# Software Architecture Design (SAD) Document

**Submission 3 Specification**

**Runtime Terror**

**SWEN90007 SM2 2021 Project**

**Release Tag:**
**https://github.com/SWEN900072021/Covid19BookingSystem/releases/tag/SWEN90007_2021_Part3_RuntimeTerror**

**Team Members:**

**Thomas Chen | 916183 | thomasc3@student.unimelb.edu.au**

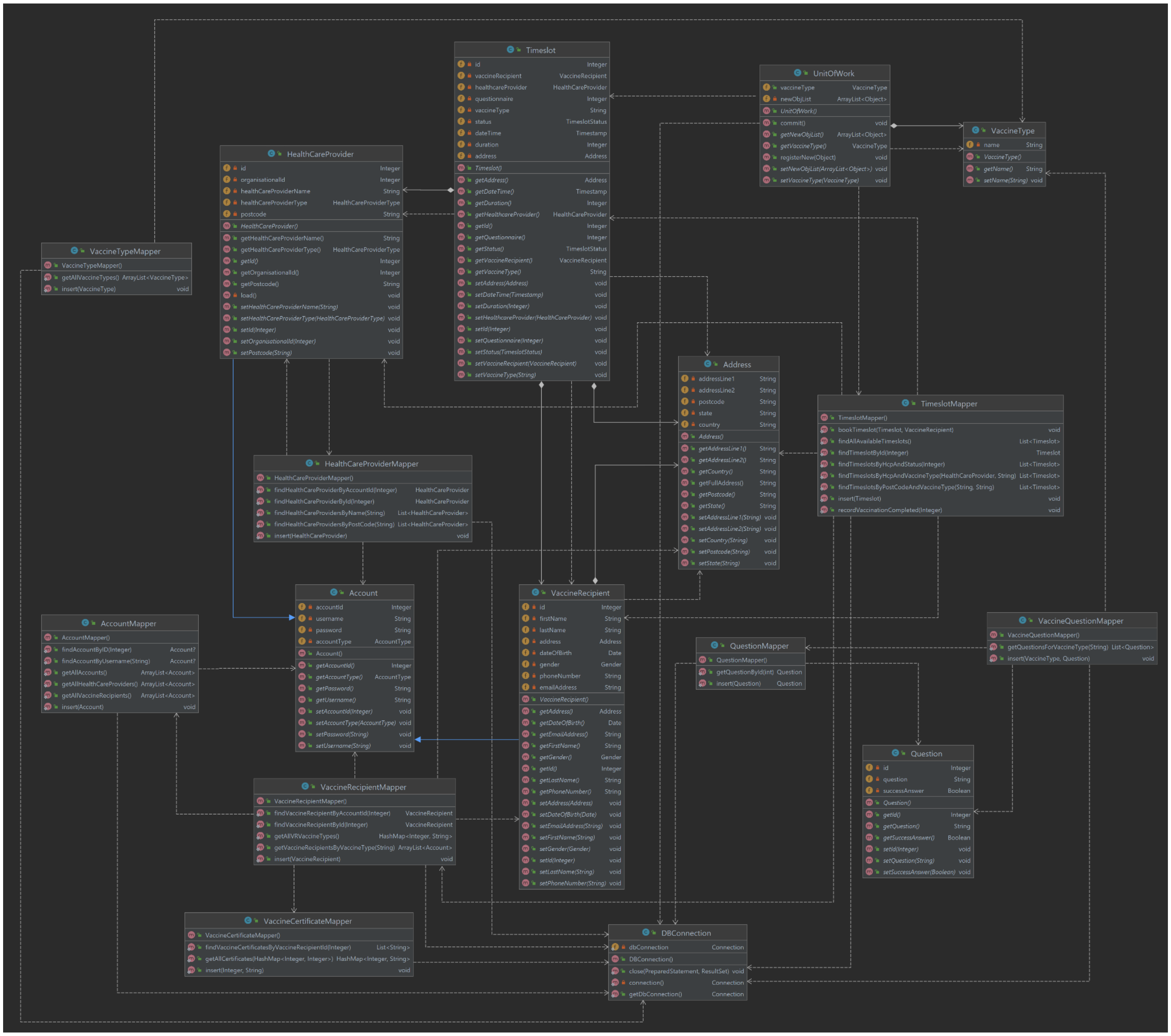
**Kah Chun Lee | 860318 | kahl2@student.unimelb.edu.au**

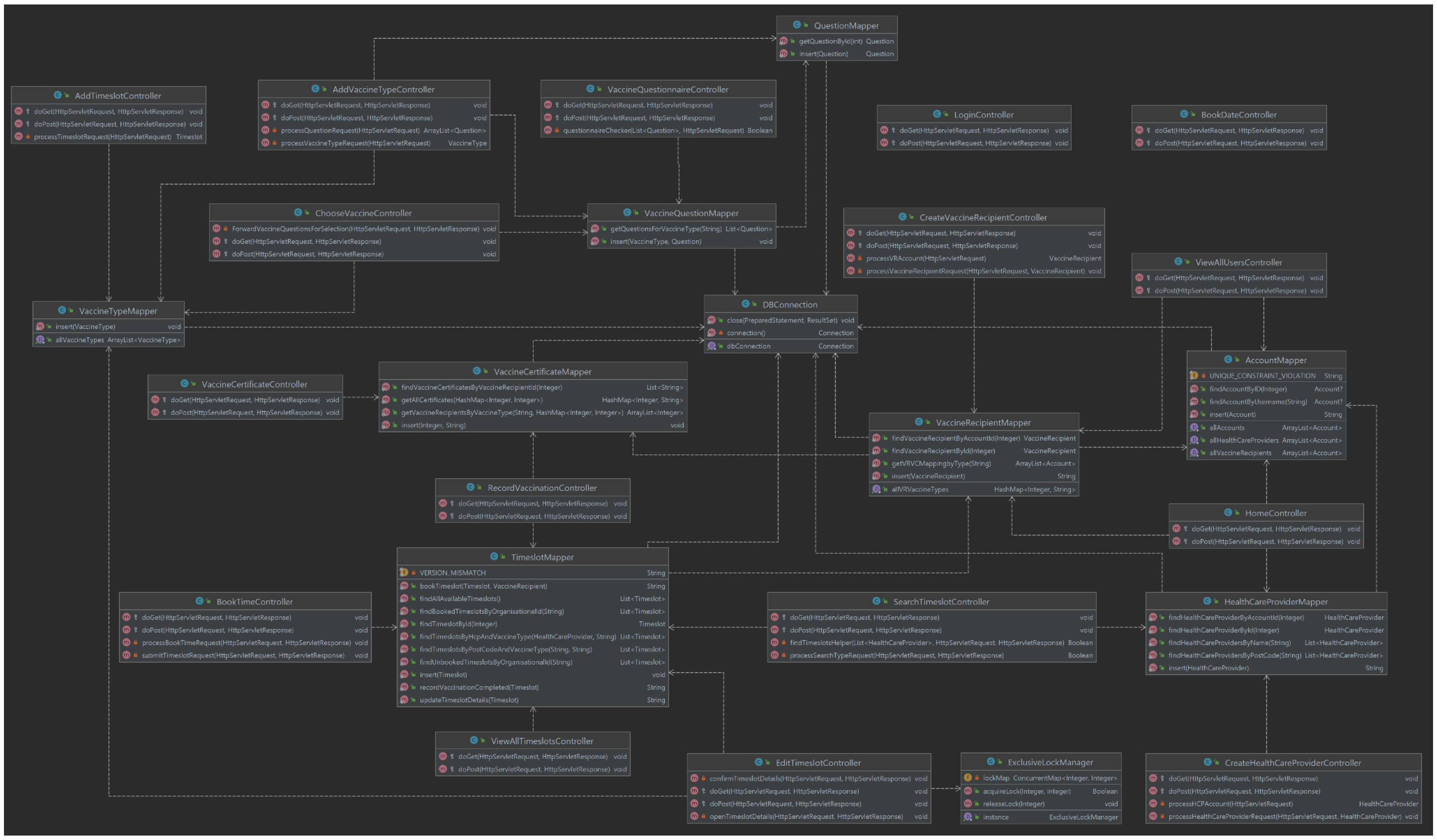
**Jay Parikh | 864675 | jparikh@student.unimelb.edu.au**

**Jad Saliba | 1014680 | jads@student.unimelb.edu.au**

# Table of Contents

# Class Diagram


Timeslot
id Integer
vaccineRecipient VaccineRecipient
healthcareProvider HealthCareProvider
questionnaire Integer
vaccineType String
status TimeslotStatus
dateTime Timestamp
duration Integer
address Address
Timeslot()
getAddress() Address
getDateTime() Timestamp
getDuration() Integer
getHealthcareProvider() HealthCareProvider
getId() Integer
getQuestionnaire() Integer
getStatus() TimeslotStatus
getVaccineRecipient() VaccineRecipient
getVaccineType() String
setAddress(Address) void
setDateTime(Timestamp) void
setDuration(Integer) void
setHealthcareProvider(HealthCareProvider) void
setId(Integer) void
setQuestionnaire(Integer) void
setStatus(TimeslotStatus) void
setVaccineRecipient(VaccineRecipient) void
setVaccineType(String) void
UnitOfWork
vaccineType VaccineType
newObjList ArrayList&lt;Object&gt;
UnitOfWork()
commit() void
getNewObjList() ArrayList&lt;Object&gt;
getVaccineType() VaccineType
registerNew(Object) void
setNewObjList(ArrayList&lt;Object&gt;) void
setVaccineType(VaccineType) void
VaccineType
name String
VaccineType()
getName() String
setName(String) void
HealthCareProvider
id Integer
organisationalId Integer
healthCareProviderName String
healthCareProviderType HealthCareProviderType
postcode String
HealthCareProvider()
getHealthCareProviderName() String
getHealthCareProviderType() HealthCareProviderType
getId() Integer
getOrganisationalId() Integer
getPostcode() String
load() void
setHealthCareProviderName(String) void
setHealthCareProviderType(HealthCareProviderType) void
setId(Integer) void
setOrganisationalId(Integer) void
setPostcode(String) void
VaccineTypeMapper
VaccineTypeMapper()
getAllVaccineTypes() ArrayList&lt;VaccineType&gt;
insert(VaccineType) void
Address
addressLine1 String
addressLine2 String
postcode String
state String
country String
Address()
getAddressLine1() String
getAddressLine2() String
getCountry() String
getFullAddress() String
getPostcode() String
getState() String
setAddressLine1(String) void
setAddressLine2(String) void
setCountry(String) void
setPostcode(String) void
setState(String) void
TimeslotMapper
TimeslotMapper()
bookTimeslot(Timeslot, VaccineRecipient) void
findAllAvailableTimeslots() List&lt;Timeslot&gt;
findTimeslotById(Integer) Timeslot
findTimeslotsByHcpAndStatus(Integer) List&lt;Timeslot&gt;
findTimeslotsByHcpAndVaccineType(HealthCareProvider, String) List&lt;Timeslot&gt;
findTimeslotsByPostCodeAndVaccineType(String, String) List&lt;Timeslot&gt;
insert(Timeslot) void
recordVaccinationCompleted(Integer) void
HealthCareProviderMapper
HealthCareProviderMapper()
findHealthCareProviderByAccountId(Integer) HealthCareProvider
findHealthCareProviderById(Integer) HealthCareProvider
findHealthCareProvidersByName(String) List&lt;HealthCareProvider&gt;
findHealthCareProvidersByPostCode(String) List&lt;HealthCareProvider&gt;
insert(HealthCareProvider) void
Account
accountId Integer
username String
password String
accountType AccountType
Account()
getAccountId() Integer
getAccountType() AccountType
getPassword() String
getUsername() String
setAccountId(Integer) void
setAccountType(AccountType) void
setPassword(String) void
setUsername(String) void
AccountMapper
AccountMapper()
findAccountByID(Integer) Account?
findAccountByUsername(String) Account?
getAllAccounts() ArrayList&lt;Account&gt;
getAllHealthCareProviders() ArrayList&lt;Account&gt;
getAllVaccineRecipients() ArrayList&lt;Account&gt;
insert(Account) void
VaccineRecipient
id Integer
firstName String
lastName String
address Address
dateOfBirth Date
gender Gender
phoneNumber String
emailAddress String
VaccineRecipient()
getAddress() Address
getDateOfBirth() Date
getEmailAddress() String
getFirstName() String
getGender() Gender
getId() Integer
getLastName() String
getPhoneNumber() String
setAddress(Address) void
setDateOfBirth(Date) void
setEmailAddress(String) void
setFirstName(String) void
setGender(Gender) void
setId(Integer) void
setLastName(String) void
setPhoneNumber(String) void
QuestionMapper
QuestionMapper()
getQuestionById(int) Question
insert(Question) Question
VaccineQuestionMapper
VaccineQuestionMapper()
getQuestionsForVaccineType(String) List&lt;Question&gt;
insert(VaccineType, Question) void
Question
id Integer
question String
successAnswer Boolean
Question()
getId() Integer
getQuestion() String
getSuccessAnswer() Boolean
setId(Integer) void
setQuestion(String) void
setSuccessAnswer(Boolean) void
VaccineRecipientMapper
VaccineRecipientMapper()
findVaccineRecipientByAccountId(Integer) VaccineRecipient
findVaccineRecipientById(Integer) VaccineRecipient
getAllVRVaccineTypes() HashMap&lt;Integer, String&gt;
getVaccineRecipientsByVaccineType(String) ArrayList&lt;Account&gt;
insert(VaccineRecipient) void
VaccineCertificateMapper
VaccineCertificateMapper()
findVaccineCertificatesByVaccineRecipientId(Integer) List&lt;String&gt;
getAllCertificates(HashMap&lt;Integer, Integer&gt;) HashMap&lt;Integer, String&gt;
insert(Integer, String) void
DBConnection
dbConnection Connection
DBConnection()
close(PreparedStatement, ResultSet) void
connection() Connection
getDbConnection() Connection


QuestionMapper
getQuestionById(int) Question
insert(Question) Question
AddTimeslotController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
processTimeslotRequest(HttpServletRequest) Timeslot
AddVaccineTypeController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
processQuestionRequest(HttpServletRequest) ArrayList<Question>
processVaccineTypeRequest(HttpServletRequest) VaccineType
VaccineQuestionnaireController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
questionnaireChecker(List<Question>, HttpServletRequest) Boolean
LoginController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
BookDateController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
ChooseVaccineController
ForwardVaccineQuestionsForSelection(HttpServletRequest, HttpServletResponse) void
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
VaccineQuestionMapper
getQuestionsForVaccineType(String) List<Question>
insert(VaccineType, Question) void
CreateVaccineRecipientController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
processVRAccount(HttpServletRequest) VaccineRecipient
processVaccineRecipientRequest(HttpServletRequest, VaccineRecipient) void
ViewAllUsersController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
VaccineTypeMapper
insert(VaccineType) void
allVaccineTypes ArrayList<VaccineType>
DBConnection
close(PreparedStatement, ResultSet) void
connection() Connection
dbConnection Connection
AccountMapper
UNIQUE_CONSTRAINT_VIOLATION String
findAccountByID(Integer) Account?
findAccountByUsername(String) Account?
insert(Account) String
allAccounts ArrayList<Account>
allHealthCareProviders ArrayList<Account>
allVaccineRecipients ArrayList<Account>
VaccineCertificateController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
VaccineCertificateMapper
findVaccineCertificatesByVaccineRecipientId(Integer) List<String>
getAllCertificates(HashMap<Integer, Integer>) HashMap<Integer, String>
getVaccineRecipientsByVaccineType(String, HashMap<Integer, Integer>) ArrayList<Integer>
insert(Integer, String) void
VaccineRecipientMapper
findVaccineRecipientByAccountId(Integer) VaccineRecipient
findVaccineRecipientById(Integer) VaccineRecipient
getVRVCMappingbyType(String) ArrayList<Account>
insert(VaccineRecipient) String
allVRVaccineTypes HashMap<Integer, String>
RecordVaccinationController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
HomeController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
TimeslotMapper
VERSION_MISMATCH String
bookTimeslot(Timeslot, VaccineRecipient) String
findAllAvailableTimeslots() List<Timeslot>
findBookedTimeslotsByOrganisationalId(String) List<Timeslot>
findTimeslotById(Integer) Timeslot
findTimeslotsByHcpAndVaccineType(HealthCareProvider, String) List<Timeslot>
findTimeslotsByPostCodeAndVaccineType(String, String) List<Timeslot>
findUnbookedTimeslotsByOrganisationalId(String) List<Timeslot>
insert(Timeslot) void
recordVaccinationCompleted(Timeslot) String
updateTimeslotDetails(Timeslot) String
BookTimeController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
processBookTimeRequest(HttpServletRequest, HttpServletResponse) void
submitTimeslotRequest(HttpServletRequest, HttpServletResponse) void
SearchTimeslotController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
findTimeslotsHelper(List<HealthCareProvider>, HttpServletRequest, HttpServletResponse) Boolean
processSearchTypeRequest(HttpServletRequest, HttpServletResponse) Boolean
HealthCareProviderMapper
findHealthCareProviderByAccountId(Integer) HealthCareProvider
findHealthCareProviderById(Integer) HealthCareProvider
findHealthCareProvidersByName(String) List<HealthCareProvider>
findHealthCareProvidersByPostCode(String) List<HealthCareProvider>
insert(HealthCareProvider) String
ViewAllTimeslotsController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
EditTimeslotController
confirmTimeslotDetails(HttpServletRequest, HttpServletResponse) void
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
openTimeslotDetails(HttpServletRequest, HttpServletResponse) void
ExclusiveLockManager
lockMap ConcurrentMap<Integer, Integer>
acquireLock(Integer, Integer) Boolean
releaseLock(Integer) void
instance ExclusiveLockManager
CreateHealthCareProviderController
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
processHCPAccount(HttpServletRequest) HealthCareProvider
processHealthCareProviderRequest(HttpServletRequest, HealthCareProvider) void

To view a better version of this diagram, please visit:
https://github.com/SWEN900072021/Covid19BookingSystem/blob/master/docs/part3/diagrams/ClassDiagram1.png
https://github.com/SWEN900072021/Covid19BookingSystem/blob/master/docs/part3/diagrams/ClassDiagram2.png

# Concurrency Issues

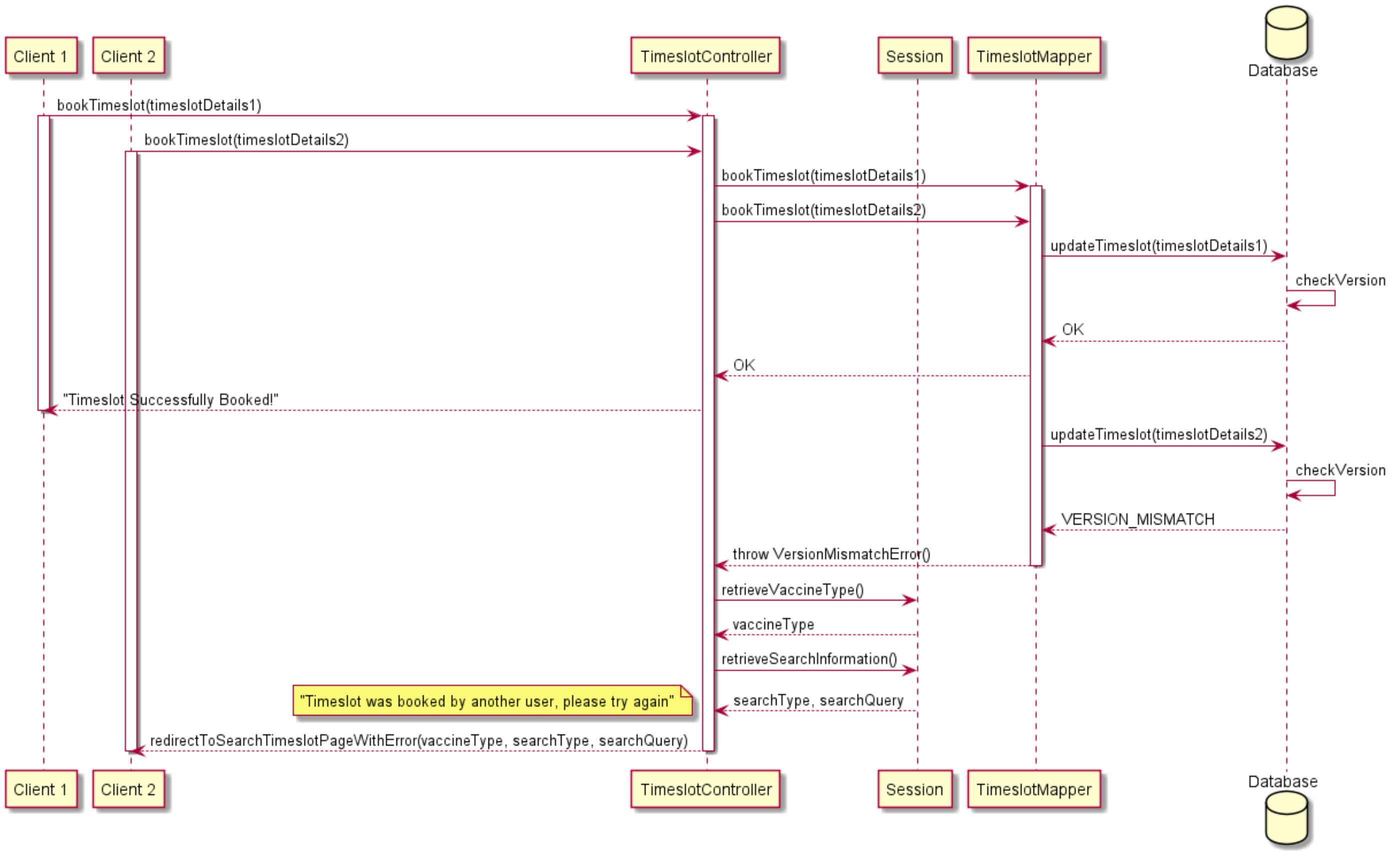
## Concurrency Issue 1

**Use Case**: Book Timeslot

**Issue**: Users can view the same timeslot, then try to book the same timeslot concurrently, meaning that there would be a conflict, and the system needs to ensure that only 1 of the multiple users can successfully book that timeslot.

**Choice of pattern**: Optimistic locking, meaning that at the time of booking the timeslot, the system will check whether the timeslot is actually suitable to be booked, and if not, throw an error back to the user saying that the timeslot has been taken.

**Implementation**: The system uses a function within the database, and a trigger that is attached to the timeslot table that runs before every update, it means that every time an update occurs on the timeslot table, the function will check if the incoming version is the same as the existing version within the database. If the version is different, it will raise a version mismatch exception, otherwise if the versions match, it will increment the version by one, and then proceed with the update. As it utilises a trigger, it also means that if the update fails, the database automatically rolls back the version increment too, upholding the ACID property for the transaction.

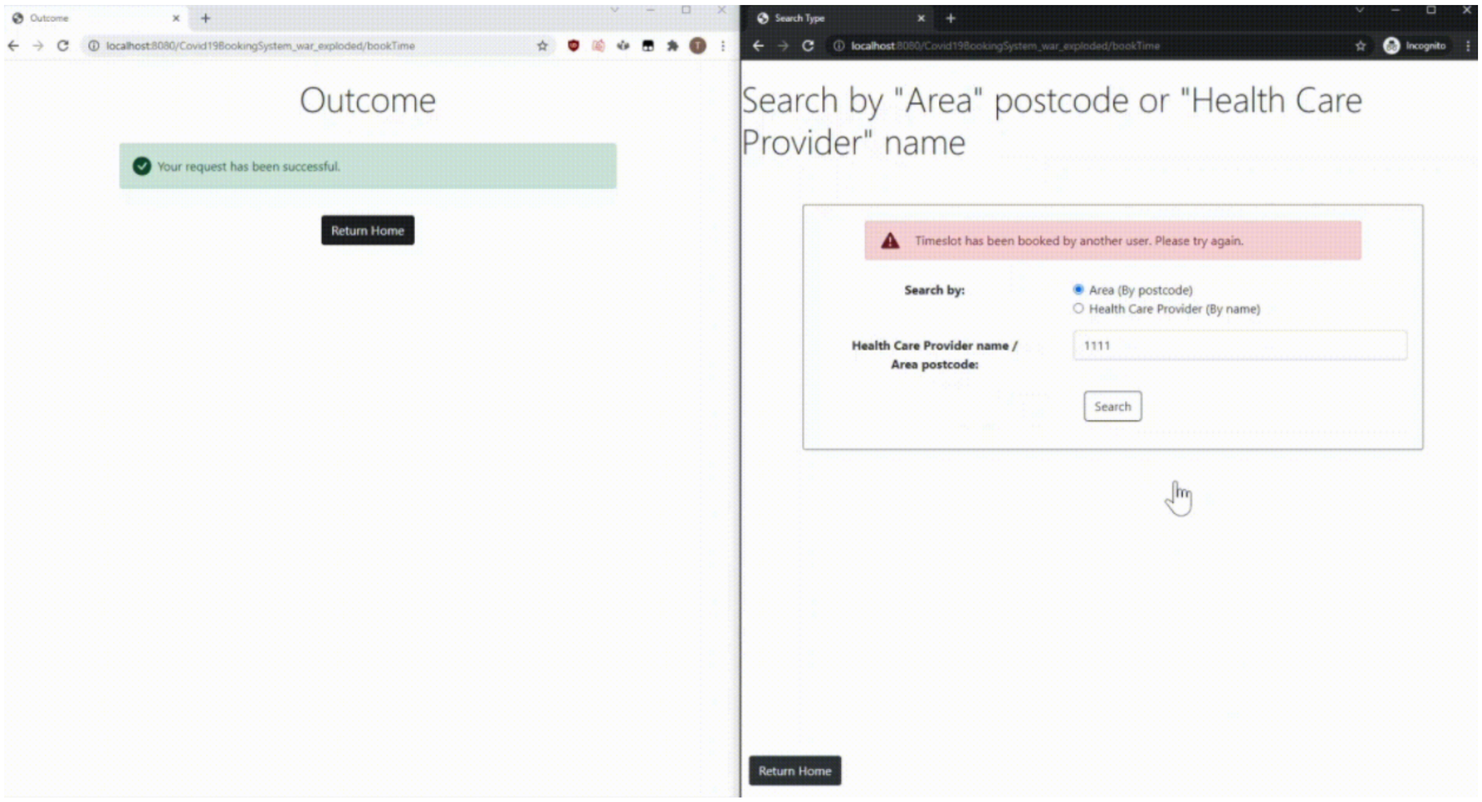

Client 1
Client 2
TimeslotController
Session
TimeslotMapper
Database
bookTimeslot(timeslotDetails1)
bookTimeslot(timeslotDetails2)
bookTimeslot(timeslotDetails1)
bookTimeslot(timeslotDetails2)
updateTimeslot(timeslotDetails1)
checkVersion
OK
OK
"Timeslot Successfully Booked!"
updateTimeslot(timeslotDetails2)
checkVersion
VERSION_MISMATCH
throw VersionMismatchError()
retrieveVaccineType()
vaccineType
retrieveSearchInformation()
searchType, searchQuery
"Timeslot was booked by another user, please try again"
redirectToSearchTimeslotPageWithError(vaccineType, searchType, searchQuery)

When booking a timeslot, the TimeslotController calls the TimeslotMapper which then tries to update the record into the database by passing in the timeslot (including version). Then the ‘timeslot’ table calls the ‘verify_version’ function which checks that the version provided is the correct one. In the case of an error, it throws a SQLException (SQLSTATE VER01) which is caught, and passed onto the TimeslotMapper. The TimeslotMapper redirects the user back to the searchTimeslot page and displays an error message informing them that the timeslot has been booked by another user, and to please try again.

**Justification**: Using optimistic locking makes sense for the use case of booking timeslots. If a pessimistic approach was taken, it would mean that if a user views another timeslot, it would have to lock out any other user from viewing or editing that same timeslot. Locking on update (exclusive write lock) only won’t work, as users could have already loaded the timeslot onto their page, and then when they go to make the update, it will act the same as an optimistic lock, in that scenario, it will fail the update and throw an error at the time of committing the transaction. This is why the team implemented an optimistic lock, which catches the version mismatch at the time of committing the update to the database. This has two benefits, it ensures that there are no race conditions and ensures that the transaction is atomic (as mentioned above, if the transaction fails, the version increment gets rolled back).
From a user experience perspective, the team has minimized the impact by redirecting the user back to the search for timeslot page, where their previous search is prefilled (by utilising session information), meaning that they are one click away from being back to the same search, or if they want to change their search query, they can easily do so as well. The reason the user doesn’t get redirected to the calendar page, is because they could have been potentially been contesting for the last timeslot available for that specific search query, and redirecting them to a calender view with no timeslots available would not be ideal for the user, so there was a conscious choice to redirect them to the search page, and pre-fill their previous search query to ensure high usability and improve their user experience.

**Testing strategy**:
1. Open two or more tabs (incognito) and log in with multiple users.
2. Navigate to the book timeslot page, and choose the same vaccine type for all the users.
3. Fill out the questionnaire and proceed to the search page.
4. Enter the same search query for all the users.
5. On the calendar view, proceed to select the same timeslot for all the users, and navigate to the confirmation page.
6. Click ‘Book Timeslot’ on all the timeslots at the same time. The expected result is only one of the multiple users should be redirected to a success page. The rest of the users should be redirected to the search timeslot page with an error message describing that another user has already booked the timeslot.

**Outcome**: Only one user can successfully book a timeslot as expected, rest of the users trying to book the same timeslot are redirected to the search timeslot page, and are prompted to try again, as the timeslot they were trying to book was booked by another user.


Outcome

Your request has been successful.

Search by "Area" postcode or "Health Care Provider" name

Timeslot has been booked by another user. Please try again.

Search by: Area (By postcode) / Health Care Provider (By name)

Health Care Provider name / Area postcode: 1111

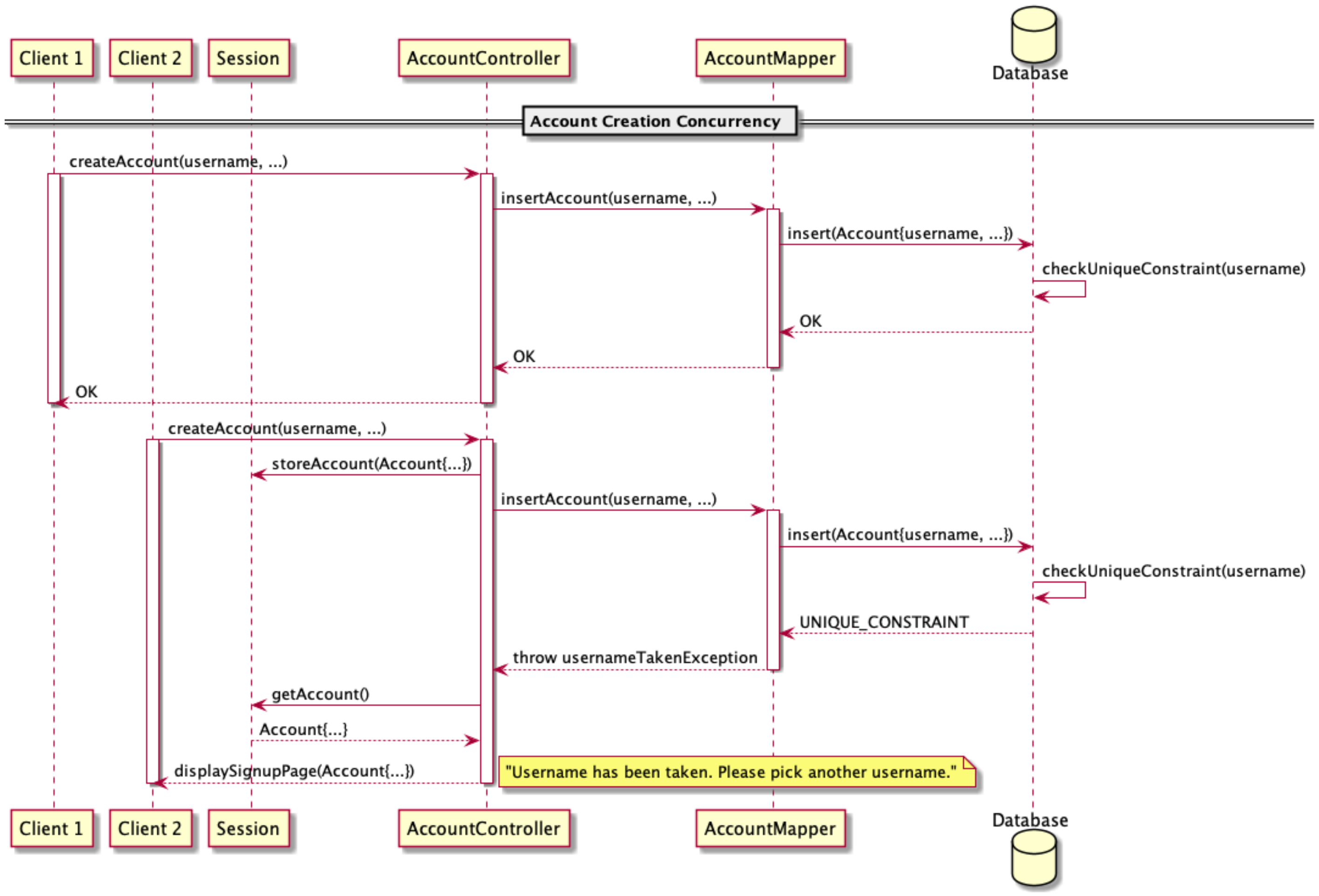
## Concurrency Issue 2

**Use Case**: Create Vaccine Recipient Account & Create Health Care Provider Account

**Issue:** When there are two or more users who want to create accounts with the same username, the system needs to ensure that only one account creation request is successful. The proceeding requests with the same username should be rejected and be marked as duplicates. This concurrency issue can occur whenever an account is created, including both VaccineRecipients or HealthCareProviders.

**Choice of Pattern:** A unique constraint is used to handle the concurrency issue of multiple users creating an account at the same time. The system needs to ensure that only one account can exist with a particular username. In the scenario where multiple users are trying to create an account with the same username, at the time of creation, only one account would be allowed to be created, whereas the rest of the users are shown an error message on the account creation page prompting them to try a different username.

**Implementation**
As the ‘username’ column has a unique constraint in the database, any requests attempted to create an account with a non-unique username will be rejected. The sequence diagram below shows the use case when two users attempt to create Vaccine Recipient accounts with the



Client 1
Client 2
Session
AccountController
AccountMapper
Database
Account Creation Concurrency
createAccount(username, ...)
insertAccount(username, ...)
insert(Account{username, ...})
checkUniqueConstraint(username)
OK
OK
OK
createAccount(username, ...)
storeAccount(Account{...})
insertAccount(username, ...)
insert(Account{username, ...})
checkUniqueConstraint(username)
UNIQUE_CONSTRAINT
throw usernameTakenException
getAccount()
Account{...}
displaySignupPage(Account{...})
"Username has been taken. Please pick another username."

same username. The Account Mapper catches a SQLException with SQLSTATE 23505 (a violation of the constraint imposed by a unique index or a unique constraint has occurred) which gets thrown from the database. It then passes on the error to the controller where the error is handled gracefully, and the appropriate message is returned to the user. The same process occurs with HealthCareProvider accounts as they are handled in the same way (through the same AccountMapper).

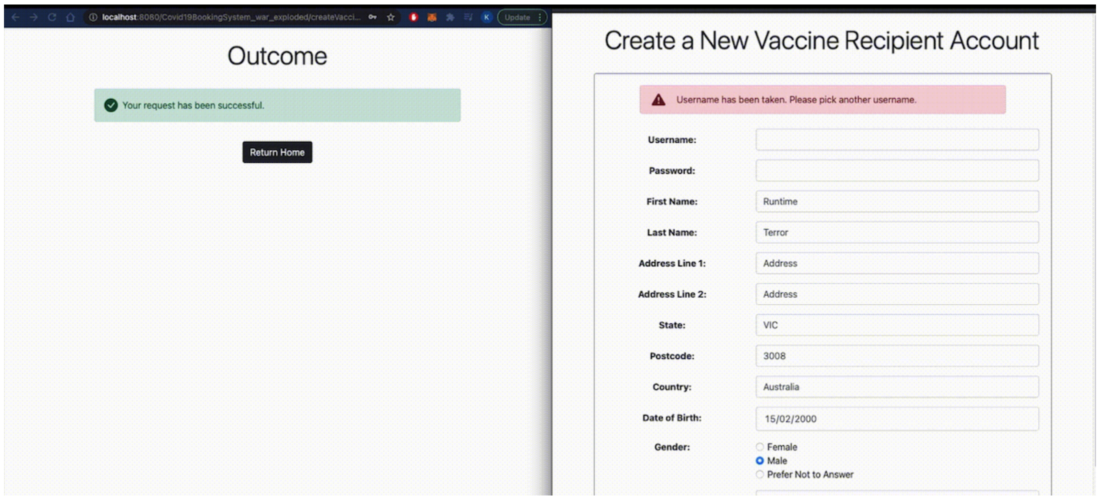
**Justification**
Using a unique constraint allows for good liveness, as user creation is a feature that most new VaccineRecipients and HealthCareProviders require. It would be an unpleasant experience if pessimistic locking was to be applied here, as it would mean only one user could create an account at any given instance. Using a unique constraint, it ensures that there are no race conditions, as all of the checks are applied at the database level. To minimise loss of work for the user, all the fields except for ‘username’ and ‘password’ are pre-filled from their previous

attempt, with an error message informing them that the username they are trying to use is taken, and to please try again with a different one.

**Testing Strategy**

1. Open 2 or more tabs (incognito) and click on the "Create Vaccine Recipient Account" button on all tabs.
2. Fill in the "Username" field with an identical username and complete the form with the required information.
3. Fill in the rest of the form with arbitrary data.
4. Click 'Create Account' on all active tabs at the same time. The expected result is only one of the multiple users should be redirected to a success page. The rest of the users should be presented with an 'username taken' error, prompting them to create the account with a different username.

**Outcome:** All users must create accounts with unique usernames. Users who attempt to create an account with a duplicate username will receive an error message and be prompted to try again with a different username.


localhost:8080/Covid19BookingSystem_war_exploded/createVacci...
Update
Return Home

Outcome

Your request has been successful.

Create a New Vaccine Recipient Account

Username has been taken. Please pick another username.

| | |
|---|---|
| Username: | |
| Password: | |
| First Name: | Runtime |
| Last Name: | Terror |
| Address Line 1: | Address |
| Address Line 2: | Address |
| State: | VIC |
| Postcode: | 3008 |
| Country: | Australia |
| Date of Birth: | 15/02/2000 |
| Gender: | Female [✓ Male] Prefer Not to Answer |

## Concurrency Issue 3

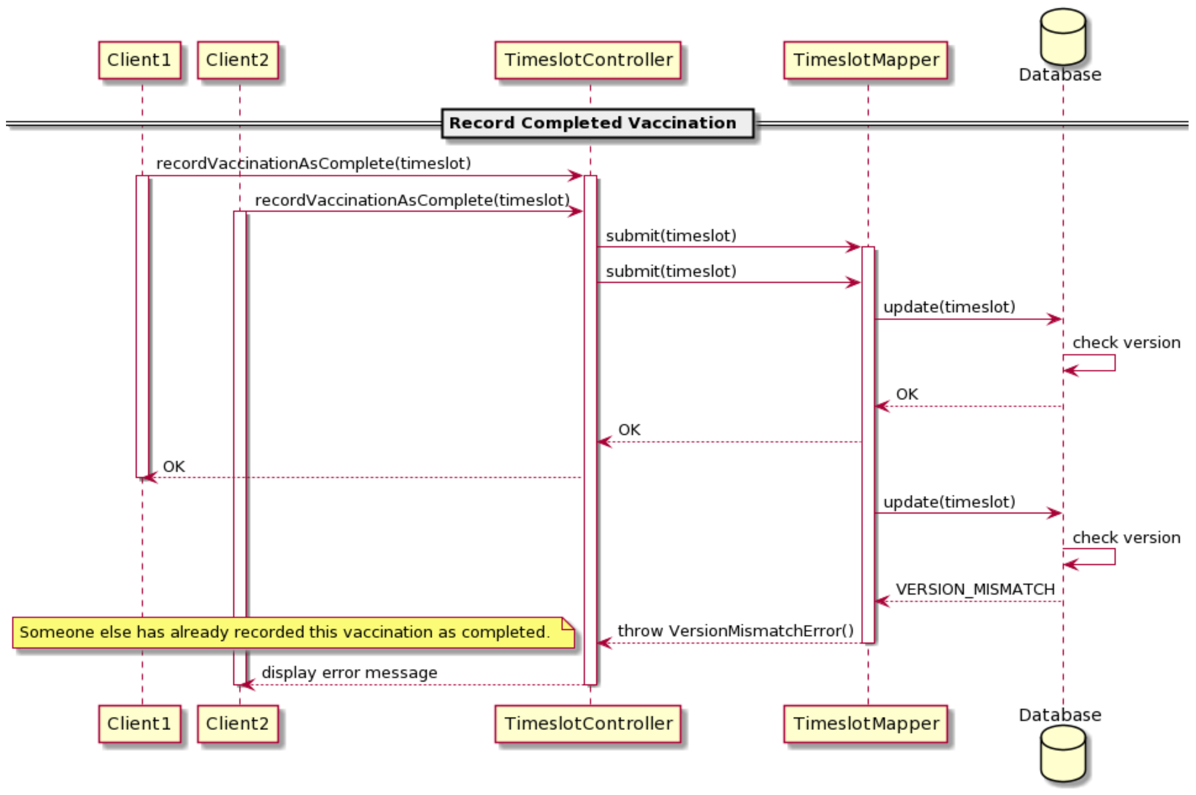
**Use Case:** Record completed vaccination

**Issue:** When there are two or more users who want to record the same timeslot as completed, the system needs to ensure that only one of those requests is successful. The other requests for the same timeslot should show an error to the respective user that the vaccination has already been recorded.

**Choice of pattern**: Optimistic (version) locking is used, meaning that when a user selects a timeslot to be marked as completed, the system will check whether the version of the timeslot is unchanged. If it has been changed, we throw an error back to the user specifying that the timeslot has already been marked as completed.

**Implementation**: The system uses a function within the database, and a trigger that is attached to the timeslot table that runs before every update, it means that every time an update occurs on the timeslot table, the function will check if the incoming version is the same as the existing version within the database. If the version is different, it will raise a version mismatch exception, otherwise if the versions match, it will increment the version by one, and then proceed with the update. As it utilises a trigger, it also means that if the update fails, the database automatically rolls back the version increment too.

When recording a timeslot as complete, the TimeslotController calls the TimeslotMapper which then tries to update the record into the database by passing in the timeslot (including version). Then the ‘timeslot’ table calls the ‘verify_version’ function which checks that the version provided is the correct one. In the case of an error, it throws a SQLException (SQLSTATE VER01) which is caught, and passed onto the TimeslotMapper. The TimeslotMapper then sends an error message informing the user that the timeslot has been marked as complete by another user.

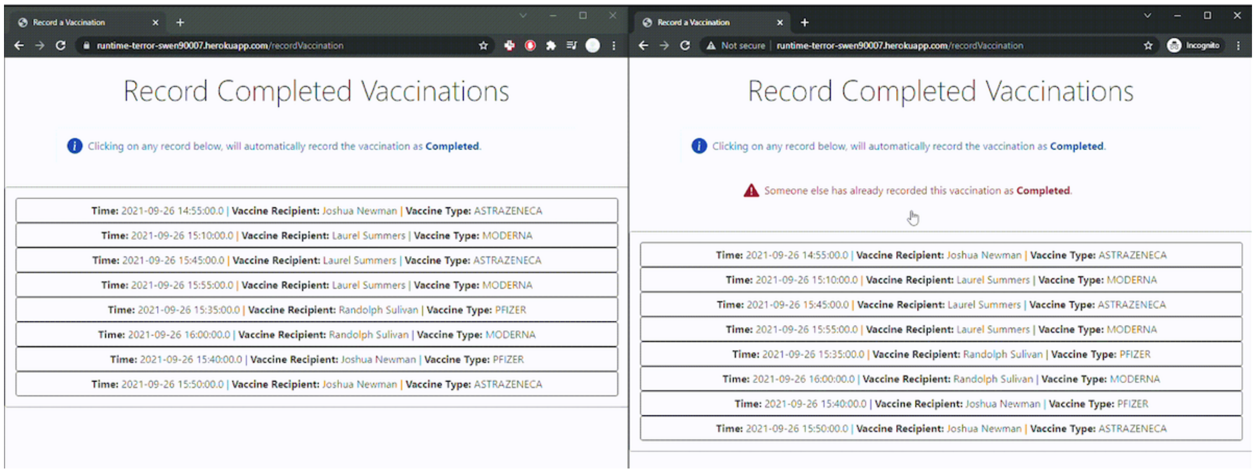

Client1
Client2
TimeslotController
TimeslotMapper
Database
Record Completed Vaccination
recordVaccinationAsComplete(timeslot)
recordVaccinationAsComplete(timeslot)
submit(timeslot)
submit(timeslot)
update(timeslot)
check version
OK
OK
OK
update(timeslot)
check version
VERSION_MISMATCH
Someone else has already recorded this vaccination as completed.
throw VersionMismatchError()
display error message

**Justification**: Similarly to the booking timeslot use case, we felt that using optimistic locking here also makes more sense than a pessimistic approach. An optimistic lock catches version mismatches when it commits the update to the database which ensures no race conditions and atomic transactions. The user is simply shown an error message whenever they select a timeslot which has already been marked as completed which makes it so the workflow is not disrupted and the user can carry on to mark other vaccinations as completed. Moreover, there will be no loss of data which is the main disadvantage of optimistic concurrency approach, since the user only clicks on a particular timeslot if they want to mark it as completed.

**Testing strategy**:

1. Open two or more tabs (incognito).
2. Login with different HealthCareProvider accounts from the same organisation (same organisational Id) or the same account.
3. Navigate to the record completed timeslot page for both users, and choose the same timeslot to record it as complete.
4. For one of the users, the timeslot is marked as completed and disappears from the list as expected.
5. For the other users, the page is reloaded and a message appears indicating that it has already been marked as completed.

**Outcome**: Only one user can successfully record a timeslot as completed, the other users trying to record the same timeslot as complete are shown a message indicating that it has already been marked as completed by another user.


Record a Vaccination
runtime-terror-swen90007.herokuapp.com/recordVaccination
Record a Vaccination
Not secure
runtime-terror-swen90007.herokuapp.com/recordVaccination
Incognito

Record Completed Vaccinations

Clicking on any record below, will automatically record the vaccination as **Completed**.

**Time:** 2021-09-26 14:55:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** ASTRAZENECA
**Time:** 2021-09-26 15:10:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:45:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** ASTRAZENECA
**Time:** 2021-09-26 15:55:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:35:00.0 | **Vaccine Recipient:** Randolph Sulivan | **Vaccine Type:** PFIZER
**Time:** 2021-09-26 16:00:00.0 | **Vaccine Recipient:** Randolph Sulivan | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:40:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** PFIZER
**Time:** 2021-09-26 15:50:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** ASTRAZENECA

Record Completed Vaccinations

Clicking on any record below, will automatically record the vaccination as **Completed**.

Someone else has already recorded this vaccination as **Completed**.

**Time:** 2021-09-26 14:55:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** ASTRAZENECA
**Time:** 2021-09-26 15:10:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:45:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** ASTRAZENECA
**Time:** 2021-09-26 15:55:00.0 | **Vaccine Recipient:** Laurel Summers | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:35:00.0 | **Vaccine Recipient:** Randolph Sulivan | **Vaccine Type:** PFIZER
**Time:** 2021-09-26 16:00:00.0 | **Vaccine Recipient:** Randolph Sulivan | **Vaccine Type:** MODERNA
**Time:** 2021-09-26 15:40:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** PFIZER
**Time:** 2021-09-26 15:50:00.0 | **Vaccine Recipient:** Joshua Newman | **Vaccine Type:** ASTRAZENECA

## Concurrency Issue 4

**Use Case:** Edit Timeslots

### Issue

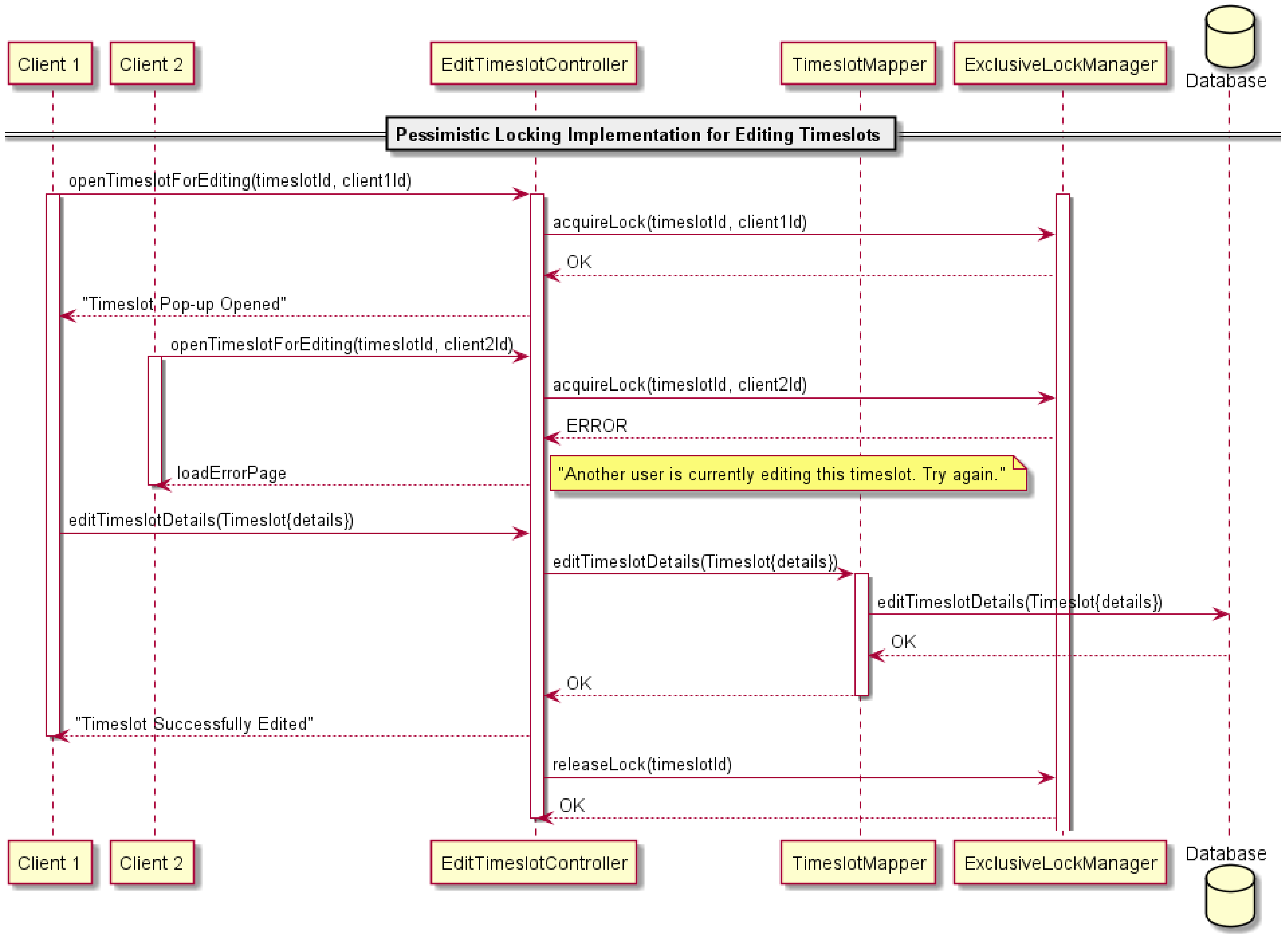
When editing a timeslot, if another user was to edit the same timeslot, there could be a potential conflict and a 'lost update' could occur, where a user's update is overridden by another user's changes. To prevent this, the system would need to ensure that only one of the users can successfully edit a particular timeslot at any given time.

### Choice of pattern

Exclusive write lock (pessimistic offline locking) is used, meaning that at the time of editing any timeslot, the system will have to first acquire an exclusive write lock on the timeslot being edited. If the lock for that timeslot is not acquired by someone else, then the user will be able to edit the timeslot. However, if the lock is already acquired by someone else, the user will be sent to an error page and will hence not be able to edit the chosen timeslot.

### Implementation

Pessimistic locking is implemented through a class called "ExculsiveLockManager". This class is a singleton throughout the application, meaning that there is only one instance of it in the application. It maintains a lockMap in the form of a ConcurrentHashMap which stores the mapping between timeslotId and userId. When a healthcare provider user visits the "Edit Timeslots" page, they are shown a list of timeslots that they can edit. When the user attempts to open a particular timeslot, the "EditTimeslotController" is sent a POST request with the 'timeslotId' and the 'userId' of the healthcare provider. Through this request, the user attempts to acquire a lock on the chosen timeslot.
The acquireLock method first checks whether the lockMap already contains the timeslotId as a key. This would determine if another user has already locked this timeslot or not. If the lockMap does not contain the chosen timeslotId as a key, then the user is able to acquire the lock which entails adding the key-value pair of timeslotId and userId to the lockMap. Following this, the user is shown a pop-up window through which they can proceed with editing the chosen


Client 1
Client 2
EditTimeslotController
TimeslotMapper
ExclusiveLockManager
Database
Pessimistic Locking Implementation for Editing Timeslots
openTimeslotForEditing(timeslotId, client1Id)
acquireLock(timeslotId, client1Id)
OK
"Timeslot Pop-up Opened"
openTimeslotForEditing(timeslotId, client2Id)
acquireLock(timeslotId, client2Id)
ERROR
loadErrorPage
"Another user is currently editing this timeslot. Try again."
editTimeslotDetails(Timeslot{details})
editTimeslotDetails(Timeslot{details})
editTimeslotDetails(Timeslot{details})
OK
OK
"Timeslot Successfully Edited"
releaseLock(timeslotId)
OK

timeslot. However, if the lockMap already contains the timeslotId as a key, then the user is shown an error message indicating that another user is currently editing the timeslot and they would need to try again later. Finally, when the user completes editing the chosen timeslot, the lock on the timeslot is released by removing the key-value pair of the timeslotId and userId from the lockMap.

**Justification**

Pessimistic locking seemed to be the most appropriate strategy for the use case of editing timeslots. The biggest advantage of pessimistic locking, when compared to optimistic locking, is that it avoids conflict occurring all together by locking the entity throughout the entire life of the use case. This results in data never being lost which saves the user's efforts of editing a timeslot that might not end up being committed. However, it does reduce liveness as for the whole duration of editing a given timeslot, no other user would be able to edit it. However, the use case of editing timeslots is local to healthcare provider organisations and would not often be conducted concurrently for the exact same timeslots.

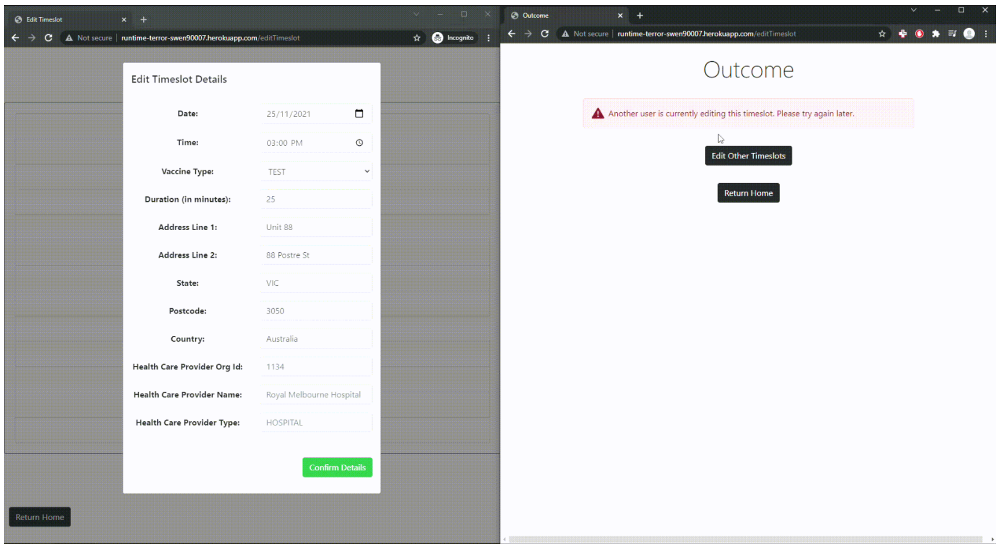
Moreover, if we only used optimistic locking, two or more users could open the same timeslot and edit them at the same time. Editing timeslots can be cumbersome as there are many fields in each timeslot such as date, time, location, etc. The user would be unsatisfied if they tried to confirm the new timeslot details and the application notified them of an error. Using pessimistic locking, it is ensured that the user would not be able to reach the pop-up window for editing the chosen timeslot if another user is currently editing it, saving the user's time and wasted efforts.

Finally, exclusive write lock is used instead of exclusive read lock, meaning that all users can view timeslots that they can potentially edit in a list view. However, only one user can acquire a write lock on a particular timeslot through which they can edit its details. Unrepeatable or inconsistent reads in the case of editing timeslots mean that when a user views (reads) timeslots in a list view, another user could update (write) one of those timeslots. This would result in the current user viewing an older version of the timeslot, which is avoided by using exclusive read locks. However, avoiding inconsistent reads is not a priority for this use case because it would be unreasonable to assume that a user would stay on the edit timeslots page for the entire duration of another user editing a particular timeslot. Moreover, we have avoided the scenario in which a user attempts to edit an "inconsistent" timeslot by using version (optimistic) locking.

**Testing strategy**:

1. Open two or more tabs (incognito).
2. Login with different HealthCareProvider accounts from the same organisation (same organisational Id).
3. Navigate to the "Edit Timeslots" on all tabs through which you should be able to see all the timeslots from the future from all healthcare providers in your organisation.
4. On all tabs, attempt to click on (open) the same timeslot.
5. The editing pop-up will only open up for one of the users (only one of the tabs), whereas the rest will be sent to an error screen indicating that another user is currently editing the timeslot.

**Outcome**: Only one user can successfully open the edit timeslot pop-up for a given timeslot at a time. For the entire duration of a user editing a given timeslot, all the other users are sent to an error page indicating that another user is currently editing the timeslot and they would have to try again later.


Edit Timeslot Details

| | |
|---|---|
| Date: | 25/11/2021 |
| Time: | 03:00 PM |
| Vaccine Type: | TEST |
| Duration (in minutes): | 25 |
| Address Line 1: | Unit 88 |
| Address Line 2: | 88 Postre St |
| State: | VIC |
| Postcode: | 3050 |
| Country: | Australia |
| Health Care Provider Org Id: | 1134 |
| Health Care Provider Name: | Royal Melbourne Hospital |
| Health Care Provider Type: | HOSPITAL |

Outcome

Another user is currently editing this timeslot. Please try again later.