Software Architecture Design

SWEN90007

SWEN90007 SM2 2020 Project

Magic Pigeons

In charge of:
Simai Deng: simaid@student.unimelb.edu.au
Jiayu Li: jiayul3@student.unimelb.edu.au
Yiran Wei: yirwei@student.unimelb.edu.au

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
31/10/2020	03.00	Initial draft	Simai Deng, Jiayu Li, Yiran Wei
31/10/2020	03.10	Updated class diagrams	Simai Deng, Jiayu Li, Yiran Wei
1/11/2020	03.20	Final version	Simai Deng, Jiayu Li, Yiran Wei

Contents

1. Introduction

This document is a Software Architecture Design report of the Part 3 development of the online examination application, which mainly focuses on concurrency and security design. This document records the design rationale behind the system and the design patterns used, explained with the aid of class diagrams and sequence diagrams.

1.1 Proposal

This document will be updated as the project progresses and will last for the whole software development lifecycle. It aims to present an overview of the architecture design of the system at a higher level and detailed views of the use of design patterns at the lower level.

1.2 Target Users

This document is aimed at the project team, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

2. Class Diagram

In this section, we will highlight the changes to the class design of our system made in Part 3. Since showing the whole system in a single class diagram is impossible due to its complexity, we will break the system down into components and introduce each component and relationship between them.

2.1 Domain Objects

Compared to the class diagram in Part 2, a StudentSubjectMark class is added to represent the users_has_subjects table in the database. It is coupled with the UserSubjectMapper class, as shown in figure 2.1.1. This class is added because of the necessity of optimistic lock in the table view for marking, which will be discussed in further detail in section 3.4.

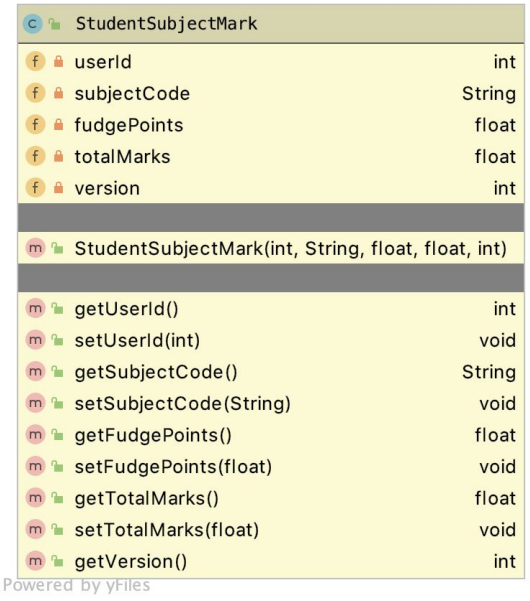

Figure 2.1.1 StudentSubjectMark class

Figure 2.1.2 shows all the domain objects used in the system. The StudentSubjectMark class does not have any dependencies with other domain classes.

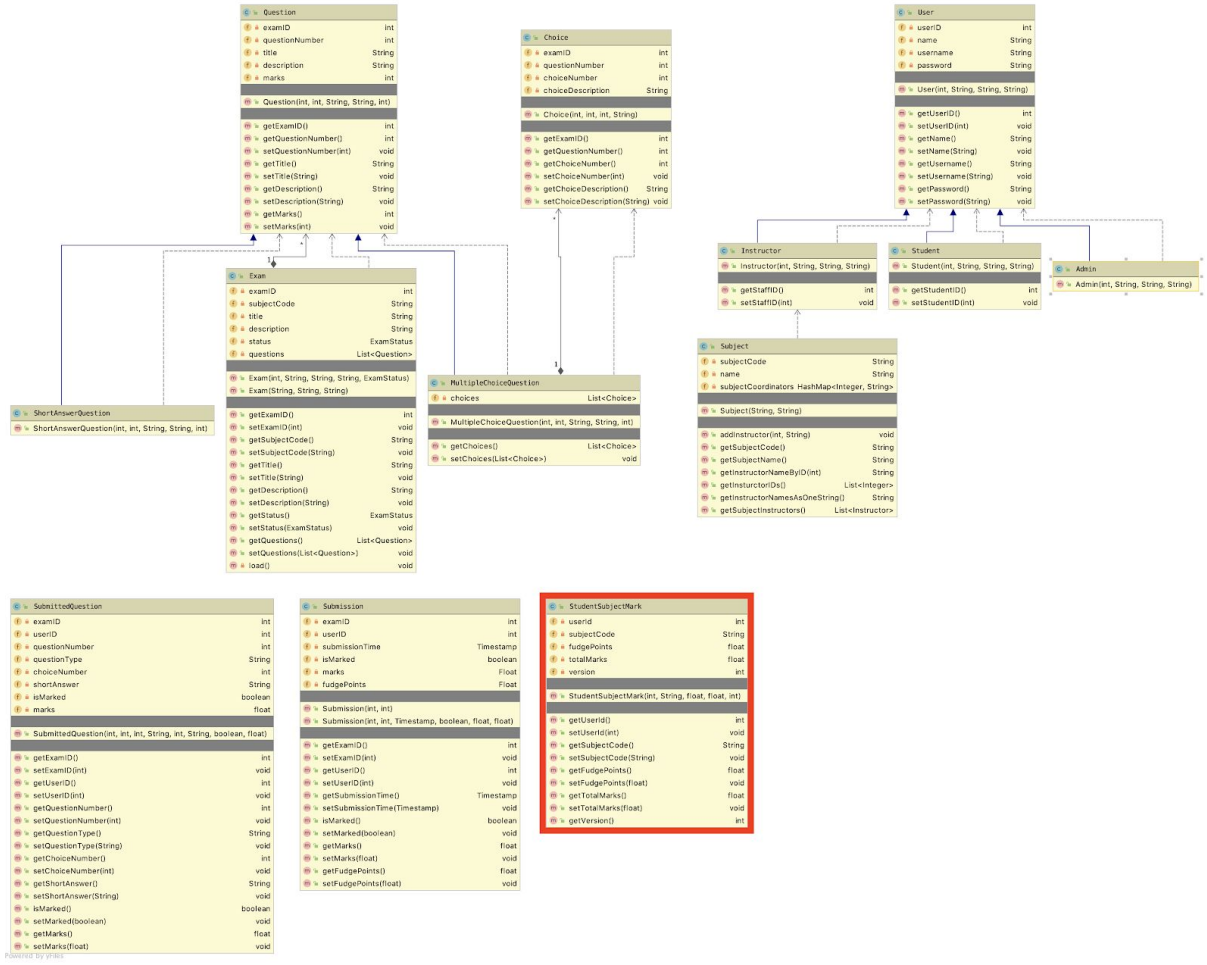

Figure 2.1.2 Class diagram of domain objects

The StudentSubjectMark class is coupled with the UserSubjectMapper class. Their relationship is shown in Figure 2.1.3.

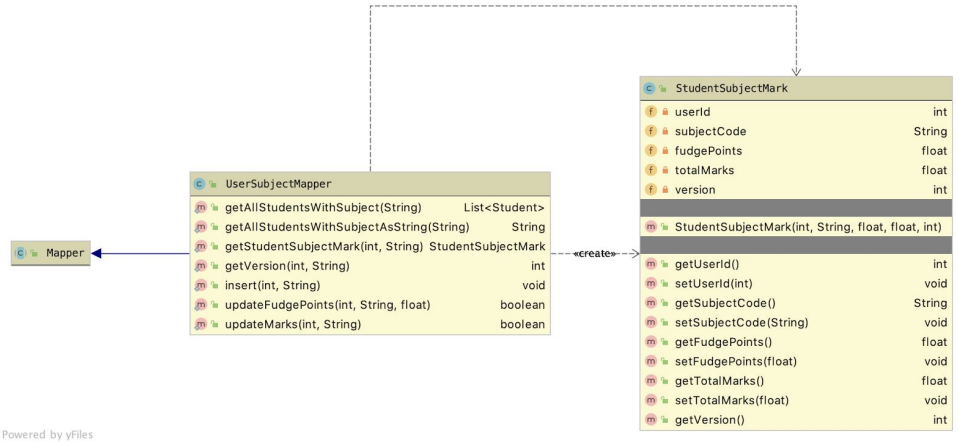

Figure 2.1.3 StudentSubjectMark class and the UserSubjectMapper class

2.2 Data Mappers

To handle concurrency, locks need to be saved in the database. Therefore, a LockMapper and a SubmissionLockMapper are added to the system in order to handle the mapping of locks. Figure 2.2.1 shows the details of the two new mapper classes.

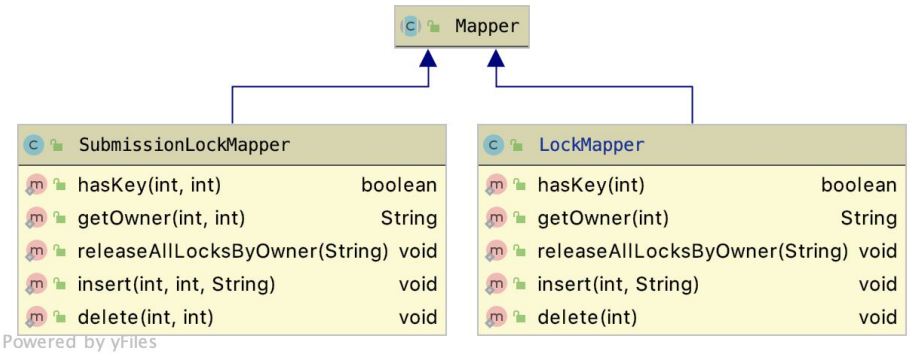

Figure 2.2.1 New data mappers added in Part 3

Figure 2.2.2 shows the details of all data mapper classes.

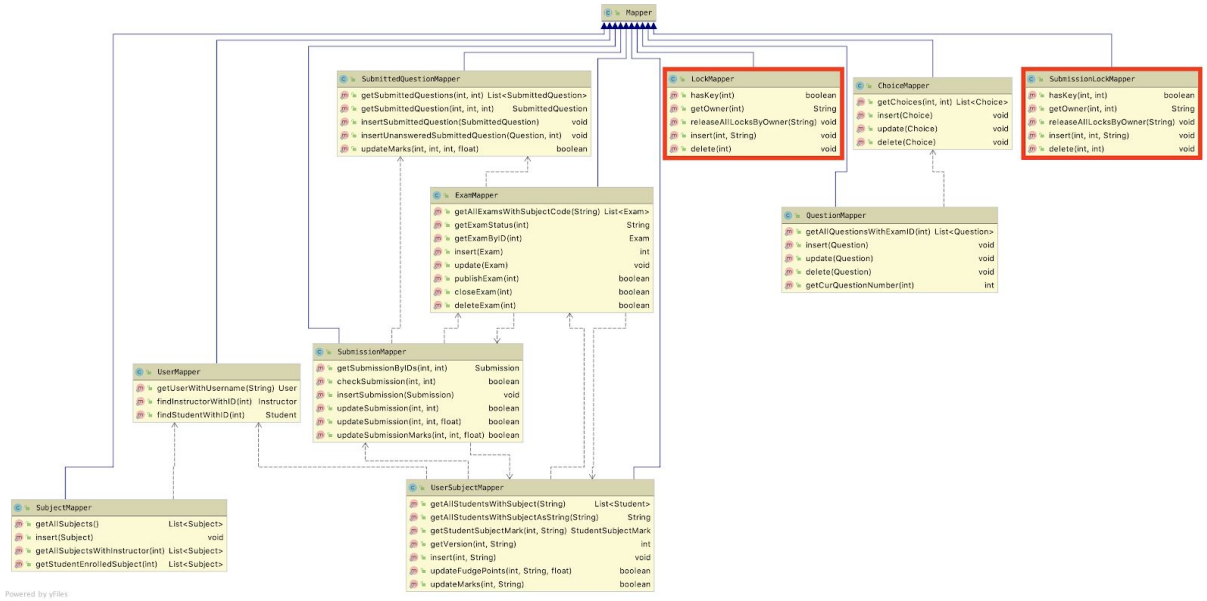

Figure 2.2.2 class diagram of all data mappers

2.3 Database Connection

DBConnection is the class that is used for getting connections with the PostgreSQL database and preparing query statements. Instances of DBConnection are created by data mappers when needed, and closed when the task is finished.

Figure 2.3.1 shows the dependencies between DBConnection and all the mapper classes.

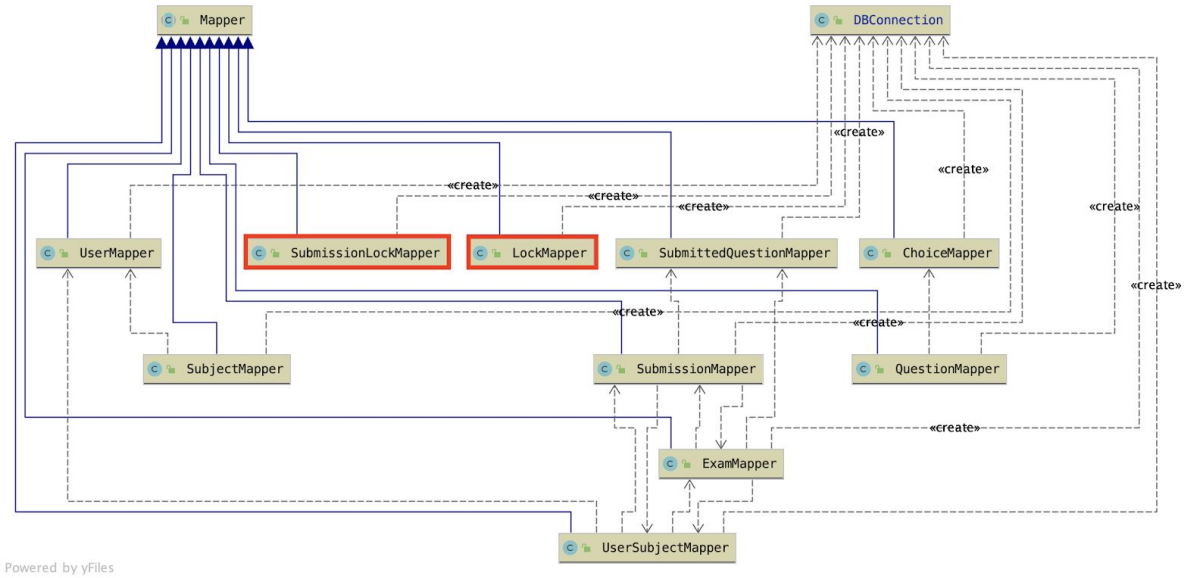

Figure 2.3.1 Dependencies between all data mappers and DBConnection

2.4 Controllers

Controllers handle the HTTP GET and POST methods from the front end by extending the HttpServlet class and overriding the doGet and doPost methods. The controllers are shown in Figure 2.4.1. The fields and methods in the controllers are unchanged compared to the version in Part 2.

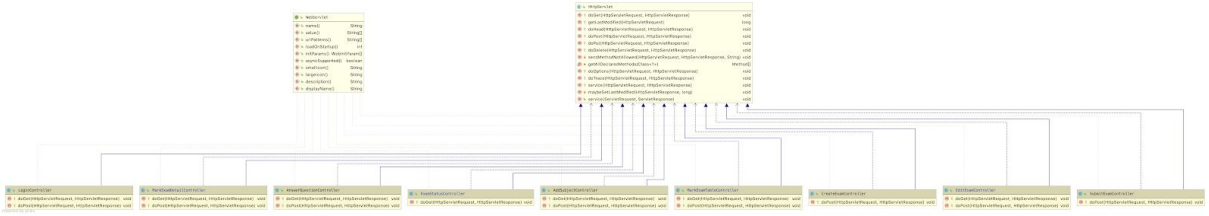

Figure 2.4.1 All controllers

Figure 2.4.2 shows the use of data mappers in controllers.

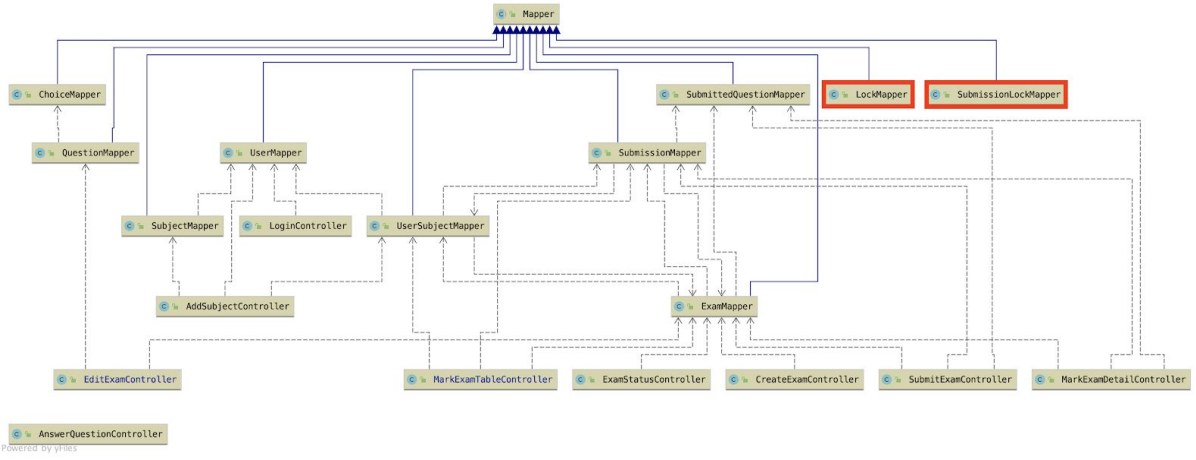

Figure 2.4.2 Dependencies between all controllers and all data mappers

2.5 Lock Management

In order to implement the locks used for concurrency control, new classes are added to the system in order to manage the lock. For the editing exam functionality, we chose to implement a pessimistic online lock. a LockManager class, a LockRemover class and a LockMapper class are introduced to the system for this purpose. In addition, AppSession and AppSessionManager classes are added as well to manage the sessions. Figure 2.5.1 shows the newly added classes for the edit exam functionality, with the involved controller EditExamController. Figure 2.5.2 gives an overview of the relationship between these classes.

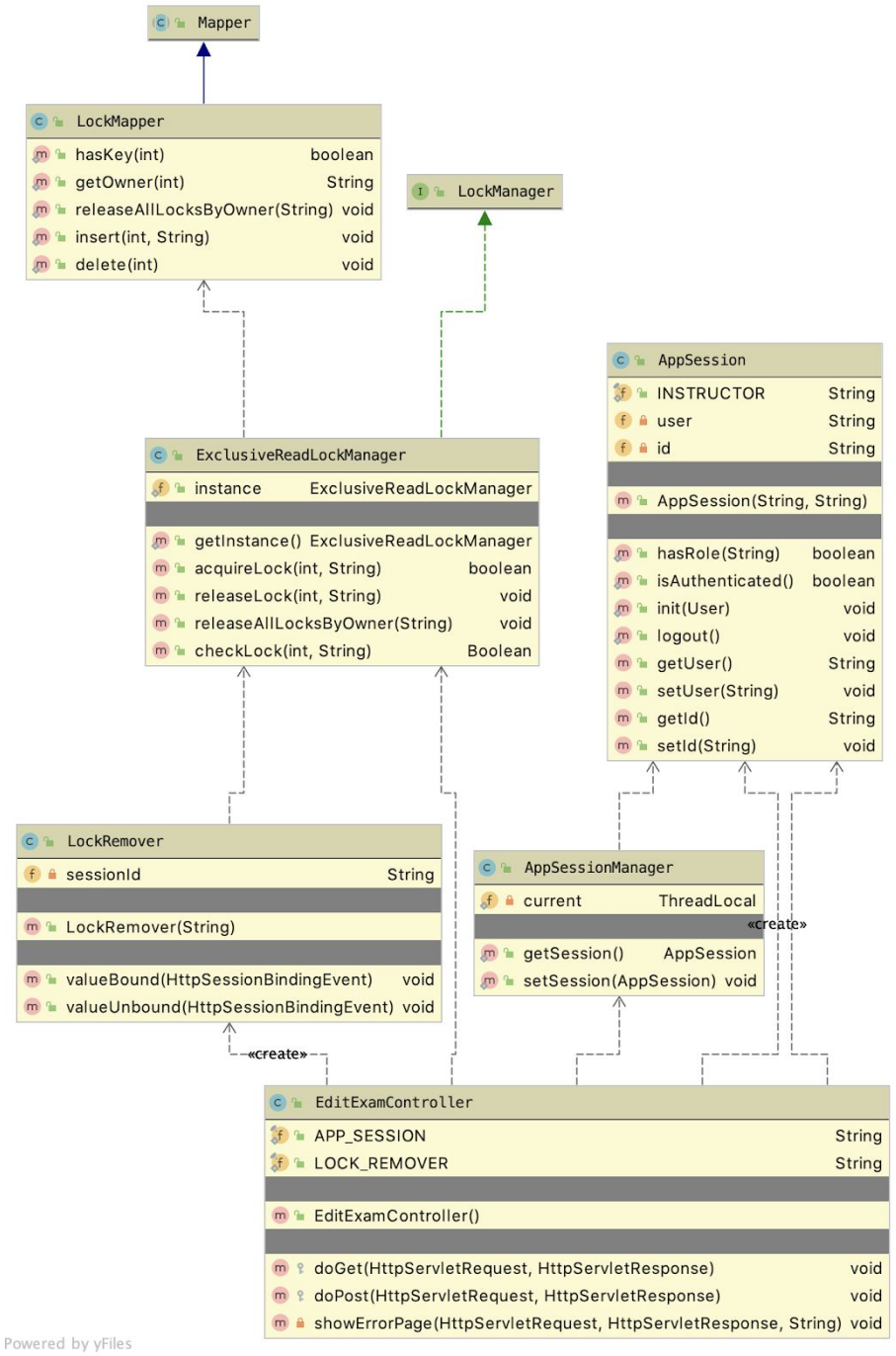

Figure 2.5.1 Class diagram of lock management for editing exams

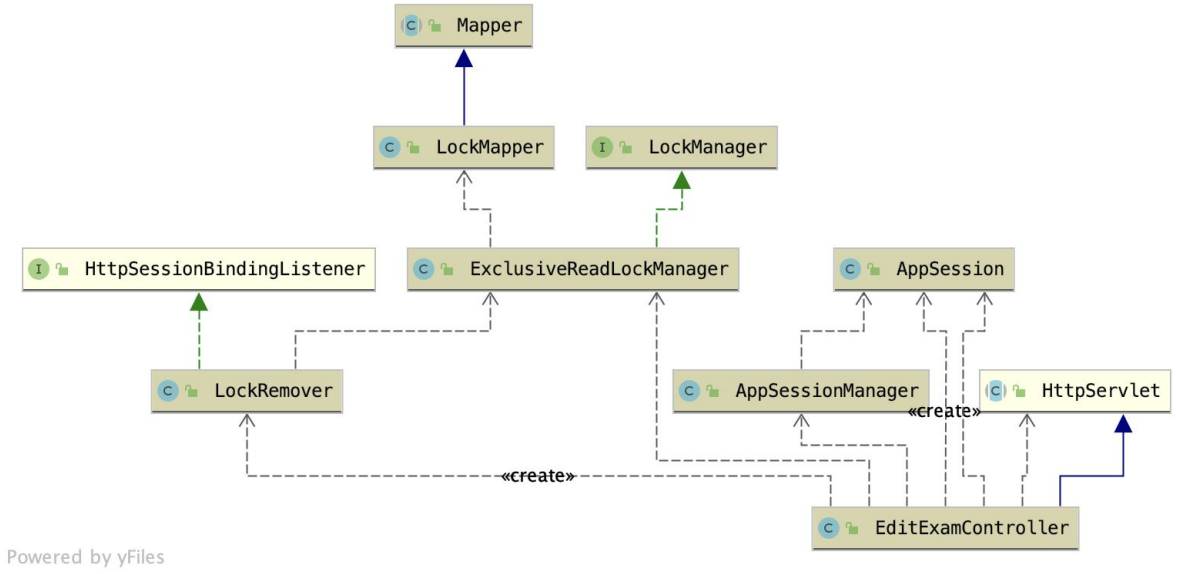

Figure 2.5.2 Overview of lock management for editing exams

There are also new classes added for the marking functionality. It also uses the pessimistic online lock so another LockManager class, LockRemover class and LockMapper class are added to manage the locks for the marking functionality. It shares the AppSession and the AppSessionManager class with the edit exam functionality. Figure 2.5.3 shows the relationship between these classes, and Figure 2.5.4 shows the classes with more details.

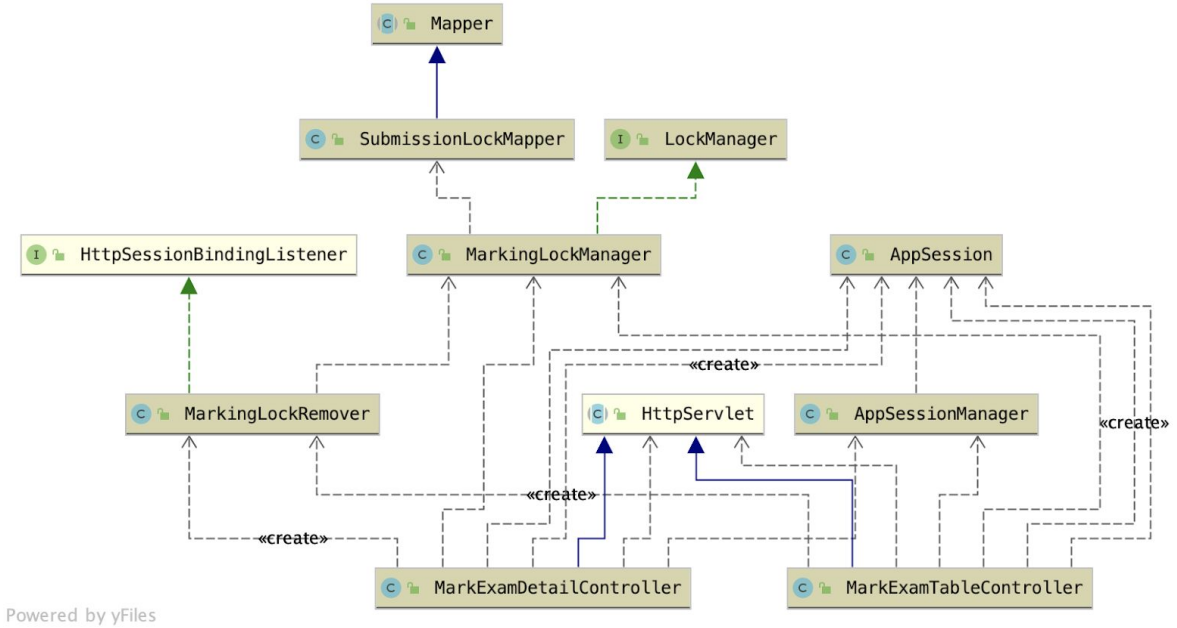

Figure 2.5.3 Overview of lock management for marking exams

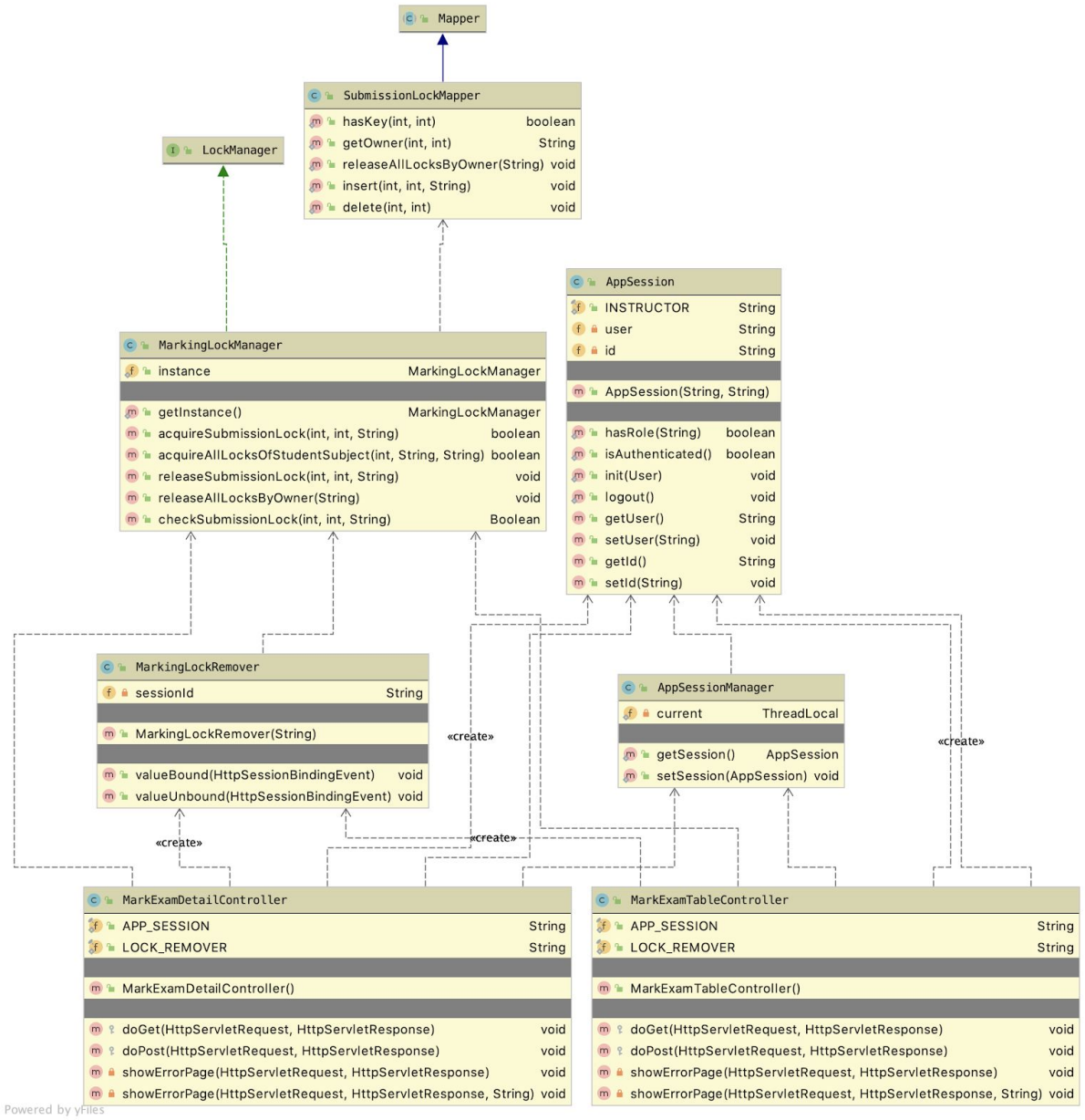

Figure 2.5.4 Class diagram of lock management for marking exams

3. Concurrency Design Patterns

3.1 Instructor creating exams

In our implementation, the change to the database is committed when the instructor clicks *save* on the createExam page and an exam id will be generated and returned by the database before questions and choices relating to this exam are saved. Therefore, there is no need to handle concurrency for this as two or more instructors cannot be creating the “same” exam.

3.2 Instructor editing exams

To handle the concurrency for multiple instructors editing the same exam, an *exclusive read lock* is used.

As we do not want the instructor to edit the exam and then find out someone else has also edited the exam and all his changes are lost, an optimistic lock is not very ideal. We did not implement a separate function for the instructor to view the questions of the exam. In other words, if an instructor wants to see the questions, he has to use the edit exam function. Therefore, there is no point in implementing the *read/write* lock because if an instructor is viewing the exam, he can also edit the exam. That leaves us with two options: the *exclusive read lock* and the *exclusive write lock*. In order to prevent inconsistent reads, the final decision is to use the *exclusive read lock*.

When the instructor clicks the edit exam button, a lock will be acquired. In this case, the lock is a data entry consisting of two parts: *lockable* and *owner*. *Lockable* is the exam id for the exam that the instructor is trying to edit, and *owner* is the session id of the request that comes in. When the instructor saves the exam or clicks the "go back" button, the lock will be released. However, if the user determines to exit in an "abnormal" way such as closing the tab directly, this will result in a deadlock. To solve this problem, all the sessions are set to time out after 120 minutes. When the instructor tries to save the exam, the controller will first check if the owner (i.e. the current session) in fact owns the lock before committing the changes to prevent a timed-out session to edit the exam. I understand this is not ideal and I am sure there are better ways to manage the session such as sending a request to the client to check if the session is still active. However, it is not covered in this course, so this is a compromise I made.

Another place where the pattern is used is in the ExamStatusController, the controller for handling, publishing, closing, or deleting an exam. In order to prevent inconsistent data, the controller will first check if the exam is locked by someone before publishing or deleting the exam (one cannot edit an exam that is published and one cannot close an unpublished exam so no need to worry about that) . It will not acquire a lock though, as the transaction happens with a single database update.

When the instructor clicks the edit exam button and tries to enter the edit exam page, the controller needs to handle the http get request. It first checks whether there is already an app session. If so, it calls the lock manager to release all locks owned by the user, and start a new app session. This process is shown in Figure 3.2.1.

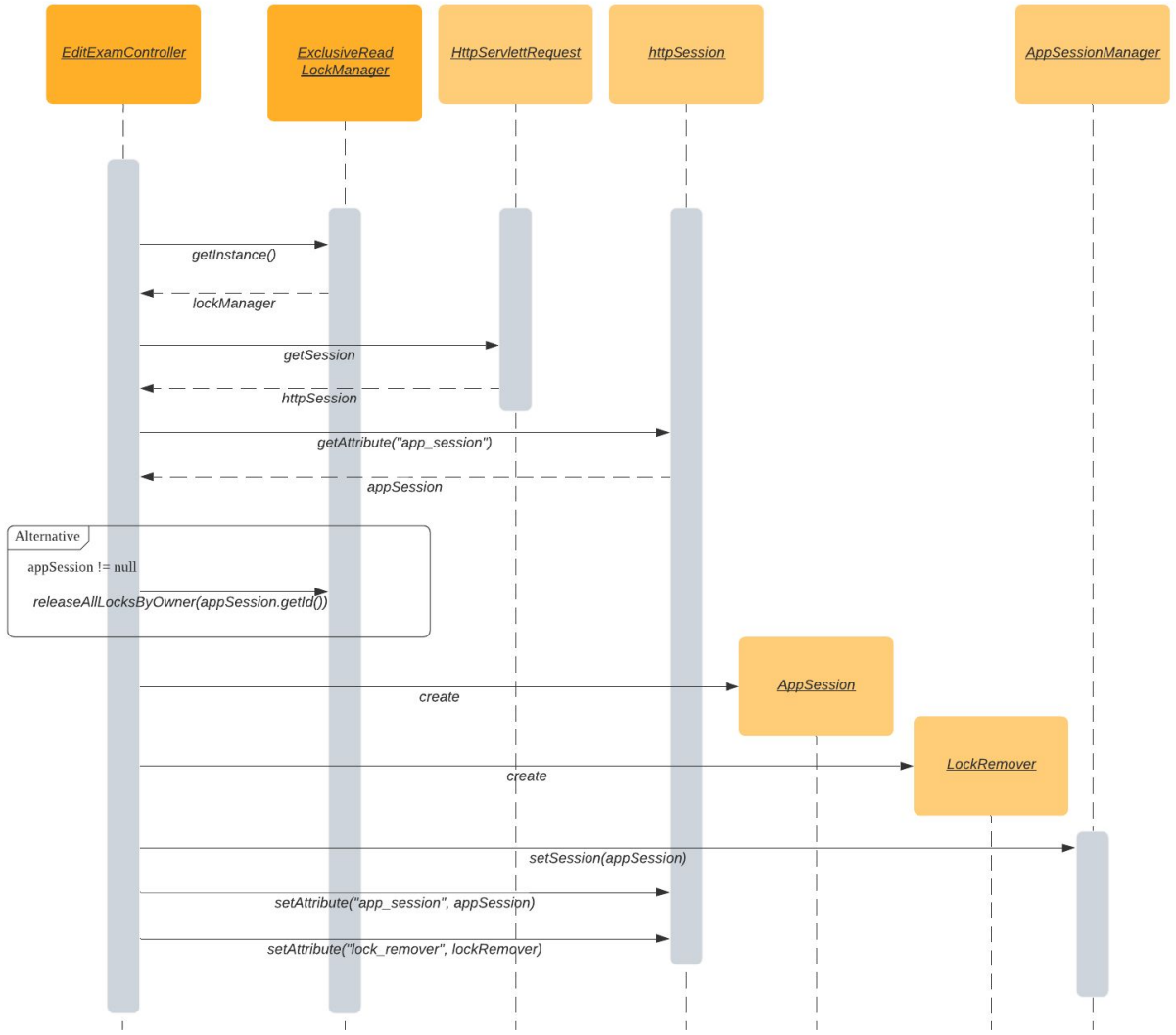

Figure 3.2.1 Setting up the app session

After the app session is set, the controller tries to acquire the lock on the exam that the instructor wants to edit. If the lock can be acquired or the user already owns the lock, the user can access the page. This process is shown in Figure 3.2.2.

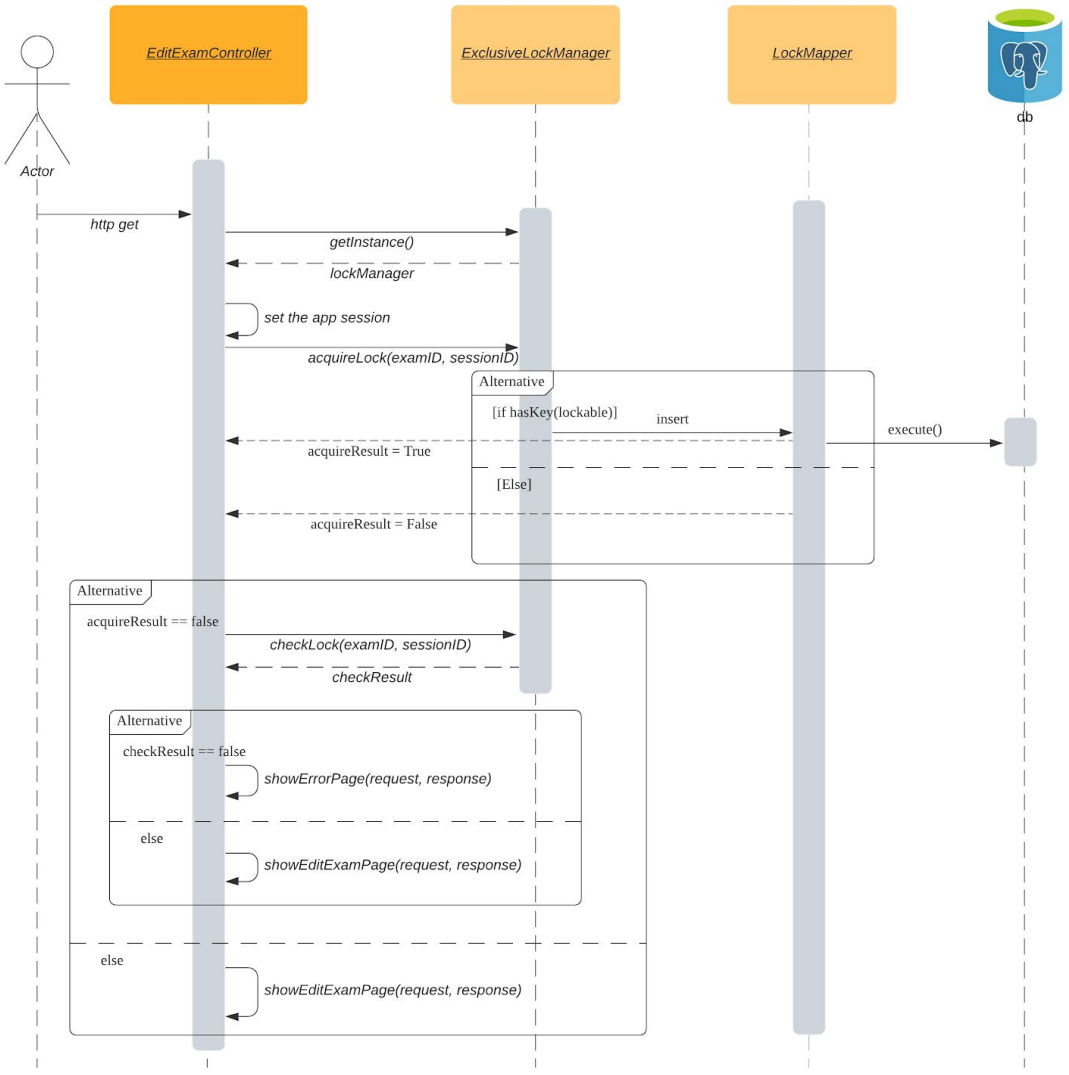

Figure 3.2.2 doGet method of editing exams

After the instructor finishes all the changes, when they press the save button, the controller needs to handle the http post request. It checks if the user still owns the lock. If so, it calls the mapper classes to make the changes to the database, and releases the lock after all changes have been performed. Otherwise, the user will be directed to an error page. This process is shown in Figure 3.2.3.

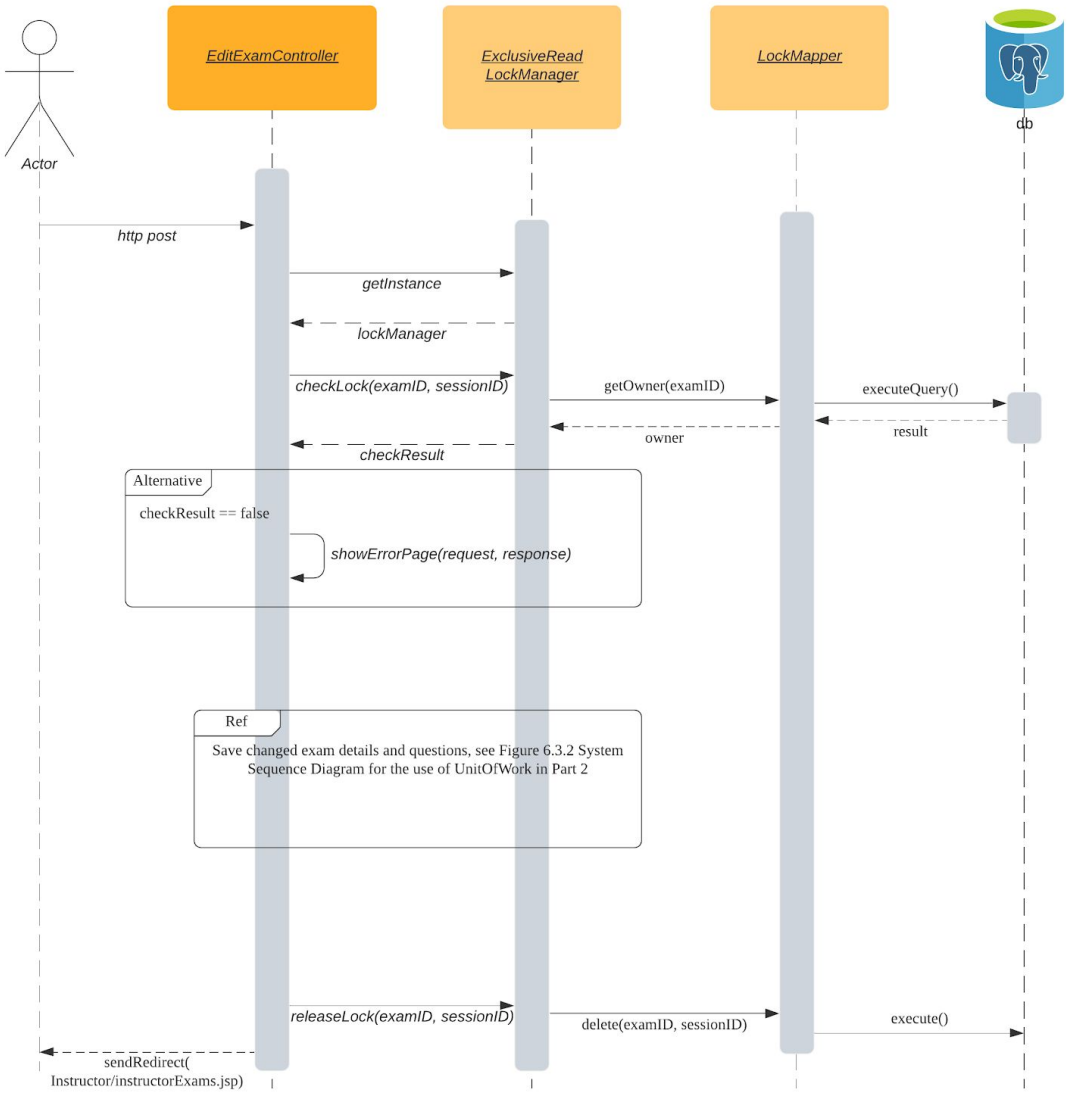

Figure 3.2.3 doPost method of editing exams

3.3 Instructor marking exams in detailed view

To handle the concurrency of multiple instructors marking submissions in detailed view at the same time, we chose to implement a pessimistic online lock. There are several reasons for our choice of lock. First of all, in an exam system, we imagine that it will be a very common situation when multiple instructors need to mark students' submissions at the same time. Therefore, the probability of conflicts happening can be high if concurrency is not handled. Also, it can be very frustrating for an instructor, who just finished marking a submission and is trying to save the marking, to learn that their work just got lost. Hence avoiding conflicts at the beginning seems to be a more sensitive approach than detecting conflicts at commit time. In addition, since there are many submissions that need to be marked, liveness seems to be a less important factor than correctness. If a submission is locked, an instructor can always choose to mark another submission, and go back to the locked submission later.

The type of lock we chose is exclusive read lock because we believe that ensuring consistency is very important in this scenario.

When the instructor enters the webpage of the detailed view, the controller first sets up the app session as shown in Figure 3.3.1. After the app session is set, the controller checks whether the lock on the submission can be acquired. If so, the lock is acquired, and the user can access the page. If not, it checks whether the lock is already being held by the user. If it is, the user can access the page. If both conditions are not true, the controller forwards an error page indicating that the submission cannot be shown because someone else is marking the submission. This process is shown in Figure 3.3.2.

When the instructor presses the "update marks" button, a http post request is sent to the controller. In the backend, both the submitted_questions table and submissions table need to be updated. The controller first checks whether the lock is still being held by the user. If not, it forwards the error page. Otherwise, it updates all the rows that need to be updated. When it finishes updating, it releases the lock and forwards the result page. This process is shown in Figure 3.3.3.

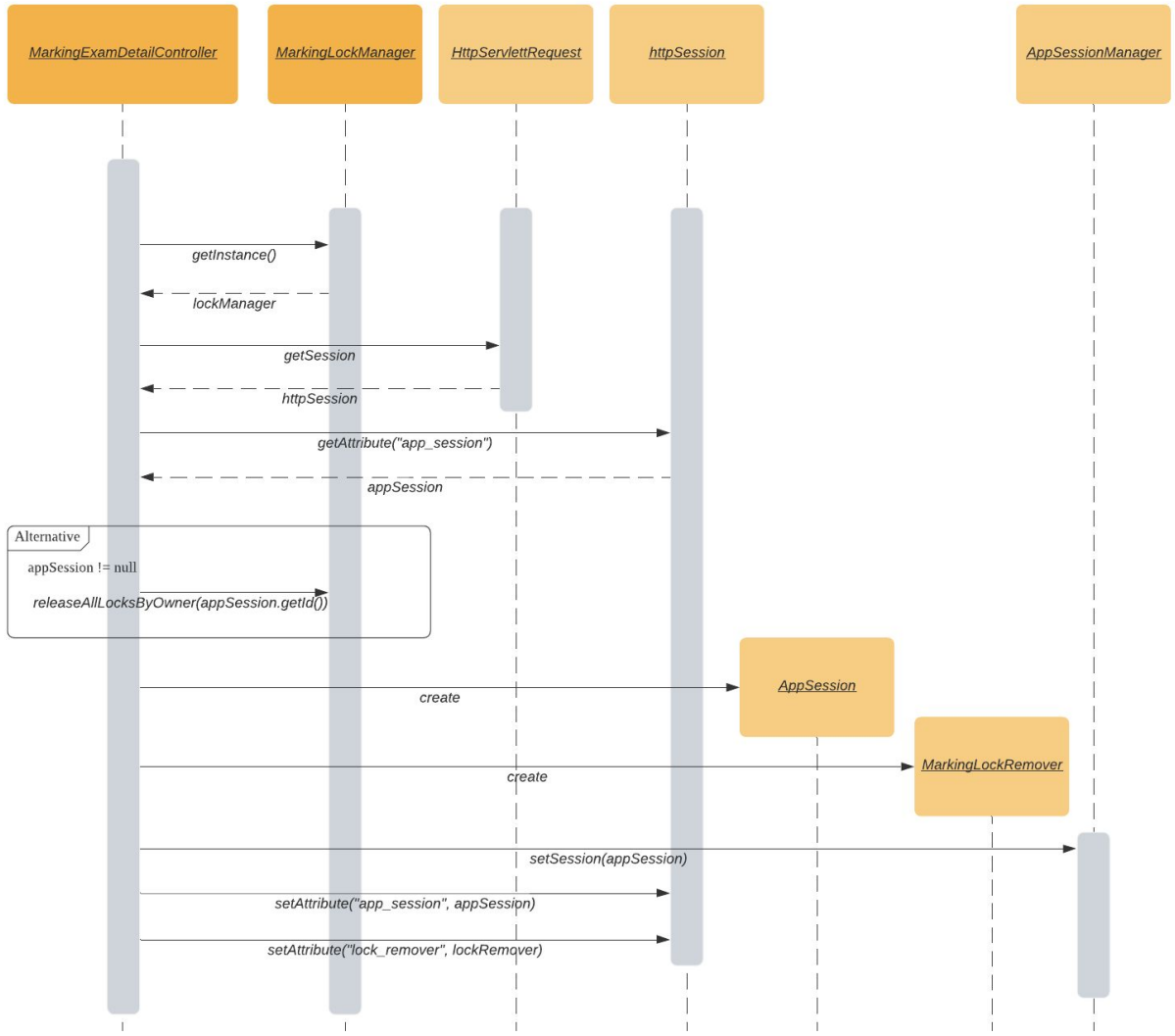

Figure 3.3.1 Setting the app session

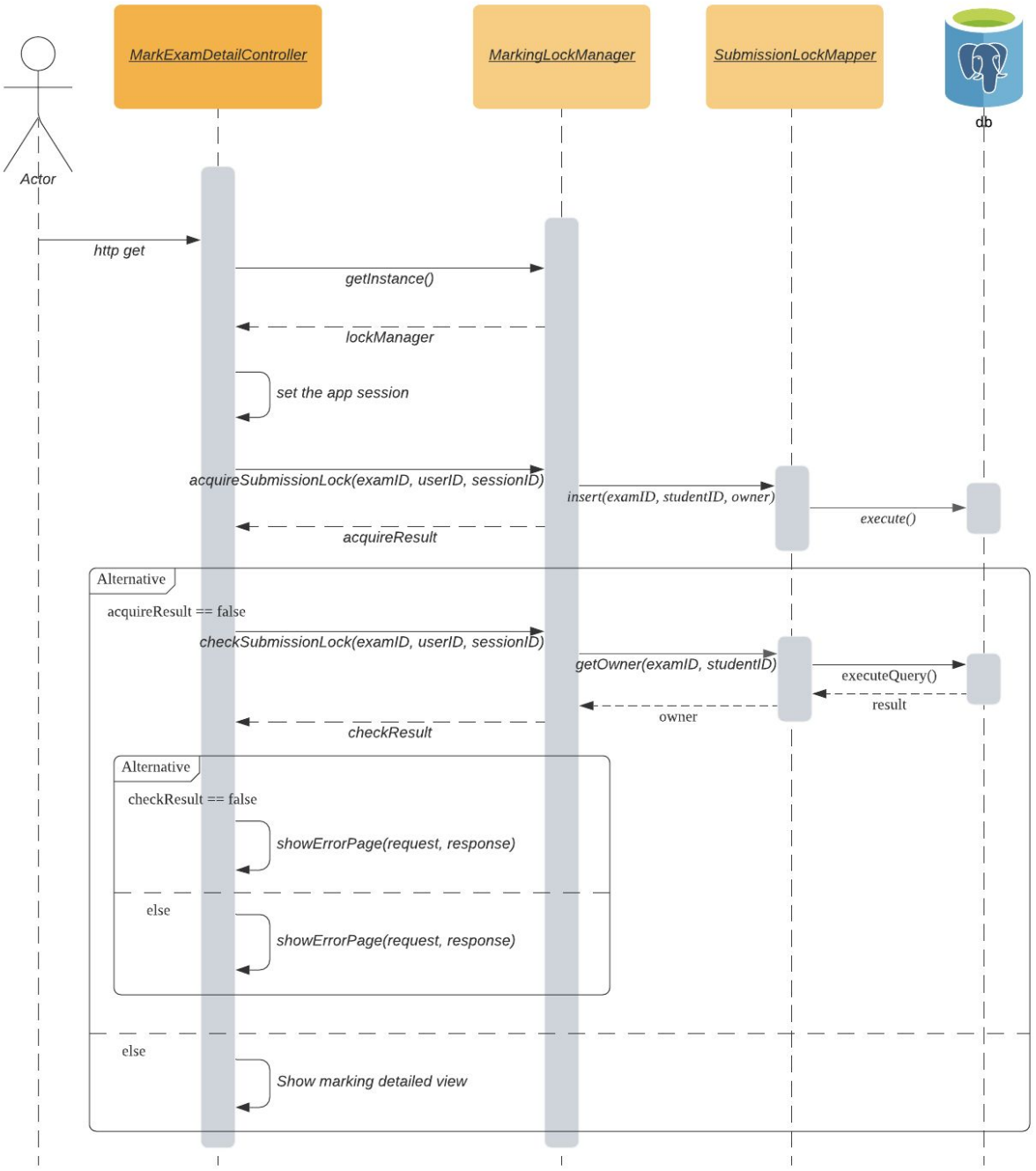

Figure 3.3.2 doGet method of marking in detailed view

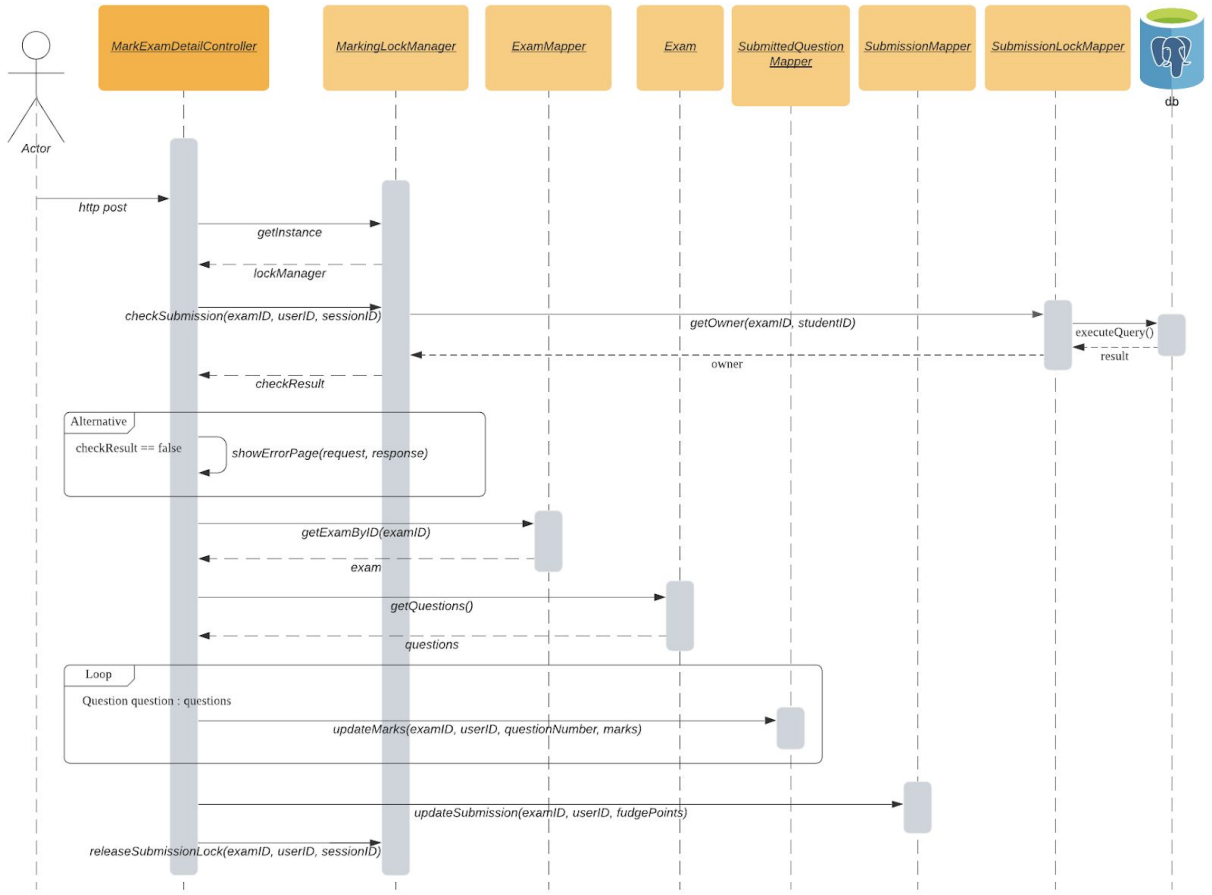

Figure 3.3.3. doPost method of marking in detailed view

3.4 Instructor marking exams in table view

For the marking in table view, we have chosen to use optimistic online locks. We deemed that the table view serves more of the functionality of displaying all student's marks as an overview, but less of updating students' marks. Consequently, liveness must be ensured. In addition, we assumed that the probability of the occurrence of conflicts in table view will be relatively low. Furthermore, since we have adopted exclusive read locks for updates of the submission table, it does not make sense to lock all submissions when an instructor enters the table view webpage.

To implement the lock, a version is added to the users_has_subjects table as a field. This version number is incremented every time a submission of this user and of this subject is updated, no matter from the table view or the detailed view. This increment is done in the updateSubmission method in the SubmissionMapper.

Every time a row is read and a StudentSubjectMark object is created, the version number will also be recorded. When an instructor tries to update marks from the table view, the version number will be compared to the version read from the database. If all fields are not being locked by other users and have the up-to-date version number, the marks will be updated. Otherwise no updates will be executed. This process is shown in Figure 3.4.1.

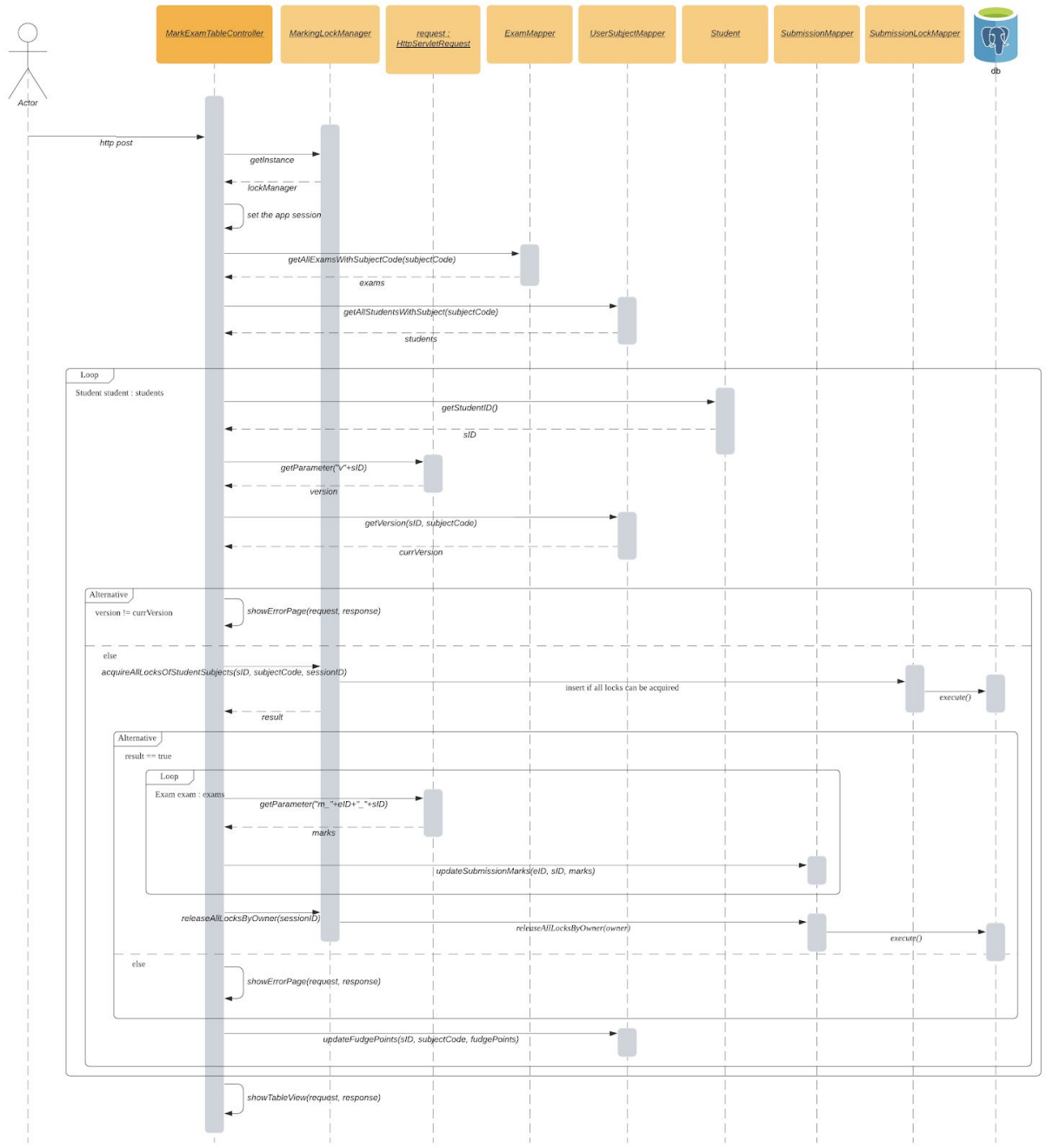

Figure 3.4.1 doPost method of marking in table view

4. Security Design Patterns

As functional requirements are met in our implementation, we start to consider non-functional aspects of this project in this section. One important thing to consider is security in the presentation layer. As authentication, authorisation, and sure pipe patterns are implemented in this application, the security is ensured from many aspects.

The Apache Shiro framework is used to implement security patterns, because this framework is already proven to be secure and reliable, and we do not want the security part to be vulnerable.

We used three security patterns to improve the integral security of this system. Except for the patterns covered in lectures, the security is also improved by not remembering the password for users in the `login.jsp` page, and add the function logout for users to make sure their information is protected.

4.1 Authentication

Authentication is one of the primary aspects of security in an enterprise application. The main idea of Authentication is to ensure users are who they say they are. It acts as a gateway to allow access to valuable information and resources only to those who are approved.

In our implementation, the authentication mechanism consists of two pieces of information according to the business requirements. One part is a unique and public identifier claimed by username, another private piece of information is the password that can verify the claim of identity. The usernames and passwords can be looked up in the database using SQL queries.

This pattern improves the security by reducing the resources in the presentation layer that a mechanism can access, and therefore eliminates the chance of being attacked using potential leak in some way. Also it would be easier to maintain the project as it is less time consuming to modify different role permissions according to the changes in the business requirements.

We choose to use an external library, Apache Shiro framework to implement this pattern, because this mature framework is elegant and reliable.

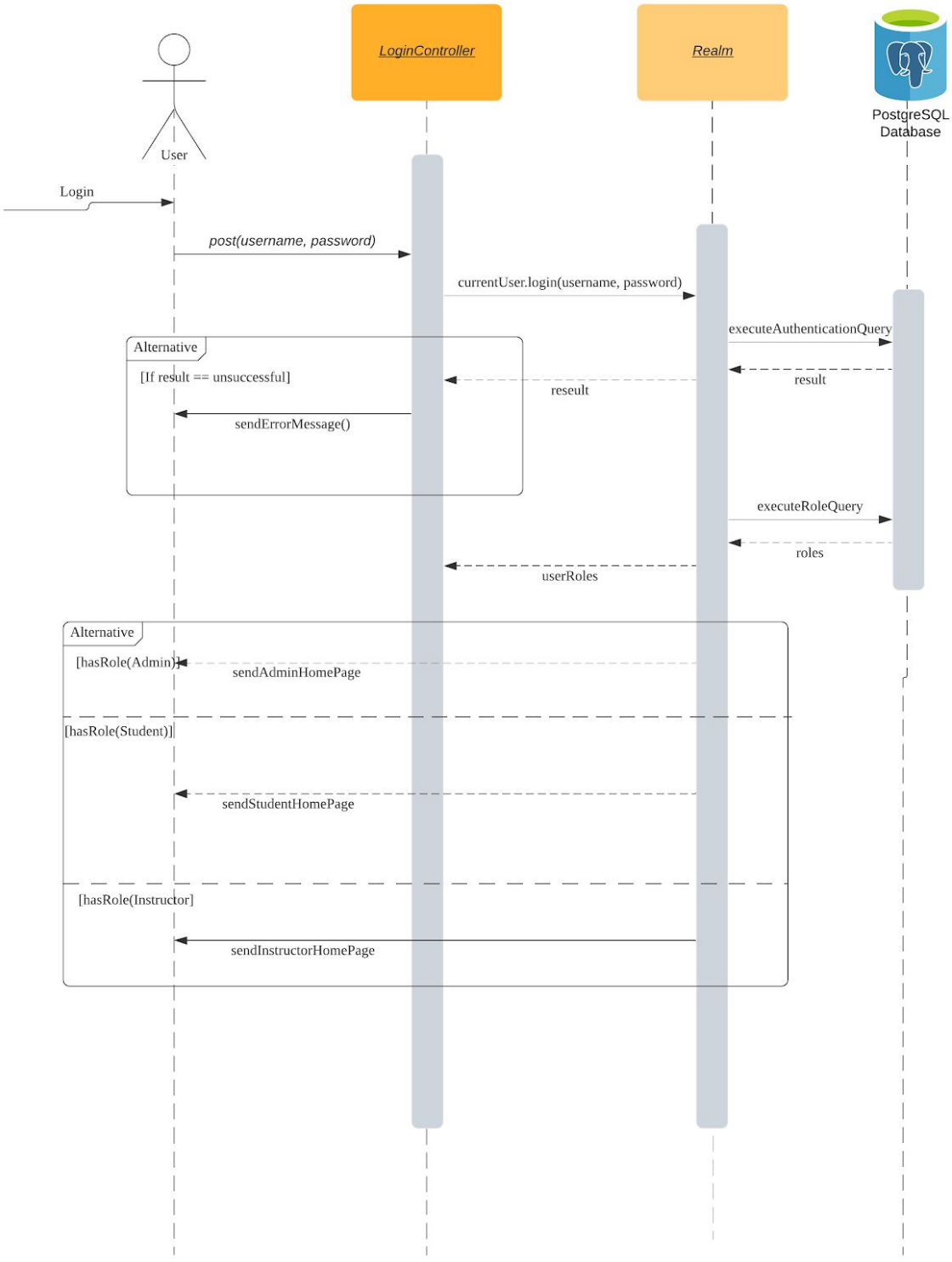

Figure 4.1.1. DoPost method of LoginController

The above figure explains how the authentication works by illustrating the ordered process of interactions when a user tries to login. A subject will be created, and username and password entered in the `Login.jsp` will be taken by the UsernamePasswordToken. Then a query will be made to check if the authentication mechanism's two pieces of information is valid. The password is stored in

the database as a SHA256-HASH and the password matched defined in the `shiro.ini` file will automatically hash the input password to check against the database. A user can only be authenticated if the username is in the database, and the password entered matches. If the user is authenticated, the `LoginController.java` will redirect the user to the home page according to their role. For example, admin will be directed into `subjects.jsp`, instructors will be directed to `instructorSubjects.jsp`, and students will be directed into `StudentHomePage.jsp`. If the authentication fails the user needs to resubmit the form.

4.2 Authorisation

Authorisation and authentication are always used together. As the authentication pattern verifies and determines the identity of who is executing the action, authorisation applies access permissions and privileges to execute the required functions based on the certification result of authentication. Authorisation ensures users get their user permissions and allows them to do what they are trying to do according to their authority.

This pattern provides more secure and elegant access than simply restricting the URL links to users. Restricting URL access is more like a hard-code way, and it is time consuming and clumsy to make modifications according to changes in business requirements.

Authentication is set up as the precondition for authorisation. A filter is used to determine the authorities because our source code is well-organised and neat. As we organised resources grouped by the authorised roles, it is effective and efficient to filt different authorisations of different types of users using their package.

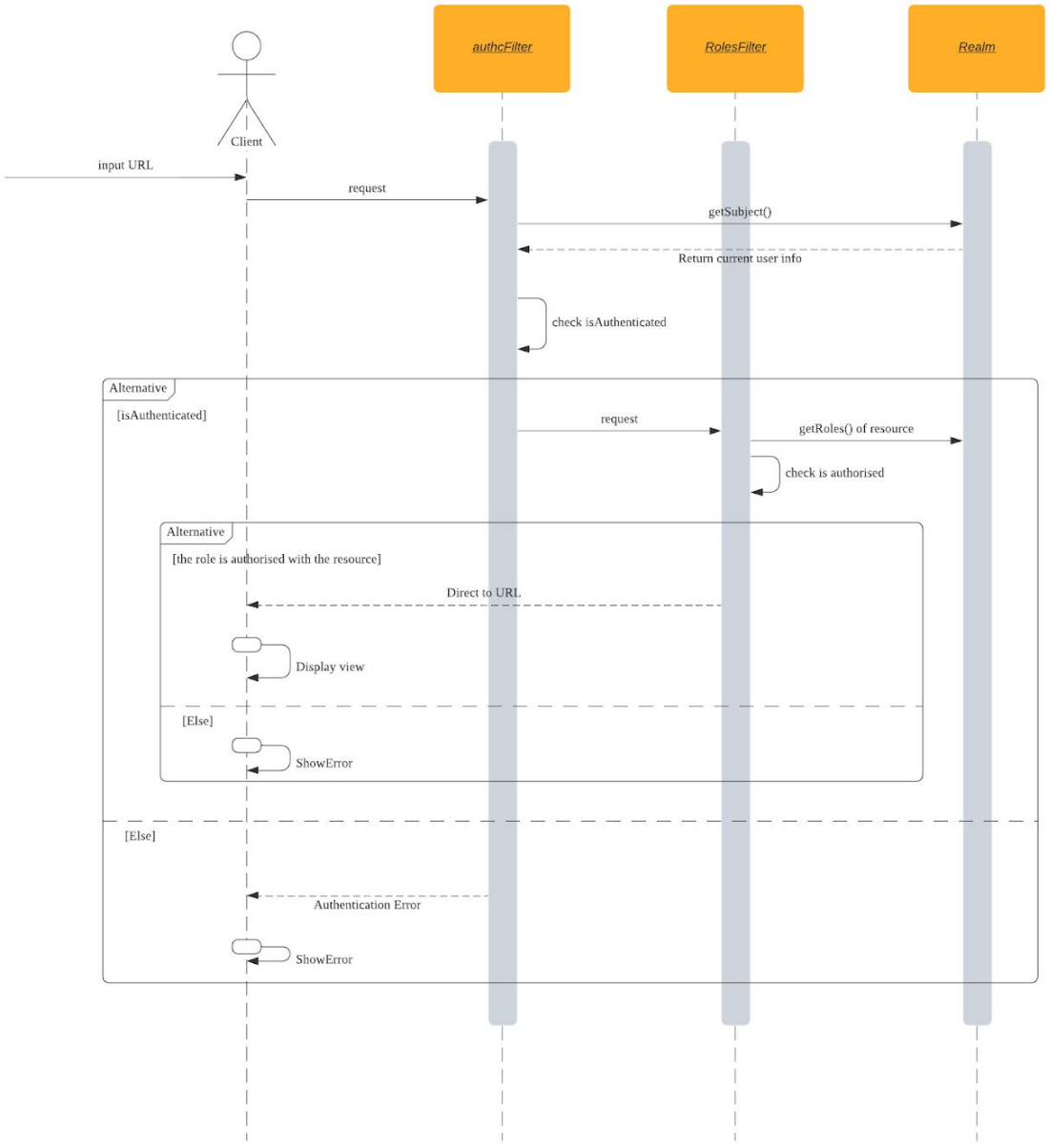

Figure 4.2.1. DoGet method of authorisation filter

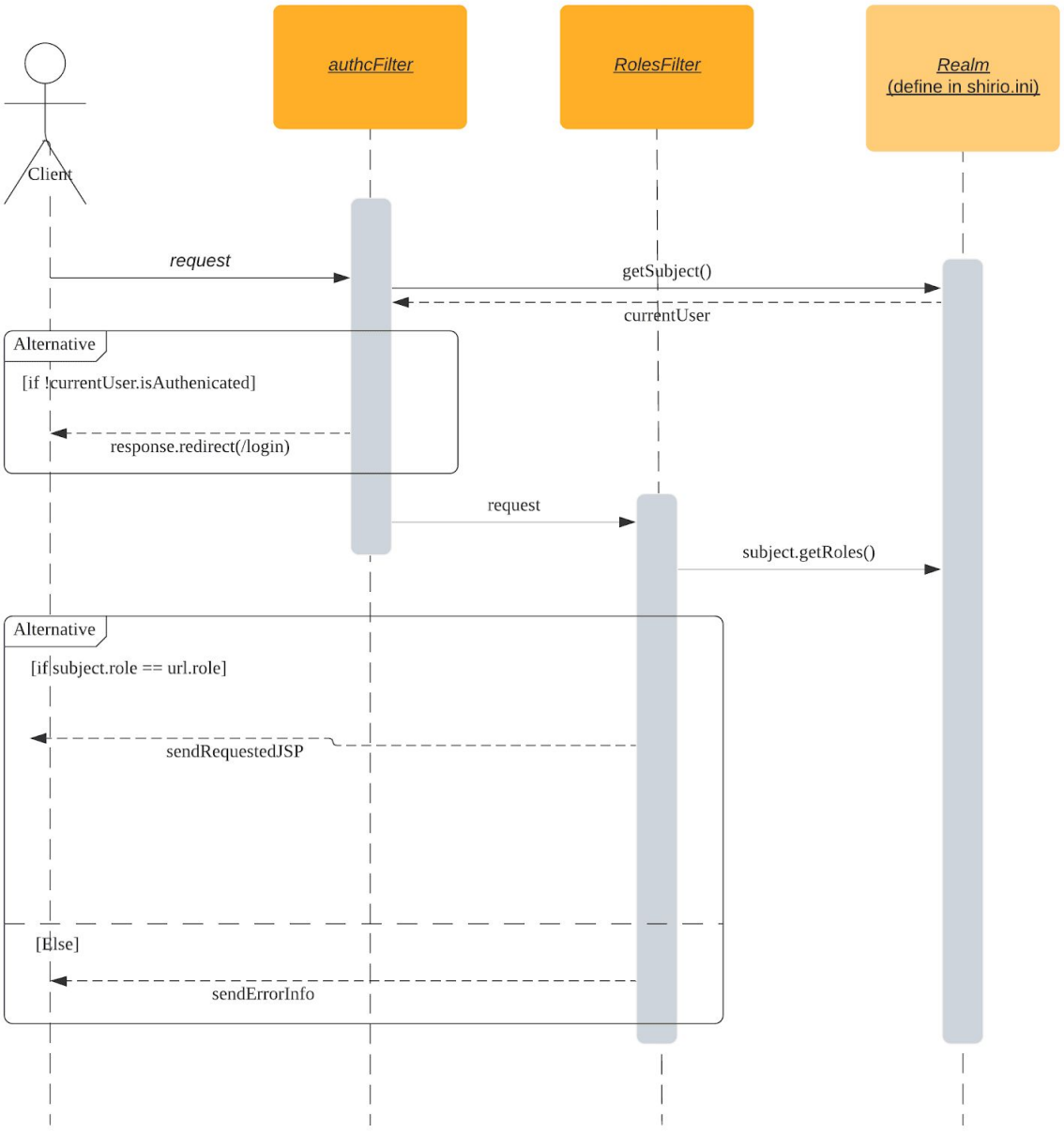

Figure 4.2.2. DoPost method of authorisation filter

Figure 4.2.1 and Figure 4.2.2 demonstrate the process when a user tries to access a resource by simply entering URL. As authentication is a precondition for authorisation, the user must login before accessing the URL successfully. As a user login successfully, it will get its role and a collection of authorisation information will be generated according to its role. If the user is authorised with the given URL and the required session attributes are provided, the page will be displayed successfully. But if the user is not authorised, the server will prompt 401 not authorised error, indicating that the request cannot be applied because valid authentication credentials are lacking. This could also happen when an unlogged user tries to access any resources except for the `login.jsp`.

```
[urls]
/logout = logout
/Instructor/**=authc,roles["INSTRUCTOR]
/Student/** = authc, roles["STUDENT"]
/Admin/** = authc, roles["ADMIN"]
```

Figure 4.2.3. Shiro initialization configuration

As figure 4.2.3. shows, the authorisation of each role is set in the shiro.ini. We organized the resources grouped by user permissions to keep the maintainability. An user request to access a resource would be firstly filtered by the authentication filter to get authentication, and then filtered by the role filter and checked if its role is authorised to the resource.

4.3 Secure pipe

Security pipe is a standardised and common way to minimize the risk of sensitive data sent over the network being read and tampered. The pattern sets up a method to send requests over a secure network with the use of encryption and decryption processing.

The Apache Shiro filter-chain automatically includes SSL encryption. Together with the https protocol used in Heroku, the data in the presentation is encrypted and secured.

In this system, password is considered to be the information that needs to be protected. A potential security vulnerability can be caused by eavesdropping and spoofing on the unprotected password. In our implementation, instead of making clients communicate with the server directly, a pipe is used to handle the communication. Besides, a secure hash algorithm (SHA256) is used to decrypt the password.

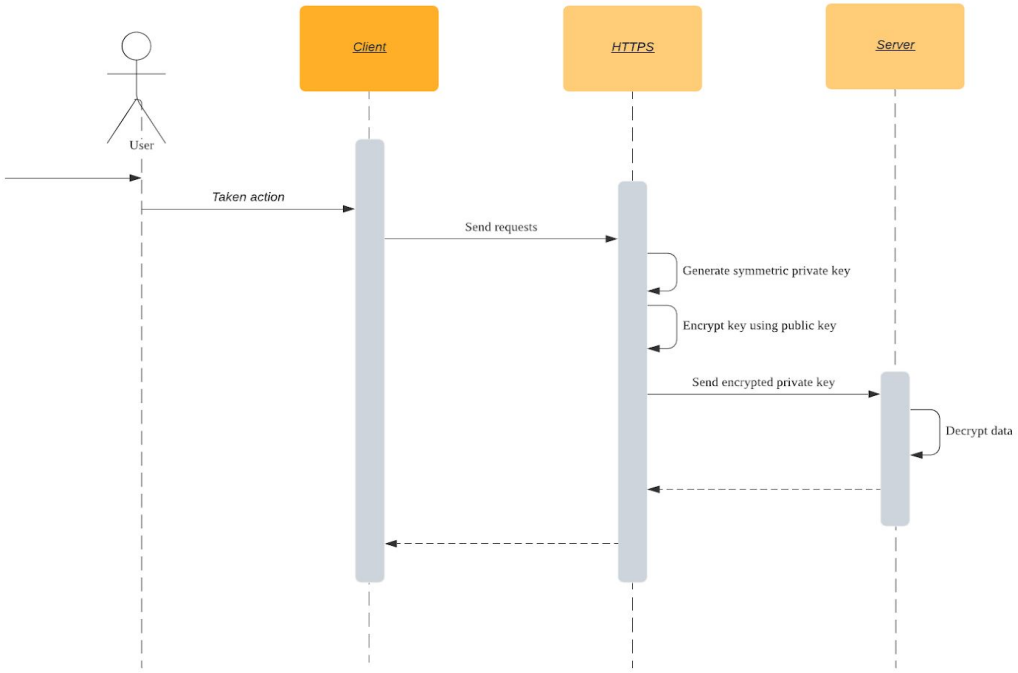

Figure 4.3. Sequence diagram

The figure above shows the process of transmission of data between client and server using a secure pipe. The adding of a pipe would not affect the performance of the system as the application behaves like the pipe does not exist. The data transmission is ensured to be secure as standard secure protocols are used for encryption and decryption.

5. Deployment to Heroku

Link to Heroku deployed app: swen90007-magic-pigeons.herokuapp.com

6. Test URLs

6.1 Login and Logout

Users login

- /login

Users logout

- /logout

6.2 URLs for admin

Viewing all subjects details:

- /Admin/subjects.jsp

6.3 URLs for instructors

Viewing all subjects:

- /Instructor/instructorSubjects.jsp

Viewing all exams for a particular subject:

- /Instructor/instructorExams.jsp?subject_code=[...]

Marking submissions:

- /Instructor/submissions_table?subject_code=[...]
- /Instructor/submissions_detail?examID=[...]&userID=[...]

6.4 URLs for students

Viewing all exams:

- /Student/studentHomePage.jsp

Viewing exam details

- /Student/studentViewExams.jsp?studentID=[...]&exam_id=[...]

Taking and submitting exams:

- /Student/studentTakeExams.jsp?studentID=[...]&exam_id=[...]
- /Student/studentAnswerQuestions.jsp?studentID=[...]&exam_id=[...]&question_index=[...]
- /Student/studentSubmitExams.jsp?studentID=[...]&exam_id=[...]

7. Test Data

7.1 User account data: <users>

Description:
This table corresponds to the users table in the database which stores all information related to the users. Correct usernames and passwords need to be used to login into the system.
Changes made in Part 3:
The passwords are changed to the hashed values of the original passwords..

user_id	user_type	name	username	password
0	ADMIN	Administrator	admin	**SHA256 hash for “admin”**
1	INSTRUCTOR	Eduardo Oliveira	eduardo	**SHA256 hash for “000000”**
2	INSTRUCTOR	Maria Rodriguez Read	maria	**SHA256 hash for “000000”**
713551	STUDENT	Jiayu Li	jiayul3	**SHA256 hash for “111111”**
904601	STUDENT	Simai Deng	simaid	**SHA256 hash for “111111”**
1049166	STUDENT	Yiran Wei	yirwei	**SHA256 hash for “111111”**

Table 7.1.1 Testing data for users

7.2 Subjects data: <subjects>

Description:
This table corresponds to the subjects table in the database that stores the data of all subjects. Users of ADMIN type are allowed to add new entries to this table and add instructors to existing subjects.

subject_code	name
SWEN90007	Software Design and Architecture
SWEN90009	Software Requirement Analysis

Table 7.2.1 Testing data for subjects

7.3 Association table of users and subjects: <users_has_subjects>

Description:
This table corresponds to the users_has_subjects table in the database which stores the subjects that each user is involved in (instructors teaching subjects or students enrolled in subjects). It also stores marks and fudge points for students as embedded values.
Changes made in Part 3:
A version number is added.

user_id	subject_code	marks	fudge_points	version
1	SWEN90007	-1 (default)	0 (default)	1 (default)
2	SWEN90007	-1 (default)	0 (default)	1 (default)
713551	SWEN90007	-1 (default)	0 (default)	1 (default)
904601	SWEN90007	-1 (default)	0 (default)	1 (default)
1049166	SWEN90007	-1 (default)	0 (default)	1 (default)
1	SWEN90009	-1 (default)	0 (default)	1 (default)
713551	SWEN90009	-1 (default)	0 (default)	1 (default)
904601	SWEN90009	-1 (default)	0 (default)	1 (default)
1049166	SWEN90009	-1 (default)	0 (default)	1 (default)

Table 7.3.1 Testing data for users and subjects

7.4 Exams data: <exams>

Description:
This table illustrates the information regarding the exams. Note that only subject_code, exam_title

and exam_status are the rows that are actually stored in the exams table in the database. Other rows are for explanatory purpose only. Exam description is not shown in this table for simplicity.			
subject_code	**exam_title**	**exam_status**	**Number of Questions**
SWEN90007	Week 3 Quiz	Closed	2
SWEN90007	Week 5 Quiz	Published	1
SWEN90007	Final exam	Unpublished	1
SWEN90009	Mid-Sem exam	Published	3
SWEN90009	Final exam	Unpublished	1

Table 7.4.1 Testing data for exams

As per requirements, students can see the exams with published or closed status, and can only answer published exams. Instructors can edit any unpublished exam, publish any unpublished exam, close any published exam, mark any closed exam, and delete any exam that has no submission.

The default state of newly created exams is unpublished.

7.5 Submission data: <submission>

Description: This table illustrates the information regarding the exam submissions. Note that some rows are for explanatory purpose only and are not actually stored in the submissions table in the database.					
exam_id	**Exam Title**	**submission_time**	**is_marked**	**marks**	**fudge_points**
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)

Table 7.5.1 Testing data for submissions

For more details about the data, please refer to the SQL script in Appendix A.

8. Git Release Tag

Git repository: https://github.com/hedgehog7453/SWEN90007_2020_MagicPigeons

Git release tag: SWEN90007_2020_Part3_MagicPigeons

Appendix A. Database implementation

```
-------------------------------------------------------------------------------
--                                  subjects                                 --
-------------------------------------------------------------------------------
DROP TABLE subjects CASCADE;
CREATE TABLE subjects (
   subject_code VARCHAR(20) NOT NULL UNIQUE,
   name VARCHAR(100) NOT NULL,
   PRIMARY KEY (subject_code)
);

INSERT INTO subjects VALUES ('SWEN90007', 'Software Design and Architecture');
INSERT INTO subjects VALUES ('SWEN90009', 'Software Requirement Analysis');

-------------------------------------------------------------------------------
--                                   users                                   --
-------------------------------------------------------------------------------
DROP TYPE user_type CASCADE;
CREATE TYPE user_type AS ENUM ('ADMIN', 'STUDENT', 'INSTRUCTOR');

DROP TABLE users CASCADE;
CREATE TABLE users (
   user_id INT NOT NULL UNIQUE,
   type user_type NOT NULL,
   name VARCHAR(50) NOT NULL,
   username VARCHAR(50) NOT NULL UNIQUE,
   password VARCHAR(100) NOT NULL,
   PRIMARY KEY (user_id)
);

INSERT INTO users VALUES (000000, 'ADMIN', 'Administrator', 'admin',
'8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918');
INSERT INTO users VALUES (000001, 'INSTRUCTOR', 'Eduardo Oliveira', 'eduardo',
'91b4d142823f7d20c5f08df69122de43f35f057a988d9619f6d3138485c9a203');
INSERT INTO users VALUES (000002, 'INSTRUCTOR', 'Maria Rodriguez Read', 'maria',
'91b4d142823f7d20c5f08df69122de43f35f057a988d9619f6d3138485c9a203');
INSERT INTO users VALUES (904601, 'STUDENT', 'Simai Deng', 'simaid',
'bcb15f821479b4d5772bd0ca866c00ad5f926e3580720659cc80d39c9d09802a');
INSERT INTO users VALUES (713551, 'STUDENT', 'Jiayu Li', 'jiayul3',
'bcb15f821479b4d5772bd0ca866c00ad5f926e3580720659cc80d39c9d09802a');
INSERT INTO users VALUES (1049166, 'STUDENT', 'Yiran Wei', 'yirwei',
'bcb15f821479b4d5772bd0ca866c00ad5f926e3580720659cc80d39c9d09802a');

-------------------------------------------------------------------------------
--                            users_has_subjects                             --
-------------------------------------------------------------------------------
```

```
DROP TABLE users_has_subjects CASCADE;
CREATE TABLE users_has_subjects (
   user_id INT NOT NULL REFERENCES users(user_id),
   subject_code VARCHAR(20) NOT NULL REFERENCES subjects(subject_code),
   fudge_points FLOAT DEFAULT 0,
   marks FLOAT DEFAULT -1,
   version INT NOT NULL DEFAULT 1
);

INSERT INTO users_has_subjects VALUES(000001, 'SWEN90007', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(000001, 'SWEN90009', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(000002, 'SWEN90007', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(904601, 'SWEN90007', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(904601, 'SWEN90009', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(713551, 'SWEN90007', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(1049166, 'SWEN90007', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(713551, 'SWEN90009', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO users_has_subjects VALUES(1049166, 'SWEN90009', DEFAULT, DEFAULT,
DEFAULT);

-------------------------------------------------------------------------------
--                                  exams                                    --
-------------------------------------------------------------------------------
DROP TYPE exam_status CASCADE;
CREATE TYPE exam_status AS ENUM ('UNPUBLISHED', 'PUBLISHED', 'CLOSED');

DROP TABLE exams CASCADE;
CREATE TABLE exams (
   exam_id SERIAL NOT NULL UNIQUE,
   subject_code VARCHAR(20) REFERENCES subjects(subject_code),
   title VARCHAR(100) NOT NULL,
   description VARCHAR(200) NOT NULL,
   status exam_status NOT NULL DEFAULT 'UNPUBLISHED',
   PRIMARY KEY (exam_id)
);

INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Week 3 Quiz', 'A quiz about data
source layer', 'CLOSED');
```

```
INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Week 5 Quiz', 'A quiz about object to relational structural patterns', 'PUBLISHED');
INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Final exam', '2020S2 Final exam of Software Design and Architecture (60% of total)', DEFAULT);
INSERT INTO exams VALUES (DEFAULT, 'SWEN90009', 'Mid-Sem exam', '2020S1 Mid semester exam', 'PUBLISHED');
INSERT INTO exams VALUES (DEFAULT, 'SWEN90009', 'Final exam', '2020S1 Final exam of Software Requirement Analysis', DEFAULT);


-------------------------------------------------------------------------------
--                                 questions                                 --
-------------------------------------------------------------------------------
DROP TYPE question_type CASCADE;
CREATE TYPE question_type AS ENUM ('MULTIPLE_CHOICE', 'SHORT_ANSWER');

DROP TABLE questions CASCADE;
CREATE TABLE questions (
   exam_id INT REFERENCES exams(exam_id) ON DELETE CASCADE,
   question_number INT NOT NULL,
   question_type question_type NOT NULL,
   title VARCHAR(45) NOT NULL,
   description VARCHAR(500) NOT NULL,
   marks FLOAT NOT NULL,
   PRIMARY KEY (exam_id, question_number)
);

INSERT INTO questions VALUES (1, 1, 'SHORT_ANSWER', 'Question 1', 'What is the object that wraps a row in a DB table or view, encapsulates the DB access, and adds domain logic on that data?', 50);
INSERT INTO questions VALUES (1, 2, 'MULTIPLE_CHOICE', 'Question 2', 'What is the layer of software that separates the in-memory objects from the database?', 50);

INSERT INTO questions VALUES (2, 1, 'SHORT_ANSWER', 'Question 1', 'How to structurally map our domain objects to a relational database?', 100);

INSERT INTO questions VALUES (3, 1, 'SHORT_ANSWER', 'Question 1', 'What does unit of work do?', 100);

INSERT INTO questions VALUES (4, 1, 'SHORT_ANSWER', 'Question 1', 'What is software engineering?', 20);
INSERT INTO questions VALUES (4, 2, 'SHORT_ANSWER', 'Question 2', 'What is software requirements analysis?', 50);
INSERT INTO questions VALUES (4, 3, 'MULTIPLE_CHOICE', 'Multiple Choice 1', 'Choose the WRONG statement.', 30);

INSERT INTO questions VALUES (5, 1, 'SHORT_ANSWER', 'Question 1', 'What is requirement elicitation?', 100);
```

```
-------------------------------------------------------------------------------
--                                  choices                                  --
-------------------------------------------------------------------------------
DROP TABLE choices CASCADE;
CREATE TABLE choices (
   exam_id INT,
   question_number INT,
   choice_number INT NOT NULL,
   choice_description VARCHAR(200) NOT NULL,
   PRIMARY KEY (exam_id, question_number, choice_number)
);

INSERT INTO choices VALUES (1, 2, 1, 'Table data gateway');
INSERT INTO choices VALUES (1, 2, 2, 'Row data gateway');
INSERT INTO choices VALUES (1, 2, 3, 'Active record');
INSERT INTO choices VALUES (1, 2, 4, 'Data mapper');

INSERT INTO choices VALUES (4, 3, 1, 'Software engineering is meaningless.');
INSERT INTO choices VALUES (4, 3, 2, 'Software engineering is the systematic
application of engineering approaches to the development of software.');

-------------------------------------------------------------------------------
--                                submissions                                --
-------------------------------------------------------------------------------
DROP TABLE submissions CASCADE;
CREATE TABLE submissions (
   exam_id INT REFERENCES exams(exam_id),
   user_id INT REFERENCES users(user_id),
   submission_time TIMESTAMP NOT NULL,
   is_marked BOOLEAN NOT NULL DEFAULT FALSE,
   marks FLOAT DEFAULT -1,
   fudge_points FLOAT DEFAULT 0,
   PRIMARY KEY (exam_id, user_id)
);

INSERT INTO submissions VALUES (1, 904601, '2020-09-28 01:00:00', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO submissions VALUES (1, 713551, '2020-09-28 01:00:00', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO submissions VALUES (1, 1049166, '2020-09-28 01:00:00', DEFAULT,
DEFAULT, DEFAULT);

-------------------------------------------------------------------------------
--                            submitted questions                            --
-------------------------------------------------------------------------------
DROP TABLE submitted_questions CASCADE;
```

```
CREATE TABLE submitted_questions (
   exam_id SERIAL REFERENCES exams(exam_id),
   user_id INT REFERENCES users(user_id),
   question_number SMALLINT NOT NULL,
   question_type question_type NOT NULL,
   choice_number SMALLINT DEFAULT null,
   short_answer VARCHAR(500) DEFAULT null,
   is_marked BOOLEAN NOT NULL DEFAULT FALSE,
   marks FLOAT DEFAULT -1,
   PRIMARY KEY (exam_id, user_id, question_number)
);

INSERT INTO submitted_questions VALUES (1, 904601, 1, 'SHORT_ANSWER', DEFAULT,
'Active record', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 904601, 2, 'MULTIPLE_CHOICE', 3,
DEFAULT, DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 713551, 1, 'SHORT_ANSWER', DEFAULT,
'Active record', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 713551, 2, 'MULTIPLE_CHOICE', 2,
DEFAULT, DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 1049166, 1, 'SHORT_ANSWER', DEFAULT,
'Data mapper', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 1049166, 2, 'MULTIPLE_CHOICE', 4,
DEFAULT, DEFAULT, DEFAULT);

-------------------------------------------------------------------------------
--                                  locks                                    --
-------------------------------------------------------------------------------
DROP TABLE exam_locks CASCADE;
CREATE TABLE exam_locks (
   lockable INT NOT NULL UNIQUE ,
   owner VARCHAR(50) NOT NULL ,
   PRIMARY KEY (lockable)
);

DROP TABLE submission_locks CASCADE;
CREATE TABLE submission_locks (
   exam_id INT NOT NULL,
   student_id INT NOT NULL,
   owner VARCHAR(50) NOT NULL ,
   PRIMARY KEY (exam_id, student_id)
);
```