Product Requirements Document

Part 2 Specification

Team Member's name	Student ID	Unimelb Usernames	Github Username	Emails
Zhiming Deng	981607	zhimingd	ZhimingDeng5	zhimingd@student.unimelb.edu.au
Gaoli Yi	1048049	gaoliy	GAOLIY	gaoliy@student.unimelb.edu.au
Lingge Guo	1211499	linggeg1	linggeg1	linggeg1@student.unimelb.edu.au
Chutong Wang	1185305	CHUTONGW	CHUTONGW	CHUTONGW@student.unimelb.edu.au

Release tag: SWEN90007_2021_Part2_<Flying Tiger>

Revision History

Date	Version	Description	Author
14/08/2021	02.00-D01	Initial draft	zhiming
15/08/2021	02.00-D02	Review and the first version of the document	chutong
20/08/2021	02.00-D03	Update the use case list	gaoli
30/8/2021	02.00-D04	Add the class diagram	zhiming
15/09/2021	02.00-D05	Update the class diagram	lingge
17/09/2021	02.00-D06	Add descriptions of the patterns used	lingge
19/09/2021	02.00-D07	Update descriptions of the patterns used	chutong
21/09/2021	02.00-D08	Add design rationale for patterns	gaoli
23/09/2021	02.00-D09	Update design rationale for patterns	zhiming
24/09/2021	02.00-D10	Add pattern implementation	lingge
25/09/2021	02.00-D11	Report formatting and final review	chutong
26/09/2021	02.00	The final version of the document	gaoli

Contents

Introduction

1.1 Proposal

This document specifies use cases, actors to be implemented, and the system's domain model of the COVID-19 Vaccine Booking and Management System.

1.2 Target Users

This document is mainly intended for vaccine recipients, health care providers and administrators of the system.

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	Description
COVID-19	Coronavirus disease 2019, which is an infectious disease caused by Severe Acute Respiratory Syndrome coronavirus type 2 (abbreviation SARS-CoV-2).

Actors

Actor	Description
Administrator	The person who manages the COVID-19 Vaccine Booking and Management System.
Vaccine recipients	People who can make a book on vaccination.
Health care providers	People who provide vaccines, add injection timeslots and confirm the vaccination of a Vaccine recipient.

Use Cases

3.1 Use case 1: Sign in/ out

Actors: Administrators, Vaccine Recipients, Health Care Providers

Description: Each user signs in to the Covid-19 vaccine booking and management system including Administrators, Vaccine Recipients, and Health Care Providers.

Basic Flow:

- One administrator, or vaccine recipient, or health care provider user opens the web, which is this booking and management system.
- He signs in by selecting his role and inputting his userID and password.

3.2 Use case 2: Create accounts

Actors: Administrators

Description: Administrators create recipients accounts with biographical information, or create provider accounts.

Basic Flow:

- Administrators get to know recipients' biographical information, including name and date of birth. Or administrators get to know health care providers' information, including name, postcode and type of provider.
- They create a relative account by inputting relative information on the web.

3.3 Use case 3: Add vaccine types

Actors: Administrators

Description: Administrators add vaccine types which will be injected into incidents afterwards.

Basic Flow:

- Administrators get to know what vaccine types will be added.
- They click vaccine types and choose to add a new vaccine type, for example, AstraZeneca.

3.4 Use case 4: View all users names and all time-slots

Actors: Administrators

Description: Administrators view all users' names that have been created before and all time-slots available to monitor time-slots vacancy.

Basic Flow:

- Administrators click viewing all the users' names or all the time-slots.
- They can view all the users' names or all the time-slots available on the web page.
- They click filters and choose the requirements, like the type of vaccine received for filtering users, and then they can view the users' names or the time-slots.

3.5 Use case 5: View booking detail

Actors: Vaccine Recipients

Description: Vaccine recipients can view their booking details, including name, date, time, health care provider name and booking status.

Basic Flow:

- Vaccine recipients sign in the system.
- They can view all booking details on the page.

3.6 Use case 6: Book vaccines online

Actors: Vaccine Recipients

Description: Vaccine recipients book vaccine injection after viewing all available time-slots at all Health Care Providers and then choosing a timeslot.

Basic Flow:

- Vaccine recipients select health care providers or dates and then search for all available timeslots.
- They click a timeslot at one health care provider and choose the type of vaccine they want to have.
- They have to answer a questionnaire.
- They book the vaccine injection successfully.

3.7 Use case 7: Answer a short questionnaire

Actors: Vaccine Recipients

Description: Vaccine recipients answer a short questionnaire before submitting one booking and submit it.

Basic Flow:

- Vaccine recipients view a short questionnaire before booking on the web.
- They answer the questionnaire.
- They submit it.

3.8 Use case 8: View proof of vaccination

Actors: Vaccine Recipients

Description: Vaccine recipients view proof of vaccination after injection and get one.

Basic Flow:

- Vaccine recipients view proof of vaccination after injection on the system.
- They can have one proof of vaccination and download it.

3.9 Use case 9: Manage questionnaire

Actors: Health Care Providers

Description: Health Care Providers add questionnaires for a new type of vaccine and edit questionnaires.

Basic Flow:

- Health Care Providers click the add questionnaires button and choose a type of vaccine.
- They can add questions and submit them.
- Health Care Providers click the edit questionnaires button and view all questionnaires.
- They can edit questions and delete questionnaires.

3.10 Use case 10: List dates and times of bookings

Actors: Health Care Providers

Description: Health Care Providers view a list of dates and times(time-slots) of bookings that book themselves.

Basic Flow:

- Health Care Providers click viewing time-slots of bookings.
- They can view a list of dates and times of bookings which book themselves.

3.11 Use case 11: Add any number of time-slots

Actors: Health Care Providers

Description: Health Care Providers add any number or themselves' available time-slots to the system for recipients to book.

Basic Flow:

- Health Care Providers click adding time-slots
- They can choose the date and time of time-slots.
- They add the chosen time slots to the system.

3.12 Use case 12: Manage vaccine

Actors: Health Care Providers

Description: Health Care Providers add the number of vaccines to the system for recipients to book.

Basic Flow:

- Health Care Providers click the edit button on my timeslot page
- They edit the number of vaccines for booking.

3.13 Use case 13: Manage booking list

Actors: Health Care Providers

Description: Health Care Providers view all booking detail and change recipients' status.

Basic Flow:

- Health Care Providers click booking list.
- They can choose the status for recipients, including in-progress and completed.

Domain Model

The project aims to implement a covid-19 vaccine management system. The system involves three types of users: recipients, health care providers, and system administrators. The core functions of the system include managing the time slots and quantity of vaccines, as well as the process of recipients booking vaccines. As a Web-based enterprise application, it has complex domain logic and concurrency issues. In this part, we applied different design patterns to realize its domain logic.

The following figure depicts our domain model, which describes a series of responsibilities an object assumes in the system and the relationship between objects. We created 8 main classes based on object-oriented design to handle different domain logic.

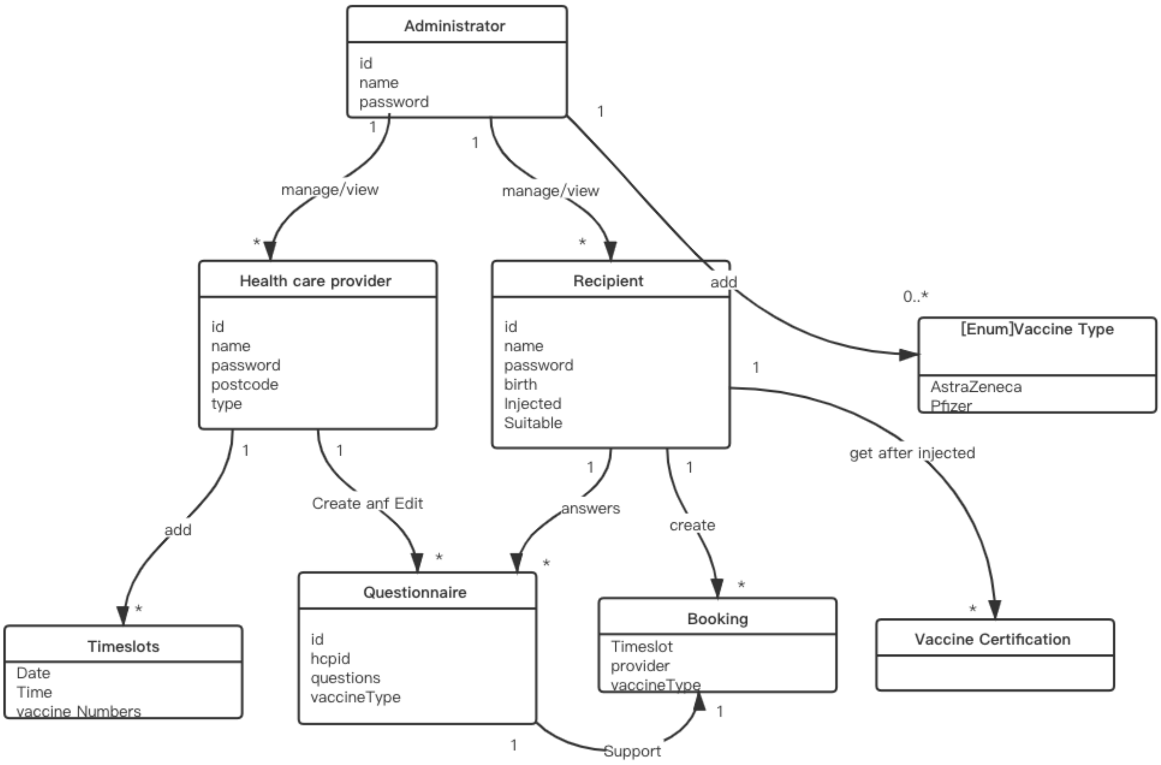
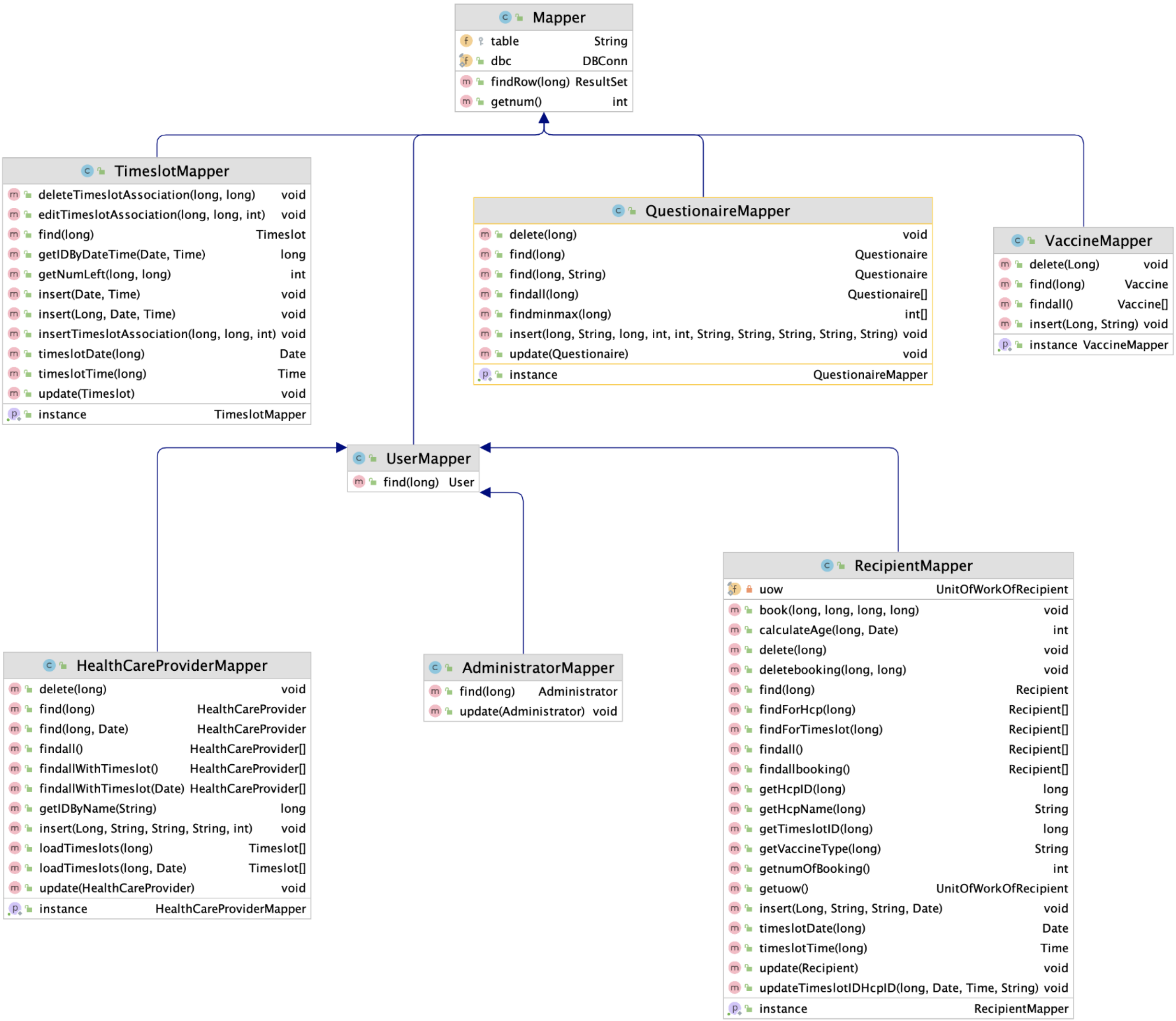
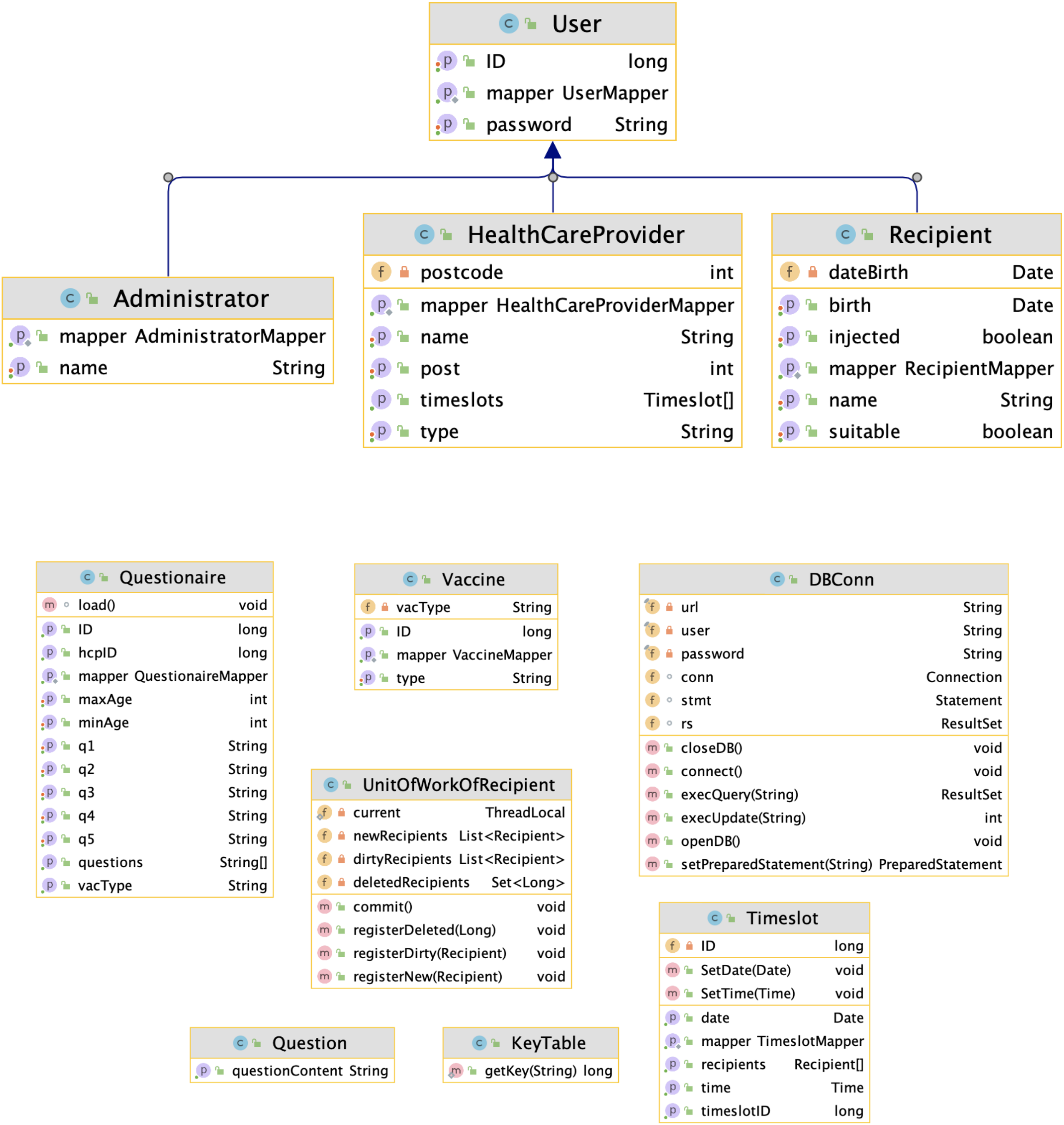


Figure: domain model

Class Diagram


Mapper
table String
dbc DBConn
findRow(long) ResultSet
getnum() int
TimeslotMapper
deleteTimeslotAssociation(long, long) void
editTimeslotAssociation(long, long, int) void
find(long) Timeslot
getIDByDateTime(Date, Time) long
getNumLeft(long, long) int
insert(Date, Time) void
insert(Long, Date, Time) void
insertTimeslotAssociation(long, long, int) void
timeslotDate(long) Date
timeslotTime(long) Time
update(Timeslot) void
instance TimeslotMapper
QuestionaireMapper
delete(long) void
find(long) Questionaire
find(long, String) Questionaire
findall(long) Questionaire[]
findminmax(long) int[]
insert(long, String, long, int, int, String, String, String, String, String) void
update(Questionaire) void
instance QuestionaireMapper
VaccineMapper
delete(Long) void
find(long) Vaccine
findall() Vaccine[]
insert(Long, String) void
instance VaccineMapper
UserMapper
find(long) User
HealthCareProviderMapper
delete(long) void
find(long) HealthCareProvider
find(long, Date) HealthCareProvider
findall() HealthCareProvider[]
findallWithTimeslot() HealthCareProvider[]
findallWithTimeslot(Date) HealthCareProvider[]
getIDByName(String) long
insert(Long, String, String, String, int) void
loadTimeslots(long) Timeslot[]
loadTimeslots(long, Date) Timeslot[]
update(HealthCareProvider) void
instance HealthCareProviderMapper
AdministratorMapper
find(long) Administrator
update(Administrator) void
RecipientMapper
uow UnitOfWorkOfRecipient
book(long, long, long, long) void
calculateAge(long, Date) int
delete(long) void
deletebooking(long, long) void
find(long) Recipient
findForHcp(long) Recipient[]
findForTimeslot(long) Recipient[]
findall() Recipient[]
findallbooking() Recipient[]
getHcpID(long) long
getHcpName(long) String
getTimeslotID(long) long
getVaccineType(long) String
getnumOfBooking() int
getuow() UnitOfWorkOfRecipient
insert(Long, String, String, Date) void
timeslotDate(long) Date
timeslotTime(long) Time
update(Recipient) void
updateTimeslotIDHcpID(long, Date, Time, String) void
instance RecipientMapper



Patterns

6.1 Domain model

In the domain layer, we used the domain model as the design pattern. It is highly correlated with object-oriented design and has good scalability. We used the second implementation method, which is to store the information required in a database, and then only create the objects as required. As we talked above, we divide the domain logic into 8 separate parts, each part is a class that contains related data and behaviours.

In the following sequence diagram, you will see that these classes interact to complete the overall process of a vaccine from booking to completion of the injection, including the administrator adding the vaccine category, the health care provider adding the time slots and the number of vaccines, and finally booking and vaccinating by the vaccinator. The whole process includes complex class interactions, and encapsulating behaviour in different objects reduces redundant code and reduces the coupling between different objects.

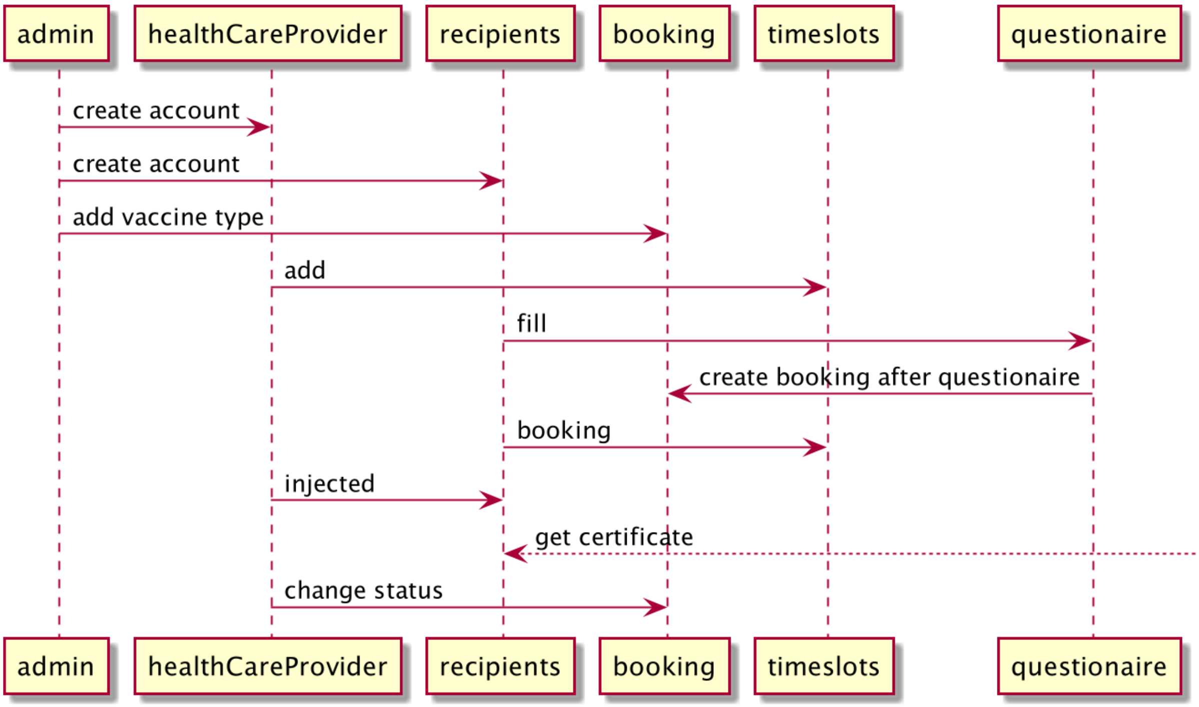


Figure: sequence diagram of the domain model

6.2 Data mapper

It is a medium layer between the domain layer and database, keeping them independent of each other. We implemented this pattern by creating a data mapper class for each domain object class and setting up the mapping between domain object and database. Each time a domain object loads data or writes back data into the database, the intermediate mapping class takes result sets from database queries and creates domain objects representing that data, as well as doing the reverse.

For example, the following diagram shows how an administrator user logs into the system and the data mapper works. Here, the login servlet uses *AdministratorMapper* to find this row in the database table and return an *Administrator* object.

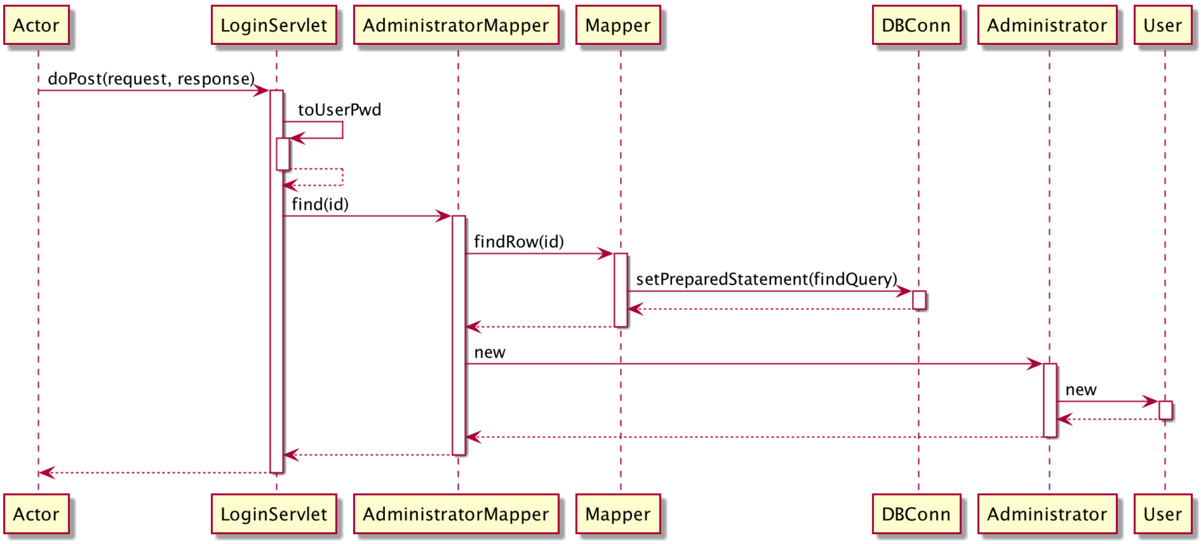
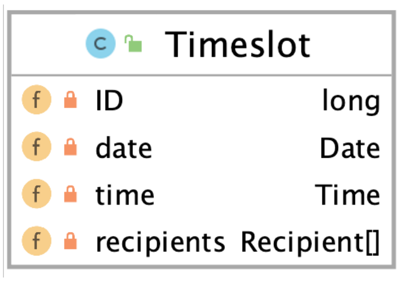


Figure: sequence diagram of login as an administrator

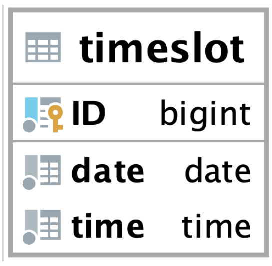
6.3 Identity field

We aim to use this pattern to maintain object identity between an in-memory object and a database row, which means the object explicitly stores the primary key to the corresponding row for any object that corresponds to a row. Here we put a timeslot class and corresponding table as an example.



Timeslot	
ID	long
date	Date
time	Time
recipients	Recipient[]

Figure: class diagram of Timeslot



timeslot	
ID	bigint
date	date
time	time

Figure: table diagram of timeslot

Then, we use a key table to generate a new key, and for a table-unique key, the key table has one row per table, where each row records the name of the table and the relevant maximum key currently in this table in column *table* and *nextID*.

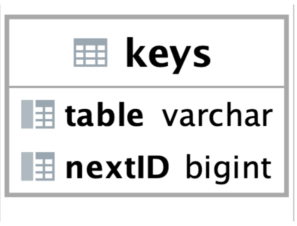


Figure: Key table definition

Every time, when we need to create a new user, for example, health care provider, we will get the new hcpID(health care provider ID), which is the current *nextID* plus 1 in this key table where column *table* equals the *health care provider*, and we will update the key table in our database with this new hcpID in the column *nextID*. Then, we could insert a new healthcare provider user with this new hcpID in the table *healthcare provider*. In this case, the Sequence diagram is as follows:

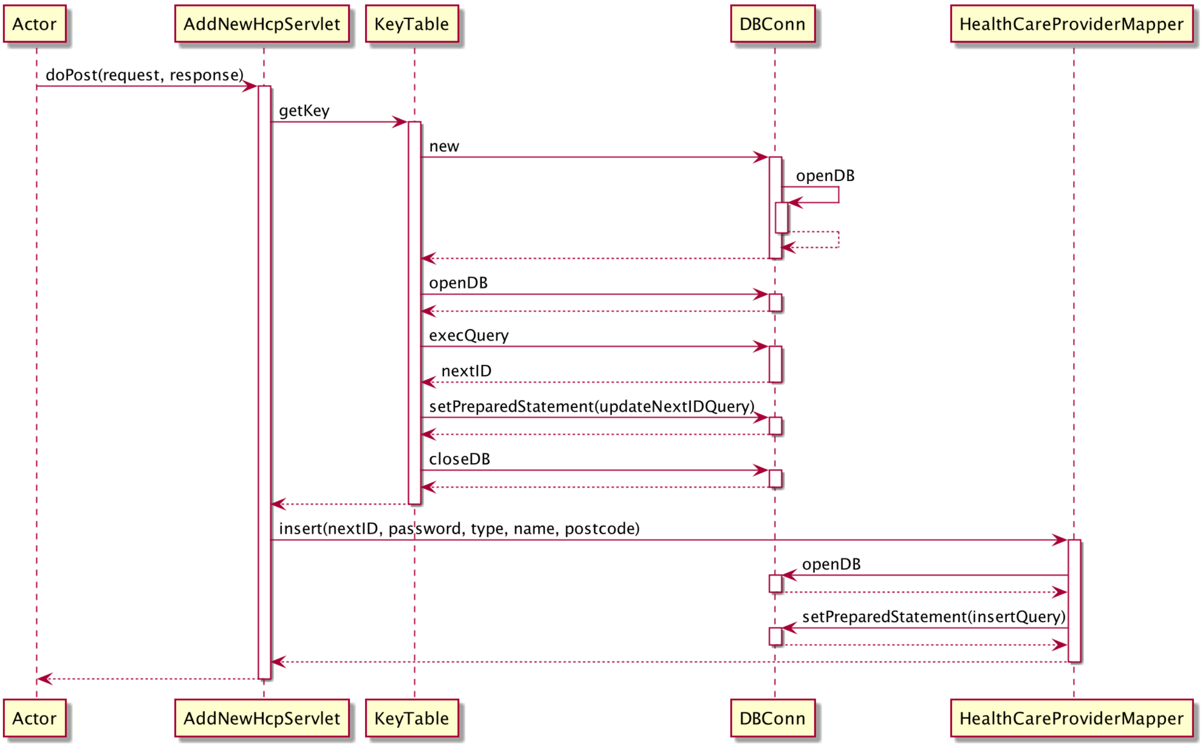


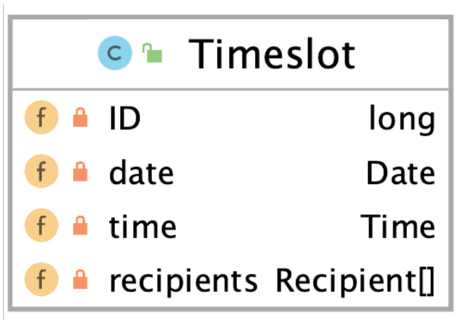
Figure: sequence diagram of creating a new health care provider

In doPost(request, response), it will get the *name, password, type, postcode* in the request for creating a new health care provider.

6.4 Foreign key mapping

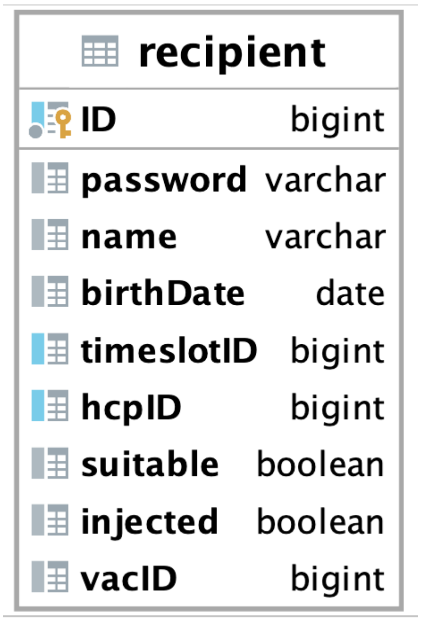
A foreign key is used when one object refers to another and then in the database schema, the association should be mapped using this. More complicatedly, if object *O* refers to a collection of objects*(O1,...On)*, then the rows corresponding to objects *O1,...On* should contain a foreign key field that refers to the row corresponding to object *O*. In our project, we use foreign key mapping in many situations, and we put timeslot and recipient as an example here.

As we know, each recipient could choose one timeslot to book for the vaccine, but on each timeslot, there may be many recipients booking. To implement, we put a list of recipients objects in class *Timeslot.* Then, in table *recipient,* each row should be containing a foreign key which is timeslotID for mapping such association after booking timeslot.



Timeslot	
ID	long
date	Date
time	Time
recipients	Recipient[]

Figure: class diagram of class Timeslot



recipient	
ID	bigint
password	varchar
name	varchar
birthDate	date
timeslotID	bigint
hcpID	bigint
suitable	boolean
injected	boolean
vacID	bigint

Figure: table diagram for the recipient

We can see there is one column called *timeslotID* as a foreign key in table *recipient,* it's mapping such association here. When the recipient books one timeslot successfully, AnswerJudgeServlet will use RecipientMapper to update the database table *recipient* and put *timeslotID* in the column *timeslotID* of this row. A sequence diagram is as follows:

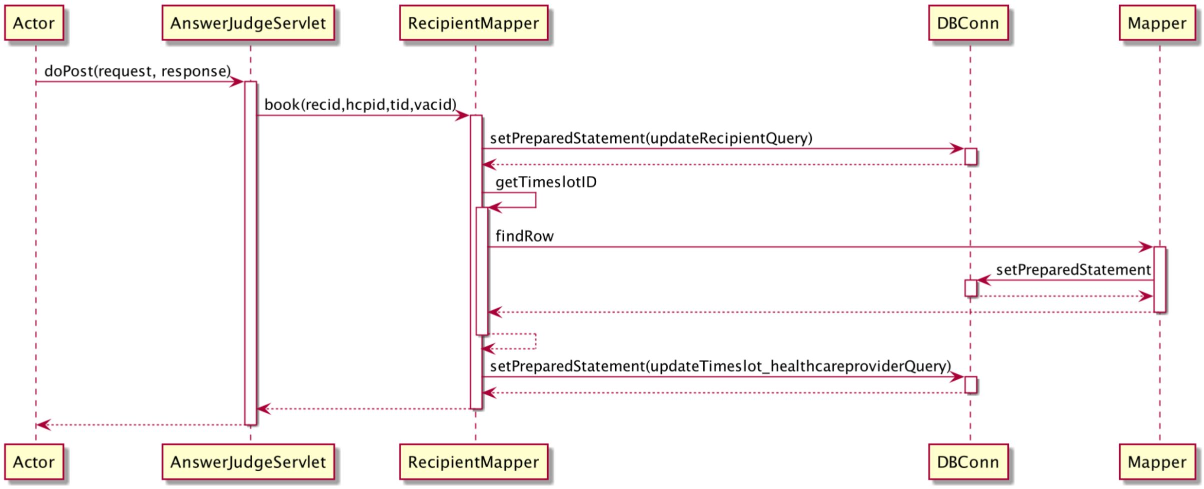

Actor
AnswerJudgeServlet
RecipientMapper
DBConn
Mapper
doPost(request, response)
book(recid,hcpid,tid,vacid)
setPreparedStatement(updateRecipientQuery)
getTimeslotID
findRow
setPreparedStatement
setPreparedStatement(updateTimeslot_healthcareproviderQuery)

Figure: Sequence diagram for recipient booking one timeslot given by one hcp

In doPost(request, response), it will get *recid,hcpid,tid,vacid* and answers of five questions in questionnaire *q1, q2, q3, q4, q5* in the actor's request for booking timeslot.

6.5 Association table mapping

The association table mapping pattern creates a new table that contains pairs of foreign keys, in which a pair represents the individual relationships to handle many-to-many associations. In our project, one health care provider can add any number of timeslots, while at one timeslot, there could be many health care providers giving health care. This is a kind of many-to-many association. Therefore, we need to add a new table *timeslot_healthcareprovider* for recording such individual relationships between *timeslot* and *health care provider*, which contains pairs of foreign keys of *timeslot* and *health care provider.*

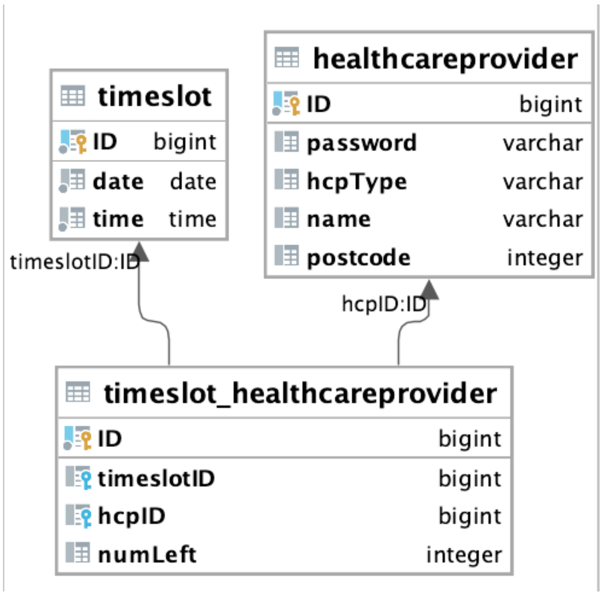


Figure: table diagram of timeslot_healthcareprovider

When one health care provider wants to add one new timeslot, let's say, a timeslot that no other providers create before. Then, other than one row inserted in the table *timeslot*, there's one row inserted in the association table between *timeslot* and *health care provider.* A sequence diagram is as follows:

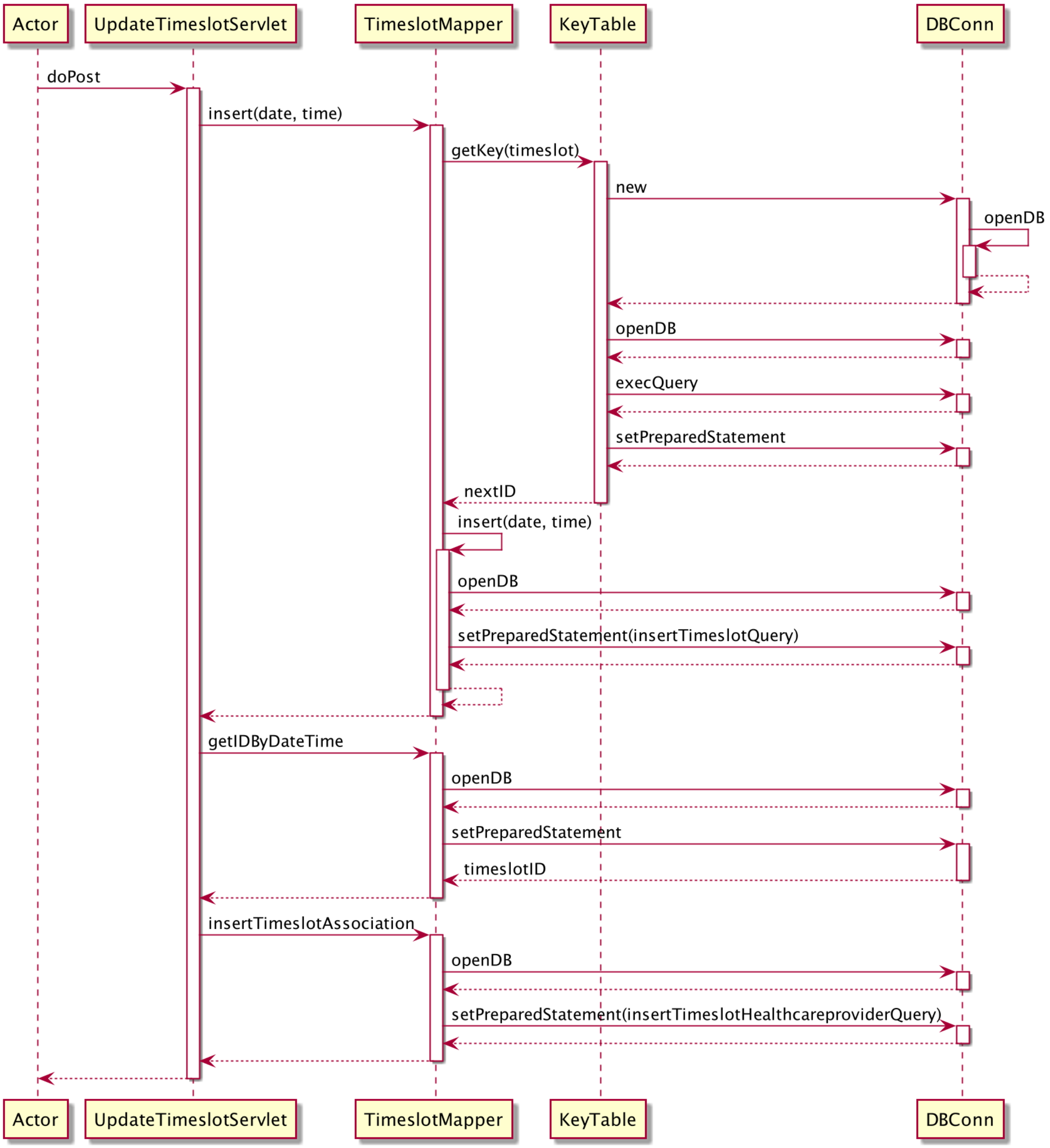


Figure: sequence diagram of the health care provider's adding a new timeslot

6.6 Embedded value

There are small classes/objects for recording information that does not make sense to represent in a database. In this project, class Question is used for recording each question content, which is a small class. Also, only when the owning object (*Questionaire*) is loaded, the corresponding embedded values(*question*) are loaded as well. Therefore, in the database, we only store each question content in each column in the table *questionnaire* instead of generating a new table *question.*

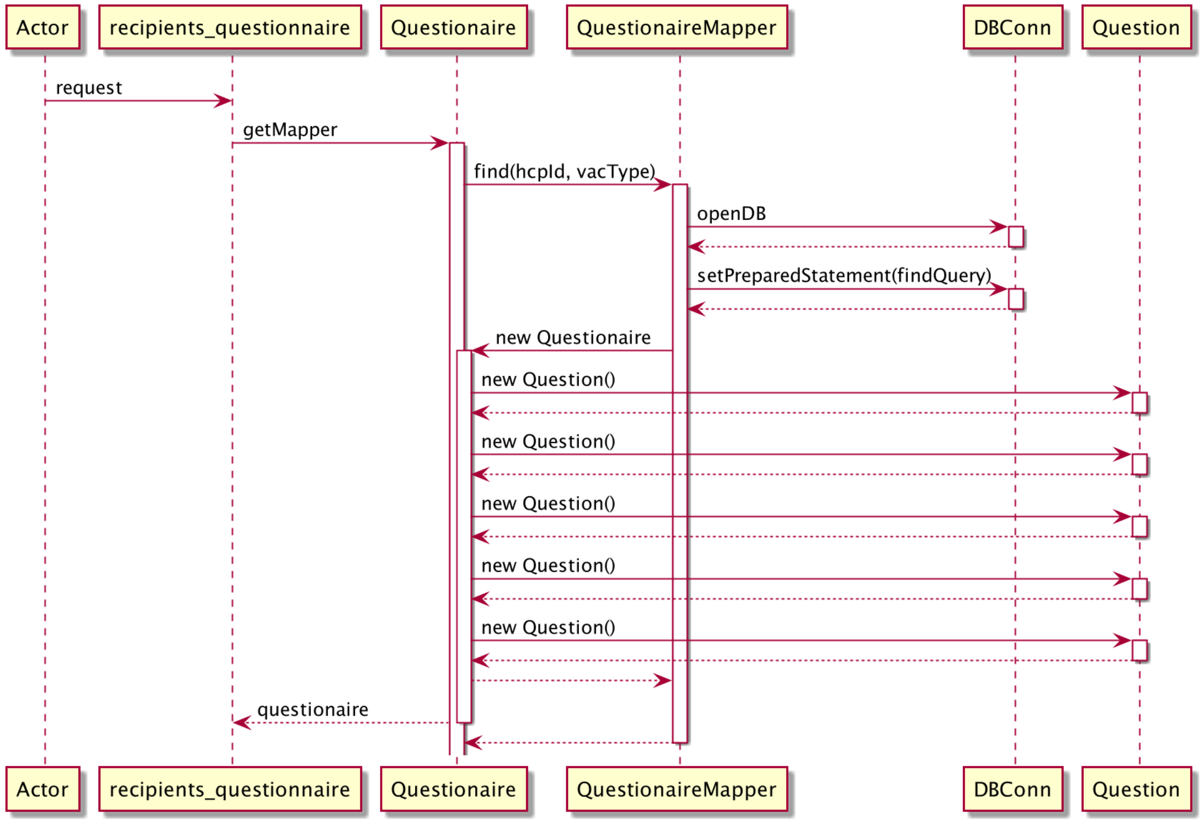


Figure: Sequence diagram of getting questionnaire based on the health care provider and vaccine type that recipient book.

6.7 Concrete table inheritance pattern

We apply specific table inheritance patterns to user objects. There is an abstract user class and three subclasses representing recipients, health care providers and administrators. In this project, we created three tables for three concrete classes. This pattern simplifies the design of tables and does not require joins or multiple queries when loading objects. The following figure is the sequence diagram of login operations. The system searches for the user name in the corresponding table until the record is found.

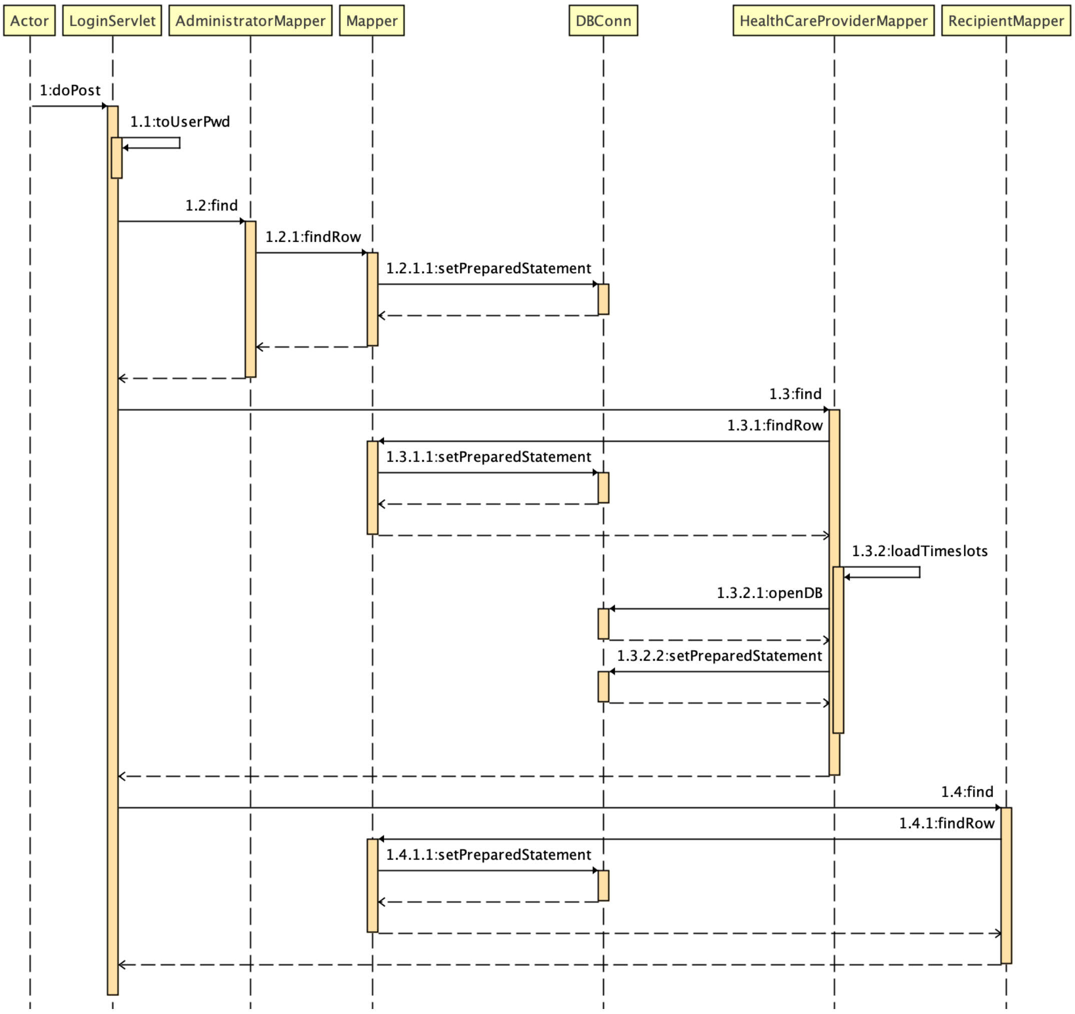


Figure: sequence diagram of the login process

6.8 Authentication and Authorization

- **Authentication**

The system uses session-based authentication and authorization and uses the SHA-1 algorithm to encrypt and verify the password. After the user authentication is successful, the server generates user-related data and saves it in the session, and the session_id sent to the client is stored in the cookie. In this way, when the client requests, it can verify whether there is session data on the server with the session_id, This completes the user's legal authentication. When the user exits the system or the session expires, the client session is invalid. The sequence diagram shows the detailed login authentication process.

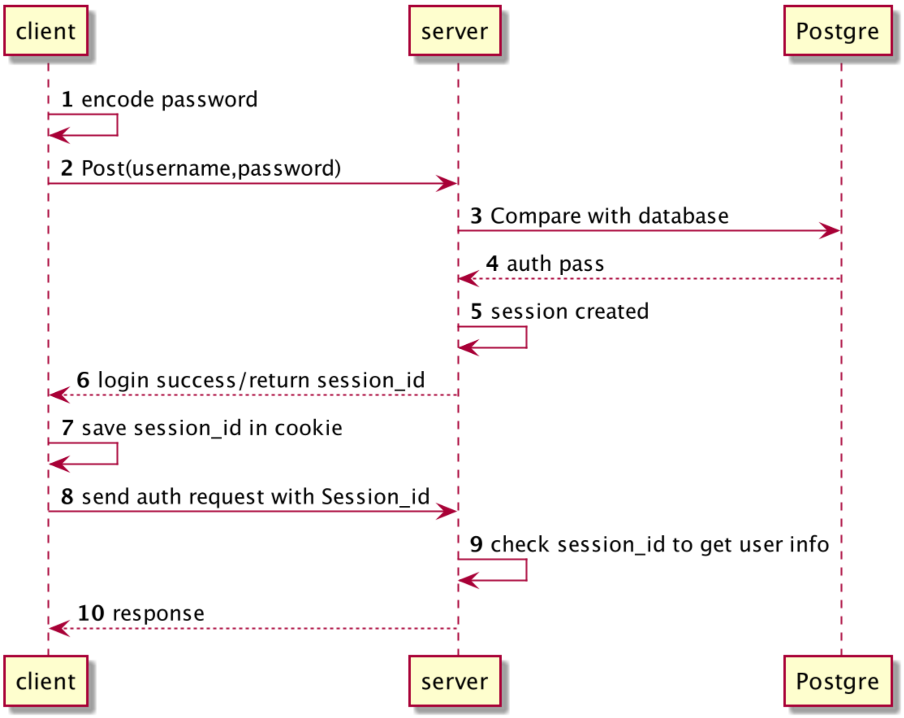

client
server
Postgre
1 encode password
2 Post(username,password)
3 Compare with database
4 auth pass
5 session created
6 login success/return session_id
7 save session_id in cookie
8 send auth request with Session_id
9 check session_id to get user info
10 response
client
server
Postgre

Figure: sequence diagram of user login authentication

- **Authorization**

Authorization mainly realizes two functions:

1. Access interception of anonymous users (non-logged users): anonymous users are prohibited from accessing the system (except the landing page).

2. Log in user access interception: determine whether certain resources can be accessed according to the user's permissions.

The system sets page logic that does not interfere with each other for three types of users. Based on this, we only need to set authority on pages. Bind the above two access permissions on each page, verify the session when the user accesses, and the pages that do not meet the permissions cannot be accessed (for example, recipients cannot access the pages of admin and provider), and jump back to the login page.

We also tried the spring security and Shiro frameworks, which provide simple implementation methods. They centralize authority and are better suited for large projects. But compared with the above methods, the framework is not easy to transplant and is complex to deploy. Using the above method is more suitable for the page logic of system design, which is flexible and easy to use.

The diagram shows the two functions we mentioned above, set two check methods when users want to view a page through URLs.

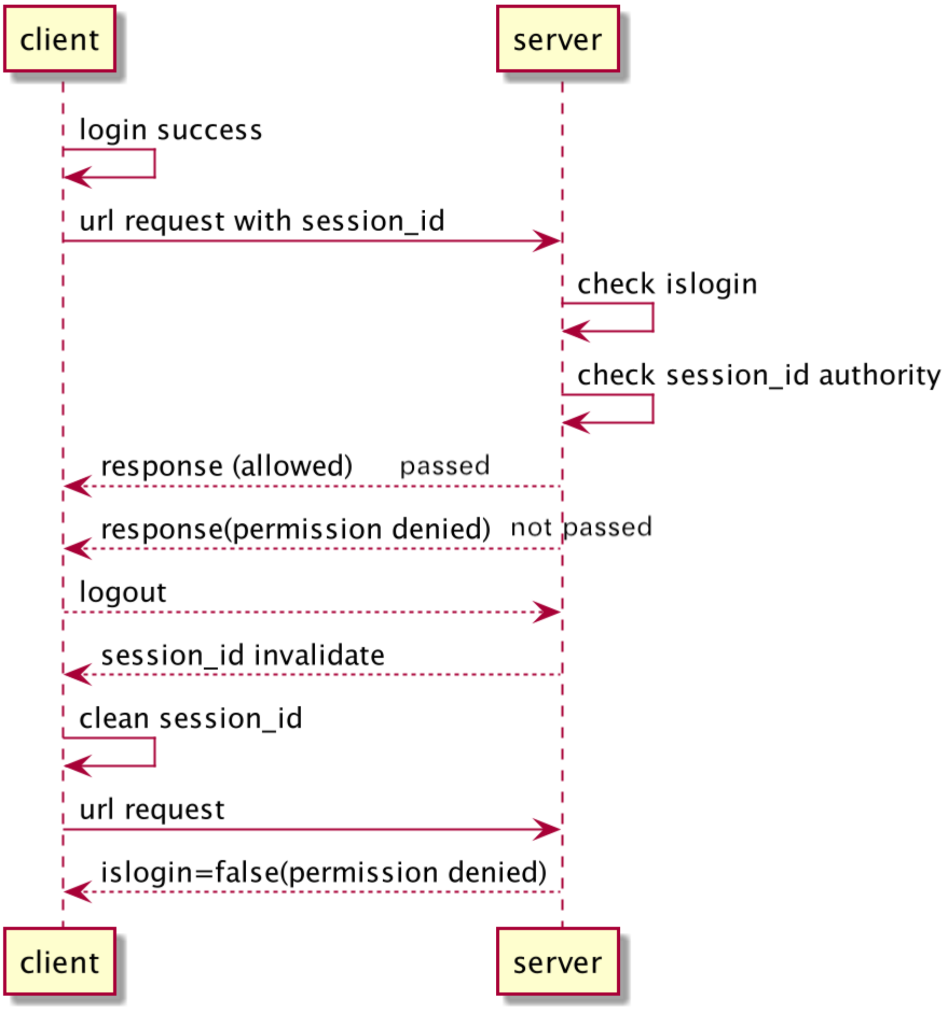


Figure: sequence diagram of user authorization

Design Rationale

7.1 Unit of work

The unit of work pattern aims to keep track of modifications of objects. And through this, we can reduce the workload of the database and better resolve concurrent problems in the next part. In our project, we apply UOW on recipient creation, deletion, and modification. As the sequence diagram shows, add, edit, delete operation will only register the change in memory but not upload them to the database. The user needs to click the 'confirm change' button to commit all the changes made. The benefits of using the unit of work here are the convenience of the objects modification management and less written-back operations to the database.

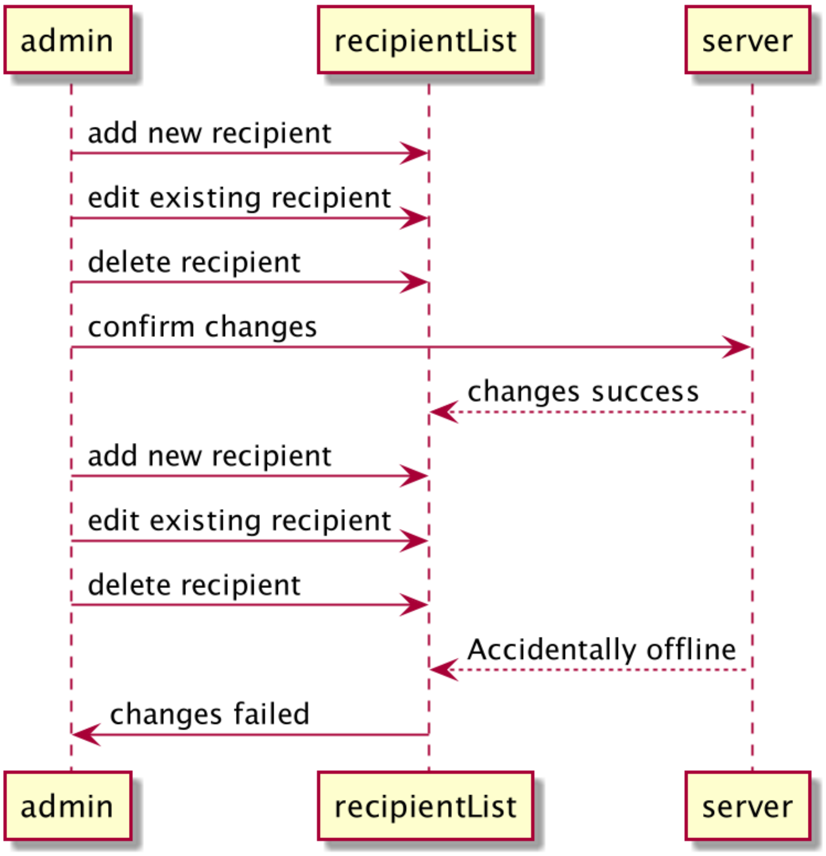


Figure: sequence diagram of the unit of work

7.2 Lazy load

Lazy load aims to reduce the amount of data read from the database by only reading what it needs. In this project, we use Ghost to implement it. Sometimes, we don't use the *question* content in the object *questionnaire.* Therefore, we could use lazy load here and only load questions from the database when we need to use questions and it's null in the object. Here, when we want to use the function *getQuestions()*, and there is one question variable in the questionnaire class that is null, it will use *QuestionaireMapper* to load the value of each question. As we store a string in column question of table *questionnaire*, we need to generate Question objects by using string values and then put them back to the object *questionnaire*.

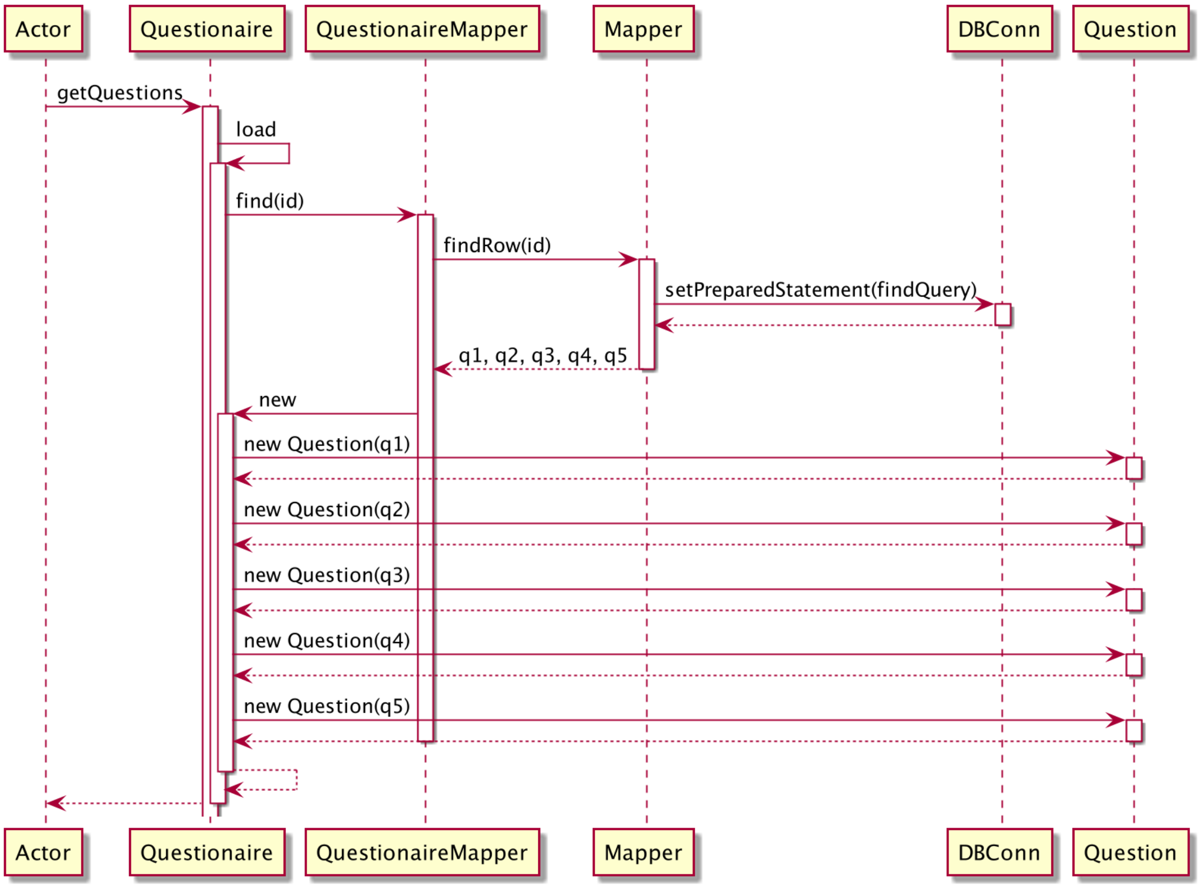


Figure: sequence diagram of getting all questions of one questionnaire object