist	1	1	2023-11-03 17:49:01.508164	2023-11-03 21:49:01.508164	0

Total rows: 52 of 52 Query complete 00:00:00.102 Ln 1, Col

Figure 10 - Screenshot of the Table Events



	id [PK] integer	event_id integer	name character varying (255)	unit_price numeric (10,2)	currency character	capacity integer	remaining_tickets integer	version_number integer
1	21	9	YijieXie3Bronze	100.00	AUD	30	30	0
2	22	9	YijieXie3Silver	100.00	AUD	20	20	0
3	23	9	YijieXie3Gold	100.00	AUD	10	10	0
4	24	9	YijieXie3VIP	100.00	AUD	5	5	0
5	25	11	QuanchiChen4Bronze	100.00	AUD	30	30	0
6	26	11	QuanchiChen4Silver	100.00	AUD	20	20	0
7	27	11	QuanchiChen4Gold	100.00	AUD	10	10	0
8	28	11	QuanchiChen4VIP	100.00	AUD	5	5	0
9	53	25	YijieXie11Bronze	100.00	AUD	30	30	0
10	54	25	YijieXie11Silver	100.00	AUD	20	20	0
11	55	25	YijieXie11Gold	100.00	AUD	10	10	0
12	56	25	YijieXie11VIP	100.00	AUD	5	5	0
13	61	30	YijieXie13Bronze	100.00	AUD	30	30	0
14	62	30	YijieXie13Silver	100.00	AUD	20	20	0
15	63	30	YijieXie13Gold	100.00	AUD	10	10	0
16	64	30	YijieXie13VIP	100.00	AUD	5	5	0
17	77	37	QuanchiChen17Bronze	100.00	AUD	30	30	0
18	78	37	QuanchiChen17Silver	100.00	AUD	20	20	0

Total rows: 208 of 208 Query complete 00:00:00.114 Ln 3, Col

Figure 11 - Screenshot of the Table Sections