SDA part3 Report

1 Handling Concurrent RSVPs for Events

Context

This issue is common in scenarios where multiple students are trying to RSVP to the same event simultaneously. When many students try to RSVP at once, the backend processes each request concurrently without locking or queuing access to the available spots. If there is no proper concurrency control, multiple students may believe they have successfully reserved a seat when they actually haven't, leading to overbooking and user disappointment.

Decision

We have decided to use an **optimistic locking mechanism combined with a retry strategy** to handle concurrency issues.

Optimistic locking allows us to detect conflicts when students RSVP simultaneously. Each transaction checks the version number of the data before committing. If the data has been modified by another transaction during this period, the transaction will fail and retry. The retry mechanism automatically re-executes the user's operation in case of conflicts, without requiring the user to manually resubmit (note that if the concurrency is too high and the retry limit is reached, an exception will be thrown). Users will hardly notice the conflict and retry process, resulting in a smoother user experience.

Implementation Strategy

When a student tries to RSVP, the system will read the current availability of capacity. If the available capacity at the time of submission is inconsistent with what the student initially saw, the RSVP will be rejected, and the student will be prompted to try again. The steps are as follows (Figure 1):

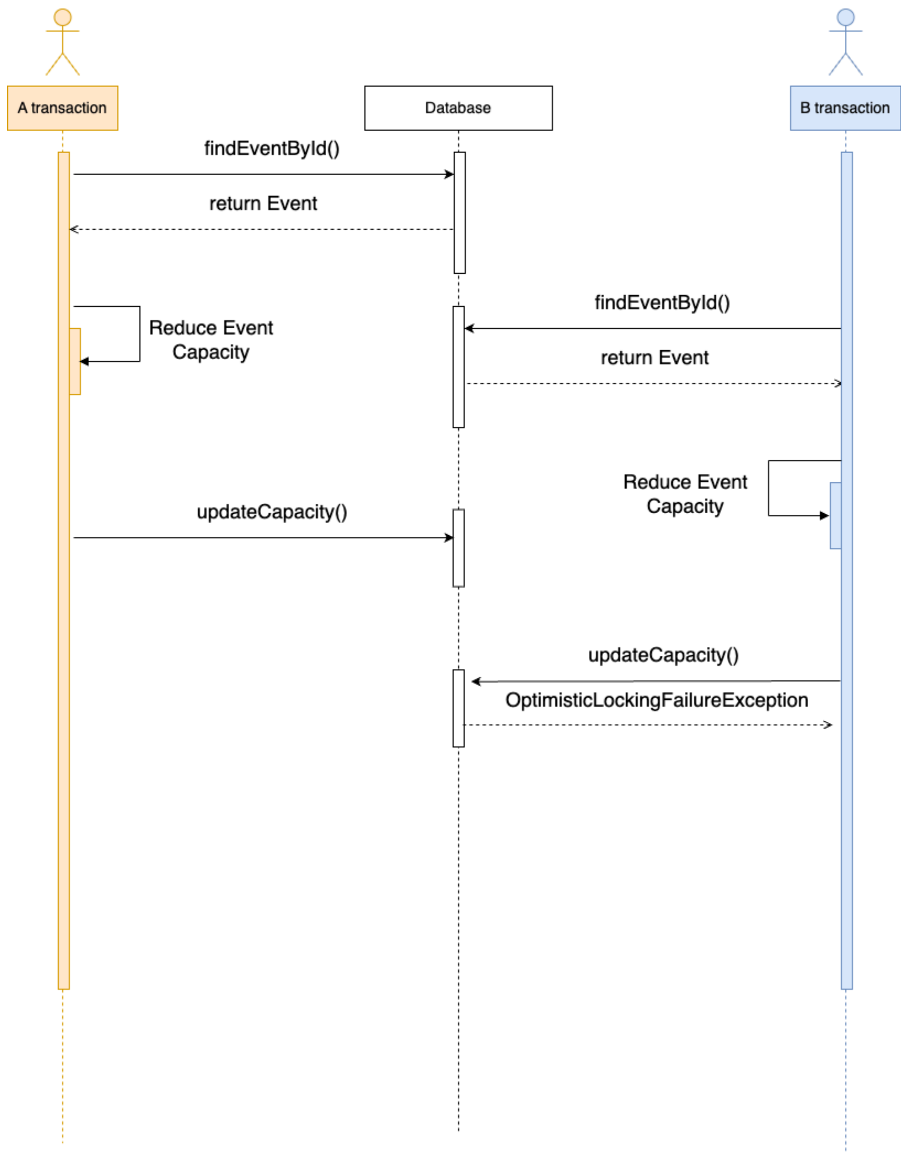


Figure 1 - Optimistic locking for concurrent event updates

Step 1: Read the Record and Store the Version Number

In the `applyForRSVP` method, first read the `Event` object using `eventRepository.findEventById(eventId)`. At this point, the read `Event` entity includes the version number `capacity_version`.

Step 2: Modify the Record

Modify the capacity of the `Event` object by executing `event.setCapacity(event.getCapacity() - numTickets)`, reducing the number of available tickets.

Step 3: Check the Version Number Before Writing

Before updating the event capacity by calling `updateCapacity(event, connection);`, read the latest version number of the `Event` record in the database and compare it with the previously read version number.

Step 4: If Version Numbers Match, Write Data

If the read version number matches the stored version number, it means that the `Event` has not been modified by other transactions during the execution of this transaction. Therefore, we can safely write the modified `Event` object back to the database and update the version number.

Step 5: If Version Numbers Do Not Match, Throw Optimistic Lock Exception

If the version numbers do not match, it means another transaction has modified the `Event` object, and the current transaction's update operation will fail. In the code, this is handled by throwing an `OptimisticLockingFailureException`.

Consequences

Positive

- Optimistic locking allows multiple users to try to RSVP simultaneously without causing blocking issues, providing better system throughput.
- Users will hardly notice the conflict and retry process when they apply for RSVP, resulting in a smoother user experience.

Negative

- In scenarios of extremely high concurrency (e.g., a highly anticipated event with limited spots and many users waiting to RSVP), the probability of conflicts is higher. This means the system may need to retry transactions multiple times, increasing the server's load.

Compliance

The lock is acquired at the start and released after the transaction is committed or rolled back, following the locking practices.

Alternatives Considered

Pessimistic Locking

Basically, pessimistic locking is suitable for high-conflict scenarios that require strict consistency. However, in the scenario where students are simultaneously RSVPing to events—which requires quick responses—using pessimistic locking would lead to reduced concurrency, and affect user experience.

Consequences

Positive

- With pessimistic locking, multiple students cannot modify the same event's capacity at the same time, which prevents data conflicts. This ensures that overbooking or incorrect reservations will not happen, as only one student at a time can successfully book a ticket.

Negative

- Since only one student can access the event data at a time, other students would experience delays or even miss out on the opportunity to RSVP due to waiting for locks to be released.
- In extremely high concurrency scenarios, although it ensures absolute data consistency, it can cause numerous database lock waits, greatly reducing system performance and possibly leading to deadlocks.

2 Handling Concurrent Event Edits by Club Administrators

Context

This issue happens because when multiple administrators access and try to modify the same event, the system processes these changes independently without coordinating the updates. It is crucial to solve this concurrency issue to ensure the integrity and reliability of event management within the system. When multiple administrators simultaneously amend an event, conflicting updates can result in confusion among administrators and students. We assume that there is usually a clear division of labor within the club (there is a dedicated person responsible for the event edits), so the scenario of multiple people making changes at the same time may not be common.

Decision

In the context of event management, we chose **optimistic offline lock** because it allows multiple administrators to make updates without blocking each other, which is ideal since simultaneous edits are not common.

Implementation Strategy

When an administrator attempts to amend an event, the system reads the event's current version number. Before saving any changes, the system checks if the version number has changed in the meantime. If it has changed (indicating another administrator has already modified the event), the operation is rejected, and an exception is thrown (Figure 2).

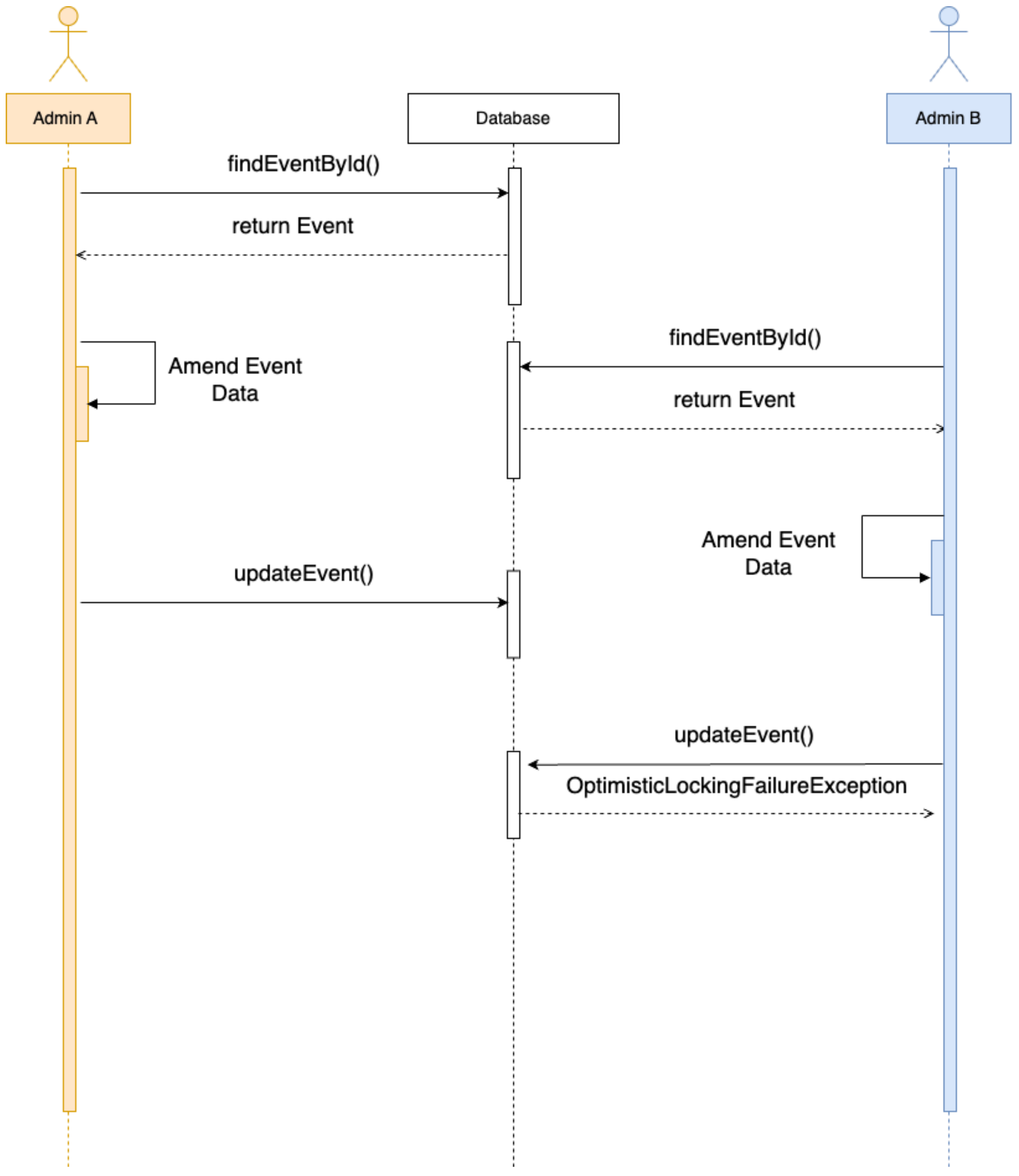


Figure 2 - Optimistic locking for concurrent event data amendments

Step 1: Read the record and store the version number

In the updateEvent method, we use eventRepository.findEventById(eventId, connection) to retrieve the Event object. The retrieved Event object contains version number (event_version).

Step 2: Modify the record

The administrator makes changes to the Event object (e.g., modifying capacity, cost, etc.) without saving them to the database yet.

Step 3: Compare the version number before writing

Before updating the event in the database, the updateEvent method compares the current event version number with the stored version number. The SQL query checks the version using WHERE event_id = ? AND event_version = ?.

Step 4: Match version numbers and update the data

If the version number in the database matches the version number read earlier, the update proceeds, and the event is updated in the database, with the version number incremented.

Step 5: Version number mismatch, throw optimistic lock exception

If the version numbers do not match, the system throws an OptimisticLockingFailureException, indicating that another administrator has already modified the event. The transaction is then rolled back.

Consequences

Positive

Optimistic locking doesn't lock database records during the amending process, allowing multiple club administrators to read and modify event information simultaneously. This improves concurrency and resource efficiency, avoiding situations where administrators have to wait for locks to be released. As a result, system responsiveness is improved, leading to a better user experience.

Negative

If multiple administrators modify activities at the same time (i.e., a high-conflict scenario), it may cause multiple retries and bad user experience.

Compliance

The lock is acquired at the start and released after the transaction is committed or rolled back, following the locking practices.

Alternatives Considered

Pessimistic Locking

Pessimistic locking would involve locking the event record when an administrator starts making changes to ensure that no other administrators can access it until the lock is released

Consequences

Positive

Because some event information is important (such as available capacity), guarantees that only one administrator can amend an event at any given time, ensuring no conflicts and avoiding potential data loss.

Negative

Since only one administrator is allowed to edit events at a time, other administrators need to wait for the lock to be released. This will reduce the concurrent processing capability of the system, especially in the case of a large number of administrators and frequent editing events, resulting in bad user experience.

3 Handling Concurrent Funding Submissions for Clubs

Context

The issue occurs when more than one administrator of a student club simultaneously submit funding applications for the same club. This can lead to race conditions where two or more administrators might think they have successfully submitted an application, causing duplicate submissions or overwriting of data. Note that we consider the whole operations(including editing) of creating a new funding application as a submit.

Resolving this concurrency issue is important to maintain accurate and financial records for the club. Duplicate or conflicting funding applications could be submitted, leading to delays in approving funding and mismanagement of club finances.

Decision

We implement a **pessimistic offline lock** to handle concurrency, using **exclusive write locks** to prevent simultaneous submissions to the same club’s funding application. Given the critical nature of these submissions, involving detailed documents and financial data, and any data conflict may lead to confusion in the approval process and even the risk of misallocation of funds. pessimistic locking ensures that no two administrators can submit or modify the same application at the same time, preventing data overwrites and conflicts. This approach saves administrators time by blocking conflicting submissions upfront, avoiding rework and ensuring data consistency.

Implementation Strategy

Our pessimistic locking mechanism is implemented using a thread-level locking with a lock manager (Figure 3). Specifically,

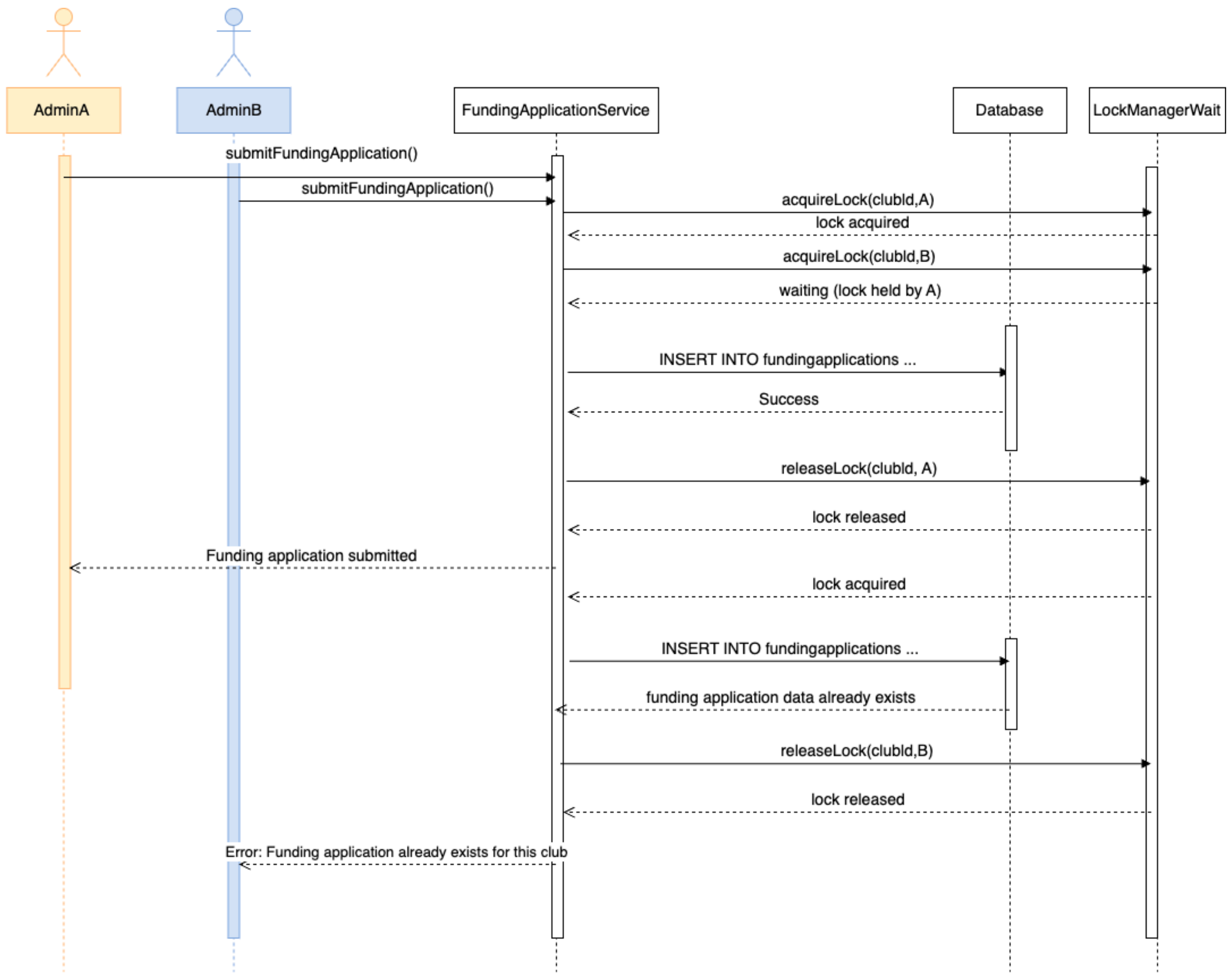

AdminA
AdminB
FundingApplicationService
Database
LockManagerWait
submitFundingApplication()
submitFundingApplication()
acquireLock(clubId,A)
lock acquired
acquireLock(clubId,B)
waiting (lock held by A)
INSERT INTO fundingapplications ...
Success
releaseLock(clubId, A)
lock released
Funding application submitted
lock acquired
INSERT INTO fundingapplications ...
funding application data already exists
releaseLock(clubId,B)
lock released
Error: Funding application already exists for this club

Figure 3 - Pessimistic locking for concurrent funding application submissions

Step 0: Define a Lock Manager

We implement a thread-level locking mechanism named LockManagerWait to control access to shared resources in a concurrent environment, such as a club's funding application in this scenario. It uses a `ConcurrentHashMap` to manage locks, where the key represents the lockable resource (`clubId`), and the value is the thread that holds the lock.

Step 1: Acquire the Thread-Level Lock

Before processing the funding application, a thread-level lock is acquired using `LockManagerWait.getInstance().acquireLock(clubId, threadName)`. This ensures that only one thread can process the funding application for the same `clubId` at a time, preventing concurrent submissions.

Step 2: Fetch the Funding Application and Validate Uniqueness

The method `fundingApplicationRepository.existsByClubIdAndSemester` checks if there is already a funding application for the same `clubId` and `semester`. If a duplicate is found, an exception is thrown, preventing further submission.

Step 3: Save the Funding Application

If no duplicate application is detected, the funding application is saved to the database by calling `fundingApplicationRepository.saveFundingApplication`.

Step 4: Commit the Transaction

After successfully saving the funding application, the transaction is committed by invoking `conn.commit()`, ensuring the changes are persisted in the database.

Step 5: Release the Thread-Level Lock

Once the transaction is committed, the thread-level lock is released by calling `LockManagerWait.getInstance().releaseLock(clubId, threadName)`, allowing other threads to process funding applications for the same club.

Step 6: Error Handling and Rollback

In case an error occurs during the process, `conn.rollback()` is called to revert any uncommitted changes. Regardless of success or failure, the thread-level lock is released in the `finally` block to avoid deadlocks or resource contention.

Consequences

Positive

Pessimistic offline lock saves time for other administrators, as they are blocked from starting the submission process until the lock is released, preventing them from spending time on changes that would later fail due to a conflict.

Ensures that only one funding application for a specific club is processed at a time, maintaining data consistency.

Negative

Pessimistic locking can result in longer wait times for transactions as only one can hold the lock at a time, though it is not a common case.

Compliance

Any new entity that involves pessimistic concurrency control should implement a similar two-level locking mechanism to ensure concurrency.

Alternatives Considered

Optimistic Locking

We considered using optimistic locking because this scenario is relatively uncommon. However, given the high integrity requirements and the significant cost of creating a funding application (as it often involves extensive written statements and attachments as supporting materials), using optimistic locking to handle concurrent conflicts could negatively impact the user experience.

Consequences

Positive

Since the occurrence of concurrent submissions is not common, optimistic locking reduces the resource overheads, leading to better performance.

Negative

New funding application creation is usually accompanied by a large number of documents uploaded, detailed filling, and approval (in the future). If club administrators encounter concurrency conflicts when creating new funding requests and have to repeat operations, this will result in a lot of wasted time and effort.

4 Handling Concurrent Reviews of Funding Applications

Context

This issue arises when multiple faculty administrators attempt to review and approve the same funding application at the same time. Different administrators may approve or reject the same application without knowing that others are doing the same, resulting in conflicting decisions.

Addressing this concurrency issue ensures that only consistent updates are applied when multiple administrators are involved in managing the same funding application. Note that "managing" refers specifically to the actions of approving or rejecting funding applications, not reviewing them.

Decision

We separated the review and approval/rejection actions. The review process is considered a read-only operation. However, when it comes to approval or rejection involving modifying the state of the application, we apply a lock to prevent concurrent modifications.

We are using an optimistic lock to ensure that each review of a funding application occurs on the latest version of the application, preventing conflicting updates without unnecessarily locking records. In our case, the faculty administrators are not expected to frequently approve/reject the same application simultaneously as they will likely be assigned to review and make decisions for specific funding applications. In this way, it detects this by checking the version of the application and raises an exception if faculty administrators and club admins try to modify the same application.

Implementation Strategy

The implementation reads the funding application, checks its version before updating, and increments the version upon success. If a version conflict occurs, the user is prompted to refresh and retry (Figure 4). Specifically,

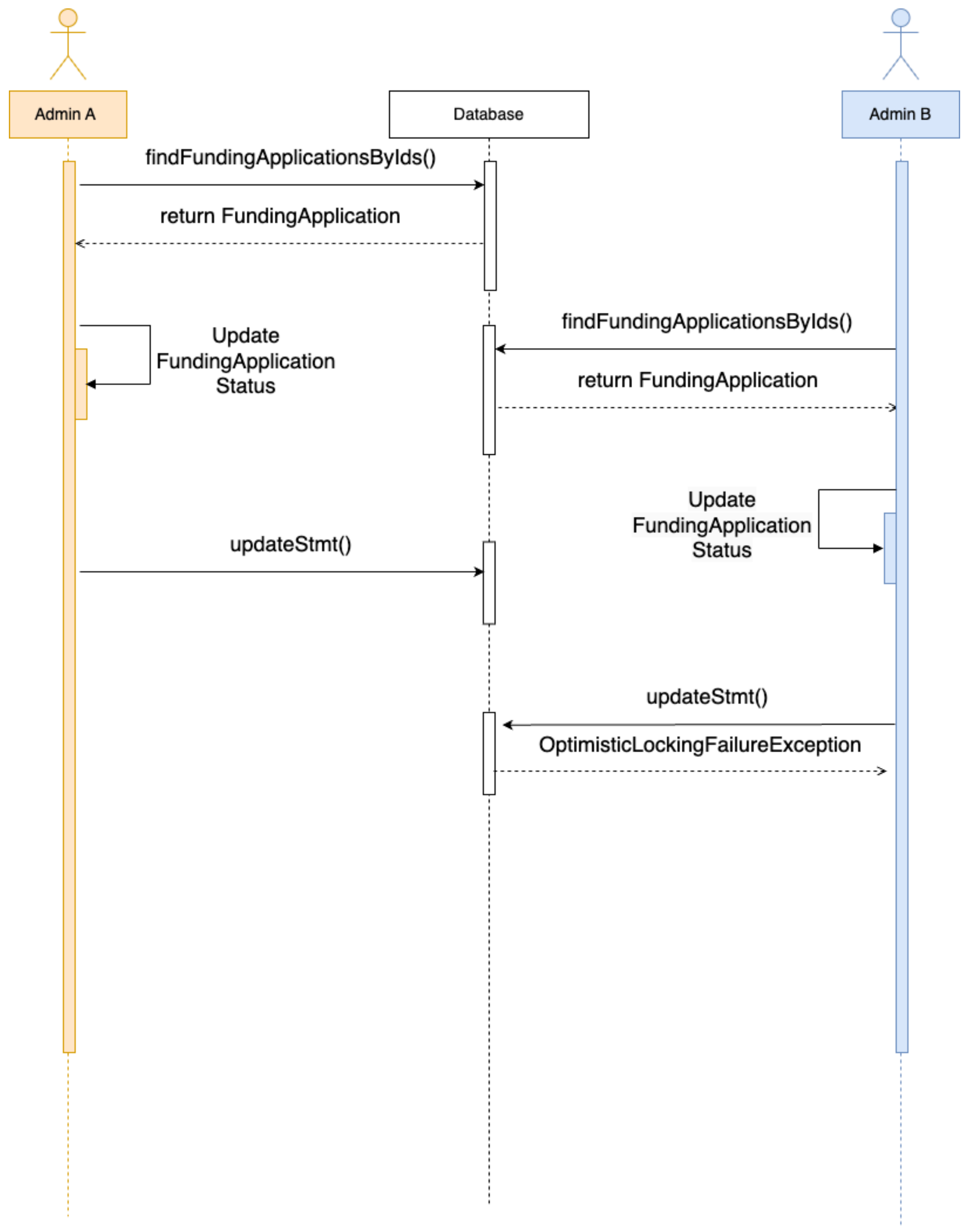

Admin A
Database
Admin B
findFundingApplicationsByIds()
return FundingApplication
Update FundingApplication Status
findFundingApplicationsByIds()
return FundingApplication
Update FundingApplication Status
updateStmt()
updateStmt()
OptimisticLockingFailureException

Figure 4 - Optimistic locking for concurrent updates to funding application status

Step 0: Version Field in FundingApplication Class
The FundingApplication class contains a version field, tracking the version of the funding application.

Step 1: Read the Record and Store the Version Field
Retrieve the FundingApplication object by using findFundingApplicationsByIds, which the FundingApplication entity includes the version field.

Step 2: Check Funding Application Status
Perform conditional checks within the reviewFundingApplication method to make sure the application's status is Submitted before any further actions. If the status is not Submitted, an exception is thrown.

Step 3: Check If Duplicated Funding Application Exists
Perform conditional checks within the reviewFundingApplication method - using existsByClubIdAndSemester(clubId, semester) function to make sure only one application per club is active for a given semester. If a duplicate is detected, an exception is thrown.

Step 4: Approve/Reject the Funding Application and Check the Version
After checks, allow the administrator to review and approve.reject the application by changing status attributes. Before saving the changes, compare the stored version number with the current version in the database.

Step 5: Increment the Version Number
If the version number check is successful, increment the version number by 1 during the update process. This is done to ensure that subsequent updates will detect any changes that occurred in the meantime.

Step 6: Handle Version Mismatch
If the version numbers do not match, throw an OptimisticLockingFailureException.

Consequences

Positive

By using optimistic locking, we avoid the overhead of locking records during reads, allowing multiple Faculty Administrators to access and read the same application concurrently without waiting. Also, it addresses concurrency between Faculty Administrators but also potential conflicts between Faculty Administrators and Student Club Administrators who might modify the same application.This reduces the risk of data inconsistency when multiple parties are involved.

Negative

If the application or system grows, with many Faculty admins Administrators working on the same applications, conflicts could become more frequent, which might bring bad user experience.

Compliance

The system ensures consistency by preventing stale data from being overwritten, and isolation by ensuring that each user works on the most recent version of the data.

Alternatives Considered

Pessimistic Locking

We could use optimistic locking as an alternative, where the system would lock the funding application record when a Faculty Administrator starts reviewing or modifying the status of it. This lock would prevent any other users from accessing or editing the application until the first user has completed their review and released the lock.

Consequences

Positive

Users would not need to handle version conflicts or retries, as only one person can make changes at a time, reducing the chance of errors due to concurrent modifications.

Negative

It might be challenging to manage cross-transaction concurrency with thread-level locks alone. For instance, when both Faculty Administrators and Student Club Administrators are modifying the same application, we would need to introduce database row-level locks to handle concurrent changes.

5 Handling Concurrent Edits of Club Funding Applications

Context

This issue arises when multiple administrators of the same club try to edit the funding applications for their club at the same time. Without the proper concurrency control, these operations will result in conflict updates. For example, two administrators might unknowingly edit the same application at the same time, causing overwrites or conflicts.

Although this situation is rare, ensuring data accuracy and preventing conflicting updates in these cases is crucial to maintaining the integrity of the system.

Decision

To resolve this concurrency issue, we decided to use optimistic locking combined with a retry mechanism. This approach ensures that changes from administrators are detected if another admin has already modified the funding application, preventing overwriting and maintaining data consistency.

The reason for choosing optimistic locking is that, although concurrent edits by multiple administrators are rare, this approach allows for better system performance in low-conflict scenarios by avoiding the need for constant locking. Optimistic locking only checks for conflicts at the time of committing changes, allowing administrators to work concurrently without being blocked by each other. This ensures data consistency without significantly impacting the system's throughput.

Implementation Strategy

This strategy outlines how the system handles concurrent edits to funding applications using optimistic locking to ensure data consistency. It ensures that conflicting updates are detected and handled efficiently through a retry mechanism (Figure 5).

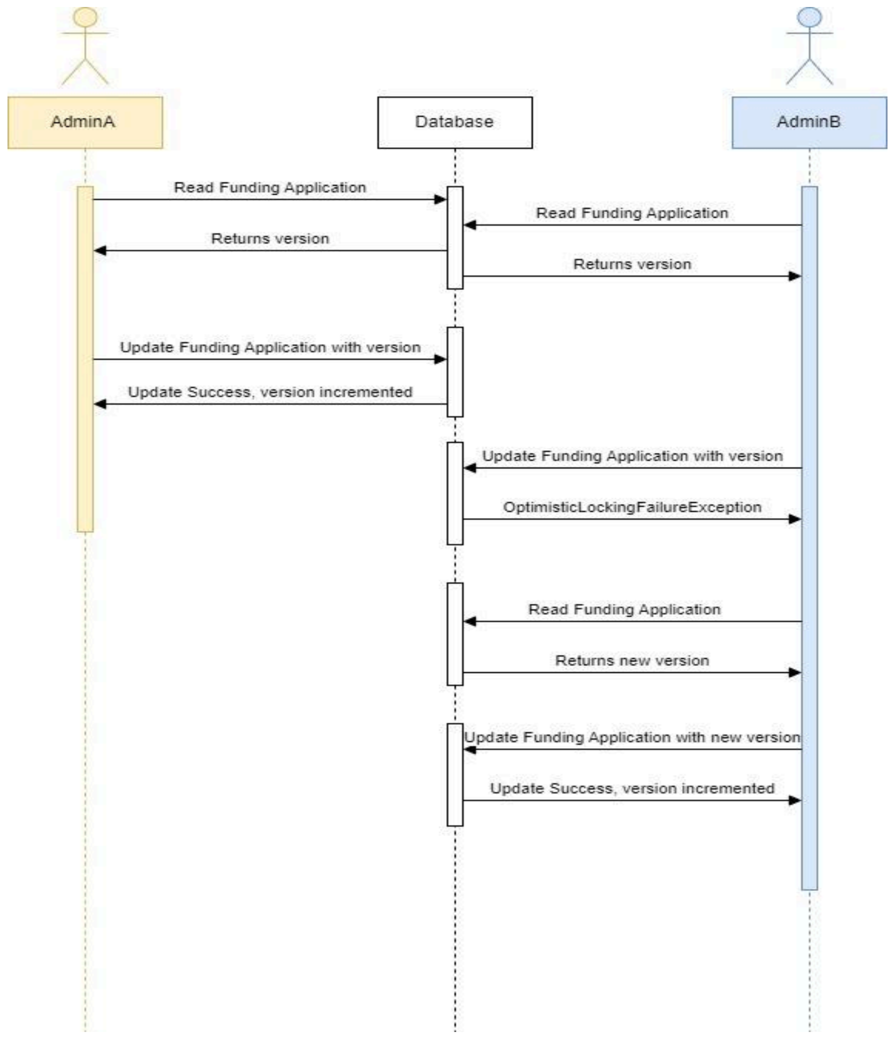

AdminA
Database
AdminB
Read Funding Application
Read Funding Application
Returns version
Returns version
Update Funding Application with version
Update Success, version incremented
Update Funding Application with version
OptimisticLockingFailureException
Read Funding Application
Returns new version
Update Funding Application with new version
Update Success, version incremented

Figure 5 - Optimistic locking with retry mechanism for concurrent funding application updates

Step 1: Read the record and store the version number
When an administrator attempts to edit a funding application, the system retrieves the funding application record by the (findFundingApplicationsByIds). This record includes the version number associated with the funding application (application_version).

Step 2: Modify the record
The administrator makes the necessary edits to the funding application without saving the changes to the database yet.

Step 3: Compare the version number before writing
Before committing the changes to the database, the system checks if the version number of the funding application in the database matches the one that was initially read.

Step 4: Match version numbers and update the data
If the version number in the database matches the stored version, the system proceeds with the update, saving the modified funding application in the database, and the version number is incremented.

Step 5: Version number mistake, throw optimistic lock exception
If the version numbers do not match, meaning another administrator has already modified the application, the system throws an OptimisticLockingFailureException. The current transaction is then rolled back, and the administrator can retry the operation.

Step 6: Retry mechanism
When an OptimisticLockingFailureException occurs, the system can automatically retry the operation. The administrator's edits are retried, starting from reading the latest version of the funding application, making the edits, and attempting to save them again. This retry process continues until the transaction succeeds.

Consequences

Positive

By using optimistic locking, administrators can edit funding applications concurrently without being blocked. For example, if one administrator begins modifying a funding application, others can still view and make changes to different applications without waiting for locks to be released. This improves the system's ability to handle multiple operations simultaneously.

Negative

In the cases where multiple administrators edit the same funding application within a short time, the system might trigger frequent optimistic locking failures, requiring several retries. This can lead to a higher load on the server and delays as administrators repeatedly attempt to save their changes.

Compliance
This decision maintains consistency with previous system designs and the established technical roadmap, preventing deviation from existing architectural patterns. Other parts of the system already utilize optimistic locking for handling concurrency issues, so this decision follows the same approach to ensure architectural cohesion and maintainability.

Alternatives Considered

Pessimistic Locking

We considered pessimistic locking, where the system would lock the funding application record as soon as one administrator begins editing it. This would ensure that no other administrators could access or modify the record until the lock is released. The lock would remain in place throughout the editing session, guaranteeing that only one administrator at a time can make changes.

This approach is effective in preventing any concurrent modifications, as it locks the resource early in the process. It is commonly used in high-conflict environments where the probability of conflicting changes is high, such as when handling critical data that must not be overwritten or compromised. However, this method requires holding the lock until the editing transaction is fully complete, meaning other users must wait until the lock is released before they can proceed.

Consequences

Positive

Prevents all concurrency conflicts and ensures data consistency, as only one administrator can modify the funding application at any given time.

Negative

Pessimistic locking would reduce system performance by forcing administrators to wait until the lock is released. This could cause delays during periods of high activity, as administrators would be blocked from making changes, leading to inefficiencies and frustration when quick access is needed.

Testing Strategy

Our tests are implemented using **JUnit 5**, leveraging **Mockito** for mocking dependencies and `ExecutorService` for simulating concurrent operations, with assertions to verify expected outcomes under concurrency. Our testing covers various concurrency scenarios, focusing on funding applications submissions and reviewing, RSVP submissions, and event updates. The goal is to evaluate how the system behaves under high concurrency, particularly verifying the effectiveness of lock mechanisms. The implementation path of testing can be found in *`Y/ClubManagement/backend/src/test/java`*. Here's a detailed explanation of your testing strategy:

1. FundingApplicationConcurrencyTest

We have three test cases about the concurrency issues of funding application, specifically,

1. testConcurrentFundingApplicationSubmissions

Objective:
Simulate concurrent submissions of funding applications by multiple administrators for the same club to test the locking mechanism. Note that it is the testing for use case 3.

Test Approach:

- Create a thread pool to simulate 10 administrators submitting applications with the same clubId and semester concurrently (Here, we set clubId=30, semester=2).
- Each thread attempts to submit the fundingApplication. We count the success and failure threads. Also, we handle the failure about duplicate submissions once the first submission is successful or an already existing funding application (we set the funding application can only submit one for each semester).
- To ensure pessimistic locking works normally, each thread prints a log when it attempts to acquire/release a lock, and the time for each thread to acquire/release the lock is recorded. This ensures only one lock can be obtained at a time.
- To guarantee the data integrity, we check the state of `fundingapplications` table in the database before and after testing, to make sure only one application has been added.

Expectation:

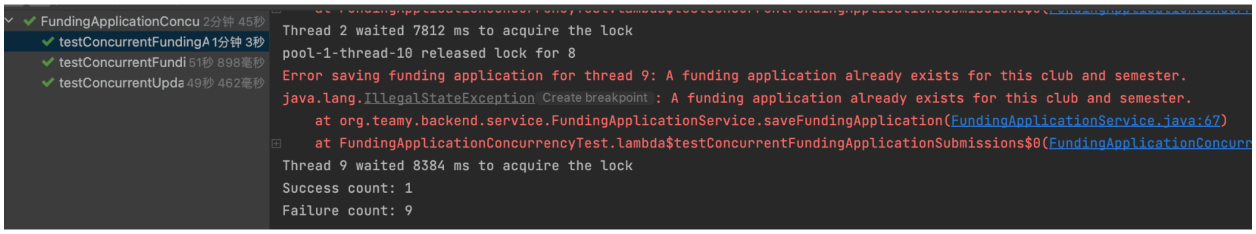
We expect:

- Only one thread is successful, while the other 9 threads are failed due to the duplications.
- The printed log shows that multiple threads acquire/release resources in the order of the lock, proving that lock exclusivity is valid.

- The time the thread waits for the lock increases, indicating that the thread is indeed waiting for the previous lock to be released, which verifies the lock's mutual exclusion.
- Only one application with preset clubId and semester is submitted in the database.

Outcome:
The results meet our expectations. The followings are screenshots of our testing results:

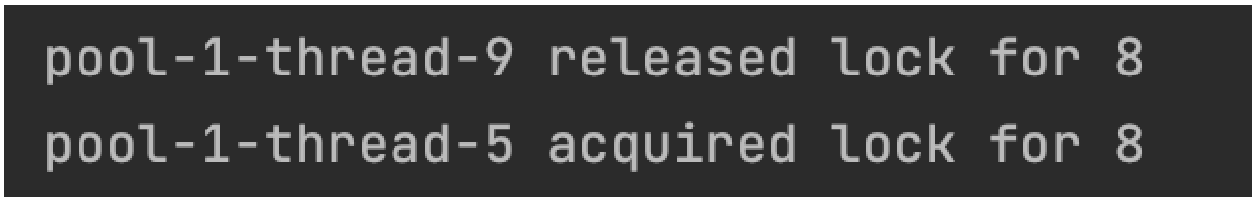
1. One thread successfully submitted the funding application, while the remaining 9 failed.



```
FundingApplicationConcu 2分钟 45秒
  testConcurrentFundingA 1分钟 3秒
  testConcurrentFundi 51秒 898毫秒
  testConcurrentUpda 49秒 462毫秒
```

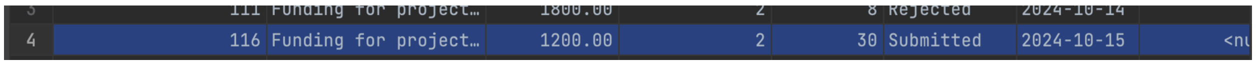
```
Thread 2 waited 7812 ms to acquire the lock
pool-1-thread-10 released lock for 8
Error saving funding application for thread 9: A funding application already exists for this club and semester.
java.lang.IllegalStateException Create breakpoint : A funding application already exists for this club and semester.
    at org.teamy.backend.service.FundingApplicationService.saveFundingApplication(FundingApplicationService.java:67)
    at FundingApplicationConcurrencyTest.lambda$testConcurrentFundingApplicationSubmissions$0(FundingApplicationConcurr
Thread 9 waited 8384 ms to acquire the lock
Success count: 1
Failure count: 9
```

2. The logs show threads acquiring and releasing locks in sequence, confirming that the locking mechanism worked as expected.



```
pool-1-thread-9 released lock for 8
pool-1-thread-5 acquired lock for 8
```

3. Only one application with the specified clubId and semester was saved in the database (we set clubId=30, semester=2 here).



3	111	Funding for project…	1800.00	2	8	Rejected	2024-10-14	
4	116	Funding for project…	1200.00	2	30	Submitted	2024-10-15	<nu

2. testConcurrentFundingApplicationReviews

Objective:

Simulate multiple reviewers concurrently reviewing the same funding application to test the optimistic locking mechanism and handle conflicts when updating the review status. It is the testing for use case 4.

Test Approach:

- Create a thread pool simulating 10 reviewers attempting to review the same funding application (we still use the applicationId = 116 generated from last testing). Each thread is randomly assigned an "Approved" or "Rejected" status and invokes the `reviewFundingApplication` method to submit their review.
- Each thread logs its review status and execution order.

- The test uses an optimistic locking mechanism to ensure that only one thread can successfully update the application at a time. If another thread modifies the funding application before a thread's submission, the version mismatch triggers an `OptimisticLockingFailureException`, terminating the operation for that thread.
- Success and failure of each thread are tracked.
- After the test, the `fundingapplications` table in the database is checked to ensure that only one final review result was saved.

Expectation:

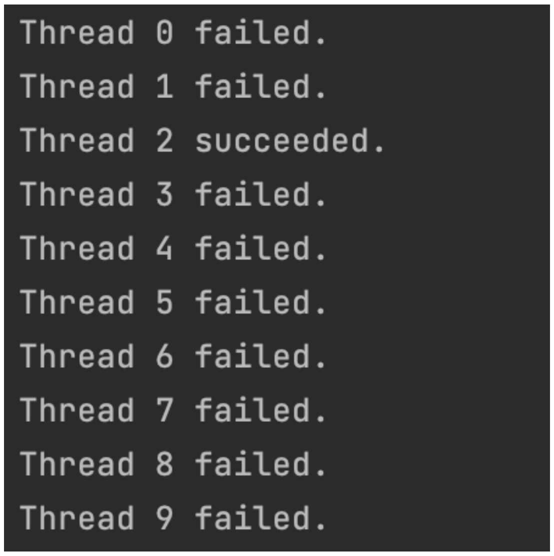
We expect:

- Only one thread will successfully complete the review, while the other 9 threads will fail due to optimistic lock conflicts.
- The log will show each thread's attempt to review, with errors from optimistic lock failures.
- The `fundingapplications` table should show only one final review status for the specified `applicationId`.

Outcome:

The results meet our expectations. The followings are screenshots of our testing results:

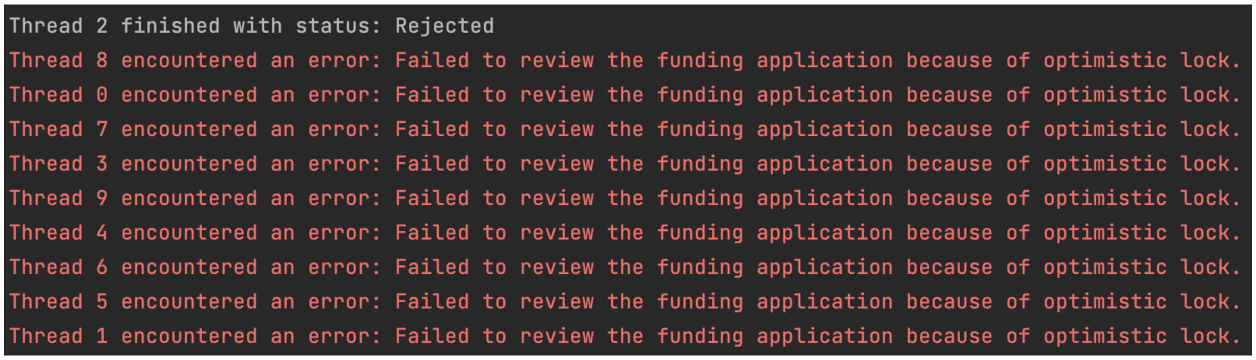
- One thread successfully completed the review, while the remaining 9 threads failed due to optimistic locking conflicts.



```
Thread 0 failed.
Thread 1 failed.
Thread 2 succeeded.
Thread 3 failed.
Thread 4 failed.
Thread 5 failed.
Thread 6 failed.
Thread 7 failed.
Thread 8 failed.
Thread 9 failed.
```

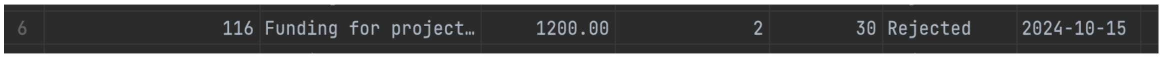
- Logs showed that most threads attempted to review but encountered `OptimisticLockingFailureException` due to a version mismatch. For example,

Thread 2 successfully completed its review, while the other threads failed due to the optimistic lock.



```
Thread 2 finished with status: Rejected
Thread 8 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 0 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 7 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 3 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 9 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 4 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 6 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 5 encountered an error: Failed to review the funding application because of optimistic lock.
Thread 1 encountered an error: Failed to review the funding application because of optimistic lock.
```

- The database shows the final review status has been changed to **Rejected** for `applicationId = 116`.



6	116	Funding for project…	1200.00	2	30	Rejected	2024-10-15

3. testConcurrentUpdateFundingApplications

Objective:
Simulate concurrent updates to the same funding application by multiple administrators to test how the system handles such updates. It is the testing for use case 5.

Test Approach:

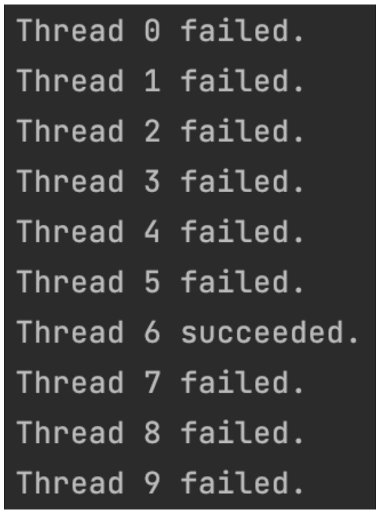
- A thread pool is created to simulate 10 administrators trying to update the same funding application concurrently. Here, we choose applicationId = 117 instead of 116 in the previous example since the status must be **Submitted (club administrators can not update once the application has been reviewed).**
- Each thread retrieves the `FundingApplication` object and attempts to update its amount to a specific value (`BigDecimal.valueOf(1000)`).
- The test leverages PostgreSQL's optimistic locking mechanism. If two threads try to update the same record concurrently, one of them will fail with an `ERROR: could not serialize access due to concurrent update`, and the transaction will be rolled back.
- The test checks which threads successfully update the application and which fail due to the version mismatch.
- After the test, the `fundingapplications` table in the database is checked to ensure that only one application was updated.

Expectation:

- Only one thread should successfully update the funding application, while the other threads should fail due to concurrent updates, triggering an optimistic locking conflict.
- The logs should reflect which thread successfully completed the update and which threads encountered an error due to the concurrent modification.
- The final state of the funding application in the database should be successfully updated, and the optimistic locking mechanism should have prevented inconsistent data changes.

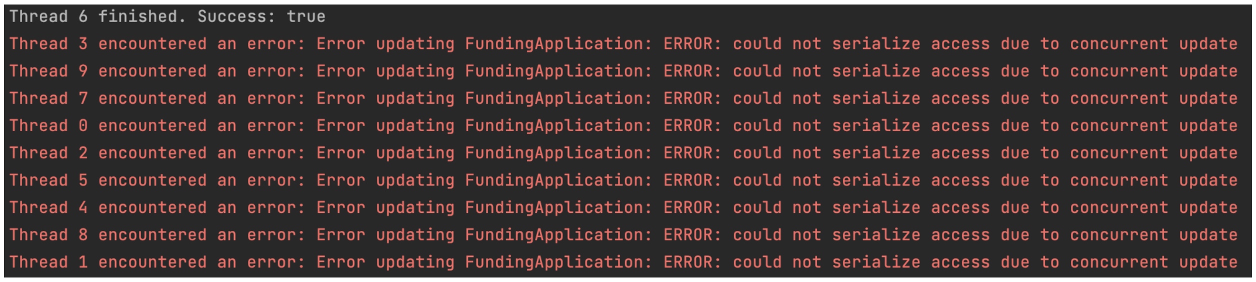
Outcome: The results aligned with the expectations:

- **Thread 6** was the only one to successfully update the funding application.



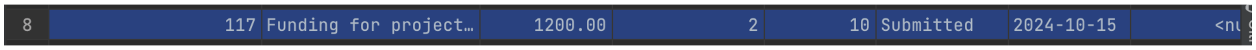
```
Thread 0 failed.
Thread 1 failed.
Thread 2 failed.
Thread 3 failed.
Thread 4 failed.
Thread 5 failed.
Thread 6 succeeded.
Thread 7 failed.
Thread 8 failed.
Thread 9 failed.
```

- The remaining 9 threads failed to update the application due to an optimistic locking conflict, as indicated by the error `ERROR: could not serialize access due to concurrent update`.

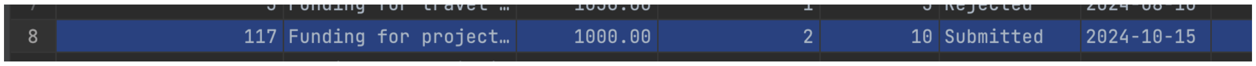


```
Thread 6 finished. Success: true
Thread 3 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 9 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 7 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 0 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 2 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 5 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 4 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 8 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
Thread 1 encountered an error: Error updating FundingApplication: ERROR: could not serialize access due to concurrent update
```

- The test confirmed that optimistic locking works as intended, preventing multiple concurrent updates and maintaining data integrity (the amount has been changed from 1200 to 1000).



8	117	Funding for project…	1200.00	2	10	Submitted	2024-10-15	<nu



8	117	Funding for project…	1000.00	2	10	Submitted	2024-10-15

2. RSVPConcurrencyTest

Objective:

Simulate concurrent RSVPs for the same event by multiple students to test the locking mechanism and ensure data integrity during simultaneous booking of event tickets. This test is focused on verifying whether the system can handle concurrent operations safely and prevent race conditions when multiple users apply for event tickets simultaneously.

Test Approach:

- We created a thread pool with 3 threads to simulate 3 students applying for RSVPs concurrently for the same event (`eventId=3`).
- Each student attempts to book 2 tickets for the event, using different `studentId`s. The test tracks the success and failure rates of each thread, focusing on situations where there might be conflicts due to the same event being accessed by multiple users at the same time.
- Each thread logs the time it took to acquire the lock and whether the operation succeeded or failed. Additionally, the system validates that the version number of the event is consistent, ensuring that optimistic locking is enforced to prevent data inconsistencies.
- After the test, the `events` table in the database is checked to ensure that only one RSVP has been applied (we check whether the capacity has been reduced correctly).

Expectation:

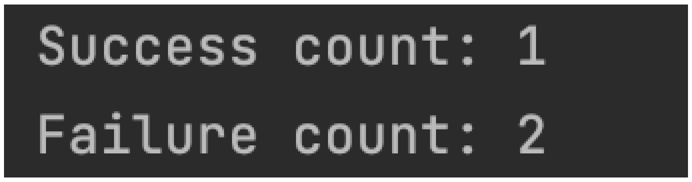
We expect:

- We expect one thread to succeed in acquiring the lock and submitting the RSVP, while other threads might fail due to conflicts (such as event capacity or version mismatch).
- The logs should show that only one thread can acquire the lock at a time, validating that the locking mechanism is functioning correctly.
- Threads that fail to acquire the lock within the expected time or encounter version mismatches will roll back their operations, preventing any inconsistencies in the event's ticket allocation.
- We expect only one successful RSVP submission, which means the capacity should only be reduced once.

Outcome:

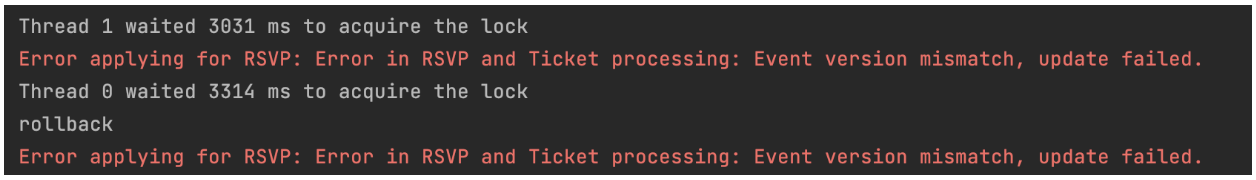
The results align with our expectations:

- The final success count was 1, and the failure count was 2, which matches the expected behavior of the system under concurrent conditions.



```
Success count: 1
Failure count: 2
```

- The logs show the time each thread waited to acquire the lock, confirming that the locking mechanism worked as intended.



```
Thread 1 waited 3031 ms to acquire the lock
Error applying for RSVP: Error in RSVP and Ticket processing: Event version mismatch, update failed.
Thread 0 waited 3314 ms to acquire the lock
rollback
Error applying for RSVP: Error in RSVP and Ticket processing: Event version mismatch, update failed.
```

- Two threads failed to apply for the RSVP due to an "Event version mismatch, update failed" error, confirming that the optimistic locking mechanism prevented conflicting updates.
- The capacity has been reduced from 30 to 28, meeting our expectations.

3. EventServiceConcurrencyTest

Objective:

The objective of this test is to simulate concurrent updates to the same event by multiple threads (in this case, 3 threads) to evaluate how the system handles concurrency and optimistic locking mechanisms. The test ensures that simultaneous updates to an event (such as changing the event capacity) do not cause conflicts or data inconsistencies.

Test Approach:

- We create a thread pool with 3 threads, each simulating an update to the capacity of the same event (`eventId=47`).
- Each thread attempts to retrieve the event, modify its capacity, and save the changes back to the database using the `eventService.updateEvent()` method. The system's optimistic lock compares the event version number, and if another thread has already updated the event, the version mismatch will result in an exception.
- We count the number of successful and failed updates and assert the results accordingly.
- After the test, the `events` table in the database is checked to ensure that only one capacity has been applied (we check whether the capacity has been updated correctly).

Expectation:

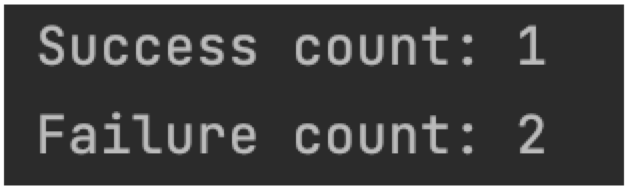
We expect:

- Only one thread should be able to update the event successfully, as the first thread that acquires the lock should be able to make changes, while subsequent threads should fail due to version mismatch errors.
- The printed logs should show that at least one update succeeds, while other threads encounter "version mismatch" errors, confirming that the optimistic locking mechanism works as expected.
- We expect only one successful event revision, which means the event data should only be updated once.

Outcome:

The test results meet our expectations:

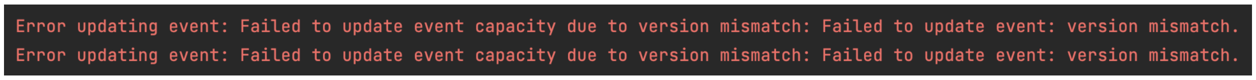
- One thread successfully updated the event, while the remaining two threads failed due to version mismatch errors.



```
Success count: 1

Failure count: 2
```

- The logs clearly show that the event's capacity was modified by one thread, while the other threads encountered the "Failed to update event capacity due to version mismatch" error.



```
Error updating event: Failed to update event capacity due to version mismatch: Failed to update event: version mismatch.
Error updating event: Failed to update event capacity due to version mismatch: Failed to update event: version mismatch.
```

- We checked the state of the event table in the database, and the capacity was revised once.