Concurrency Document

“Alfie” Music Events System

Software Design and Architecture
SWEN90007 SM2 2023 Project
Getters Setters

Joel Fressard Kenna - jfkenna@student.unimelb.edu.au - 995401
Sebastian Bobadilla Chara - sbobadillach@student.unimelb.edu.au - 1305851
Georgia Rose Lewis - grlewis@student.unimelb.edu.au - 982172
Anjaney Chirag Mahajan - anjaneychira@student.unimelb.edu.au - 1119668

Professor Eduardo Oliveira & Professor Maria Rodriguez Read
University of Melbourne
Semester 2 2023

Due: October 17th 2023 17:00 AEST

Date	Version	Description	Author
2023-10-09	1.00-D1	Initial document set-up	Georgia Lewis
2023-10-15	1.01-D1	Added concurrency issues and brief description	Anjaney C Mahajan
2023-10-17	1.02-D1	Add description of concurrency patterns	Joel Kenna
2023-10-17	1.03-D1	Add mitigation sections for concurrency cases and add mitigation sequence diagrams for edit event / edit user / edit section prices	Joel Kenna
2023-10-17	1.04-D1	Added sectins for Concurrency & Pattern Description	Sebastian Bobadilla Chara

Contents

1 Class Diagrams

Based on feedback received from our part 2 assignment, we improved our backend server by introducing a service layer for better organization. This allowed us to implement our concurrency changes for the application for the main architectural change involved adding version numbers to database data classes to track modifications. The adjustments to read and write methods in the data mapper and managing the current information's version on the frontend are implementation details discussed in other sections, ensuring system functionality and performance.

The diagrams for the Presentation layer can be seen in Figure 1, for the Domain layer in Figure 2, for the Service layer in Figure 3, and for the Persistence layer in Figure 4.

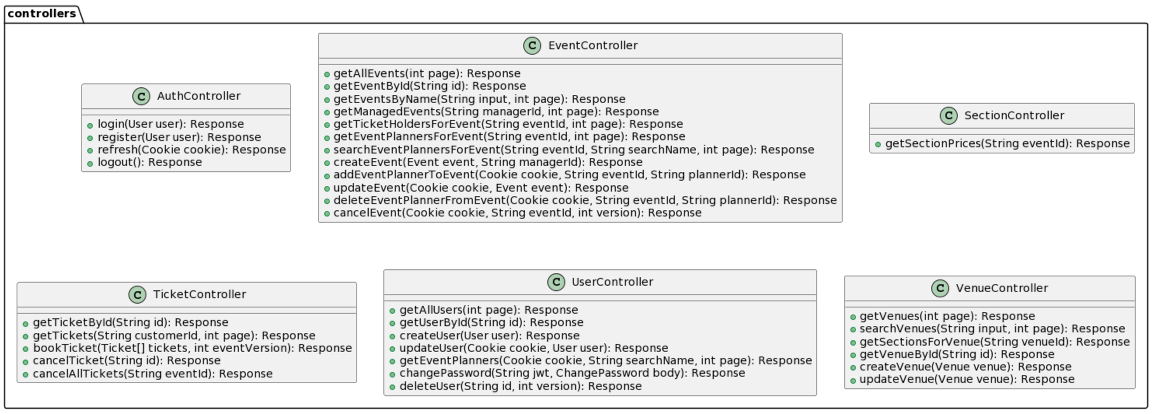

controllers
AuthController
login(User user): Response
register(User user): Response
refresh(Cookie cookie): Response
logout(): Response
EventController
getAllEvents(int page): Response
getEventById(String id): Response
getEventsByName(String input, int page): Response
getManagedEvents(String managerId, int page): Response
getTicketHoldersForEvent(String eventId, int page): Response
getEventPlannersForEvent(String eventId, int page): Response
searchEventPlannersForEvent(String eventId, String searchName, int page): Response
createEvent(Event event, String managerId): Response
addEventPlannerToEvent(Cookie cookie, String eventId, String plannerId): Response
updateEvent(Cookie cookie, Event event): Response
deleteEventPlannerFromEvent(Cookie cookie, String eventId, String plannerId): Response
cancelEvent(Cookie cookie, String eventId, int version): Response
SectionController
getSectionPrices(String eventId): Response
TicketController
getTicketById(String id): Response
getTickets(String customerId, int page): Response
bookTicket(Ticket[] tickets, int eventVersion): Response
cancelTicket(String id): Response
cancelAllTickets(String eventId): Response
UserController
getAllUsers(int page): Response
getUserById(String id): Response
createUser(User user): Response
updateUser(Cookie cookie, User user): Response
getEventPlanners(Cookie cookie, String searchName, int page): Response
changePassword(String jwt, ChangePassword body): Response
deleteUser(String id, int version): Response
VenueController
getVenues(int page): Response
searchVenues(String input, int page): Response
getSectionsForVenue(String venueId): Response
getVenueById(String id): Response
createVenue(Venue venue): Response
updateVenue(Venue venue): Response

Figure 1: Component Diagram of Presentation Layer

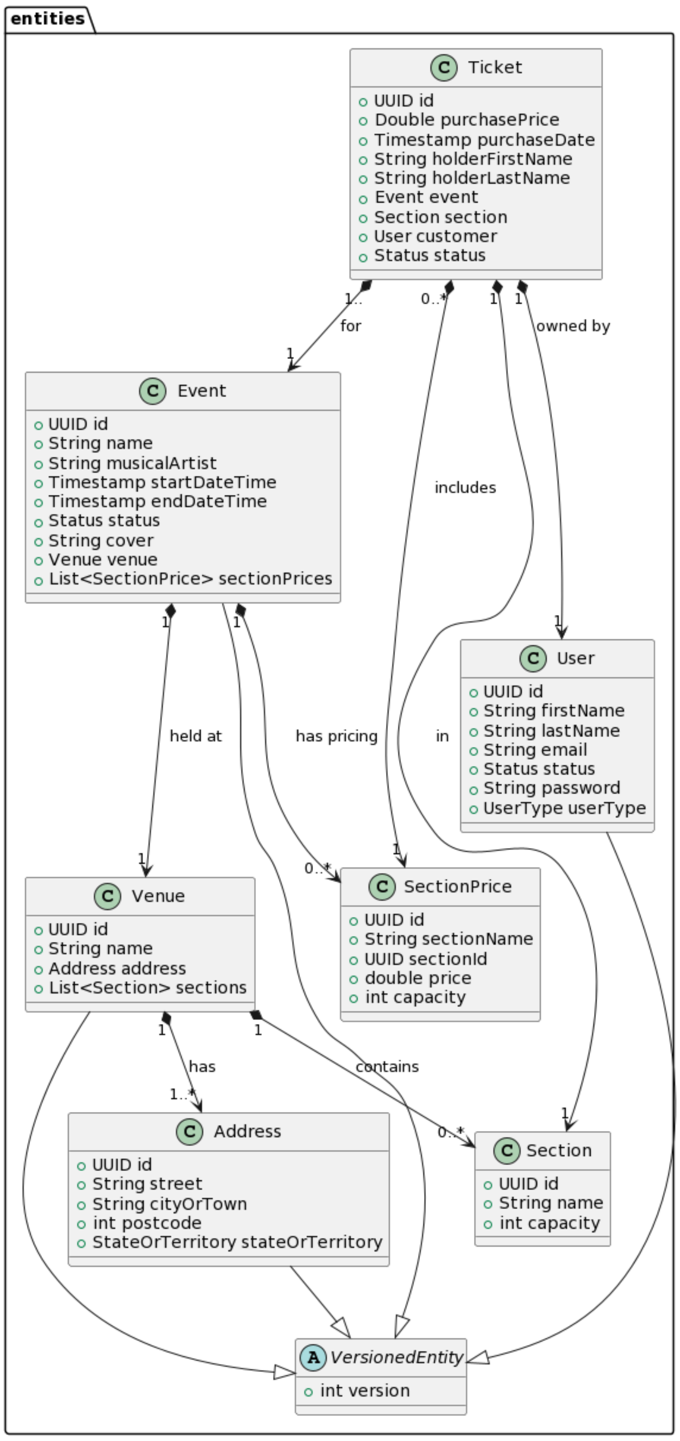


Figure 2: Domain Models of the Music Events System

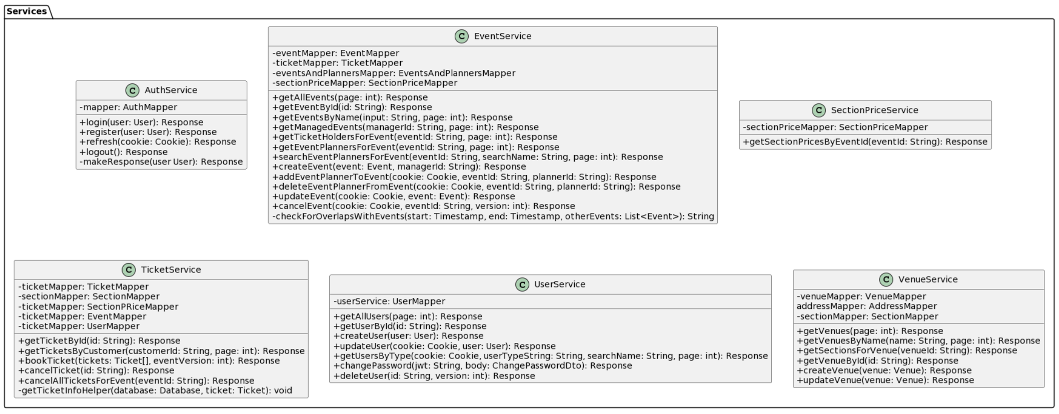

Services
AuthService
-mapper: AuthMapper
+login(user: User): Response
+register(user: User): Response
+refresh(cookie: Cookie): Response
+logout(): Response
-makeResponse(user User): Response
EventService
-eventMapper: EventMapper
-ticketMapper: TicketMapper
-eventsAndPlannersMapper: EventsAndPlannersMapper
-sectionPriceMapper: SectionPriceMapper
+getAllEvents(page: int): Response
+getEventById(id: String): Response
+getEventsByName(input: String, page: int): Response
+getManagedEvents(managerId: String, page: int): Response
+getTicketHoldersForEvent(eventId: String, page: int): Response
+getEventPlannersForEvent(eventId: String, page: int): Response
+searchEventPlannersForEvent(eventId: String, searchName: String, page: int): Response
+createEvent(event: Event, managerId: String): Response
+addEventPlannerToEvent(cookie: Cookie, eventId: String, plannerId: String): Response
+deleteEventPlannerFromEvent(cookie: Cookie, eventId: String, plannerId: String): Response
+updateEvent(cookie: Cookie, event: Event): Response
+cancelEvent(cookie: Cookie, eventId: String, version: int): Response
-checkForOverlapsWithEvents(start: Timestamp, end: Timestamp, otherEvents: List<Event>): String
SectionPriceService
-sectionPriceMapper: SectionPriceMapper
+getSectionPricesByEventId(eventId: String): Response
TicketService
-ticketMapper: TicketMapper
-sectionMapper: SectionMapper
-ticketMapper: SectionPRiceMapper
-ticketMapper: EventMapper
-ticketMapper: UserMapper
+getTicketById(id: String): Response
+getTicketsByCustomer(customerId: String, page: int): Response
+bookTicket(tickets: Ticket[], eventVersion: int): Response
+cancelTicket(id: String): Response
+cancelAllTicketsForEvent(eventId: String): Response
-getTicketInfoHelper(database: Database, ticket: Ticket): void
UserService
-userService: UserMapper
+getAllUsers(page: int): Response
+getUserById(id: String): Response
+createUser(user: User): Response
+updateUser(cookie: Cookie, user: User): Response
+getUsersByType(cookie: Cookie, userTypeString: String, searchName: String, page: int): Response
+changePassword(jwt: String, body: ChangePasswordDto): Response
+deleteUser(id: String, version: int): Response
VenueService
-venueMapper: VenueMapper
addressMapper: AddressMapper
-sectionMapper: SectionMapper
+getVenues(page: int): Response
+getVenuesByName(name: String, page: int): Response
+getSectionsForVenue(venueId: String): Response
+getVenueById(id: String): Response
+createVenue(venue: Venue): Response
+updateVenue(venue: Venue): Response

Figure 3: Component Diagram of Service Layer

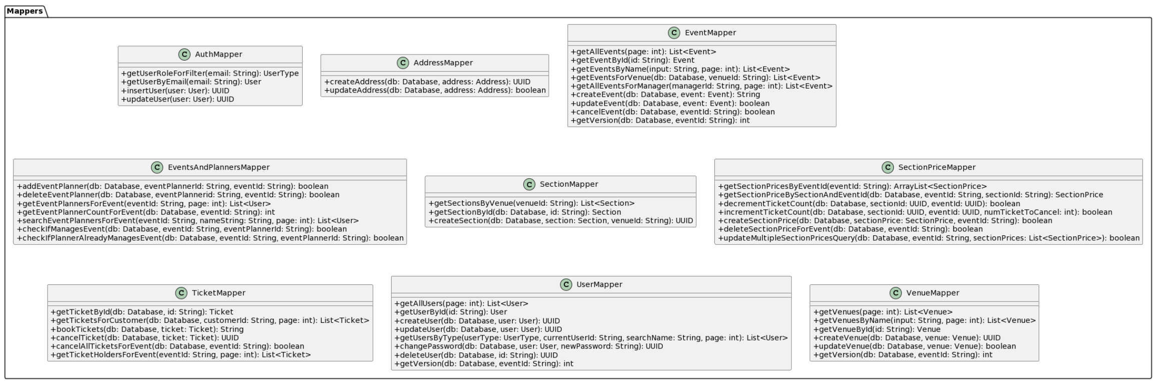

Mappers
AuthMapper
+getUserRoleForFilter(email: String): UserType
+getUserByEmail(email: String): User
+insertUser(user: User): UUID
+updateUser(user: User): UUID
AddressMapper
+createAddress(db: Database, address: Address): UUID
+updateAddress(db: Database, address: Address): boolean
EventMapper
+getAllEvents(page: int): List<Event>
+getEventById(id: String): Event
+getEventsByName(input: String, page: int): List<Event>
+getEventsForVenue(db: Database, venueId: String): List<Event>
+getAllEventsForManager(managerId: String, page: int): List<Event>
+createEvent(db: Database, event: Event): String
+updateEvent(db: Database, event: Event): boolean
+cancelEvent(db: Database, eventId: String): boolean
+getVersion(db: Database, eventId: String): int
EventsAndPlannersMapper
+addEventPlanner(db: Database, eventPlannerId: String, eventId: String): boolean
+deleteEventPlanner(db: Database, eventPlannerid: String, eventId: String): boolean
+getEventPlannersForEvent(eventId: String, page: int): List<User>
+getEventPlannerCountForEvent(db: Database, eventId: String): int
+searchEventPlannersForEvent(eventId: String, nameString: String, page: int): List<User>
+checkIfManagesEvent(db: Database, eventId: String, eventPlannerId: String): boolean
+checkIfPlannerAlreadyManagesEvent(db: Database, eventId: String, eventPlannerId: String): boolean
SectionMapper
+getSectionsByVenue(venueId: String): List<Section>
+getSectionById(db: Database, id: String): Section
+createSection(db: Database, section: Section, venueId: String): UUID
SectionPriceMapper
+getSectionPricesByEventId(eventId: String): ArrayList<SectionPrice>
+getSectionPriceBySectionAndEventId(db: Database, eventId: String, sectionId: String): SectionPrice
+decrementTicketCount(db: Database, sectionId: UUID, eventId: UUID): boolean
+incrementTicketCount(db: Database, sectionId: UUID, eventId: UUID, numTicketToCancel: int): boolean
+createSectionPrice(db: Database, sectionPrice: SectionPrice, eventId: String): boolean
+deleteSectionPriceForEvent(db: Database, eventId: String): boolean
+updateMultipleSectionPricesQuery(db: Database, eventId: String, sectionPrices: List<SectionPrice>): boolean
TicketMapper
+getTicketById(db: Database, id: String): Ticket
+getTicketsForCustomer(db: Database, customerId: String, page: int): List<Ticket>
+bookTickets(db: Database, ticket: Ticket): String
+cancelTicket(db: Database, ticket: Ticket): UUID
+cancelAllTicketsForEvent(db: Database, eventId: String): boolean
+getTicketHoldersForEvent(eventId: String, page: int): List<Ticket>
UserMapper
+getAllUsers(page: int): List<User>
+getUserById(id: String): User
+createUser(db: Database, user: User): UUID
+updateUser(db: Database, user: User): UUID
+getUsersByType(userType: UserType, currentUserId: String, searchName: String, page: int): List<User>
+changePassword(db: Database, user: User, newPassword: String): UUID
+deleteUser(db: Database, id: String): UUID
+getVersion(db: Database, eventId: String): int
VenueMapper
+getVenues(page: int): List<Venue>
+getVenuesByName(input: String, page: int): List<Venue>
+getVenueById(id: String): Venue
+createVenue(db: Database, venue: Venue): UUID
+updateVenue(db: Database, venue: Venue): boolean
+getVersion(db: Database, eventId: String): int

Figure 4: Data Persistence Layer of the Music Events System

2 Concurrency & Pattern Description

For testing our implementation, we used the testing tool K6[1]. This tool allows us to have a scenario configured for making HTTP requests through a simple JS script. In the end, the tool provides us with different metrics about the run time of the requests that were made.

For this report we will focus only on the concurrency aspect of things, so we will use the tool to generate several users that perform several requests to the specified endpoint.

2.1 Business transactions as database transactions

Each business transaction in the system is handled as a single database transaction, with a transaction level of `TRANSACTION_REPEATABLE_READ`. If any of the database operations or business logic checks for the business transaction fail, the database transaction is rolled back and an error is returned to the user. By bundling multiple operations as a single transaction at this isolation level, the database ensures that concurrent transactions will fail if any of their sub-operations (e.g. UPDATE to different rows or tables) interfere with each other - if one transaction edits a row and a second simultaneously attempts to edit that same row, PostgreSQL will cause the second to fail with an access serialisation error, as repeatable read guarantees consistency in cases of concurrent updates. If the system detects the transaction failed due to access serialization errors, it replies to the client with a 409 Conflict, which our client handles by refreshing the page and requesting the latest copy of the data they were editing.

2.2 Versioning

In our system, any database rows that correspond to business objects in the system have an additional field named “version”, that keeps track of the current entity revision. Whenever a business transaction takes place to edit or delete an entity (in our system, “deletes” are just updates that change the status of a row), the version number is also incremented. As the version will always be updated when a business transaction completes, comparisons against the version field provides a simple way of verifying if an entity visible to the client is up-to-date with the latest copy in the database. At the server level, entities with versions are implemented as extending the abstract *VersionedEntity* class. The *VersionedEntity* class currently only contains a version field, but was included to make the system more easily extensible to support more complex versioning and version logic. Note that as we have no use cases that require history to be stored or displayed, we do not keep track of previous versions of each row, as this would significantly increase system complexity and storage requirements.

The main use for versioning in our system is to ensure that users are aware when changes they’re making to an event or venue will be overwriting new changes that they haven’t yet seen - when a user attempts to update an event, user, or venue, they include the version number in their request payload. In the service layer, the server checks if the version number provided by the client is smaller than the version currently in the database.

If the version is smaller, the sever returns a 409 Conflict, indicating the data the user was editing was not up-to-date. When the client receives a 409, it displays a message indicating the record is out-of-date in the UI and reloads the page, causing it make a new request to fetch the latest data from the database.

As a consequence of these changes, users are forced to review changes before making new updates - for example, in a scenario where a planner opens an event and begins filling out some new values,

[1]K6: Load testing tool. https://k6.io/docs/

but a second planner updates the event while the first planner is still editing, the first planner's update will fail, and their page will reload to display the new changes. The first planner can then review the updated event and then make any edits they feel they are still required. By forcing these reviews, we guard our system against users unknowingly overwriting data.

3 Multiple people buying tickets

3.1 Overview and Details

The issue of multiple people buying tickets simultaneously is a critical aspect of the music events system. This system allows users to purchase tickets for various events. In the event of a high demand for tickets, it's common for multiple users to attempt ticket purchases simultaneously, which introduces concurrency challenges and potential bottlenecks. This concurrency issue can result in race conditions and potential overbooking, affecting the fairness of ticket allocation.

3.1.1 Specific Challenges

When multiple users attempt to buy tickets for the same event concurrently, several challenges arise:

- **Race Conditions**: Simultaneous access to the ticket inventory can lead to race conditions, where multiple users may believe they have purchased the same ticket, causing conflicts and inconsistencies.
- **Overbooking**: Without proper synchronization, the system may accidentally overbook tickets, leading to dissatisfaction among customers who are unable to attend the event despite purchasing tickets.

3.2 Mitigation

To mitigate this issue we took several approaches to seek a robust and safe solution for this use case.

3.2.1 Encapsulation of the business transaction

As stated in our previous report, since we have a client-server architecture, having a Unit of Work on its own doesn't make much sense because objects are short lived. To overcome this, we mapped each business transaction to each request, and in turn, to a single database transaction. This means that all actions followed in the database are encapsulated in their own transaction.

3.2.2 Transaction isolation level

As a general concurrency safety measure applied system wide to all connections obtained from the connection pool, we have set the default transaction isolation level to `TRANSACTION_REPEATABLE_READ`. However, for more "critical" use cases, like booking a ticket where we can have a great amount of transactions at the same time, we decided to implement a higher transaction isolation level, being `TRANSACTION_SERIALIZABLE`.

According to PostgreSQL documentation[2] this isolation level, being the highest, emulates a scenario where all transactions come straight one after the other. This means that all interactions with the database become serially equivalent. When having this isolation level, `SELECT` statements

[2]Serializable Isolation Level: PostgreSQL docs https://www.postgresql.org/docs/7.2/xact-serializable.html

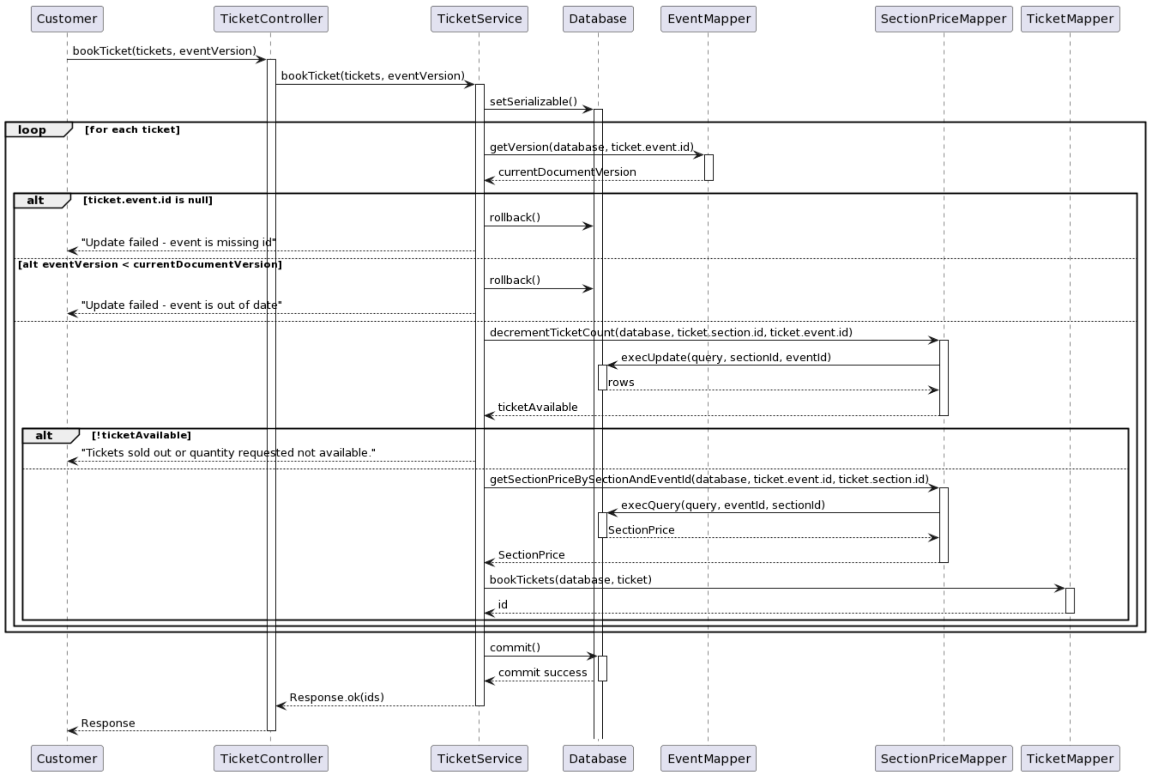
will only read committed data before the transaction began, and `UPDATE` statements will wait for previous statements to finish before executing. In a situation where we want a first come first serve service, this isolation level is ideal.

3.2.3 Software validation checks

Additionally, a validation in the logic was made. Inside the service layer of the code, before proceeding to book the ticket when a requests is being processed, we have a previous step in the which we check for ticket availability. If so, we return a value of `true` to indicate that we ticket is valid, we record it in the tickets table, and then return the successful response to the user.

By doing this we not only ensure that every successful request will have a valid ticket, but we also avoid going beyond the specified amount of tickets available. However, the correct functionality of this implementation depends on the transaction isolation level of the database connection.

The sequence diagram for a user purchasing a ticket can be seen in Figure 5



Customer
TicketController
TicketService
Database
EventMapper
SectionPriceMapper
TicketMapper
bookTicket(tickets, eventVersion)
bookTicket(tickets, eventVersion)
setSerializable()
loop
[for each ticket]
getVersion(database, ticket.event.id)
currentDocumentVersion
alt
[ticket.event.id is null]
rollback()
"Update failed - event is missing id"
[alt eventVersion < currentDocumentVersion]
rollback()
"Update failed - event is out of date"
decrementTicketCount(database, ticket.section.id, ticket.event.id)
execUpdate(query, sectionId, eventId)
rows
ticketAvailable
alt
[!ticketAvailable]
"Tickets sold out or quantity requested not available."
getSectionPriceBySectionAndEventId(database, ticket.event.id, ticket.section.id)
execQuery(query, eventId, sectionId)
SectionPrice
SectionPrice
bookTickets(database, ticket)
id
commit()
commit success
Response.ok(ids)
Response

Figure 5: Control flow for purchasing a ticket, illustrating error flows for outdated versions and lack of ticket availability

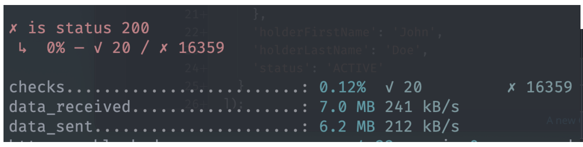
3.3 Testing

Using K6, we defined a very simple scenario where 30 virtual users make a request trying to book a ticket to the same event, a situation that would be very common for a system like this. All the users, aware that tickets are limited, keep on making requests during 30 seconds, trying to ensure a ticket for themselves and their friends.

The script defined can be seen in Listing 1. As a result of running the script, we can see that we obtain the desired behaviour thanks to all the checks that we implemented previously. In the database, we have 0 remaining tickets, as seen in Figure 6b, and as a result from running K6, we see that the tool gathered only 20 successful responses, as seen in Figure 6a, the amount of tickets that we had available at the beginning.

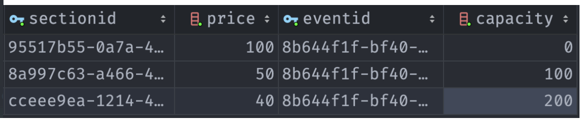
```javascript
import http from 'k6/http';
import { sleep, check } from 'k6';

export const options = {
  vus: 30,
  duration: '5s'
};

export default function() {
  const url = 'http://localhost:8080/ticket';
  const payload = JSON.stringify([
    {
      'event': {
        'id': '8b644f1f-bf40-4ae6-a303-d9e6dabd5c79'
      },
      'section': {
        'id': '95517b55-0a7a-478b-b6e6-80503345c6cc'
      },
      'customer': {
        'id': '55ee14a9-0254-4900-947b-2531d807f45d'
      },
      'holderFirstName': 'John',
      'holderLastName': 'Doe',
      'status': 'ACTIVE'
    }
  ]);

  const params = {
    headers: {
      'Content-Type': 'application/json'
    }
  };

  const res = http.post(url, payload, params);
  check(res, {
    'is status 200': (r) => r.status === 200
  });
}
```

Listing 1: K6 configuration script.



```
✗ is status 200
 ↳  0% — √ 20 / ✗ 16359

checks.........................: 0.12%  √ 20          ✗ 16359
data_received..................: 7.0 MB 241 kB/s
data_sent......................: 6.2 MB 212 kB/s
```

(a) K6 results



sectionid	price	eventid	capacity
95517b55-0a7a-4…	100	8b644f1f-bf40-…	0
8a997c63-a466-4…	50	8b644f1f-bf40-…	100
cceee9ea-1214-4…	40	8b644f1f-bf40-…	200

(b) Database output

Figure 6: Screenshots of the result of the concurrency testing performed on the "Book ticket" use case.

4 Event updated after user views the book tickets page but before they actually book their tickets

4.1 Overview and Details

When the user is purchasing tickets, it is important that they are aware of the most up-to-date details about the price they'll be paying and the time / location of the actual event. Otherwise, the user could receive an unexpected costs or book a ticket for an event they won't be able to attend.

4.1.1 Specific Challenges:

- **Outdated pricing and event information:** When a user carries out the ticket purchase process, they select specific sections and corresponding prices. If an administrator or event organizer updates the section prices or event details for the event after the user has originally loaded the event data, it can result in scenarios where the users purchase a ticket for an event based on outdated or inconsistent pricing information.

4.2 Mitigation

Note: As previously described, the book ticket transaction takes place at a `TRANSACTION_SERIALIZABLE` level.

4.2.1 Version check for event tickets are being booked for in service layer

Incoming payloads for purchasing tickets to an event include a version number. At the beginning of the transaction, the service layer checks the version number from the purchase request against the latest event version in the database. If the incoming version number is less than the current version number of the event, the server returns a 409 Conflict to the client to indicate that their data is out of date, which causes the client to display a message about a failed purchase and reload the page with the latest data.

The sequence diagram for mitigation can be seen in Figure 5

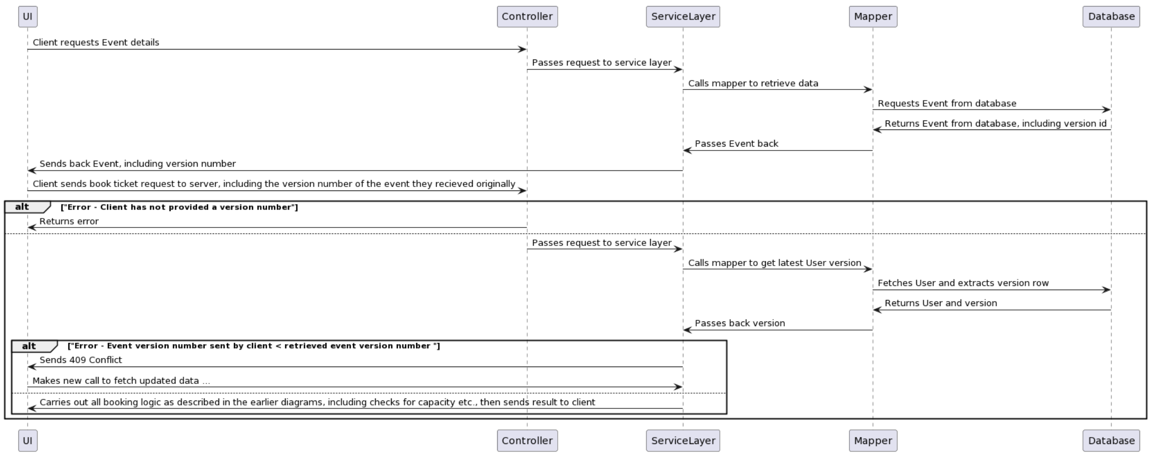

UI
Controller
ServiceLayer
Mapper
Database
Client requests Event details
Passes request to service layer
Calls mapper to retrieve data
Requests Event from database
Returns Event from database, including version id
Passes Event back
Sends back Event, including version number
Client sends book ticket request to server, including the version number of the event they recieved originally
alt
["Error - Client has not provided a version number"]
Returns error
Passes request to service layer
Calls mapper to get latest User version
Fetches User and extracts version row
Returns User and version
Passes back version
alt
["Error - Event version number sent by client < retrieved event version number "]
Sends 409 Conflict
Makes new call to fetch updated data ...
Carries out all booking logic as described in the earlier diagrams, including checks for capacity etc., then sends result to client

Figure 7: Event updated after user views-the book tickets mitigation sequence diagram

4.3 Testing

5 User details update

5.1 Overview and Details

In our system, administrators have the ability to update users' information, and each user can modify their own details. This can lead to issues when both the administrator and a user attempt to perform account updates or deletions at the same time.

5.1.1 Specific Challenges:

- **Data overwrites:** If a administrator loads a user, performs some updates to it, and then saves, they may inadvertently overwrite another change that was submitted by user after they originally loaded the record. This is non-ideal, as the administrator will not even be aware that they're overwriting the latest changes. Ideally, the administrator would be notified of any new updates and forced to review them before they can complete their edit.

5.2 Mitigation

5.2.1 Transaction isolation level

This transaction was isolated at a 'REPEATABLE READ' level. This will cause a serialization failure when multiple simultaneous transactions update the same row. We detect these serialization failures and return a 409 Conflict to the client whose update failed, which causes the client to display a message about a failed update and re-load the latest data. This ensures that no data 'disappears' due to multiple simultaneous updates.

5.2.2 Version check in service layer

Incoming update payloads include a version number. At the beginning of the transaction, the service layer checks the version number from the update payload against the latest version in the database.

If the incoming version number is less than the current version number, the server returns a 409 Conflict to the client to indicate that their data is out of date, which causes the client to display a message about a failed update and re-load the latest data.

The sequence diagram for multiple event planners editing an event can be seen in Figure **??**

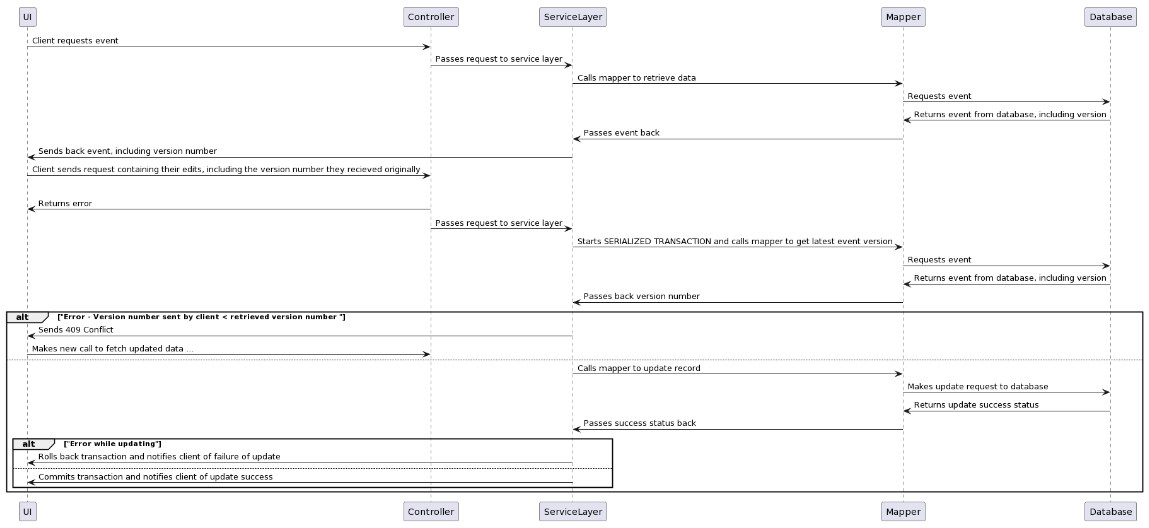

UI
Controller
ServiceLayer
Mapper
Database
Client requests event
Passes request to service layer
Calls mapper to retrieve data
Requests event
Returns event from database, including version
Passes event back
Sends back event, including version number
Client sends request containing their edits, including the version number they recieved originally
Returns error
Passes request to service layer
Starts SERIALIZED TRANSACTION and calls mapper to get latest event version
Requests event
Returns event from database, including version
Passes back version number
alt
["Error - Version number sent by client < retrieved version number "]
Sends 409 Conflict
Makes new call to fetch updated data ...
Calls mapper to update record
Makes update request to database
Returns update success status
Passes success status back
alt
["Error while updating"]
Rolls back transaction and notifies client of failure of update
Commits transaction and notifies client of update success

Figure 8: Control flow for editing a user, illustrating error flows for outdated versions and concurrent transactions

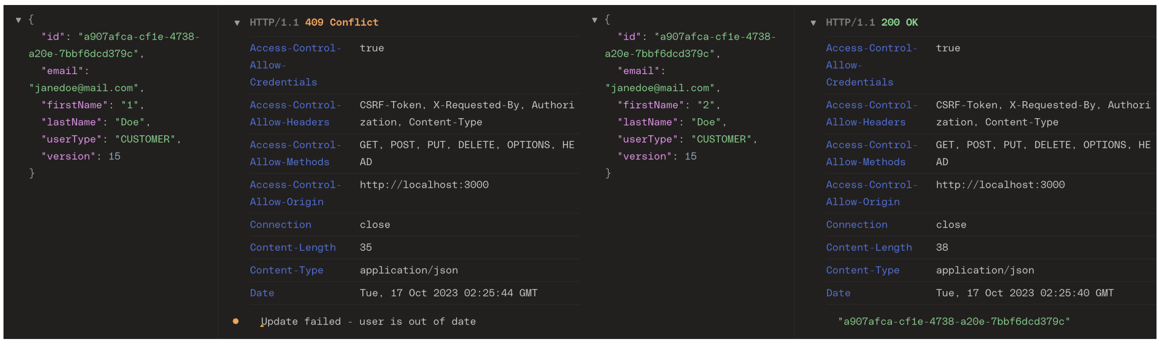
5.3 Testing

For testing our implementation and making sure that it behaves the way we intend it to do, while keeping a proper handling of concurrent access we tested the requests in a very simple yet effective way that allows us to see the behaviour we get on different scenarios.

To simulate what could happen when more than one request is received while a previous one is still being processed, we included a `Thread.sleep(5000)` call before committing the changes that were made to the database. This adds a significant delay that allows us to see in a much finer scale the results of concurrent transactions.

As a result of this scenario we got that the very first request that was received, and hence the first one that was processed, was successful in spite of the delay introduced, however the second one failed. This information is not visible to user, but due to having the database transaction isolation level set to `TRANSACTION_REPEATABLE_READ` we get a message error from PostgreSQL that reads "*could not serialize access due to concurrent update*". However, if we decrement to a lower isolation levet, being for example `TRANSACTION_READ_COMMITTED` we don't get that warning, and BOTH requests are completed successfully.

The result of the scenario defined previously to test our concurrency measures can be seen in Figure 9a for the failed request, and in Figure 9b for the successful one. It is worth noting we can consider them concurrent because they were sent milliseconds apart, as we can see in their timestamps.



```
▼ {
    "id": "a907afca-cf1e-4738-
  a20e-7bbf6dcd379c",
    "email":
  "janedoe@mail.com",
    "firstName": "1",
    "lastName": "Doe",
    "userType": "CUSTOMER",
    "version": 15
  }
```

```
▼ HTTP/1.1 409 Conflict
  Access-Control-   true
  Allow-
  Credentials
  Access-Control-   CSRF-Token, X-Requested-By, Authori
  Allow-Headers     zation, Content-Type
  Access-Control-   GET, POST, PUT, DELETE, OPTIONS, HE
  Allow-Methods     AD
  Access-Control-   http://localhost:3000
  Allow-Origin
  Connection        close
  Content-Length    35
  Content-Type      application/json
  Date              Tue, 17 Oct 2023 02:25:44 GMT
● Update failed - user is out of date
```

(a) Failed request

```
▼ {
    "id": "a907afca-cf1e-4738-
  a20e-7bbf6dcd379c",
    "email":
  "janedoe@mail.com",
    "firstName": "2",
    "lastName": "Doe",
    "userType": "CUSTOMER",
    "version": 15
  }
```

```
▼ HTTP/1.1 200 OK
  Access-Control-   true
  Allow-
  Credentials
  Access-Control-   CSRF-Token, X-Requested-By, Authori
  Allow-Headers     zation, Content-Type
  Access-Control-   GET, POST, PUT, DELETE, OPTIONS, HE
  Allow-Methods     AD
  Access-Control-   http://localhost:3000
  Allow-Origin
  Connection        close
  Content-Length    38
  Content-Type      application/json
  Date              Tue, 17 Oct 2023 02:25:40 GMT
    "a907afca-cf1e-4738-a20e-7bbf6dcd379c"
```

(b) Successful request

Figure 9: Screenshots of the result of the concurrency testing performed on the “Edit user” use case.

6 Cancel ticket

6.1 Overview and Details

When a customer is cancelling a ticket, the number of available tickets left for a particular section is incremented as this ticket is added back into the pool of tickets available. If an event planner is cancelling their ticket at the same time, this could cause the number of tickets to be incremented twice, therefore leaving the database in a state where more tickets are available than should be. If an event planner is cancelling all tickets, then this could cause conflicts where a user is cancelling their own ticket simultaneously. When a customer is purchasing a ticket, the number of available tickets left for a particular section is decremented; if this happens concurrently with ticket cancellation, the actions could conflict with each other, and leave the number of tickets left as different than expected. It is crucial that none of these actions interfere with each other, so that the number of available tickets remains consistent.

6.1.1 Specific Challenges

When a customer is cancelling their ticket, several concurrency challenges arise:

- **Concurrency Control:** Concurrent updates by separate customers and event planners can lead to conflicts, potentially overwriting changes made by one user while another user is using the data.
- **Data Consistency:** When a customer cancels their ticket, the event planner should have access to this updated information as soon as possible. Additionally, when a customer's ticket is cancelled, other customers should have access to purchase this ticket as soon as possible.
- **User Experience Impact:** When an event planner deletes a ticket, customers should be shown this as soon as possible, so that they have the most up-to-date information about their ticket. They also should not be able to cancel their ticket, as this would result in the *sectionPrice* table being incremented unnecessarily, and could lead to overbooking of the event.
- **Overbooking:** If a customer tries to cancel their ticket at the same time as an event planner, the number of tickets being added back to the pool may be 2 rather than 1, which makes it

seem as though there are more tickets available than there actually are; this could lead to overbooking of the event.

6.2 Mitigation

6.2.1 Transaction isolation level

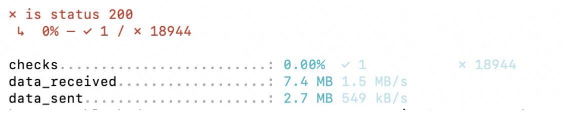
This transaction was isolated at a 'REPEATABLE READ' level. This will cause a serialization failure when multiple simultaneous transactions update the same row. We detect these serialization failures and return a 409 Conflict to the client whose update failed, which causes the client to display a message about the failed update. This prevents the data from being inconsistent due to multiple simultaneous updates.

6.2.2 Version check in service layer

Incoming delete requests include a version number. At the beginning of the transaction, the service layer checks the version number from the last request to get the ticket number against the latest version in the database. If the incoming version number is less than the current version number, the server returns a 409 Conflict to the client to indicate that their data is out of date, which causes the client to display a message about a failed update.

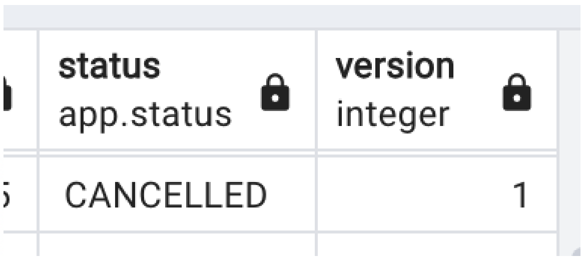
6.3 Testing

To test this, we used K6 to define a scenario where multiple requests are sent to the browser attempting to delete a single ticket at the same time. This was expected to pass once, and this was achieved, as shown in Figure 10a. The version in the table was only incremented once, as shown in Figure 10b.



```
× is status 200
 ↳  0% — ✓ 1 / × 18944

checks.........................: 0.00%  ✓ 1           × 18944
data_received..................: 7.4 MB 1.5 MB/s
data_sent......................: 2.7 MB 549 kB/s
```

(a) Failed request



status app.status	version integer
CANCELLED	1

(b) Successful request

Figure 10: Screenshots of the result of the concurrency testing performed on the "Delete ticket" use case.

7 Event planners simultaneously creating or updating events

7.1 Overview and Details

The situation that may occur related to this use case is when different event planners are trying to compete for getting a venue booked for the date and time they want. Since our system works on a first-come first-served basis, planners will compete to get it for their event. Additionally, event planners can add or remove other event planners from an event which more closely reflects reality, introducing a different set of situations that need to be accounted for.

7.1.1 Specific Challenges:

- **Race Conditions:** Event planners contend with each other to get the best date for their event. If this is not handled properly we can end with a venue being booked for two or more events at the same time.
- **Concurrent modifications:** Since more than one event planner may be handling the same event, we run into the problem that they may modify the event without knowing that they are working on outdated information.

7.2 Mitigation

7.2.1 Serialization of transactions

This use case required handling two different types of concurrency issues that can be solved with the techniques mentioned previously. First, to account for the race conditions of different event planners contending for an specific venue at an specific time and date we can increase the transaction isolation level to be `TRANSACTION_SERIALIZABLE`, that way we enforce the serial equivalence of all requests performed. This one is an “easy” fix, since we delegate the handling to the database.

On the other hand we do need to include additional checks and logic for ensuring that users can’t update information that is no longer the current one. For this a column to keep the version is maintained, and we check the offset of the versions before allowing for any modification to be made.

7.2.2 Version check in service layer

Incoming update payloads for editing an event include a version number. At the beginning of the transaction, the service layer checks the version number from the update payload against the latest version in the database. If the incoming version number is less than the current version number, the server returns a 409 Conflict to the client to indicate that their data is out of date, which causes the client to display a message about a failed update and re-load the latest data.

The sequence diagram for multiple event planners editing an event can be seen in Figure 11

7.3 Testing

To test this we can first use K6 again to send several requests to the server and see that only one succeeds for a specific date and time in a venue. The script is similar to the one seen in Listing 1, with just changing the endpoint it makes reference to, and adapting the body. The results can be seen in Figure 12.

This result is also meaningful because it also indicates that versioning control is working as well. As mentioned previously, it wouldn’t work on a lower isolation level, however on a higher one we can have an even more strict control and check of it.

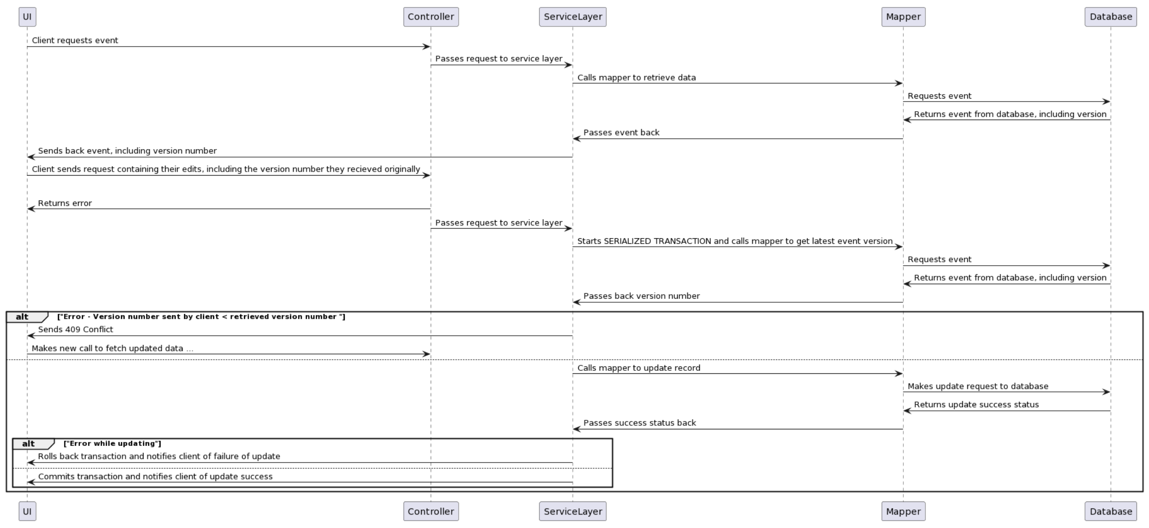
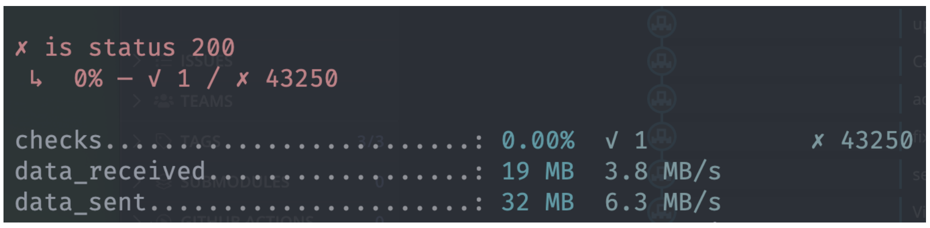

UI
Controller
ServiceLayer
Mapper
Database
Client requests event
Passes request to service layer
Calls mapper to retrieve data
Requests event
Returns event from database, including version
Passes event back
Sends back event, including version number
Client sends request containing their edits, including the version number they recieved originally
Returns error
Passes request to service layer
Starts SERIALIZED TRANSACTION and calls mapper to get latest event version
Requests event
Returns event from database, including version
Passes back version number
alt
["Error - Version number sent by client < retrieved version number "]
Sends 409 Conflict
Makes new call to fetch updated data ...
Calls mapper to update record
Makes update request to database
Returns update success status
Passes success status back
alt
["Error while updating"]
Rolls back transaction and notifies client of failure of update
Commits transaction and notifies client of update success
UI
Controller
ServiceLayer
Mapper
Database

Figure 11: Control flow for editing an event, illustrating error flows for outdated versions and concurrent transactions



```
✗ is status 200
 ↳  0% — ✓ 1 / ✗ 43250

checks.........................: 0.00%  ✓ 1          ✗ 43250
data_received..................: 19 MB  3.8 MB/s
data_sent......................: 32 MB  6.3 MB/s
```

Figure 12: Results of running the test script with K6

8 Overall testing summary

Testing was performed using K6 for all key identified concurrency risks, and results indicated concurrency risks were well addressed. Alongside this testing, manual testing of non time-critical scenarios (eg. two users editing an event simultaneously, resulting in one having an outdated version copies) was performed on both the deployed and local versions of the applications, and produced expected outcomes.

Based on the success of our automated testing and manual test plans, we can say with some confidence that we have addressed all the major concurrency risks we identified, although we can make no guarantees about any error scenarios that we overlooked and failed to capture in our report.