SWEN90007: Software Design and Architecture Part 3 Report

Team: BrogrammerBrigade

Team Details

Kevin Wu	993272	wukw@student.unimelb.edu.au
James Launder	993934	launderj@student.unimelb.edu.au
Kah Meng Lee	940711	kahl3@student.unimelb.edu.au
Henry Hamer	1173633	hhamer@student.unimelb.edu.au

Implemented Use Cases

1. As an administrative member of a Student Club, I want to modify an application for funding, so that the Student Club's event contains accurate information.
2. As an administrative member of a Student Club, I want to cancel an application for funding, so that the Faculty Administrators do not review it.
3. As a Faculty Administrator, I want to review funding applications submitted by Student Clubs.

Concurrency Use Cases

1. Students simultaneously RSVP to an event
2. Administrators of a Student Club amend an event simultaneously
3. Administrators of a Student Club submit a funding application for the same club simultaneously
4. Faculty Administrators review a funding application simultaneously
5. Administrators of Student Clubs modify an existing funding application simultaneously

Implemented Patterns

- Pessimistic Offline Lock Pattern 2: Preemptive Locking
- Pessimistic Offline Lock Pattern 2: Late Locking
- Optimistic Lock with database

Class Diagrams

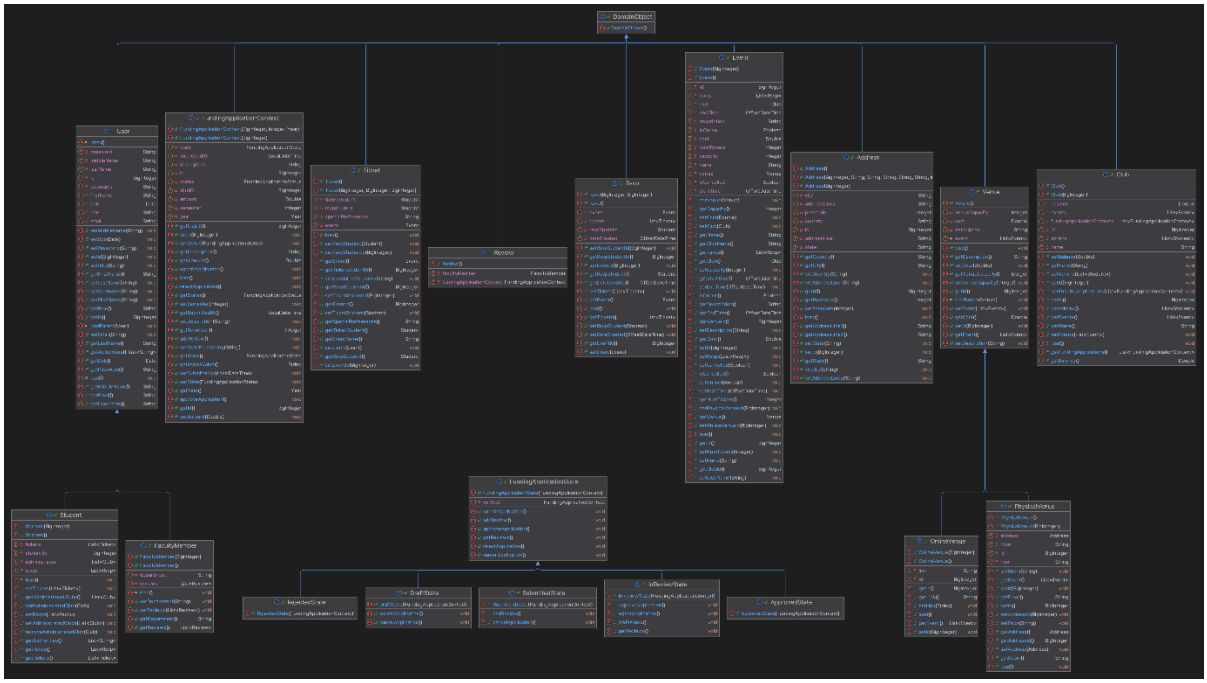
The overall structure of our application has remained the same, consisting of controller, service, domain, mapper and database connection layers. Information in the system ideally only passes between layers adjacent to it, and input and output types from each layer are standardized, as discussed in our report for part 2.

In this part, the main change to the class structure was the introduction of a LockManager class to manage pessimistic locking for the concurrency use cases (see ADR 1). These changes are represented in the LockManager -> Service Layer diagram below.

Domain Layer

This diagram shows the domain classes in our implementation, unchanged from part 2.

For better viewing:
https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part2/domain.png



Controller -> Service Layer

This diagram shows dependencies between our controller and service layers, unchanged from part 2.

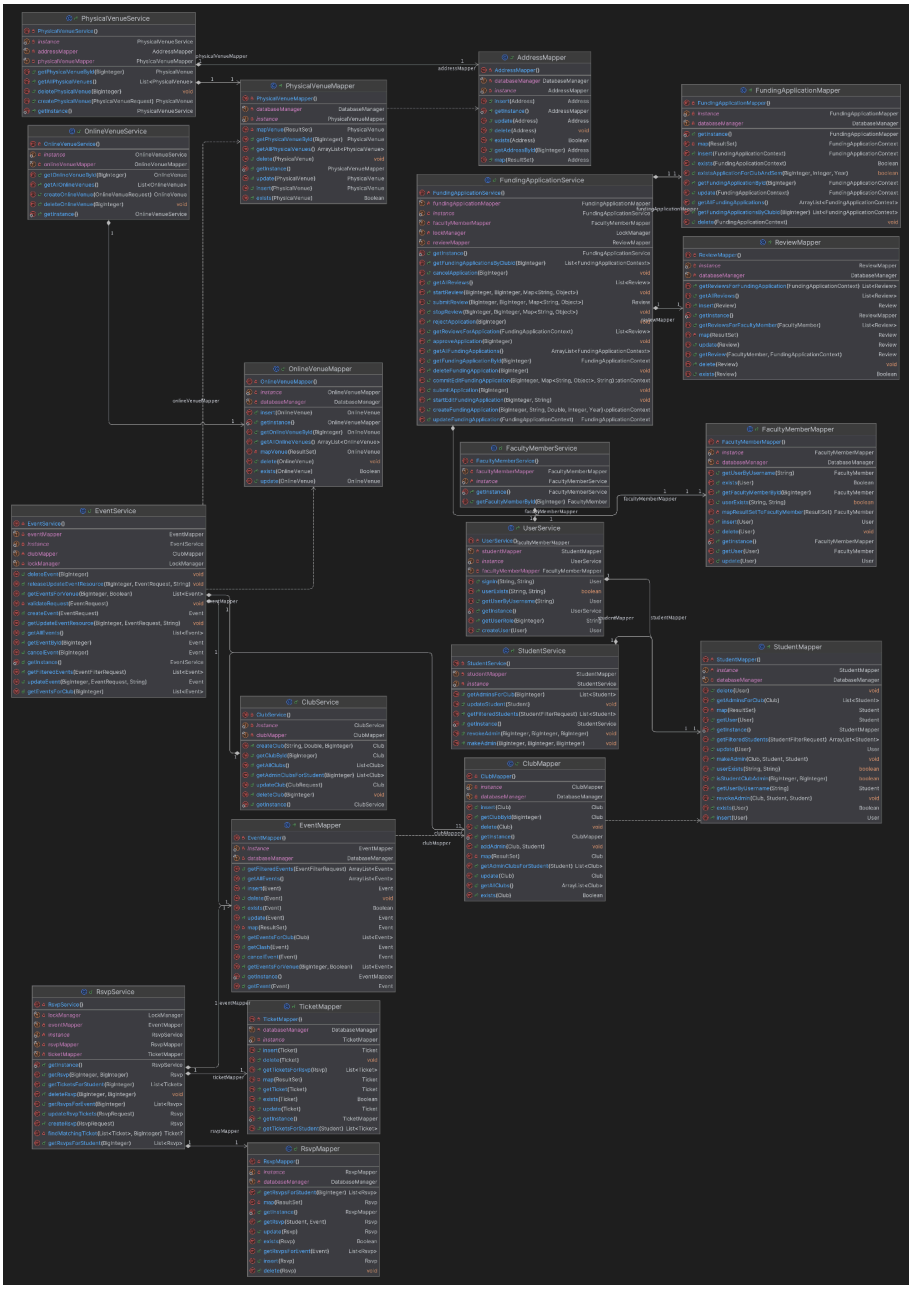
For better viewing:
https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part3/controllerService.png

Service -> Mapper Layer

The below diagram shows dependencies between the service and mapper layers. There are minor changes to this since part 2, in particular to incorporate the new funding application and review based functionality.

For better viewing:

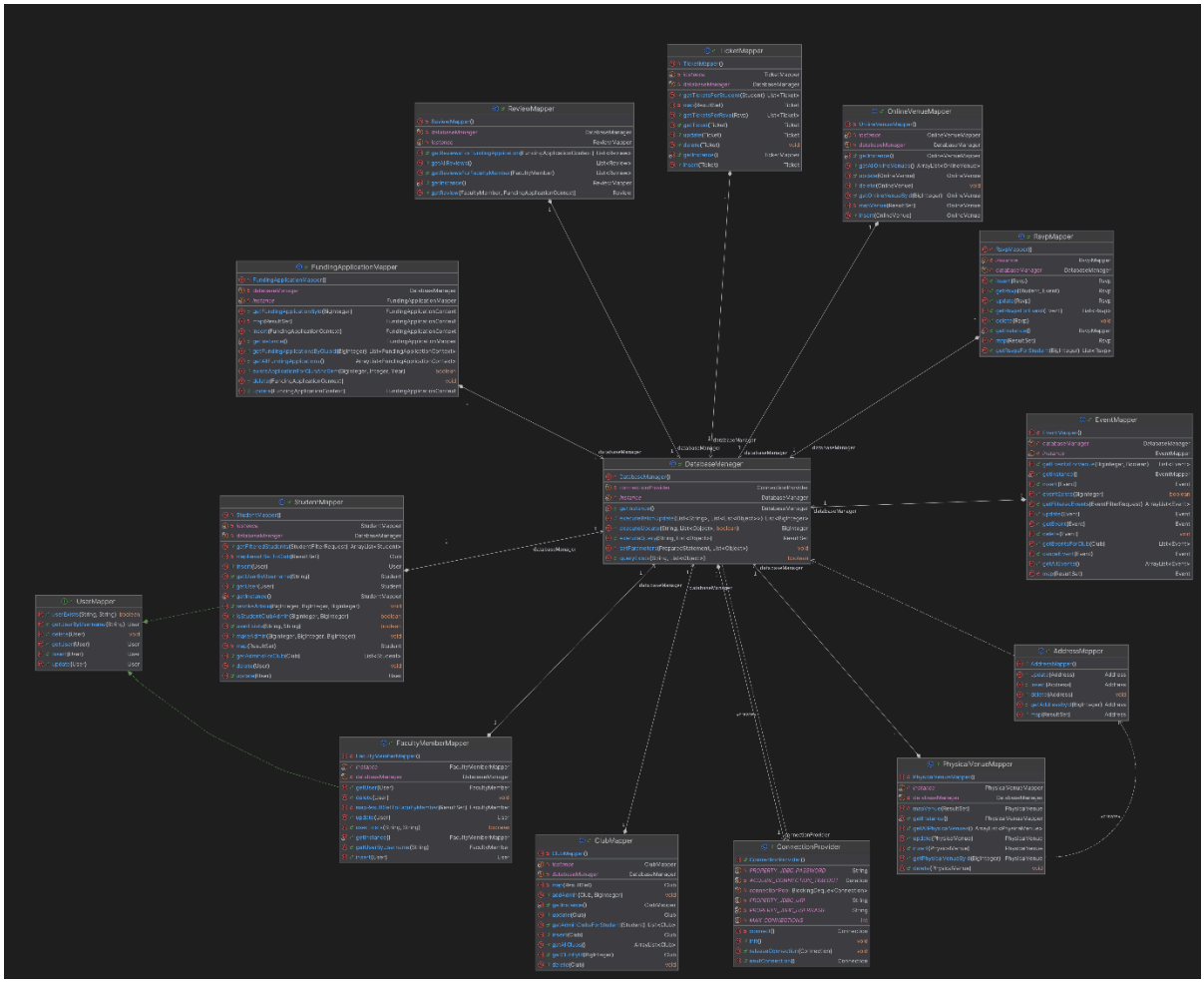
https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part3/serviceMapper.png



Mapper -> Database Connection

This diagram shows associations between classes in the mapper and DB connection layer, which acts as an adapter to the database to decouple query creation (mapper) and DB connection logic. There are no major changes to this from part 2.

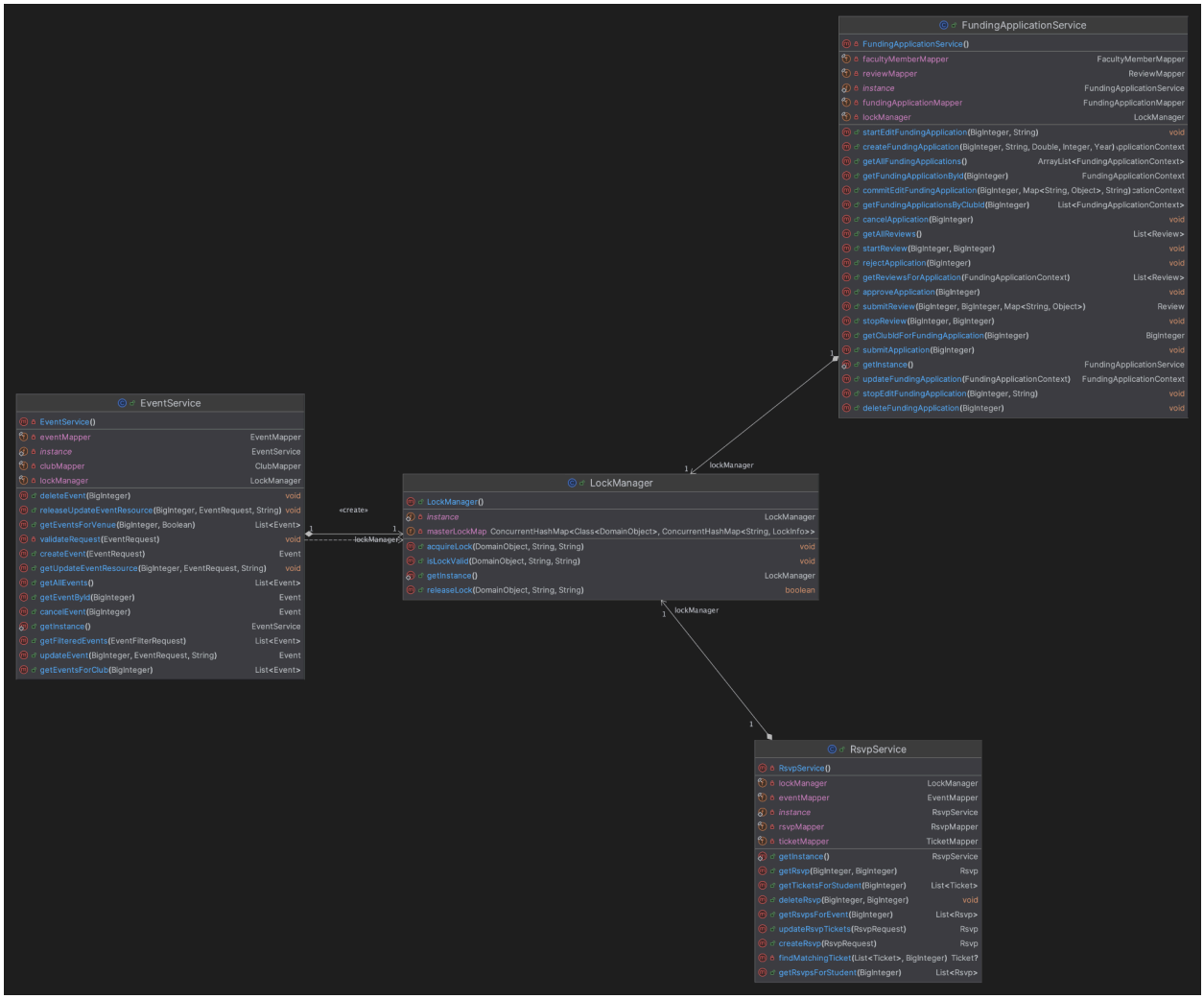
For better viewing: https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part3/mapperDb.png



Lockmanager -> Service Layer

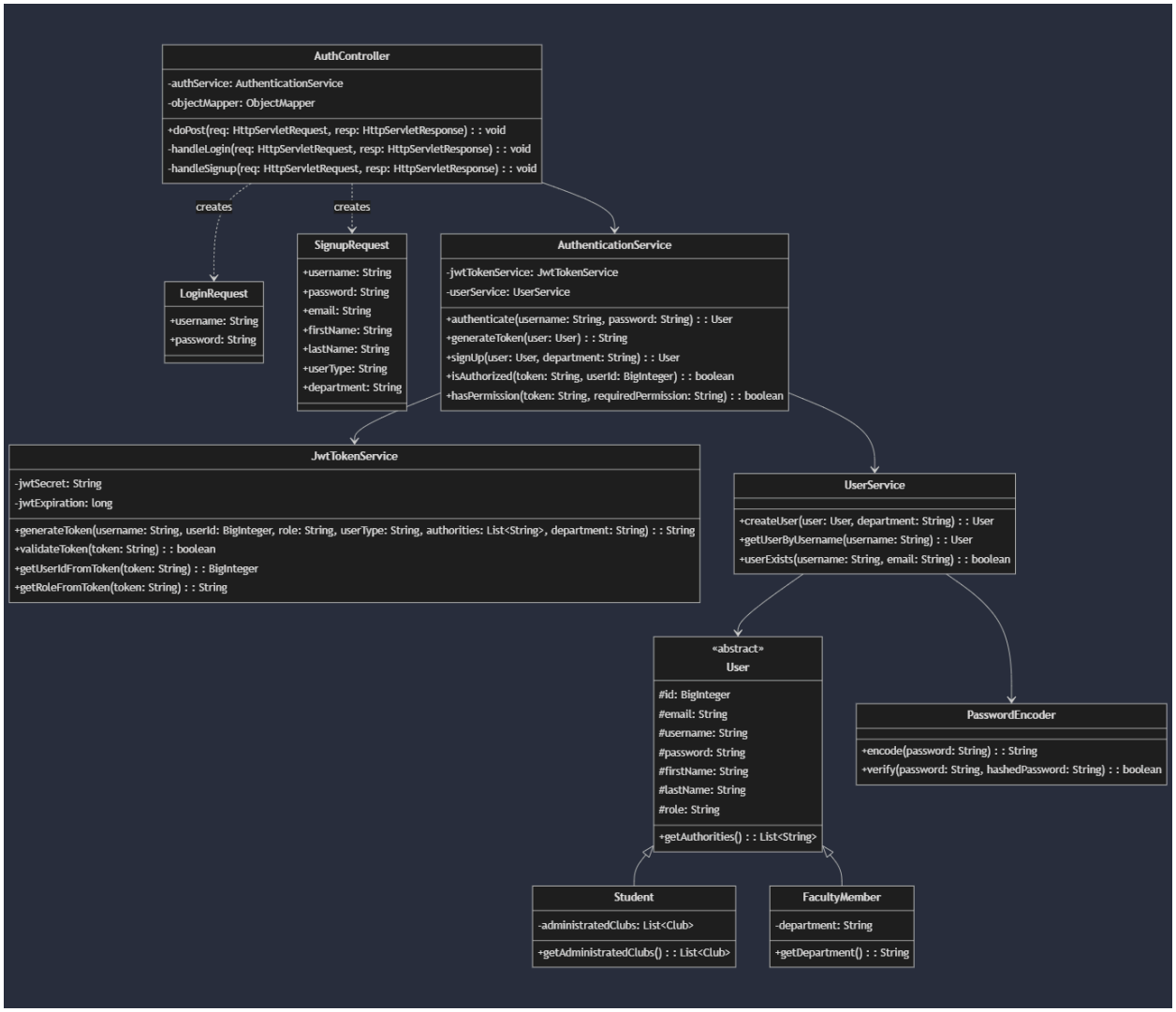
The LockManager class is new since part 2 and is used to implement pessimistic locks as discussed later in the report. It is only used from the service layer by service classes that use pessimistic lock, namely funding applications, events and RSVPs.

https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part3/lockmanager.png

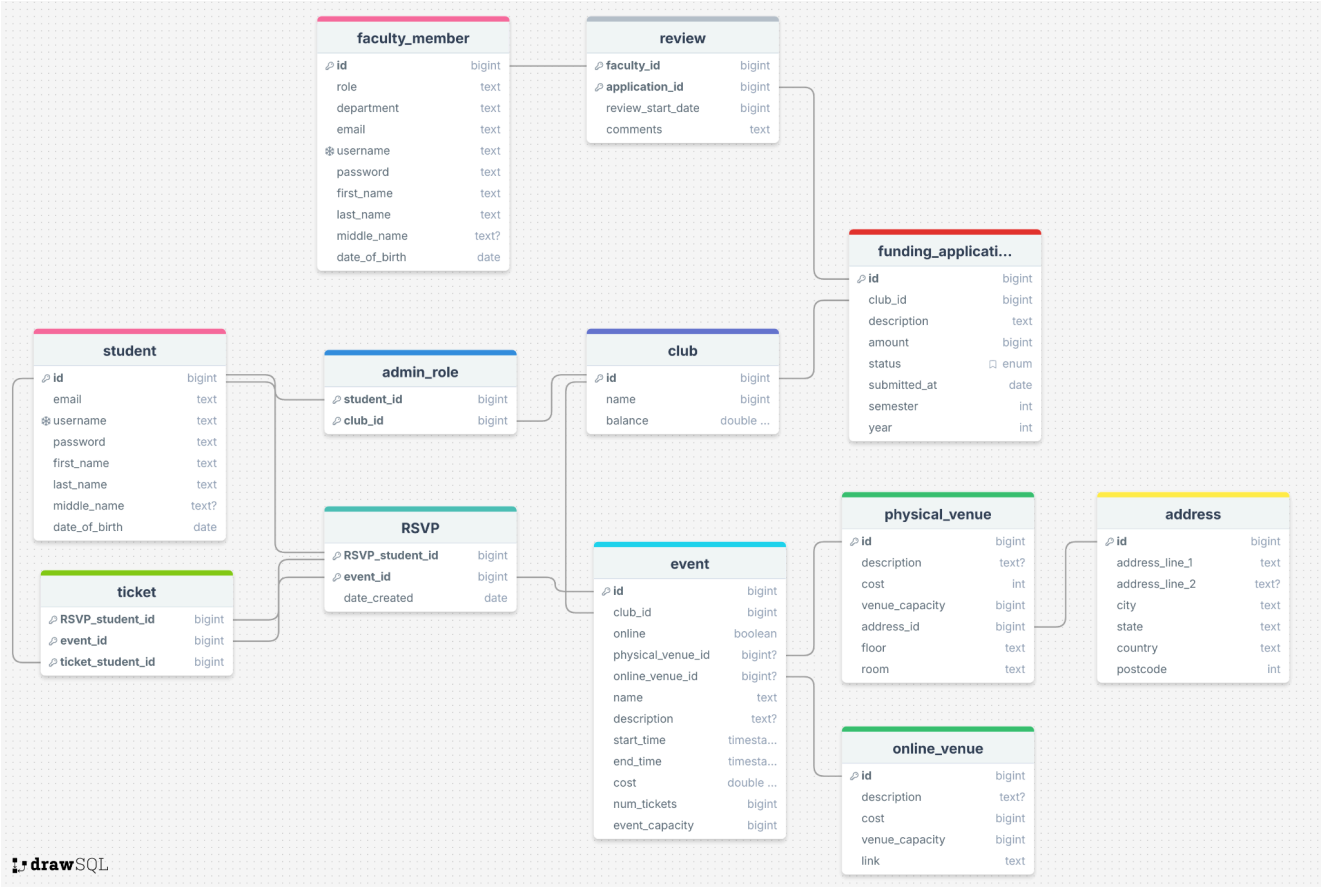


Authentication and Authorization

The authentication process has in theory been downgraded from last time to remove a lot of tech debt from the previous implementation making it more extendable and easier to apply fine grained permissions.



Database Schema



Note: Auxiliary classes such as config, util, test, filter, dto and exception classes have been excluded from the above diagrams to reduce clutter. The service layer -> domain relationships are also omitted due to impractically interpretable relationships, since each service class may interact with multiple domain classes.

Patterns and Descriptions ADR

ADR 1 - Pessimistic Offline Lock Pattern 1: Preemptive Locking

Context

In the development of our student club event management system, we've identified scenarios where concurrent data access poses significant risks to data integrity. These scenarios primarily involve multiple users attempting to concurrently modify the same or overlapping data.

Maintaining data integrity is imperative in our system, since many users have the ability to modify the same pool of data. It is important that we implement a mechanism to prevent situations where one user's edits inadvertently overwrite another's, ensuring that the system maintains consistent and accurate data.

On the other hand, we recognize that there can often be a tradeoff between data integrity and user experience. In implementing some of our concurrency patterns, we aim to minimise situations where users might lose their work due to concurrent editing conflicts, or be prevented from performing actions for excessive periods. This consideration is especially important given the collaborative nature of student club management.

Decision

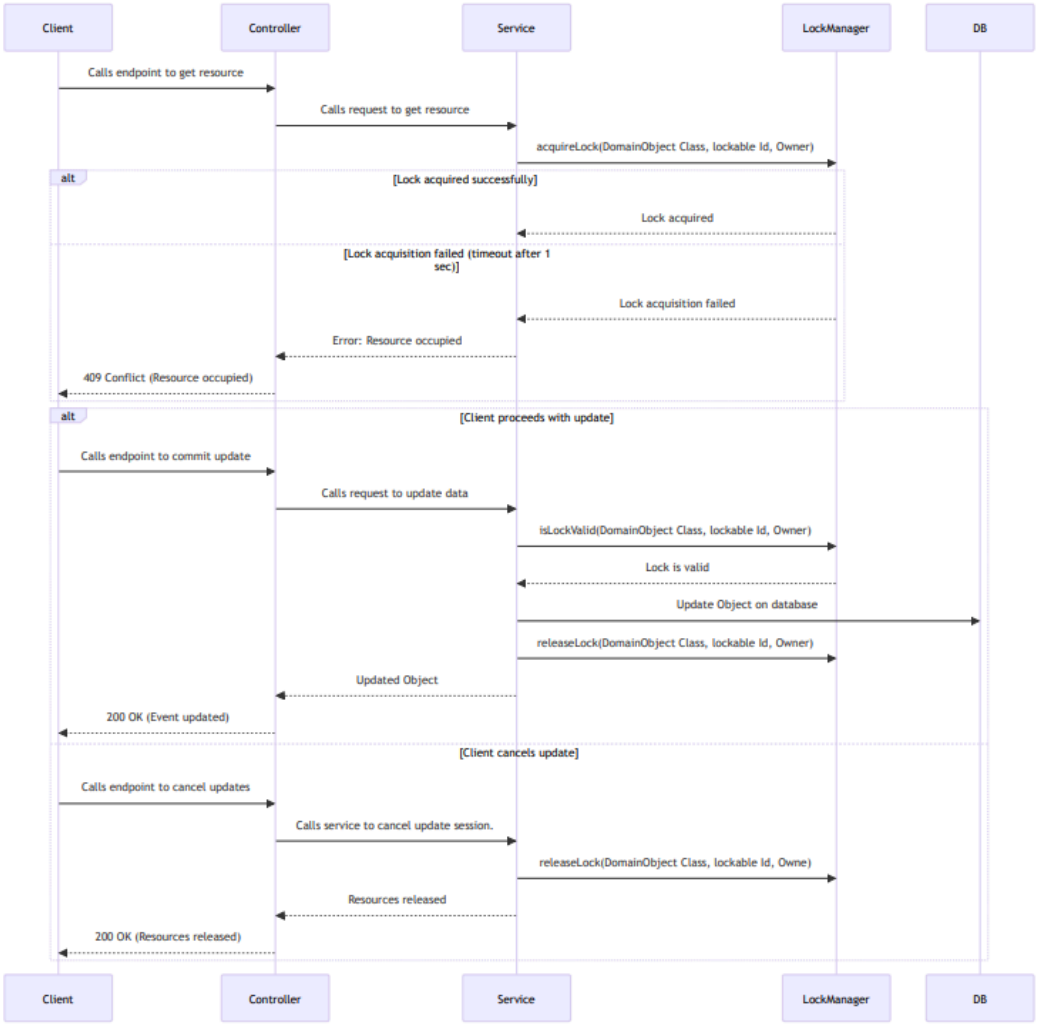
This pattern will be applied to scenarios involving editing screens, where the risk of data loss or conflict is highest.

We will deploy this pattern in three key areas of our application:

1. When multiple administrators of a Student Club attempt to amend event details simultaneously.
2. In instances where multiple Faculty Administrators are reviewing the same funding application concurrently.
3. When multiple administrators of Student Clubs are modifying an existing funding application at the same time.

Implementation Strategy

The sequence diagram below shows the approach that all implementations of this pattern will follow.



1. The Pessimistic Offline Lock is implemented using a LockManager class with three primary methods: acquire, check if valid, and release.
2. When a user initiates an edit, the system attempts to acquire a lock on the resource (identified by its primary key or unique attribute).
3. If the lock is successfully acquired, the user can proceed with editing. If not, they are informed that the resource is currently being edited by another user. So we fail first before they make any changes.
4. The lock is held until the user either commits their changes or explicitly releases the lock.
5. A timeout is set on each lock to prevent indefinite locking in case of user inaction or connection issues.
6. The system checks if the lock is still valid before committing any changes.

Consequences

Positive

The primary benefit of this approach over Optimistic locks is the prevention of edit losses for users. This mechanism significantly improves the user experience, as users will know upfront whether they can edit a resource. This transparency helps avoid the frustration that could arise from lost work due to conflicting edits.

Furthermore, this pattern ensures data integrity by maintaining consistency across the system. With only one user modifying a resource at any given time, we can be confident that our data is consistent.

Negative

The most significant impact it would have is the reduction in application liveliness. As the pattern allows only one user to edit a resource at a time, this may lead to wait times for other users attempting to access the same data.

There is also a risk of lock hogging. If a user acquires a lock and fails to release it, the resource could remain unavailable for an extended period up to our set timeout of one hour. This situation could lead to unnecessary delays and user frustration.

Implementation of this pattern introduces additional complexity to our system. The need to manage locks, handle timeout scenarios, and integrate locking mechanisms with our existing architecture increases the overall system complexity, which could impact maintainability and debugging.

Lastly, while we have implemented measures to mitigate deadlocks, such as timeouts, the potential for deadlocks may still exist for some use cases. This risk requires ongoing monitoring and may necessitate additional mitigation.

Compliance

1. Locked user actions require three endpoints to be implemented:
    a. One to obtain the locks from the database. Locks are the primary key of the table in the database.
    b. One to commit the changes. Method should check if the owner has write access before committing changes. Whole function should be in a try catch finally block, where you release the locks in the finally block.
    c. One to just release locks. In the chance where the user changes their mind and stops editing the file.
2. Frontend requests will need to call the method to acquire resources before committing.

Testing

To ensure the effectiveness and reliability of our concurrency implementations, we have developed a comprehensive testing strategy. Each use case has its own test servlet that runs a unit test that is designed to verify the correct functioning of the locking mechanism across various scenarios.

Test Environment Setup

- Each test case begins with a known, consistent state in the database.
- A specific resource (e.g., a funding application or event) is pre-created and its ID is known.
- Three concurrent threads are used to simulate multiple users attempting to access and modify the same resource simultaneously.

Test Execution

1. **Concurrent Access Simulation**
    - Multiple threads are initiated simultaneously, each representing a different user or process.
    - Each thread attempts to acquire a lock on the same pre-defined resource.
    - Once a lock is acquired, the thread proceeds to modify the resource (e.g., updating a funding application).
2. **Lock Acquisition Verification**
    - The system logs which thread successfully acquires the lock.
    - Failed lock acquisition attempts are also recorded.
3. **Resource Modification**
    - The thread that successfully acquires the lock attempts to modify the resource.
    - The modification is verified to ensure it reflects the changes made by the lock-holding thread.
4. **Concurrency Control Check**
    - The test verifies that only one thread can successfully modify the resource at a time.
    - Failed modification attempts by other threads are logged and counted.

Expected Outcomes

1. Only one thread should successfully acquire the lock and modify the resource.
2. The other two threads should fail to acquire the lock and, consequently, fail to modify the resource.
3. The final state of the resource should reflect the changes made by the single successful thread.
4. The system should maintain data integrity throughout the process, with no partial or conflicting updates.

Result Analysis

- Success is determined by verifying that:
    1. Only one thread successfully updates the resource at any given time.
    2. The resource's final state matches the update made by the successful thread.
    3. No concurrent modifications occur.

Author: Kah Meng

Changelog:

Version	**Changes**
V1.0	Designed Initial offline pessimistic lock pattern.
V1.1	Updated design to support offline locking of table unique keys. Now has MasterLockManager to store the table unique lock managers, and is accessed by the specific Class that is associated to the table
V2.0	Now requires two separate functions to work. One function is to get the lock. While the other function is to execute. Has timeout feature.
V2.1	Removed timeout feature as it was causing deadlock errors.

ADR 2 - Pessimistic Offline Lock Pattern 2: Late Locking

Context

As mentioned in ADR 1, we have identified critical scenarios where concurrent data access poses significant risks to data integrity and user experience.

In some concurrent scenarios, the user executes a write transaction that does not require extensive editing time. As such we deem that it would be acceptable to check if the update is valid on submission, as the consequence of potentially wasting a user's effort is low.

Decision

This pattern will be applied to scenarios where concurrent access control is needed, but risk of data loss or conflict is not as high.

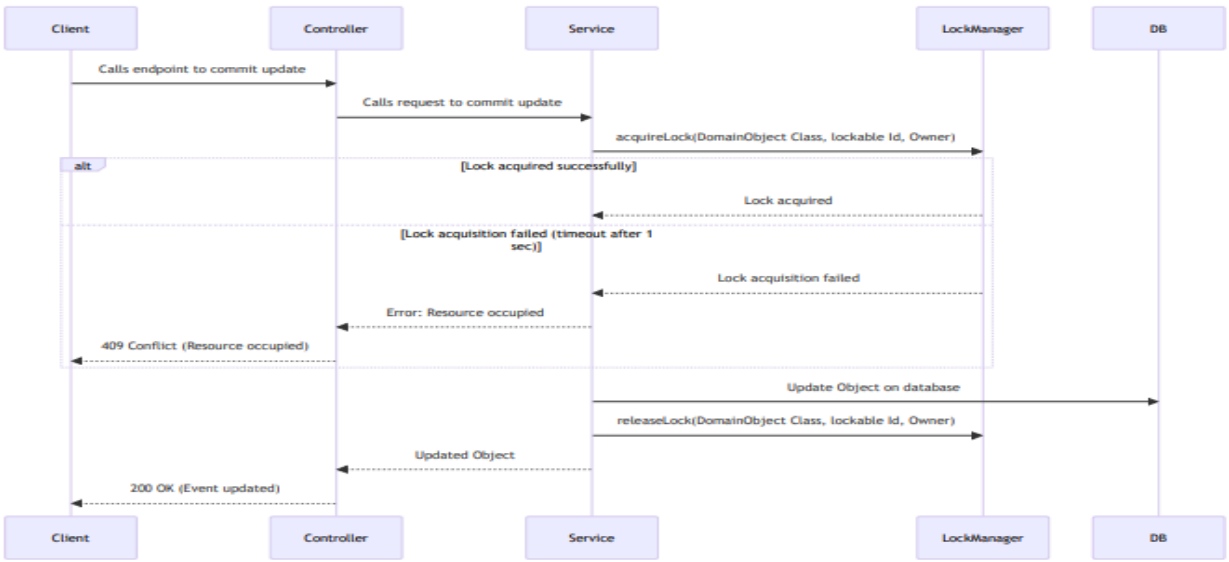
In particular we will implement this pattern in one use case of our application:

1. Concurrent creation and editing of RSVPs to an event

We have chosen this implementation of the pessimistic offline lock over the initial approach because the use case only involves users entering the student IDs for the students they are RSVPing for. In this situation, we believe that the advantage of maintaining system responsiveness outweighs the potential risk of minimal work loss for the user, as they would not have lost significant progress.

Implementation Strategy

The sequence diagram below shows the approach that all implementations of this pattern will follow.



The difference between this pessimistic approach compared to the first approach is the acquisition of the locks. In this approach, the lock is acquired and released in the same transaction. While in the first approach there are two separate functions to acquire the locks, and commit changes after.

Consequences

Positive

This allows the pattern to function similarly to an optimistic approach, but instead of checking the version numbers, it checks if there is write access at the start of the commit and obtains the locks. This guarantees that the update will go through without any race conditions when the lock is obtained. As such, it would allow different users to concurrently submit an RSVP to an event and improve liveness of the system.

Another benefit of this approach is the simplicity of implementing it. Unlike the first approach which requires 3 new endpoints, this approach would only require a refactor of the existing update service calls to acquire and release the locks.

Negative

As mentioned earlier, there is a possibility for users to have their work to be lost if the transaction fails. This however would not be an issue for the use cases we are implementing.

Compliance

To implement this pattern, service methods must contain the following logic:

1. Encapsulate current transaction code up to the return of the object in a try block
2. At the start of the transaction logic, acquire all the locks required for the transaction
3. In the finally block, release all the locks for the transaction

Testing

The pattern is tested using the testing strategy from ADR 1 - Pessimistic Offline Lock Pattern 1, where each use case would have its own test case.

Author: Kah Meng

Changelog:

Version	Changes
V1.0	This pessimistic lock pattern has been designed and implemented

ADR 3 - Error Handling

Context

In part 2, we ran into the issue of finding a consistent error handling practice, and to separate the error messages returned to the frontend vs those logged on the backend for developers. We wanted to make some error messages visible to the frontend, while hiding others that could potentially expose sensitive information or implementation details of our system to users.

Decision

We will implement an error-handling mechanism using a Jakarta web filter and a custom VisibleException class. This approach will ensure that only user-friendly error messages are sent to the frontend, while more detailed technical information (e.g., stack traces) is logged for backend developers.

We have decided to handle two types of exceptions:

- VisibleException: An exception that contains a user-friendly message and HTTP status code, designed to be returned to the frontend.
- Generic exceptions: Any other exception, which will log details such as the class name and stack trace on the backend but return a generic error message to the frontend.

Implementation Strategy

1. A global security filter intercepts all exceptions.
2. If a VisibleException is caught, it sends the specified message and status code back to the frontend, while also logging the stack trace on the backend.
3. For generic exceptions, the filter logs the stack trace for internal use but returns a standardised error message to the frontend (e.g., "An unexpected error occurred.").

Consequences

Positive

- Centralising and standardising error handling across the system simplifies maintenance, ensures consistency in responses, and reduces duplication of error-handling logic. This approach makes it easier to manage and update error behaviour globally, ensuring that all parts of the system follow the same error-handling conventions.
- We ensure that sensitive system information is not leaked to users, enhancing security.
- Backend developers retain access to detailed error information through logs, facilitating debugging.
- Custom user-friendly messages can improve the user experience by providing clear, relevant feedback when something goes wrong.

Negative

- There is added complexity in maintaining two types of error handling (user-visible vs. developer-logged).
- Developers must ensure that any custom exceptions intended for the frontend are properly handled through VisibleException.

Compliance

All future error-handling code will follow this approach. Any user-facing error must be handled using the VisibleException class, while other exceptions will be caught and logged as generic errors. This decision applies globally across our Jakarta-based system to ensure consistency in error handling.

Alternatives considered

Return all error details to the frontend

- Positive: Simplifies error handling, as developers wouldn't need to separate frontend-visible and backend-specific errors.
- Negative: Exposes sensitive information and potentially reveals system internals, increasing the risk of exploitation.

Log everything internally without distinguishing frontend error messages

- Positive: Ensures that all error details are captured for debugging purposes.
- Negative: Provides a poor user experience, as users would receive vague or overly technical error messages.

Author: Henry Hamer

Changelog:

Version	Changes
V1.0	Initial implementation of error handling strategy with VisibleException class and Jakarta web filter for separating frontend and backend error messages.

ADR 4 - Authentication

Context

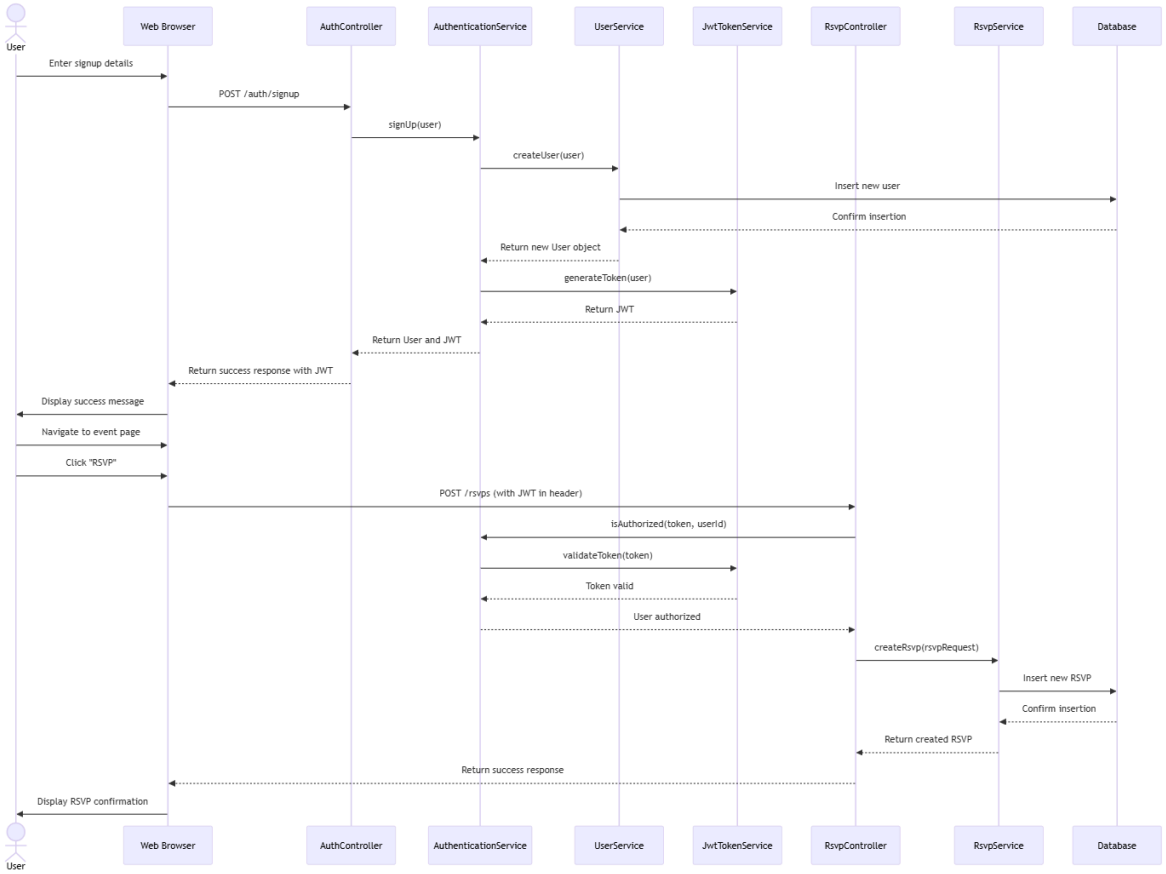
In part 2, we created an authentication and authorization system that involved JWTs and refresh tokens to provide a holistic version of authentication. This would enable full control over users states and session with the refresh tokens. However, this implementation became very bloated and was cumbersome to work with. We needed something slimmer that would be extensible to apply to the increased needs for authentication with faculty members and funding applications.

Decision

We decided to scrap the old authentication completely and reimplement a simpler version. This would just involve JWTs and a filter. Thus, enabling us to create an authentication service which will hold the authorization methods that we can use on any controller that needs authentication.

Implementation Strategy

1. A global security filter intercepts all requests
2. A public paths function is used in the filter to check if the requests needs to include an authorization header. This enables people to access some endpoints without logging in
3. If a request is not public then it will need to get process by the authentication to ensure that the token provided is a valid token
4. Once a request has been cleared by the filter it can be processed at the controller level by more fine grained authentication functions applied to specific end points



The above is a system sequence diagram of the process of a user creating an account, getting the JWT from the server. The user then uses the token to send with a request for an RSVP, the token is validated and the user is confirmed as the user making the request. This then lets them create an RSVP before returning it to the frontend.

Consequences

Positive

- Centralised processing of what requests can be passed on to the controllers and pre verifying the token when required
- A more extensible design that allows us to easily create detailed authentication for different end points as required and easily debug any issues
- Removal of lots of tech debt - this will help if we need to continue future development as we can move more quickly

Negative

- Loss of the ability to manage user sessions - this makes it difficult as there is no way to invalidate a users token before it expires
- Need to keep updating the authenticationService whenever a new type of authentication is required which will make the class bloated.

Compliance

Need to ensure that if a new end point is added - the globalSecurityFilter must be checked to ensure that your request will get passed through. For controllers that require authentication it must be instantiated in the controller

Alternatives considered

Maintain the previous implementation

- Positive: time saved on reimplementation of the new method
- Negative:increased difficulty when extending the old authentication implementation

Author: James Launder

Changelog:

Version	**Changes**
V1.0	Discussion of authentication and including of diagrams

Concurrency Use Case ADRs

ADR - Concurrent RSVP to an Event

Context

In our student club event management system, the RSVP process is a critical and frequently used feature. Multiple users may attempt to RSVP to the same event simultaneously, potentially leading to race conditions or inconsistencies in event capacity management. The RSVP process is typically quick, but ensuring data integrity is crucial.

A pessimistic locking pattern that acquires and releases the lock in a single transaction can help in this context because:

1. It would perform functionally similarly to optimistic locking, allowing for high concurrency.
2. It reduces the number of database queries compared to traditional pessimistic locking, potentially improving performance.

Decision

1. Implement a pessimistic late locking pattern (see ADR 2 above) for the RSVP creation and editing process.
2. This pattern will be applied when users attempt to RSVP to an event or modify their existing RSVP.

Implementation Strategy

We will implement the implement offline pessimistic late lock as outlined in the ADR 2:

1. When a user attempts to RSVP or edit their RSVP, initiate a transaction.
2. Within the same transaction, attempt to acquire a lock on the event record with a 1-second timeout.
3. If the lock is acquired, proceed with the RSVP creation or modification.
4. If the lock acquisition fails, return the error to the user.
5. After performing the RSVP operation, commit the updates of the transaction, which will automatically release the lock.

Consequences

Positive

Reducing the number of database queries while maintaining good liveness is the main advantage of this approach compared to an optimistic locking approach, since RSVPS are only locked for the duration of the transaction, rather than the duration of the user's edit. This can improve overall system performance and reduce the database load.

This approach also prevents race conditions and conflicts, which is crucial for managing event RSVPs consistently.

Negative

The increased complexity of implementation is a notable drawback, as it requires careful resource management to make sure that locks are only acquired for the successful transaction, potentially leading to more complex code and increased development time.

There's also a potential for decreased throughput if many operations fail to acquire the lock and must be retried by users, which could lead to user frustration and increased system load, particularly during popular events or high-traffic periods.This drawback is shared with the optimistic approach.

Testing

Method
A test case has been created to test concurrent attempts at submitting the same RSVP request to the same event from the same user. This test case can be reached using the endpoint {{base_url}}/test1, and it is handled by the controller ConcurrentTest1.

Start State
The test should have the following starting state

- An event with sufficient capacity for the RSVP
- A student who will create the rsvp
- Another student who does not have a ticket to that event, who will be RSVP'd on behalf of by student one

Results
The test shows that only one of the threads managed to get access to the resource at any given time. The other threads that failed to obtain the lock threw an error.

```
Test 1 starting..
pool-3-thread-2 acquired lock 312 of class type class com.brogrammerbrigade.backend.domain.Ticket
pool-3-thread-2 acquired lock 302 of class type class com.brogrammerbrigade.backend.domain.Ticket
Failed to acquire lock
pool-3-thread-2 acquired lock 312 of class type class com.brogrammerbrigade.backend.domain.Rsvp
pool-3-thread-2 acquired lock 2 of class type class com.brogrammerbrigade.backend.domain.Event
Lock owner mismatch. Current owner: pool-3-thread-2, Releasing owner: pool-3-thread-1
Executing query: SELECT * FROM app.rsvp WHERE RSVP_student_id = ? and event_id = ?
Failed to acquire lock
```

The thread that obtained the lock updated the RSVP successfully and released the locks. The image below shows that only one update managed to go through, demonstrating that data integrity has been maintained.

```
Thread 1 successfully updated the RSVP.
Concurrent RSVP create test completed.
Successful creations: 1
Failed creations: 2
Executing query: SELECT * FROM app.rsvp WHERE RSVP_student_id = ? and event_id = ?
Final RSVP state: 31
```

Considered Alternative 1 – Optimistic Lock

Instead of pessimistic, we considered using an optimistic locking approach.

Consequences

Positive:

- Simpler to implement, as it doesn't require explicit lock management.
- Potentially higher throughput in low-contention scenarios.

Negative:

- More calls to the database, as it requires checking version numbers or timestamps on each operation.

Ultimately the single-transaction pessimistic locking approach was chosen over optimistic locking because it provides a good balance between concurrency and data integrity, while potentially reducing the number of database calls in high-contention scenarios. It also provides immediate feedback to users about contention, rather than detecting conflicts at commit time, which can lead to a better user experience in this quick RSVP process.

Author: Kah Meng

Changelog:

Version	**Changes**
V1.0	Implemented pessimistic lock pattern 2 on this use case

ADR - Concurrent Event Editing

Context

Multiple club administrators may attempt to edit the same event simultaneously, which has the potential to cause conflicting updates. In this case, concurrent updates not only to the event but also to the hosting club need to be considered, as updates to an event's cost will affect club balance.

Decision

To address the challenges of concurrent event editing, we have decided to implement a preemptive pessimistic offline lock pattern, whereby only one user can access the edit window for an event at a time. This pattern will be specifically applied in scenarios where multiple administrators of a Student Club attempt to amend event details simultaneously.

Implementation Strategy

We will implement offline pessimistic lock as outlined in the ADR - Pessimistic Offline Lock Pattern 1:

1. Acquire locks (getUpdateEventResource): This step will lock both the event and associated club resources, ensuring exclusive access.
2. Perform update (updateEvent): Here, we will validate lock ownership, make the necessary changes, and commit updates.
3. Release locks (releaseUpdateEventResource): Locks will be released after a successful update or in case of cancellation.

To track ownership across different threads, we will use the username from the authentication token.

Consequences

Positive

Admins will be notified if they are unable to edit an event before they make any changes, hence preventing any potential lost updates. It will also ensure data integrity is maintained.

Additionally, it allows for complex, multi-resource updates involving both event details and club finances to be handled atomically, and once it obtains the resource. The update is guaranteed to go through, assuming the update is valid.

Negative

This approach may reduce system liveness as only one administrator can edit an event at a time, which could lead to lock contention if multiple administrators frequently attempt to edit popular events.

As mentioned earlier in ADR - Pessimistic Offline Lock Pattern, the issue of long-held locks are applicable here. An attempt to mitigate this issue has been made by calling the release endpoint when the browser closes on the page but it might not cover all possibilities. Issues such as added complexity are applicable as well.

Testing

Method
A test case has been created to test concurrent attempts at editing the same event from different admins. This test case can be reached using the endpoint {{base_url}}/test2, and it handled by the controller ConcurrentTest2.

Start State
The test should have the following variables

1. A test event that currently exist on the database
2. Two requests that simulates the request the event service would receive from the controller.

Results
The test shows that only one of the threads managed to get access to the resource at any given time. The other threads that failed to obtain the lock threw an error.

```
Test 2 starting..
Executing query: SELECT * FROM app.event WHERE id = ?
Executing query: SELECT * FROM app.event WHERE id = ?
Executing query: SELECT * FROM app.event WHERE id = ?
Executing query: SELECT * FROM app.club WHERE id = ?
Executing query: SELECT * FROM app.club WHERE id = ?
Executing query: SELECT * FROM app.club WHERE id = ?
0 acquired lock 1 of class type class com.brogrammerbrigade.backend.domain.Event
Failed to acquire lock
Failed to acquire lock
0 acquired lock 1 of class type class com.brogrammerbrigade.backend.domain.Club
```

The thread that obtained the lock updated the event successfully and released the locks. Image below shows that only one update managed to go through at any given time. Hence proving that our system is able to maintain data integrity.

```
Lock released successfully
Lock released successfully
Event has been updated to :Updated Annual Hackathon
Thread 0 successfully updated the event.
Concurrent event update test completed.
Successful updates: 1
Failed updates: 2
Executing query: SELECT * FROM app.event WHERE id = ?
Final event state: Join us for a 48-hour coding challenge!
```

Considered Alternative 1 – Optimistic Lock

Instead of pessimistic locking, we considered using optimistic locking for event editing.

Consequences

Positive:

- Multiple administrators could attempt to edit the event simultaneously.
- No need for explicit lock management.
- Potentially better performance in scenarios with low contention.

Negative:

- Effort might be wasted when an admin finds out he is unable to edit an event at the end of his changes.
- Potential for frequent conflicts in high-contention scenarios, leading to user frustration.

We believe that since the event is likely accessed by many different admins of the club, we believe that sacrificing liveliness and prioritising user experience by reducing conflicts is the better choice.

Author: Kah Meng

Changelog:

Version	**Changes**
V1.0	Implemented pessimistic lock pattern 1 on this use case

ADR - Concurrent Submission of Funding Application

Context

Student clubs apply for budget funding through funding applications submitted by club admins. These applications are limited to one per semester per club and include a description of purchases and amount applied for. Multiple club admins may try to submit a funding application simultaneously, so it is important to ensure the constraint is adhered to.

Decision

The optimistic offline lock pattern was chosen to be the most suited for this function. It was decided that liveliness of submitting applications was important as a club could want to submit applications for multiple semesters which would not be allowed or be complex to manage if pessimistic locking patterns were used instead.

Furthermore in this business case, we believe that the frequency and probability of conflicts is low as we expect student administrators for a club would communicate with each other to a reasonable degree. Additionally, since only one funding application may be submitted per semester, the very low frequency of creating and submitting applications so the benefits of the added liveness outweighs the negatives.

Implementation Strategy

1. Create funding application (createFundingApplication): this takes in required data and creates a funding application object.
2. Check if a funding application already exists (existsApplicationForClubAndSem): this checks the database if an application for the club already exists for the given year and semester. Stops the business transaction if an application exists.
3. Commit to database (insert): inserts the new funding application to the database. A unique constraint on the club id, year and semester keys ensures the single application a semester is maintained.

Consequences

Positive

The application has increased liveliness since there are no locking mechanisms. This would allow multiple administrators to draft and submit applications for different semesters concurrently.

Furthermore, this strategy is relatively simple to implement. In the current implementation, there is simply a check at the end of the business transaction and a constraint on the database which enforces the business rules.

The lack of explicit lock management which would be present in alternatives also improves performance.

Negative

The key drawback to this approach is the potential for lost work where an admin would try to submit an application for a semester and year that another administrator had just submitted.

There is a potential concurrency issue in the fringe case where the check in the service layer happens at exactly the same time for two applications before either of the SQL commands to create a new funding application is executed. However, the unique key constraint in the database will still reject the duplicate application.

Testing

Method
The test case for this use case tests three student administrators attempting to concurrently submit a funding application for the same club, semester and year. The test case can be reached using the endpoint {{base_url}}/test2 and it is handled by ConcurrentTest3.

Start State

1. Club with id 31 exists.
2. Authenticated as an administrator of the club with id 31.
3. Three requests that simulate the request the funding application service will get from the controller.

Results
3 threads tried to create a funding application at the same time.

```
Thread 0 successfully created a funding application.
Thread 1 failed: Failed to create funding application: ERROR: duplicate key value violates unique constraint "funding_application_club_id_semester_year_key"
  Detail: Key (club_id, semester, year)=(31, 2, 2027) already exists.
Thread 2 failed: Failed to create funding application: ERROR: duplicate key value violates unique constraint "funding_application_club_id_semester_year_key"
  Detail: Key (club_id, semester, year)=(31, 2, 2027) already exists.
Concurrent Funding app create test completed.
Successful creations: 1
Failed creations: 2
```

As seen in the above test logs, Thread 0 was first and successful in creating the funding application. Now that the funding application exists, threads 1 and 2 fail due to the unique constraint on club id, semester and key. This is evidence that the system is handling this concurrent use case correctly.

Considered Alternative 1 – Pessimistic Lock

Instead of optimistic, we considered using pessimistic locking for submitting funding applications.

Consequences

Positive:

- No lost work: Application submissions are certain to be successful once the lock is required and the system is functioning as intended.
- Prevents conflict: Administrators will be notified if there is currently a funding application in the process of being submitted, preventing conflicting submissions.

Negative:

- Reduced liveness: Due to locks preventing concurrent submission of applications potentially for different semesters which should be allowed.
- More complex implementation: Need to implement explicit lock management.

Author: Kevin Wu

Changelog:

Version	**Changes**
V1.0	Implemented optimistic strategy on the use case.

ADR - Concurrent Review of Funding Application

Context

In our student club event management system, funding applications submitted by clubs require a review by faculty members before it is approved.The review process involves making a decision (approve or reject) and providing comments. The review can only be performed on applications in a 'submitted' state, and once approved or rejected, the application cannot be further edited.

Multiple faculty members may attempt to review the same funding application simultaneously. This is expected to occur frequently, as all faculty members have access to review all submitted funding applications.

Decision

To address the challenges of review editing and submission, we have decided to implement a pessimistic offline lock pattern. This pattern will be specifically applied in scenarios where multiple faculty members attempt to review a funding application simultaneously.

Implementation Strategy

We will implement offline pessimistic lock as outlined in the ADR - Pessimistic Offline Lock Pattern 1:

1. Acquire lock (startReview): Lock the funding application resource.
2. Perform review (submitReview): Validate lock ownership, make the decision, add comments, and commit the review.
3. Release lock (stopReview): Release the lock if faculty member decides to not review the funding application anymore.

We will use faculty member IDs as lock owners to track lock ownership.

Consequences

Positive

Provides a clear indication to faculty members when an application is already under review., hence preventing any lost updates. Data integrity is also maintained.

It has additional benefit of simplifying the review process by guaranteeing exclusive access to the application during review.

Negative

It may reduce system liveliness as only one faculty member can only review an application at a time.

As mentioned earlier in ADR - Pessimistic Offline Lock Pattern, the issue of long-held locks are applicable here. An attempt to mitigate this issue has been made by calling the release endpoint when the browser closes on the page but it might not cover all possibilities. Issues such as added complexity are applicable as well.

Testing

Method
A test case has been created to test concurrent attempts at reviewing the same funding application from different faculty members.. This test case can be reached using the endpoint {{base_url}}/test4, and it is handled by the controller ConcurrentTest4.

Start State
The test should have the following variables

1. An existing applicationId that is from an application that is in the SUBMITTED state.
2. Generate reviewData that the review will be conducted on. In our testcase, it approves the funding application.

Results
The test shows that only one of the threads managed to get access to the resource at any given time. The other threads that failed to obtain the lock threw an error.

```
Thread 0 failed: Faculty member not found with ID: 0
1 acquired lock 39 of class type class com.brogrammerbrigade.backend.domain.Review
Failed to acquire lock
Thread 2 failed: Concurrency exception, 2 could not acquire lock for 39
```

The thread that obtained the lock updated the event successfully and released the locks. Image below shows that only one update managed to go through at any given time. Hence proving that our system is able to maintain data integrity.

```
Thread 1 successfully submitted review.
Concurrent review submission test completed.
Successful submission: 1
Failed submission: 2
```

Considered Alternative 1 – Optimistic Lock

Instead of pessimistic locking, we considered using optimistic locking for funding application reviews.

Consequences

Positive:

- Better system liveness, as multiple admins will be able to attempt to review the application simultaneously.

Negative:

- Use of system liveness is redundant as only one review can be done at one time. Hence it would be better to lock other users out once a review is in progress.

Author: Kah Meng

Changelog:

Version	**Changes**
V1.0	Implemented the pessimistic pattern on the use case.

ADR - Concurrent Editing of Funding Application

Context

Funding applications are critical documents that can only be submitted once per semester for each club. These applications contain important details such as the description of fund usage and the amount requested. Multiple club administrators may attempt to edit the same funding application simultaneously.

The editing process may be time-consuming, potentially lasting several minutes or longer, due to the importance and complexity of the information being entered. As of right now, the description field is very basic. But our system is built so that an admin can edit the funding application for a long period of time with the secure knowledge that their efforts will not be wasted. This is especially important given that only one application can be submitted per semester.

Decision

To address the challenges of concurrent funding application editing, we have decided to implement a pessimistic offline lock pattern. This pattern will be specifically applied in scenarios where multiple administrators of a Student Club attempt to amend funding application details simultaneously.

Implementation Strategy

We will implement offline pessimistic lock as outlined in the ADR - Pessimistic Offline Lock Pattern 1:

1. Acquire lock (startEditFundingApplication): Lock the funding application resource.
2. Perform update (commitEditFundingApplication): Validate lock ownership, make changes, and commit updates.
3. Release lock (stopEditFundingApplication): Release lock after successful update or in case of cancellation.

To track ownership across different threads, we will use the username from the authentication token.

Consequences

Positive

This will allow admins to have detailed uninterrupted editing sessions, which is beneficial for potentially complex descriptions and justifications for their funding application.

Additionally admins will be notified if they are unable to edit an funding application before they make any changes, hence preventing any potential lost updates. It will also ensure data integrity is maintained.

Negative

As with other use cases that implements pessimistic locks, it will suffer from liveliness issues as only one administrator can edit a funding application at a time, which could lead to lock contention. Other downsides such as long-held locks apply in this use case too.

Testing

Method
A test case has been created to test concurrent attempts at editing the same funding application from different admins. This test case can be reached using the endpoint {{base_url}}/test5, and it handled by the controller ConcurrentTest5.

Start State
The test should have the following states

1. A test funding application that currently exist on the database, and is in the state of SUBMITTED or IN_DRAFT
2. An updates map that simulates the update the admin might incur.

Results

The execution results show that only one user managed to obtain a lock at a time. The appropriate exception is shown where it is unable to acquire a lock.

```
Test 5 starting..
Executing query: SELECT * FROM app.funding_application WHERE id = ?
Executing query: SELECT * FROM app.funding_application WHERE id = ?
Executing query: SELECT * FROM app.funding_application WHERE id = ?
0 acquired lock 35 of class type class com.brogrammerbrigade.backend.domain.FundingApplication.FundingApplicationContext
Executing query: SELECT * FROM app.funding_application WHERE id = ?
Failed to acquire lock
Failed to acquire lock
Thread 2 failed: Concurrency exception, 2 could not acquire lock for 35
Thread 1 failed: Concurrency exception, 1 could not acquire lock for 35
```

The thread that obtained the lock updated the funding application successfully and released the locks. Image below shows that only one update managed to go through at any given time. Hence proving that our system is able to maintain data integrity.

```
Thread 0 successfully updated Funding App.
Concurrent Funding App edit test completed.
Successful edit: 1
Failed edit: 2
```

Considered Alternative 1 – Optimistic Lock

Instead of pessimistic, we considered using optimistic. Similar to updating events and reviews, the benefit of system liveliness is not as beneficial to the use case compared to potentially having work lost. This would be very frustrating, as the admin might have spent a long time editing a funding application only to realise it is not able to update at the end, and might have to re-edit the application again.

Author: Kah Meng

Changelog:

Version	**Changes**
V1.0	Implemented pessimistic lock on the use case.