Software Architecture Design (SAD) Document

Submission 2 Specification

Runtime Terror

SWEN90007 SM2 2021 Project

Release Tag:

https://github.com/SWEN900072021/Covid19BookingSystem/releases/tag/SWEN90007_2021_Part2_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

Implemented Use Cases

- **Use Case 1:** Create Accounts
- **Use Case 2:** Create Health Care Provider Account
- **Use Case 3:** Create Vaccine Recipient Account
- **Use Case 4:** Add Vaccine Types
- **Use Case 5:** View All Users
- **Use Case 6:** View All Vaccine Recipients
- **Use Case 7:** Filter by Type of Vaccine Received
- **Use Case 8:** View All Time-Slots
- **Use Case 9:** Determine Eligibility
- **Use Case 10:** Search for Time-Slot
- **Use Case 11:** Search by Area
- **Use Case 12:** Search by Health Care Provider
- **Use Case 13:** Book a Time-Slot
- **Use Case 14:** Add Time Slot
- **Use Case 15:** Customize Time Slot
- **Use Case 16:** Record a Completed Vaccination
- **Use Case 17:** Access Vaccination Certificate
- **Use Case 18**: Log in

Implemented Patterns

- Domain model
- Data mapper
- Unit of work
- Lazy load (Ghost)
- Identity field
- Foreign key mapping
- Association table mapping
- Embedded value
- Inheritance pattern (Class table inheritance)
- Authentication and Authorization

Domain Model Diagram

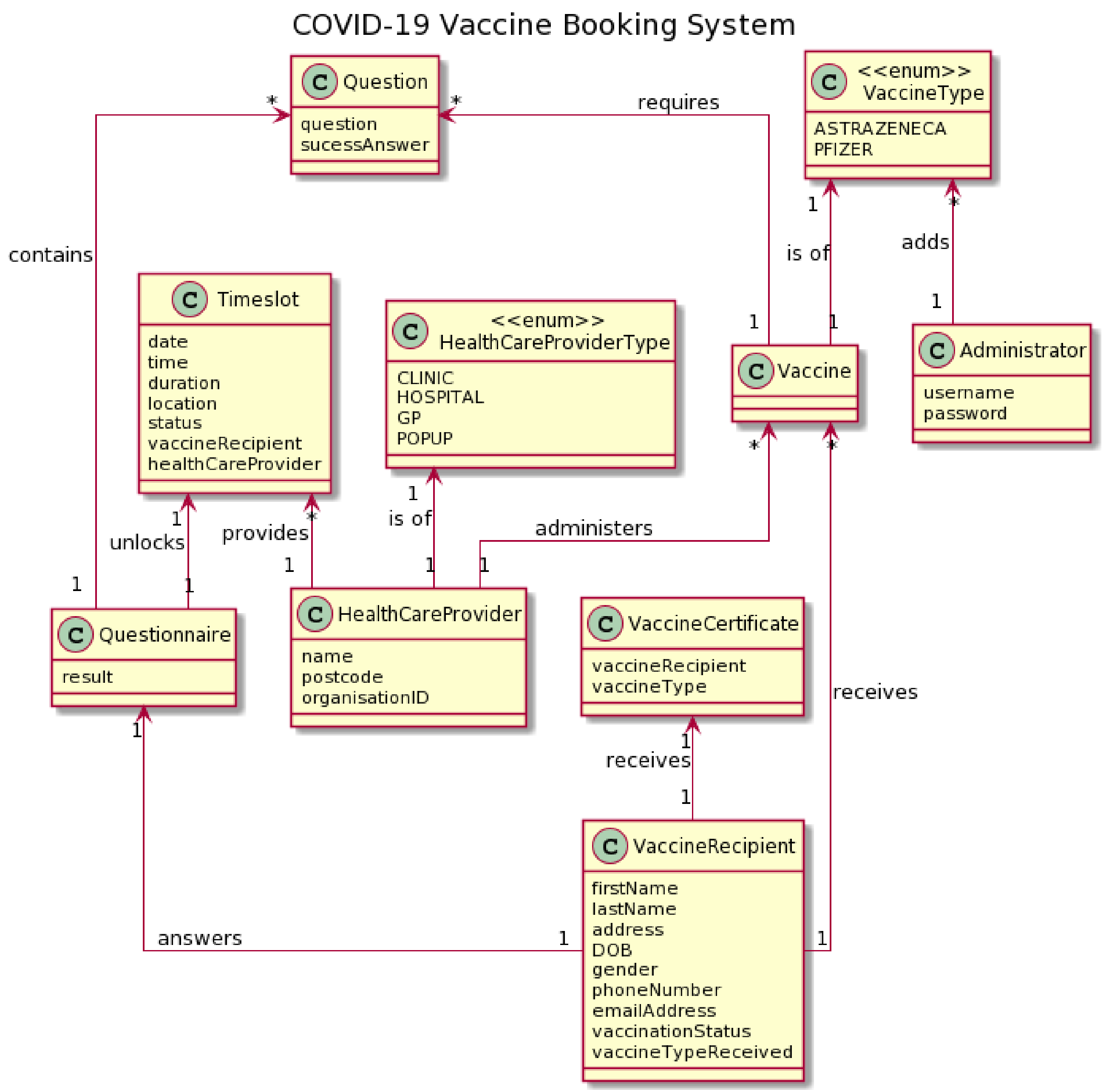


Figure 1: Domain Model

Class Diagram

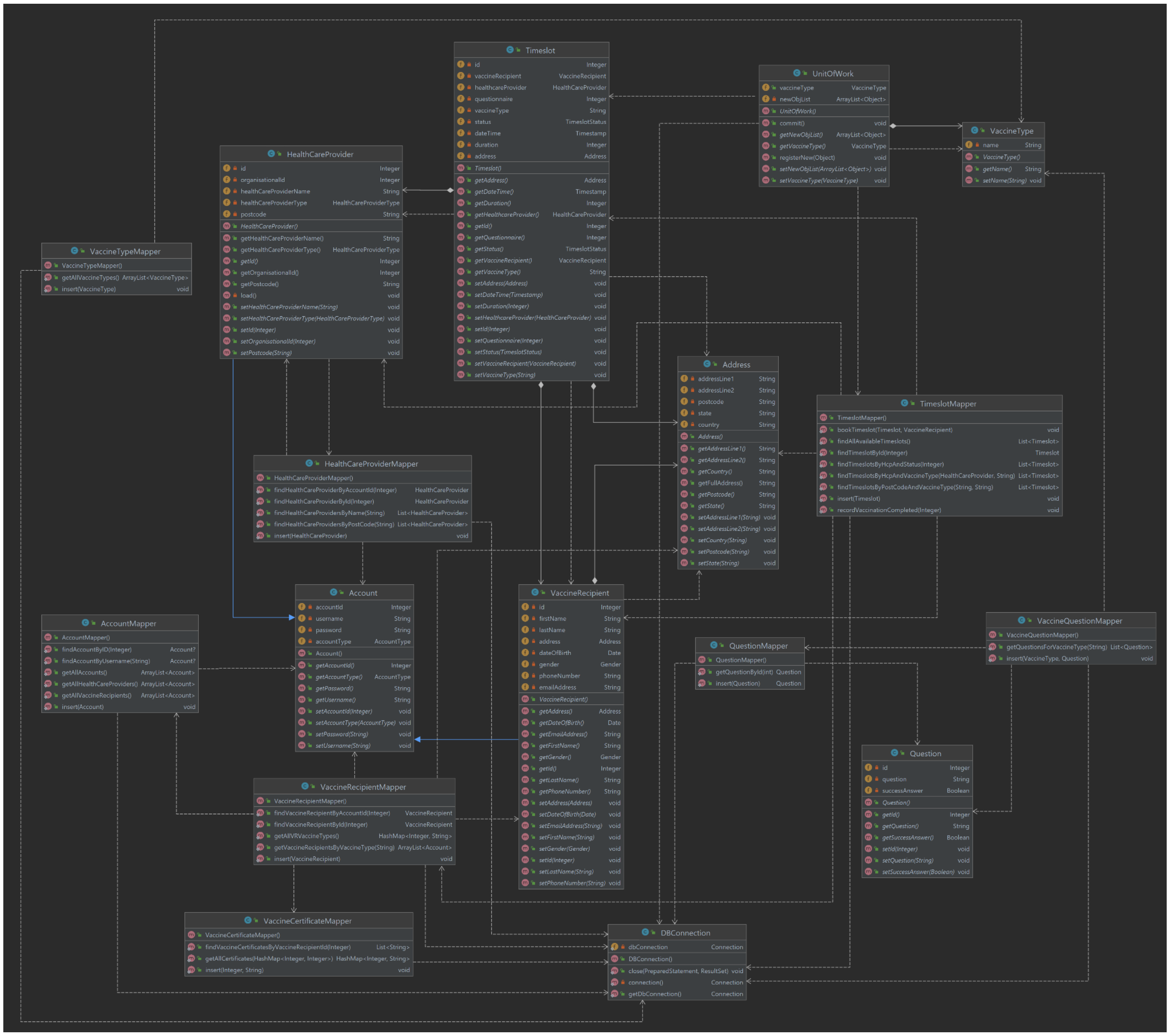

Timeslot
id Integer
vaccineRecipient VaccineRecipient
healthcareProvider HealthCareProvider
questionnaire Integer
vaccineType String
status TimeslotStatus
dateTime Timestamp
duration Integer
address Address
Timeslot()
getAddress() Address
getDateTime() Timestamp
getDuration() Integer
getHealthcareProvider() 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<Object>
UnitOfWork()
commit() void
getNewObjList() ArrayList<Object>
getVaccineType() VaccineType
registerNew(Object) void
setNewObjList(ArrayList<Object>) 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<VaccineType>
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<Timeslot>
findTimeslotById(Integer) Timeslot
findTimeslotsByHcpAndStatus(Integer) List<Timeslot>
findTimeslotsByHcpAndVaccineType(HealthCareProvider, String) List<Timeslot>
findTimeslotsByPostCodeAndVaccineType(String, String) List<Timeslot>
insert(Timeslot) void
recordVaccinationCompleted(Integer) void
HealthCareProviderMapper
HealthCareProviderMapper()
findHealthCareProviderByAccountId(Integer) HealthCareProvider
findHealthCareProviderById(Integer) HealthCareProvider
findHealthCareProvidersByName(String) List<HealthCareProvider>
findHealthCareProvidersByPostCode(String) List<HealthCareProvider>
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<Account>
getAllHealthCareProviders() ArrayList<Account>
getAllVaccineRecipients() ArrayList<Account>
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<Question>
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<Integer, String>
getVaccineRecipientsByVaccineType(String) ArrayList<Account>
insert(VaccineRecipient) void
VaccineCertificateMapper
VaccineCertificateMapper()
findVaccineCertificatesByVaccineRecipientId(Integer) List<String>
getAllCertificates(HashMap<Integer, Integer>) HashMap<Integer, String>
insert(Integer, String) void
DBConnection
dbConnection Connection
DBConnection()
close(PreparedStatement, ResultSet) void
connection() Connection
getDbConnection() Connection

Figure 2: Class diagram

To view a better version of this diagram, please visit:
https://github.com/SWEN900072021/Covid19BookingSystem/blob/master/docs/part2/diagrams/ClassDiagram.png

Database Schema Diagram

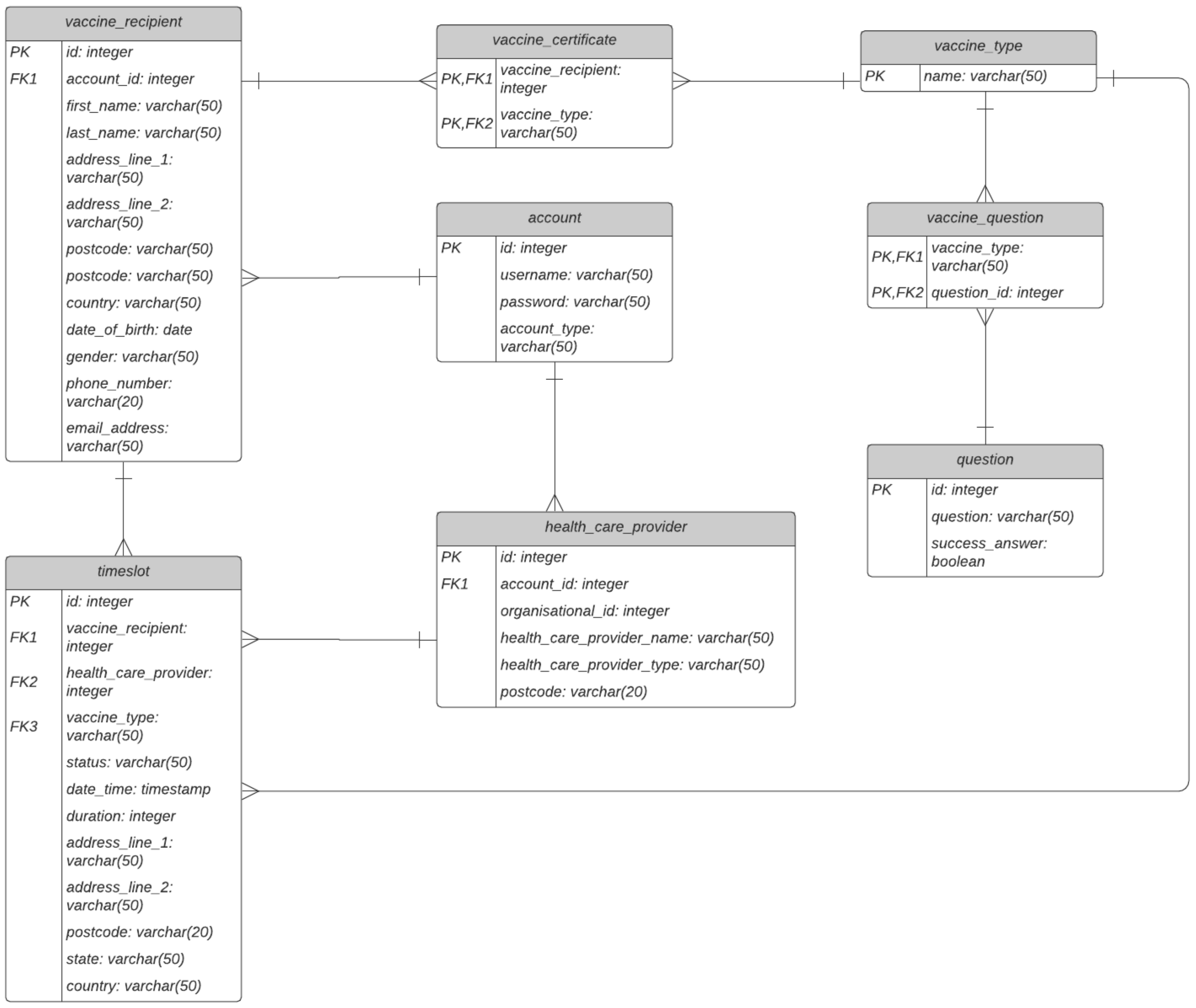


Figure 3: Database Schema Diagram

Patterns and Descriptions

Association Table Mapping

Implementation

Association Table Mapping is used as a link table that shows the association between two other tables by storing their primary keys. We decided on using association table mapping to connect the 'vaccine_type' with the 'question' table to link each VaccineType with its respective list of Questions. It is also used to track the associations between VaccineRecipients and VaccineTypes they have received.

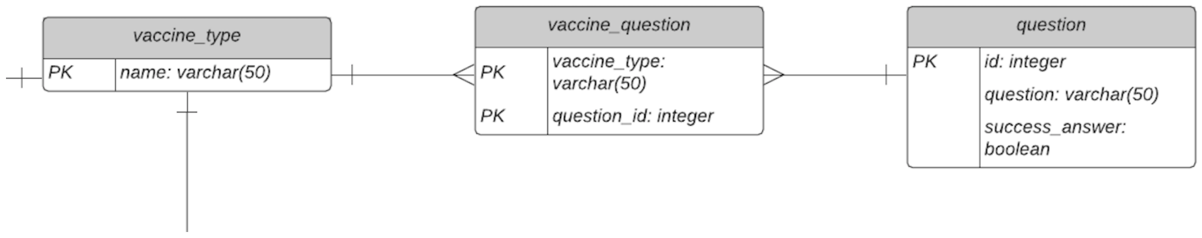


Figure 4: Vaccine Question DB Schema

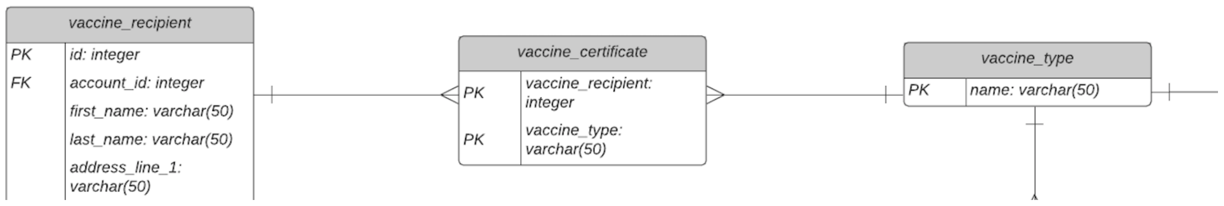


Figure 5: Vaccine Certificate DB Schema

Use of Pattern

When a vaccine recipient wants to book a timeslot, they first select their preferred vaccine type. The system then uses the association table pattern to get all the question ids related to that VaccineType from the ‘vaccine_question’ table. This list of ids is then used to retrieve the questions from the ‘question’ table. Finally, the system displays the list of Questions linked with the chosen VaccineType as a questionnaire that the VaccineRecipient has to fill out before booking a Timeslot.

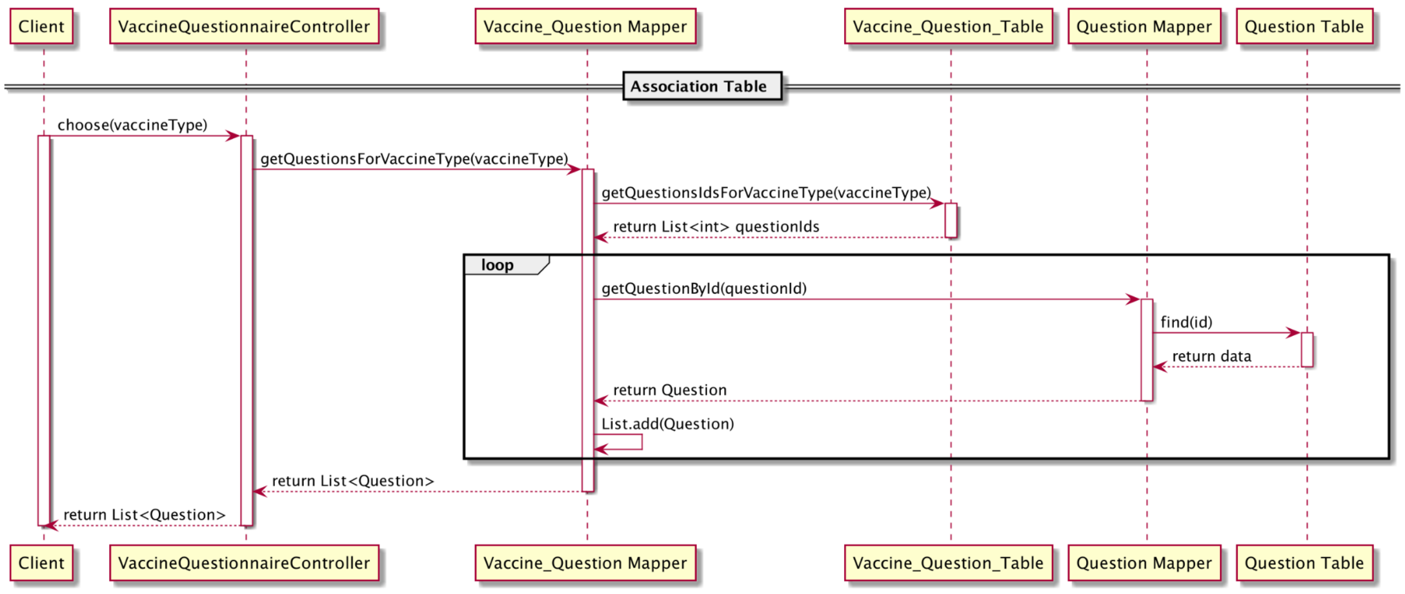


Figure 6: Sequence Diagram for Association Table

Data Mapper

Implementation

Data mappers are used to build an independent relationship between the object class and the database. The insertion of the data mapper layer happens between the domain layer and the data source layer. This reduces coupling as the domain layer would operate without being aware of any database actions.

Use of Pattern

Due to the nature of the data mapper which serves as an interface between the domain layer and the data source layers, this pattern was used each time an object had to have a direct link with the database. Therefore, the data mapper pattern was used in relation to the following objects:

- Account
- HealthcareProvider
- Question
- Timeslot
- VaccineCertificate
- VaccineQuestion
- VaccineRecipient
- VaccineType

These data mappers contained the necessary SQL queries to insert, update and read from these tables.

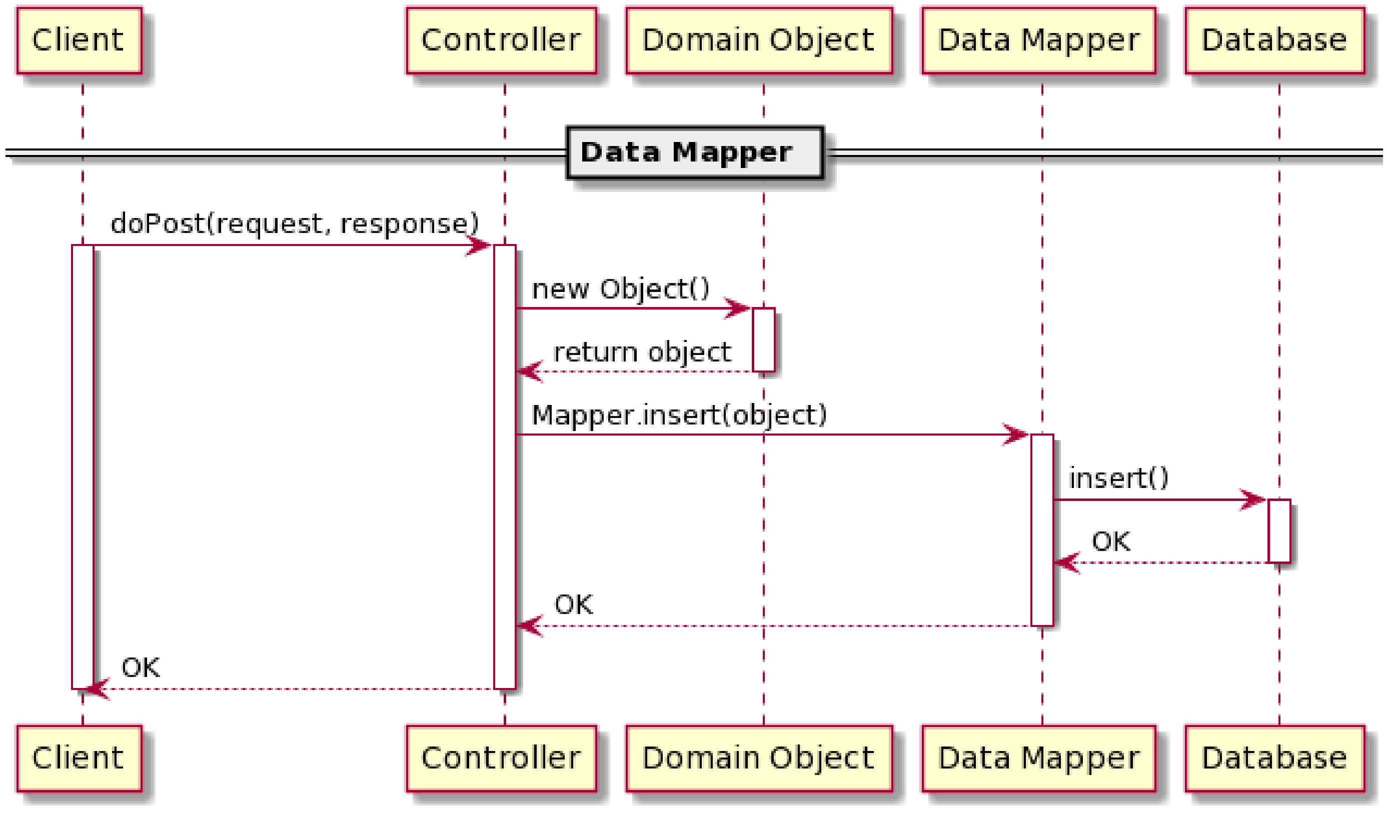


Figure 7: Sequence Diagram for Data Mapper

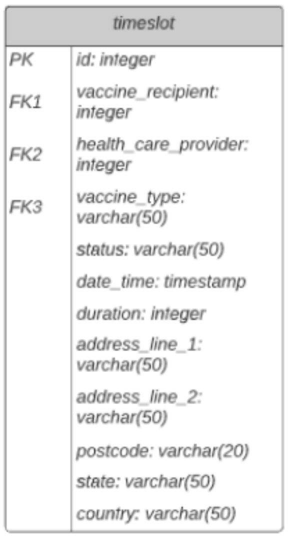
Embedded Value

Implementation

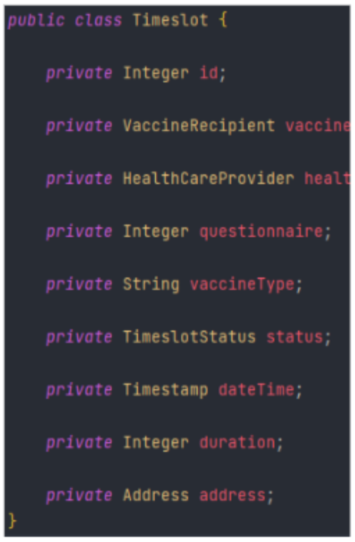
Embedded value is used for objects that don't necessarily need a table in the database. It embeds the fields of an object to values inside that object's owner. We use this pattern to map the different fields of a vaccine recipient's address (street, postcode, country, etc) into one address object, while flattening out these details on the database level.

Use of Pattern

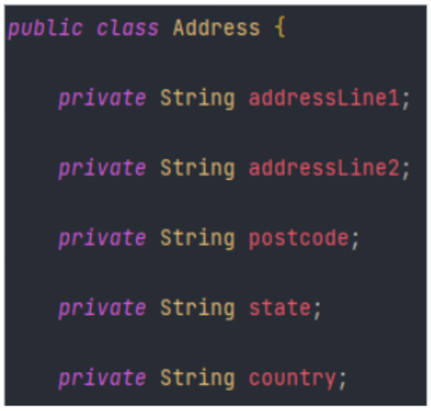
Figure 8 is the domain and database representation of the embedded value pattern for Timeslot.



timeslot	
PK	id: integer
FK1	vaccine_recipient: integer
FK2	health_care_provider: integer
FK3	vaccine_type: varchar(50)
	status: varchar(50)
	date_time: timestamp
	duration: integer
	address_line_1: varchar(50)
	address_line_2: varchar(50)
	postcode: varchar(20)
	state: varchar(50)
	country: varchar(50)



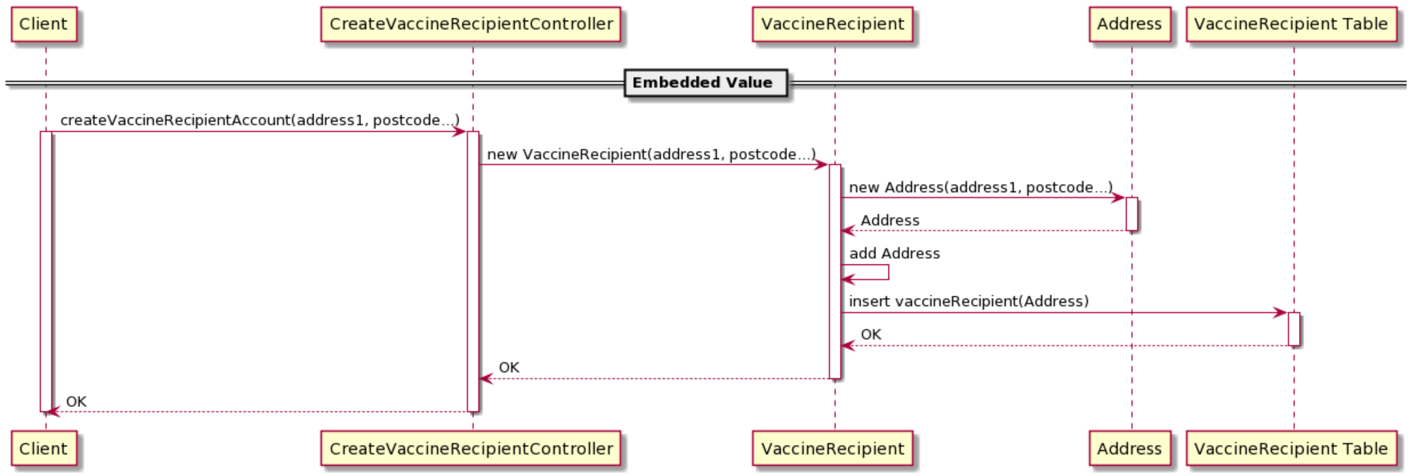
```
public class Timeslot {

    private Integer id;

    private VaccineRecipient vaccine

    private HealthCareProvider healt

    private Integer questionnaire;

    private String vaccineType;

    private TimeslotStatus status;

    private Timestamp dateTime;

    private Integer duration;

    private Address address;
}
```



```
public class Address {

    private String addressLine1;

    private String addressLine2;

    private String postcode;

    private String state;

    private String country;
```

Figure 8: The database and domain representation of Embedded Value for Timeslot

The sequence diagram below (Figure 9) represents the same pattern used for VaccineRecipient. When a vaccine recipient creates an account, they fill out different fields for their address such as address_line1, address2, postcode, state and country. These fields are then stored in an Address domain object which is part of the newly created VaccineRecipient object. The individual address fields are then stored in the same 'vaccine_recipient' table by accessing the Address object inside the VaccineRecipient object.



Client
CreateVaccineRecipientController
VaccineRecipient
Address
VaccineRecipient Table
Embedded Value
createVaccineRecipientAccount(address1, postcode...)
new VaccineRecipient(address1, postcode...)
new Address(address1, postcode...)
Address
add Address
insert vaccineRecipient(Address)
OK
OK
OK

Figure 9: The sequence diagram representation of Embedded Value for VaccineRecipient

Inheritance Pattern

Pattern chosen: Class Table Inheritance

Implementation

There are three types of accounts within the system; ‘Administrator’, ‘VaccineRecipient’ and ‘HealthCareProvider’. Using class table inheritance means that there is a parent class ‘Account’ which stores the common attributes between these three accounts, such as “username” and “password”. Within the database, only the two child classes (VaccineRecipient and HealthCareProvider) each have a table as well as the parent class, so there are three tables; ‘account’, ‘vaccine_recipient’ and ‘health_care_provider’.

Justification

The reason for using class table inheritance is that when a user signs up, the account details are stored within the incoming request, and therefore, it makes sense to store this information along with the user’s other attributes in the one domain object. However, at a database level, the account details of a user is conceptually a different concern, and there is no need to retrieve the user’s personal information when performing authentication (as we only need to validate the combination of username and password). There was also no need for an Administrator domain object or table, as by using an ‘AccountType’ attribute in the Account class, it was able to handle Administrator access, and avoid creating an unnecessary table or domain object, as an Administrator does not have any personal information or attributes in the system.

Use of Pattern

Scenario 1: Creating a VaccineRecipient account

1. A user or administrator goes to the VaccineRecipient account creation page and sends a request to the controller.
2. The controller for VaccineRecipient creation will then create a domain level object VaccineRecipient, which contains both the VaccineRecipient’s account information as well as their personal details.
3. The controller then calls both the AccountMapper and VaccineRecipientMapper to persist the VaccineRecipient’s personal information to the ‘vaccine_recipient’ table and the VaccineRecipient’s account details to the ‘account’ table’ in one atomic transaction.

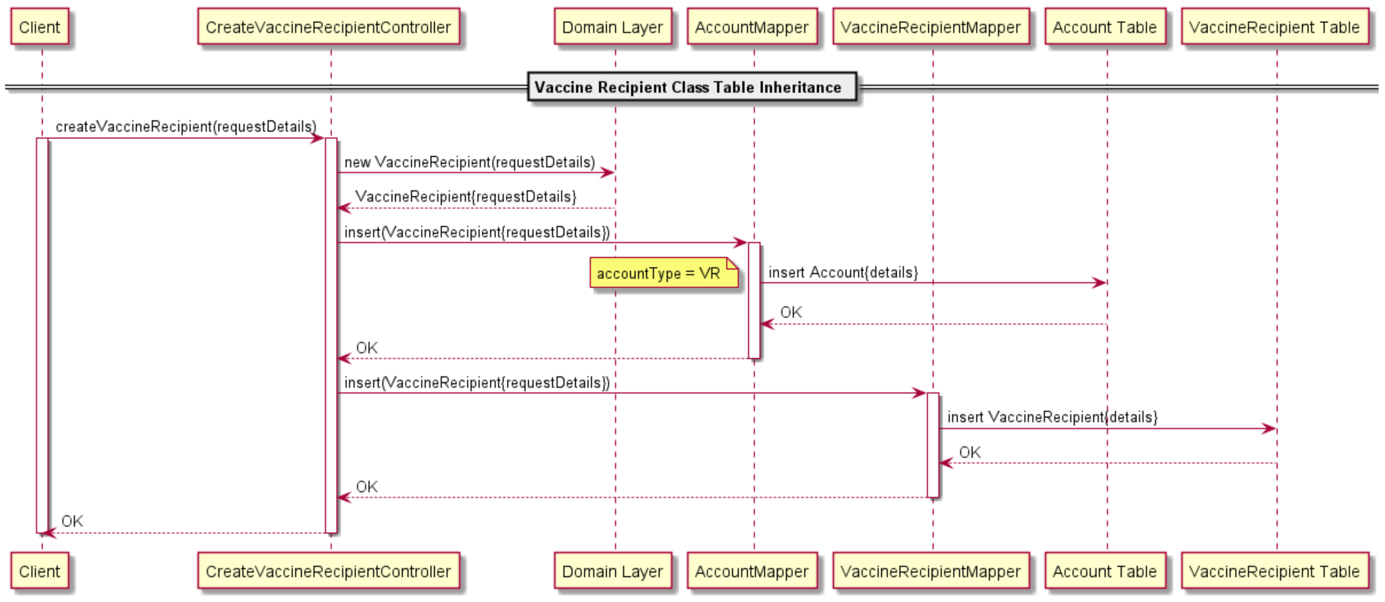

Client
CreateVaccineRecipientController
Domain Layer
AccountMapper
VaccineRecipientMapper
Account Table
VaccineRecipient Table
Vaccine Recipient Class Table Inheritance
createVaccineRecipient(requestDetails)
new VaccineRecipient(requestDetails)
VaccineRecipient{requestDetails}
insert(VaccineRecipient{requestDetails})
accountType = VR
insert Account{details}
OK
OK
insert(VaccineRecipient{requestDetails})
insert VaccineRecipient{details}
OK
OK
OK

Figure 10: Sequence Diagram representation of Class TableInheritance for VaccineRecipients

<u>Scenario 2: Creating a HealthCareProvider account</u>

1. An administrator goes to the HealthCareProvider account creation page and sends a request to the controller.
2. The controller for HealthCareProvider creation will then create a domain level object HealthCareProvider, which contains both the HealthCareProvider's account information as well as their organisational details.
3. The controller then calls both the AccountMapper and HealthCareProviderMapper to persist the HealthCareProvider's organizational information to the 'health_care_provider' table and theHealthCareProvider's account details to the 'account' table' in one atomic transaction.

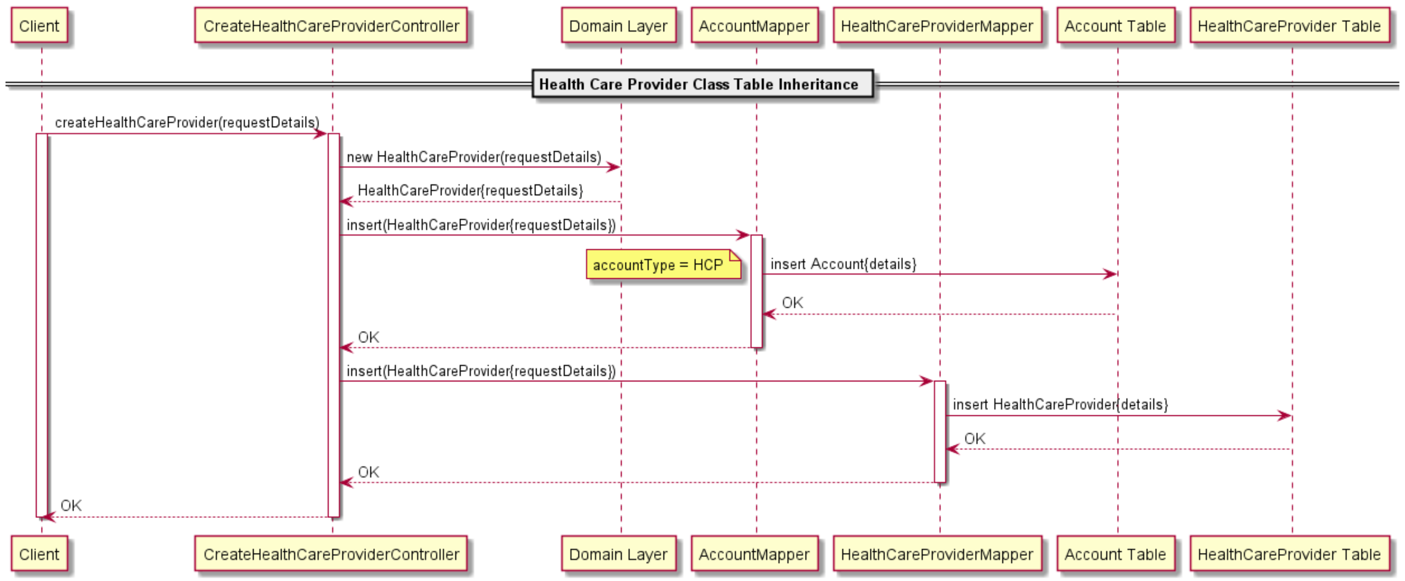

Client
CreateHealthCareProviderController
Domain Layer
AccountMapper
HealthCareProviderMapper
Account Table
HealthCareProvider Table
Health Care Provider Class Table Inheritance
createHealthCareProvider(requestDetails)
new HealthCareProvider(requestDetails)
HealthCareProvider{requestDetails}
insert(HealthCareProvider{requestDetails})
accountType = HCP
insert Account{details}
OK
OK
insert(HealthCareProvider{requestDetails})
insert HealthCareProvider{details}
OK
OK
OK

Figure 11: Sequence Diagram representation of Class Table Inheritance for Health Care Provider

Unit of Work

Implementation

Unit of Work is a utility tool in our system to help reduce the number of times we query by bulk executing the queries at the end of a multi-step business transaction. We identified the best use for a unit of work in our system is to handle bulk creation of data entries/ objects into our database. The system utilises Caller Registration to update the information on the Unit of work.

Design Rationale

In the use case of HealthCareProviders adding Timeslot(s), they could be adding up to a day's or even a week's worth of Timeslots at a time which can be a significant amount. By bulk executing the large number of individual Timeslot creation queries, we reduce the number of times we commit to the database. This ensures that all the Timeslots are inserted into the database in one transaction.

Unit of work is used in the system to not only reduce the number of hits to the database when bulk creating data entries but also to gracefully handle data entry errors during bulk creation. Postgres has a built-in rollback functionality that will rollback the entire bulk transaction in the event one of the queries fails to be executed. Currently the system does not utilise dirty and deleted lists in the unit of work as there are no suitable use cases for them.

The use of Unit of work allows good software design by having high cohesion and simplicity in design where we place all the information of newly created Timeslot(s) into one single location.

Use of pattern

Unit of work is currently used to handle bulk creation of timeslots by the Admin.

The following are the detailed steps undertaken during Unit of Work:

- At the start of the multi-step business transaction to bulk create Timeslot(s), the controller will check whether there is a unit of work in the session.
    - If there isn't already one, it will instantiate a new unit of work and place it within the session.
- For each new Timeslot created, the controller will register the newly created Timeslot into the unit of work via registerNew(Timeslot).
- Once the user wants to complete the transaction and upload the newly created objects onto the database, the controller will call the unit of work's commit method.
- Unit of work will first set the database connection's autocommit flag to false. This will allow us to essentially stage multiple queries in one bundle and only commit them when we call the database connection's own commit method.

- Then it will iterate through each newly created object in the newObjList and call the object's respective mapper insert method.
- Once all objects in the list have been iterated, it calls the database connection's commit method to execute all staged queries to the database.
- The unit of work then resets all its internal state, i.e. clears the lists, sets the database auto commit back to true and closes the database connection.

A detailed walkthrough of this process can be found in the sequence diagram below.

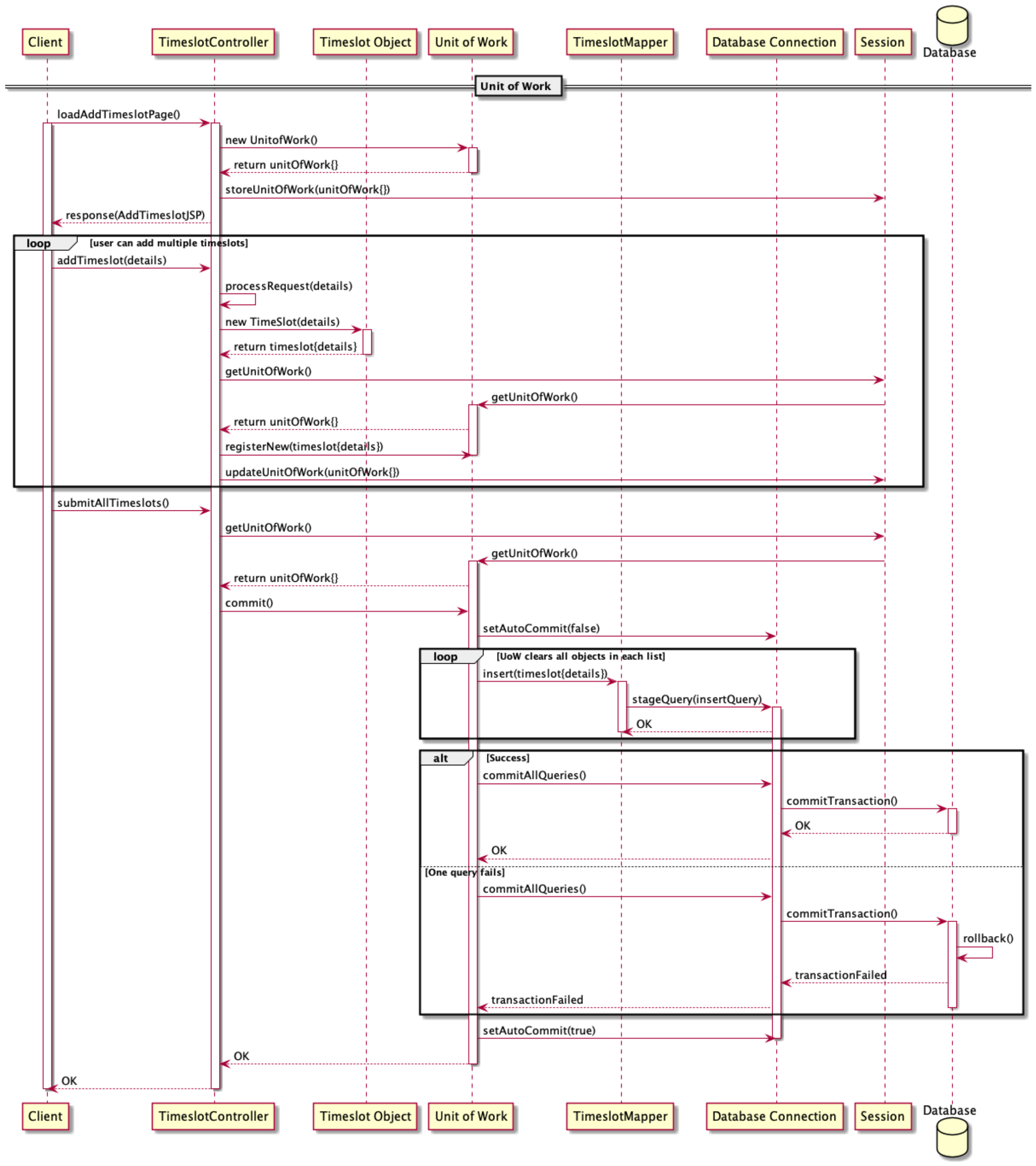


Figure 12: Sequence diagram for Unit of Work for Adding Timeslots

Identity Field

Implementation

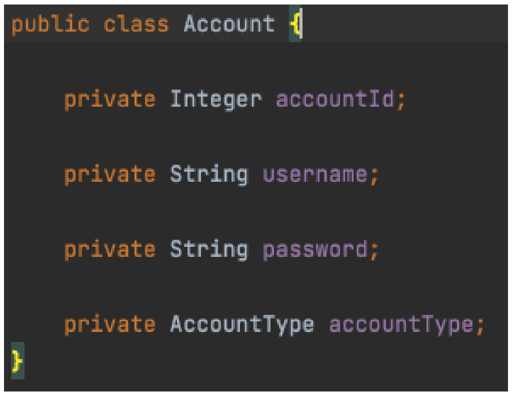
For Account and its child classes (VaccineRecipient, HealthcareProvider), they all share the same unique id. In Account it is called id but within the child classes they are called account_id. The id key here has attributes:

1. **Meaningless keys**: Serial sequentially generates integers starting from 1 and has no meaning with respect to the domain.
2. **Simple keys**: An id corresponds to a row in the database table
3. **Table-unique keys**: Id is unique in the scope of the table
4. **Auto-generated**: Auto-generated by the database

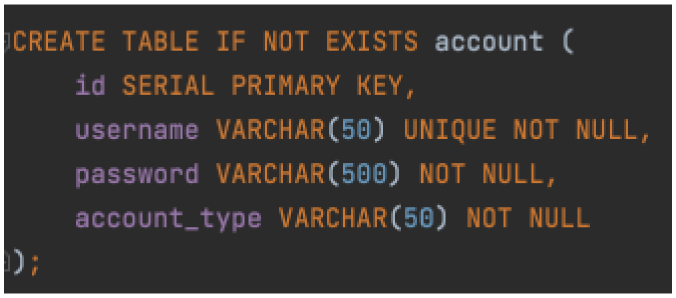
Use of Pattern

This pattern is used in every table in the database except for ‘vaccine_type’ (name is used as the unique identifier), ‘vaccine_question’ and ‘vaccine_certificate’ tables.

For the ‘account’ table (Figure 13), we can see its primary key ‘id’ that is reflected in the domain object Account as accountId (Figure 14).



```java
public class Account {

    private Integer accountId;

    private String username;

    private String password;

    private AccountType accountType;
}
```



```sql
CREATE TABLE IF NOT EXISTS account (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(500) NOT NULL,
    account_type VARCHAR(50) NOT NULL
);
```

Figure 13: Account Domain Object

Figure 14: Account Table

The sequence diagram below shows the use of the database-unique primary key of id for Account objects via the login process. As VaccineRecipient inherits from Account type, they also hold the primary key in its domain object as accountId (Figure 13). Once the VaccineRecipient has successfully logged into the website, the controller finds the account using the VaccineRecipient’s username via the Account Mapper. The mapper creates a new Account object, uses the primary key from the row and sets it as the id for the Account object before returning the object back to the controller.

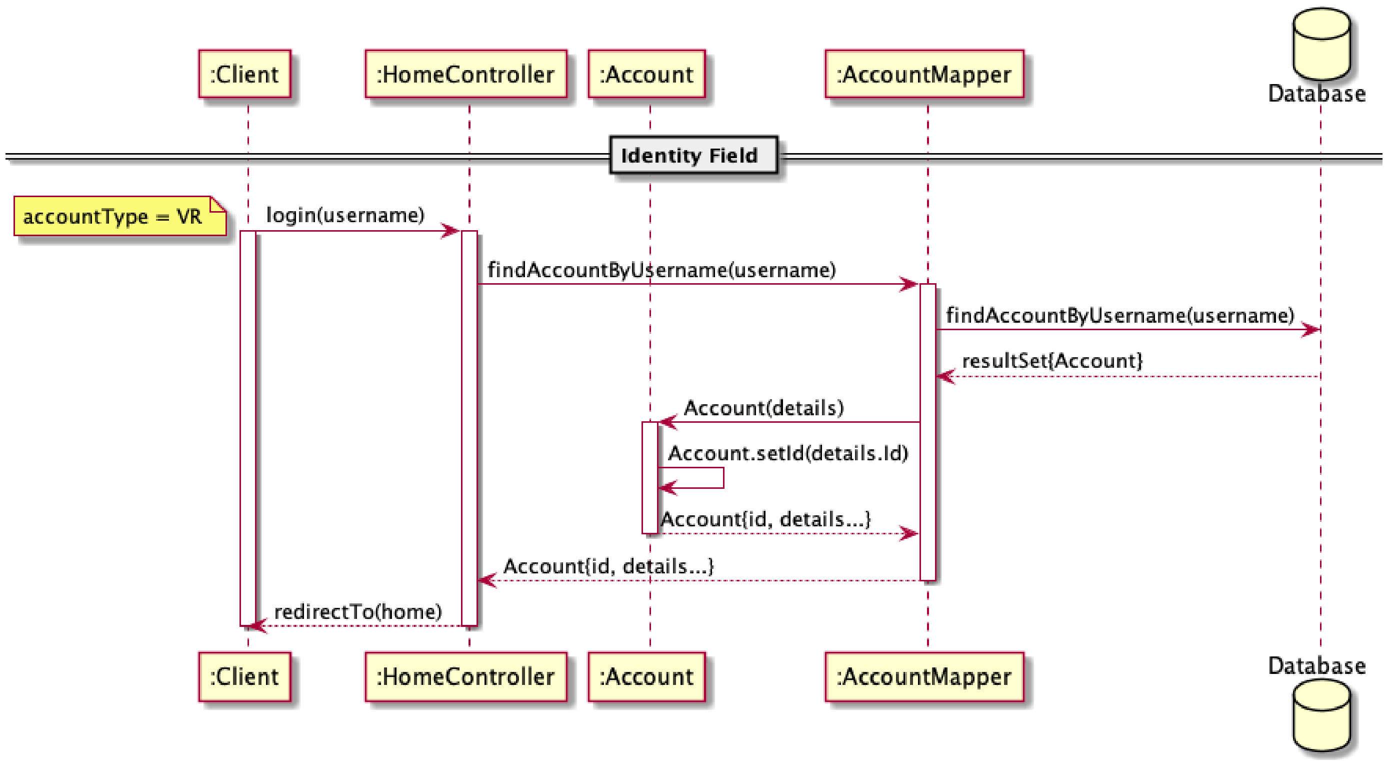


Figure 15: Sequence diagram for Identity Field

Foreign Key Mapping

Implementation

Foreign key mapping is used in the system whenever there is a link between two domain objects and there is a need to store the relationship within the database. The use of this pattern ensures that when loading these linked domain objects from their database rows, the system is able to maintain the relationship between the linked domain objects.

Use of Pattern

Foreign key mapping is used to maintain the link between Timeslot and HealthCareProvider. Each Timeslot must contain the HealthcareProvider that added/created it. In the Timeslot domain object, it stores the HealthcareProvider object as one of its variables. This link is preserved at the database level by the Timeslot having a foreign key field to store the HealthcareProvider's id.

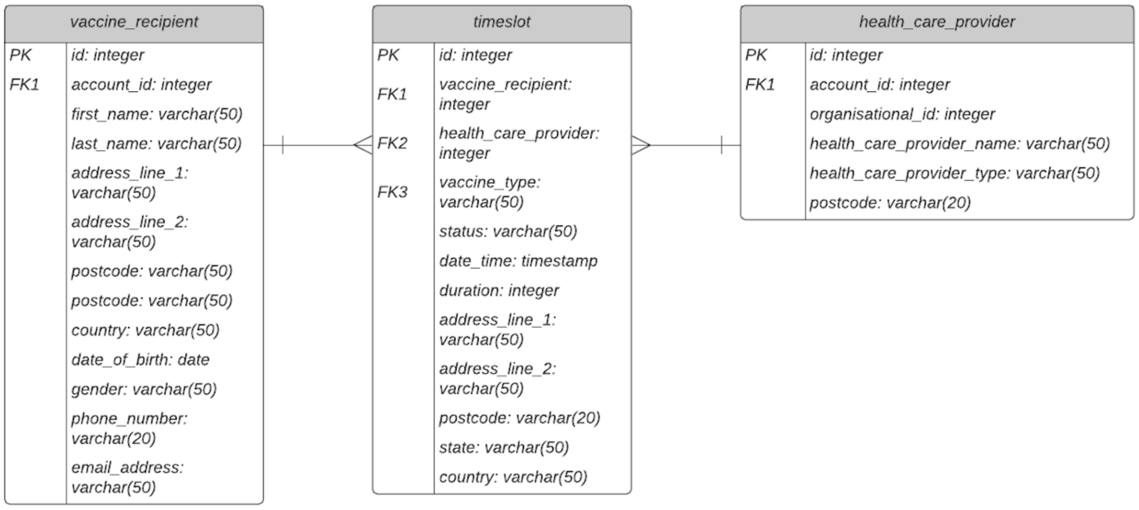
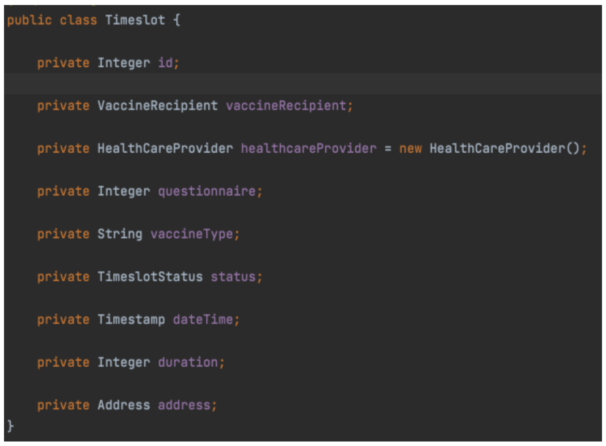
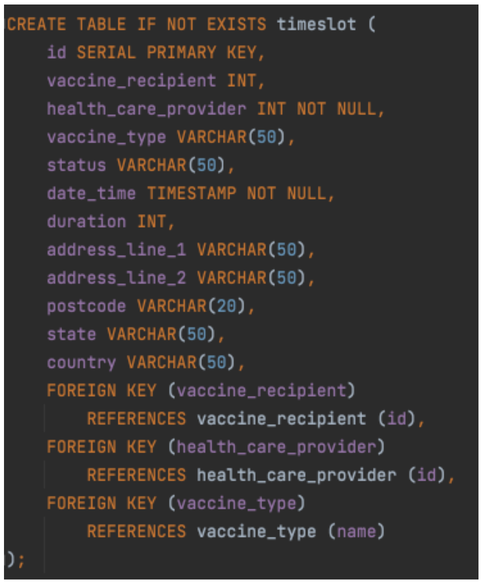


Figure 16: database representation for foreign key mapping



```
public class Timeslot {

    private Integer id;

    private VaccineRecipient vaccineRecipient;

    private HealthCareProvider healthcareProvider = new HealthCareProvider();

    private Integer questionnaire;

    private String vaccineType;

    private TimeslotStatus status;

    private Timestamp dateTime;

    private Integer duration;

    private Address address;
}
```



```
CREATE TABLE IF NOT EXISTS timeslot (
    id SERIAL PRIMARY KEY,
    vaccine_recipient INT,
    health_care_provider INT NOT NULL,
    vaccine_type VARCHAR(50),
    status VARCHAR(50),
    date_time TIMESTAMP NOT NULL,
    duration INT,
    address_line_1 VARCHAR(50),
    address_line_2 VARCHAR(50),
    postcode VARCHAR(20),
    state VARCHAR(50),
    country VARCHAR(50),
    FOREIGN KEY (vaccine_recipient)
        REFERENCES vaccine_recipient (id),
    FOREIGN KEY (health_care_provider)
        REFERENCES health_care_provider (id),
    FOREIGN KEY (vaccine_type)
        REFERENCES vaccine_type (name)
);
```

Figure 17: Timeslot Domain Object *Figure 18: Timeslot Table*

The mapping of foreign keys can be observed in the process of adding Timeslots to the system as shown in the sequence diagram below. The controller first creates a new HealthCareProvider object with the id associated with the HealthCareProvider who is currently adding the Timeslot. It is then placed into the Timeslot object before sending it to the mapper to be recorded on the database.

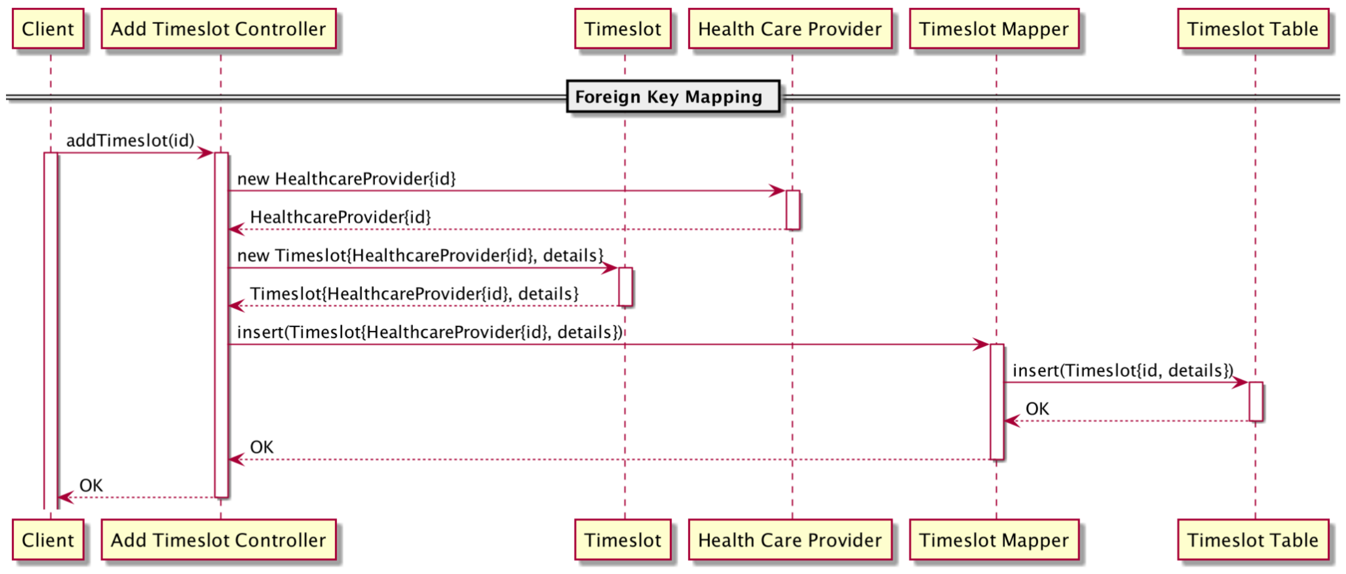

Client
Add Timeslot Controller
Timeslot
Health Care Provider
Timeslot Mapper
Timeslot Table
Foreign Key Mapping
addTimeslot(id)
new HealthcareProvider{id}
HealthcareProvider{id}
new Timeslot{HealthcareProvider{id}, details}
Timeslot{HealthcareProvider{id}, details}
insert(Timeslot{HealthcareProvider{id}, details})
insert(Timeslot{id, details})
OK
OK
OK

Figure 19: Sequence Diagram showcasing Foreign Key Mapping in the use case Adding Timeslots

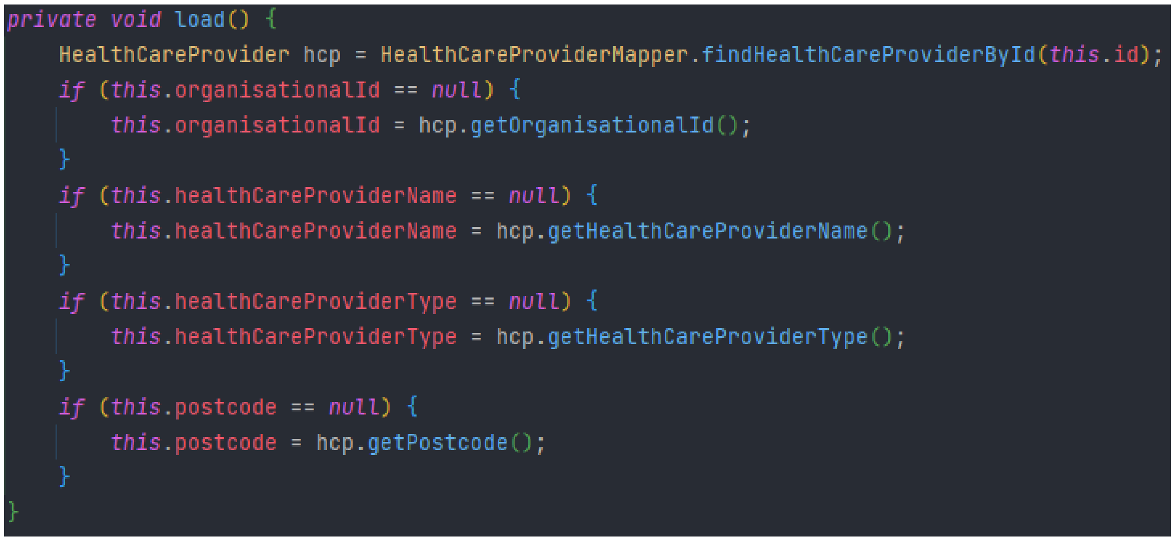
Lazy Load

Implementation

In the implementation, the 'ghost' lazy loading pattern was implemented. Through this pattern, all fields in a class are initialized to null upon creation of an object. When the get method of any field in the class is called, it checks if the field is null and if it is, all fields in the class are initialized at the same time, saving the number of hits to the database table.

In the system, when Timeslots are returned by finding all Timeslots based on a particular area (postcode), the associated HealthCareProvider is initialized as null (except for id). When a VaccineRecipient goes to view a specific timeslot, the associated HealthCareProvider's details will first be checked, and if any of the fields are null, the controller will go to the database and load all the information into the timeslot and render it to the user.

In Figure 20, the implementation of the ghost lazy loading pattern is shown through the load() method in the HealthCareProvider domain class.



```java
private void load() {
    HealthCareProvider hcp = HealthCareProviderMapper.findHealthCareProviderById(this.id);
    if (this.organisationalId == null) {
        this.organisationalId = hcp.getOrganisationalId();
    }
    if (this.healthCareProviderName == null) {
        this.healthCareProviderName = hcp.getHealthCareProviderName();
    }
    if (this.healthCareProviderType == null) {
        this.healthCareProviderType = hcp.getHealthCareProviderType();
    }
    if (this.postcode == null) {
        this.postcode = hcp.getPostcode();
    }
}
```

Figure 20. load() method in the HealthCareProvider domain class

Design Rationale

Searching for Timeslots in bulk can be a data intensive task. The system could potentially have millions of timeslot records in the database. Since the associated HealthCareProvider to a timeslot is stored in a different database table, it would be quite wasteful to do millions of queries to load all of the associated HealthCareProviders, when the user would only want to see the information of the HealthCareProvider that is associated with the particular Timeslot they want to view (and potentially book). Not only does lazy loading the HealthCareProvider save a hit onto the 'health_care_provider' table for every Timeslot that is loaded, it also saves having to store the information for all of the HealthCareProviders in memory as well, only loading them when the user actually wants to view that information. By delaying the loading of the object until the point where the user actually needs it, it avoids objects having information they don't need, increases fetching speeds, reduces processing time and increases the efficiency of the system. Lazy loading will be crucial as the system scales, as with higher loads and many users at once, the system will need to effectively and efficiently handle the way it retrieves and renders timeslots to the users.

The use of ghost lazy loading pattern seemed appropriate for the scenario being handled in this use case. Instead of using lazy initialization where the database connection would have to be called to get each field in the class individually, it makes more sense to get them all at once as the screen displays all the details of the HealthCareProvider that owns the chosen Timeslot.

Use of pattern

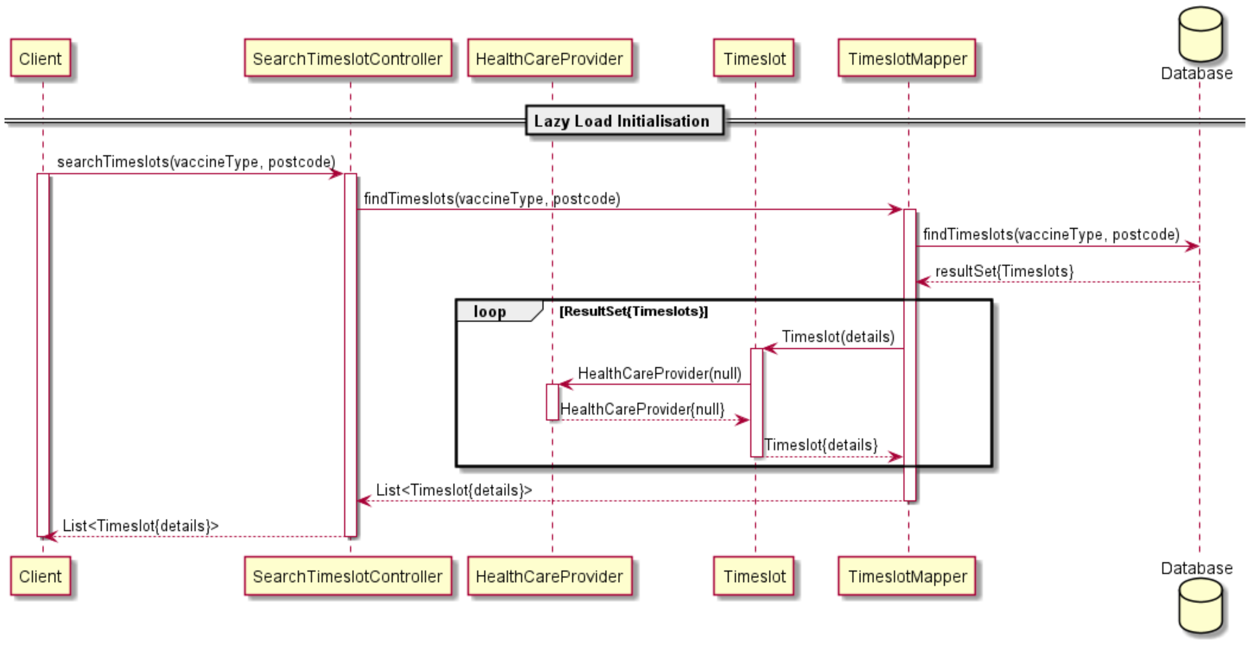


Figure 21: Sequence Diagram for Lazy Load

Lazy load is used when a user searches for a timeslot based on postcode.

1. A user searches for Timeslots by area (postcode).
2. All Timeslots based on the criteria are returned (with the HealthCareProvider details except for id initialised to null)

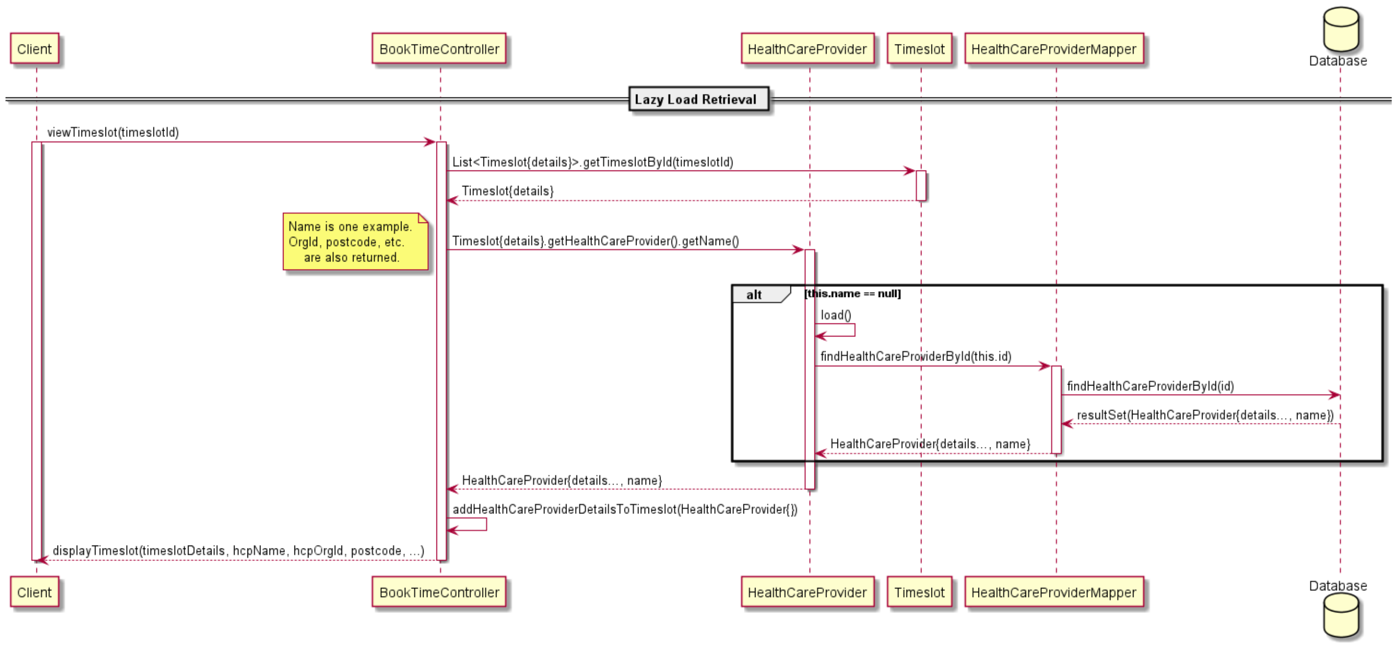

Client
BookTimeController
HealthCareProvider
Timeslot
HealthCareProviderMapper
Database
Lazy Load Retrieval
viewTimeslot(timeslotId)
List<Timeslot{details}>.getTimeslotById(timeslotId)
Timeslot{details}
Name is one example. OrgId, postcode, etc. are also returned.
Timeslot{details}.getHealthCareProvider().getName()
alt
[this.name == null]
load()
findHealthCareProviderById(this.id)
findHealthCareProviderById(id)
resultSet(HealthCareProvider{details..., name})
HealthCareProvider{details..., name}
HealthCareProvider{details..., name}
addHealthCareProviderDetailsToTimeslot(HealthCareProvider{})
displayTimeslot(timeslotDetails, hcpName, hcpOrgId, postcode, ...)

Figure 22: Sequence Diagram showcasing Lazy Load usage when retrieving Timeslots

3. When a user clicks on a particular Timeslot, the controller will fetch the details for the specific timeslot, as well as the HealthCareProvider within it.
4. If any of the fields in the HealthCareProvider domain object are null, the controller will go and retrieve the HealthCareProvider using the previously initialised id.
5. The controller will then display the specific Timeslot including all of the HealthCareProvider's details to the user.

Authentication and Authorization

Implementation

For the system's Authentication Enforcer and Authorization Enforcer, it was identified that the best strategy was to use a third-party provider strategy. The third-party Authentication and Authorization Enforcers of choice is Spring Security. This ensures that the Authentication and Authorization Enforcers in the system are proven and reliable to provide protection and confidentiality of the user's private and sensitive data.

Use of pattern

In the system, there are 3 different account types [Admin, HealthcareProvider and VaccineRecipient] each having different levels of access control and privileges. Upon landing on the website, the user is first prompted to login to their account. This process where Authentication Enforcer handles the login process can be seen in the sequence diagram below.

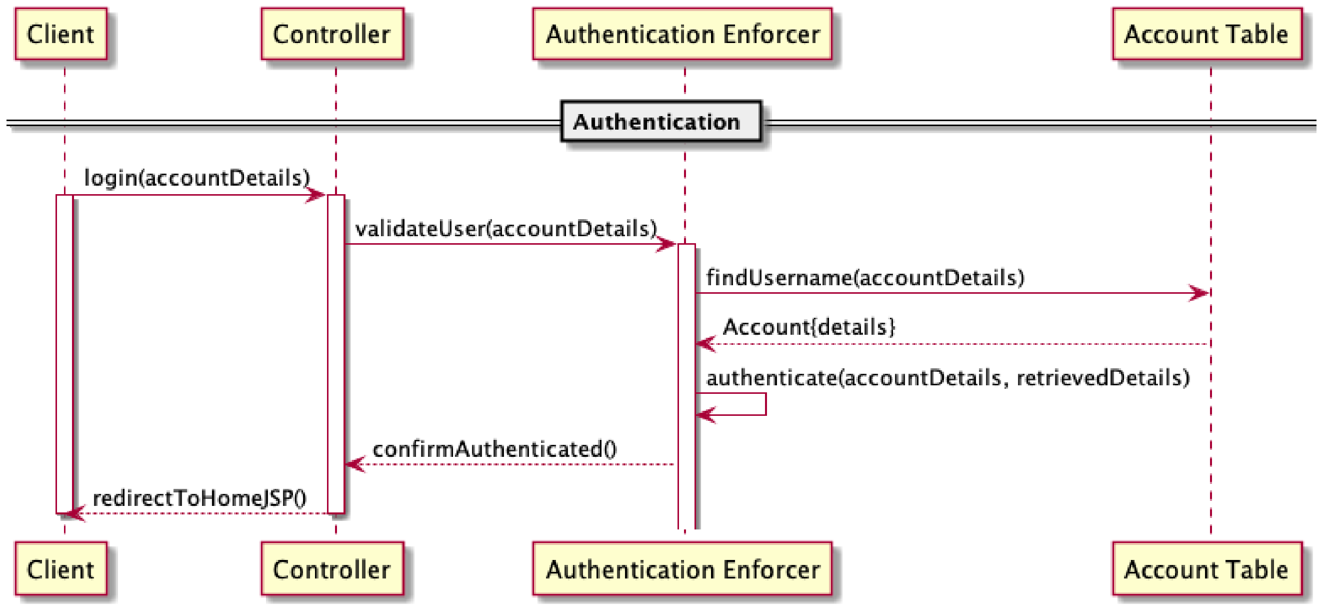
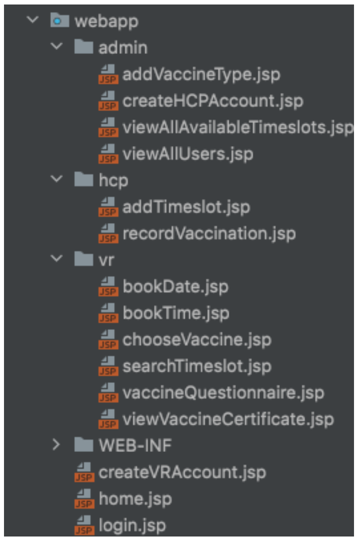


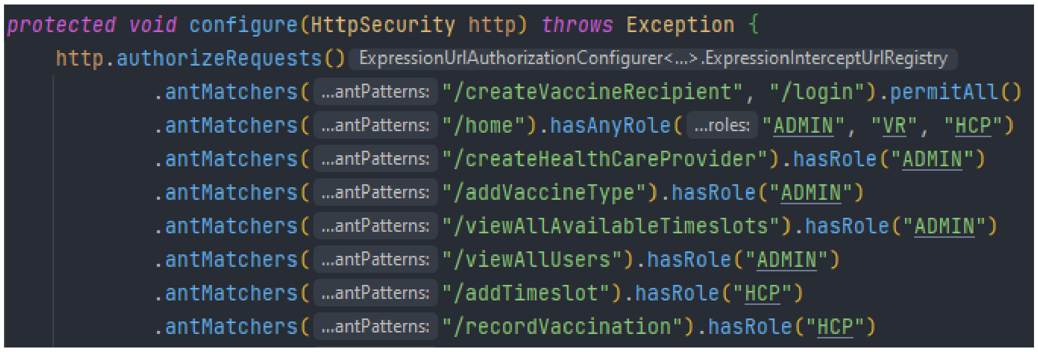
Figure 23: Sequence Diagram for Authentication



```
webapp
  admin
    addVaccineType.jsp
    createHCPAccount.jsp
    viewAllAvailableTimeslots.jsp
    viewAllUsers.jsp
  hcp
    addTimeslot.jsp
    recordVaccination.jsp
  vr
    bookDate.jsp
    bookTime.jsp
    chooseVaccine.jsp
    searchTimeslot.jsp
    vaccineQuestionnaire.jsp
    viewVaccineCertificate.jsp
  WEB-INF
  createVRAccount.jsp
  home.jsp
  login.jsp
```

Figure 24: JSP File structure

JSP pages have been divided into 3 subfolders to ease protection of routes. Each subfolder has views that are only accessible by accounts of the respective types, i.e. admin views are only accessible by Admin accounts. “createVRAccount.jsp” and “login.jsp” are views accessible by everyone (even users who aren’t logged in) with the exception of “home.jsp” where any user who is logged in can access that page. On home.jsp, different account types will have a different view, i.e. VaccineRecipients have a homepage with buttons that link to pages that they have permission to access. The access control is all handled by the Authorisation Enforcer as seen by the sequence diagram below.



```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/createVaccineRecipient", "/login").permitAll()
            .antMatchers( ...antPatterns: "/home").hasAnyRole( ...roles: "ADMIN", "VR", "HCP")
            .antMatchers( ...antPatterns: "/createHealthCareProvider").hasRole("ADMIN")
            .antMatchers( ...antPatterns: "/addVaccineType").hasRole("ADMIN")
            .antMatchers( ...antPatterns: "/viewAllAvailableTimeslots").hasRole("ADMIN")
            .antMatchers( ...antPatterns: "/viewAllUsers").hasRole("ADMIN")
            .antMatchers( ...antPatterns: "/addTimeslot").hasRole("HCP")
            .antMatchers( ...antPatterns: "/recordVaccination").hasRole("HCP")
```

Figure 25: Configuration of antMatchers to restrict routes by user roles

Similarly, the controller routes have been individually specified in the authorization config using antMatchers. Each route is accessible by one or more user roles ('ADMIN', 'VR', or 'HCP'). Hence, only the specified user role(s) are allowed to access the controller of the given route. Figure 25, shows an example of the antMatchers and how they restrict routes in the system.

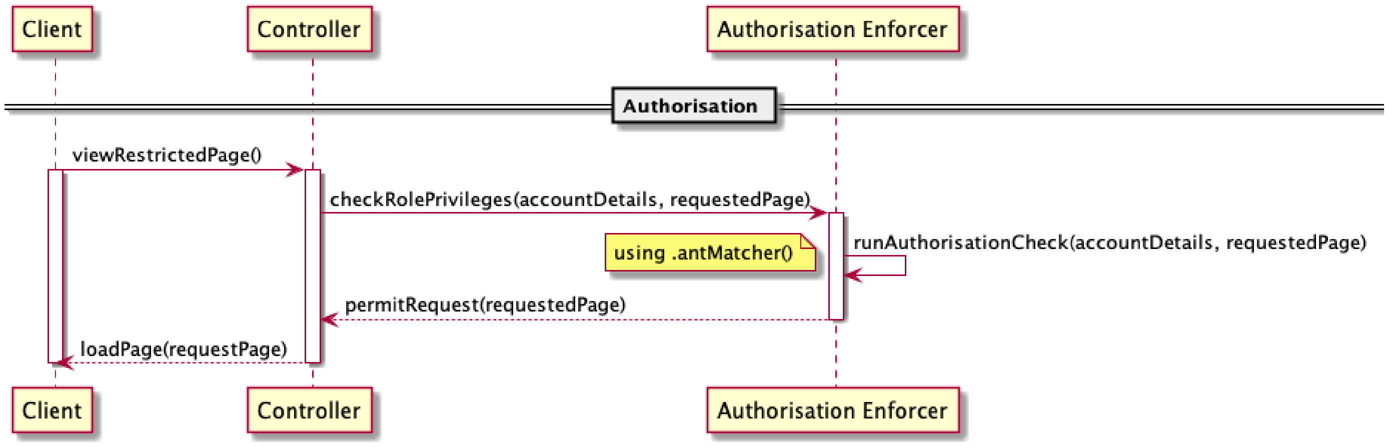


Figure 26: Sequence diagram for Authorisation