Part 3 Architecture Document

SWEN90007: Software Design and Architecture

The Team
SWEN90007 SM2 2020 Project
Github: https://github.com/emilylm/swen90007-project.git
Heroku: https://swen90007-exam-app.herokuapp.com

In charge of: Emily Marshall <emilylm@student.unimelb.edu.au>, emilylm, 587580
Luke Rosa <lrosa@student.unimelb.edu.au>, lrosa, 319522

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
14/10/2020	01.00-D01	Initial draft of part 3 architecture document	Emily Marshall Luke Rosa
31/10/2020	01.00-D02	Added authentication, authorisation, secure pipe discussions	Emily Marshall Luke Rosa
31/10/2020	01.00-D03	Added concurrency discussion	Emily Marshall Luke Rosa
01/11/2020	01.00-D04	Final review of this document	Emily Marshall Luke Rosa
01/11/2020	01.00	Final version of this document	Emily Marshall Luke Rosa

Table of contents

1. Introduction

This document specifies the SWEN90007 project system architecture describing its main standards, module, components, frameworks, and integrations.

1.1 Proposal

The purpose of this document is to give, in high level overview, a technical solution to be followed, emphasizing the components and frameworks that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target users

This document is aimed at the project team and teaching team of SWEN90007, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

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
Component	Reusable and independent software element with well-defined public interface, which encapsulates numerous functionalities and which can be easily integrated with other components.
Module	Logical grouping of functionalities to facilitate the division and understanding of software.

1.4 Glossary of synonyms

Synonyms
Marks, points, grades
Examination Application, system

2. Architectural representation

The specification of the system's architecture Examination Application follows the framework "4+1"[1], which defines a set of views, as shown in Figure 1. Each of these views approaches aspects of architectural relevance under different perspectives:

- The **logical view** shows the significant elements of the project for the adopted architecture and the relationship between them. Between the main elements are modules, components, packages and the application main classes;
- The **process view** shows the concurrency and synchronization aspects of the system, mapping the elements of the logical view to processes, *threads* and execution tasks;
- The **development view** focuses on aspects relating to the organization of the system's source code, architectural patterns used and orientations and the norms for the system's development;
- The **physical view** shows the hardware involved and the mapping of the software elements to the hardware elements in the system's environment.
- The **scenarios** show a subset of the architecturally significant use cases of the system.

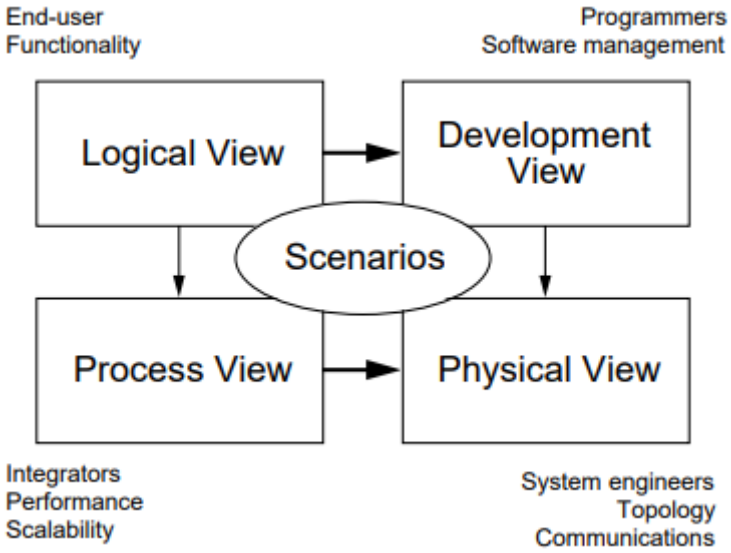


Figure 1. Views of *framework* "4+1"[1]

3. Architectural objectives and restrictions

The defined architecture's main objective is to develop a system that fully meets all functional requirements of the application specification (see specification document).

3.1 Qualities

The qualities of service of the Examination Application cover performance, tolerance to imperfections, usability, security, maintainability, portability, and concurrency.

- The system is expected to perform to a level to guarantee its usability – there are no strict performance timeframes that have placed on the system but the general qualities are that the application should feel responsive and not freeze nor crash.
- The system should also have a tolerance for imperfections. Wherever possible, the application should be tolerant of non-well-formed inputs and should provide feedback to the user.
- Data integrity should be guaranteed by the system – data stored should be accurate, complete, and reliable.
- Business rules for the business domain should be followed – for example, administrators should not be able to register in classes.
- The application is expected to guarantee the security of the system – all users have been given a role which defines the system components they have access to and the transactions they may perform. User passwords should also be encrypted in the database and not stored in plaintext, as per best practice.
- There are no strict usability requirements but the user interface and presentation layer should enable (and not impede) users' use of the application and should be clearly laid out so user's know how to operate the application.
- The application should be maintainable and extendible in future should the need arise. It should also be portable and accessible from any desktop computer and web browser.
- The application should support concurrency where users can perform actions on the same database object without these transactions violating ACID compliance for databases.

- The system should support high cohesion and low coupling – the responsibilities of an element of the system should be focused and strongly related and elements should be as independent as possible.

3.2 Constraints

- As per the project specifications:

  All exams allow a single attempt, meaning students can only start and submit their exam once.

  This has been interpreted by the project team to mean students can start and submit an exam but does not account for situations where an exam is commenced by a student and the browser is closed or the application crashes prior to submission. In this case, no responses will be recorded and so no attempt is persisted in the database.

3.3 Principles

- To ensure a tolerance for imperfections, the system has been developed with error handling so will prompt the user when inputs are not well formed. Malformed inputs include inputs that do not meet database requirements (for example, required fields must be inputted prior to querying the database) or conflict with business domain rules (for example, an exam must contain one or more questions).

3.4 Requirements of architectural relevance

This section lists the requirements that have impact on the system's architecture and the treatment given to each of them.

Requirement	Impact	Treatment
Security	The application must implement basic security behaviours: • Authentication: login using university credentials and a password; • Authorisation: according to their role, users must be granted permission to perform some specific actions (e.g. instructors can create exams); • Passwords must be encrypted on the database and not stored as plain-text; and • Data sent across the network must not be modified by a tier.	• Passwords have been persisted in the database. They have been stored as a hash, created using a salt with Spring Security. • User object has been created with a role attribute to store each user's role, which determines associated privileges. • User roles have also been persisted in the database.
Data persistence	Data persistence will be addressed by using a relational database.	A PostgreSQL relational database will be used to persist data.
Extensibility	The system should support future additions and changes.	The domain model pattern has been used to support this extensibility.
Concurrency	The system should support concurrency for certain system operations.	• In order to guarantee data integrity, concurrency patterns will be developed within the

		application making use of optimistic and pessimistic locking in conjunction with a database lock utilities class. • The use of a normalised relational database naturally supports concurrency in certain circumstances.

3.5 Reuse approach

In order to meet the architectural requirements of this system, a number of components will be reused, as they are commonly available and thoroughly tested, and a number of components will be developed by the project team.

Component	Development	Comments
Data persistence	Reuse	Utilise PostgreSQL.
Database driver	Reuse	Utilise JDBC.
Authentication	Reuse	Utilise Spring Security's authentication framework.
Authorisation	Reuse, Develop	Develop access control for different user tiers, based on roles stored in the database in conjunction with Spring Security's access-control framework.
Secure pipe	Reuse	To provide privacy and data integrity for communications between the server and client, Heroku's Transport Layer Security (TLS) will be used.
Session control	Reuse	Build on Java Servlet HttpSession.
Log	Reuse	Make use of Java's Logger for system logging.
Presentation layer	Reuse	Build on JQuery and JTSL.
Concurrency	Reuse, develop	A server-side database lock class will be developed to manage PostgreSQL's database locks.

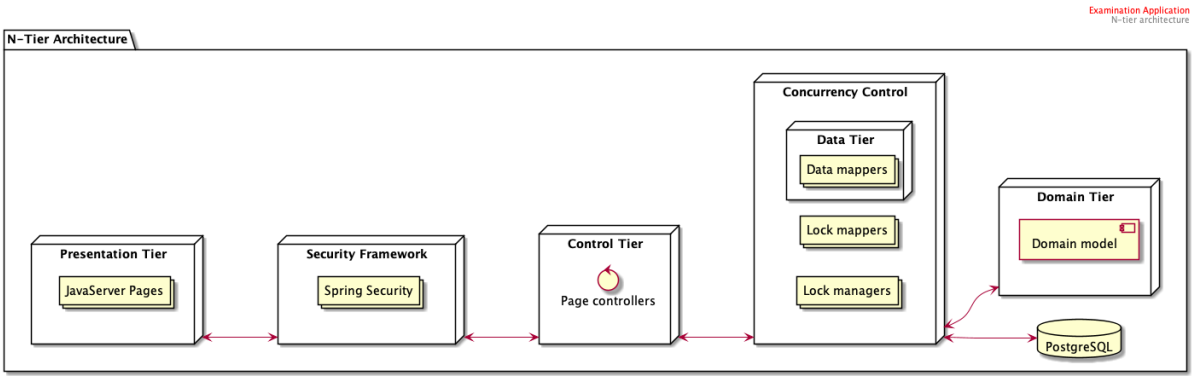
4. Logical view

This section shows the system's organisation from a functional point of view. The main elements, like modules and main components are specified. The interface between these elements is also specified.
The Examination Application is divided into layers based on N-tier architecture – it is based on a responsibility layering strategy that associates each layer with a particular responsibility. This strategy has been chosen for a number of reasons:

- It provides for data integrity as it isolates system responsibilities from one another;
- It allows for changes to one layer without impacting others, supporting extensibility; and
- The tiers allow the project team to encode finer grain security policies, especially policing roles (student, instructor, administrator) of users.

Each layer has specific responsibilities:

- The **presentation layer** deals with the presentation logic and rendering pages;
- The **control layer** manages the access to the domain layer;
- The **data layer** is responsible for the access to the PostgreSQL database;
- The **domain layer** is related to the business logic and manages the accesses to the resource layer;
- The **security layer** provides framework for authentication and authorisation;
- The **concurrency layer** manages all offline locks within the application; and
- The **lock mapper layer** handles the required acquisition and releasing of locks for each method in the data later.



In the presentation layer, Java Server Pages (JSP) have been created – this is a server-side technology that enables a dynamic, platform-independent method for building Web-based applications.
Through user input, the JSPs speak to the controllers which introduce servlet functions in the control layer. Through the use of servlet requests and responses, the servlets in the control layer pass on user requests to the data layer.
The data layer receives requests from the control layer, accesses the database and uses the returned data to instantiate domain objects.

4.1 Class diagrams

Since part 2 submission, the class diagram has seen a number of changes to manage authentication, authorisation, and concurrency:

- Security package has been created to centralise authorisation and authentication logic;
- Concurrency package has been added to handle all offline locks in the application; and

- Lock mappers package has been added which contains a layer of abstraction that provides implicit locking mechanisms around the data mapper components

Please refer to the appendix for all class diagrams.

5. Process view

5.1 Concurrency

5.1.1 Concurrency problems

There are a number of circumstances where the occurrence of two actions by two different system actors at the same time could lead to divergent system outcomes, impacting database consistency. The implementation of concurrency in the Examination Application is aimed at managing:

- Write-read conflicts, where a user reads uncommitted data;
- Write-write conflicts, where a user overwrites uncommitted data causing a lost update problem; and
- Read-write conflicts, where a user makes a non-repeatable read.

Given the sensitive nature of the Examination Application (a platform that stores exams, student grades, etc.), the system should be able to handle concurrency where it arises without causing data integrity issues.

5.1.2 Concurrency principles

The implementation of the concurrency patterns should:

- Not cause application transactions to malfunction; and
- Not cause lower throughput or bad response times than serial execution or where it does, it should be justified.

In selecting an appropriate concurrency strategy for each situation, the project team was guided by a number of principles, listed below.

5.1.2.1 Principle 1: The Examination Application must deliver acceptable user experience

In order to maintain usability and provide a good user experience on the Examination Application, application transactions that require a lot of user input or are time-consuming should adhere to degree 3 isolation, where the application pre-declares all resources that are needed and these resources are allocated in a single request prior to the user commencing the operation.

If all resources cannot be granted initially, the user should be warned they cannot attempt the transaction at this time. This saves a user spending significant time or energy inputting data into an operation to discover the data cannot be persisted only once the transaction is complete.

Using this principle for implementation across all concurrency strategies would reduce the liveliness of the application so is only used in situations where anything less than degree 3 isolation would worsen user experience to an intolerable level. Where possible, optimistic locks will be used to support the liveliness of the application.

5.1.2.2 Principle 2: No locking strategy where concurrency is naturally supported or business requirements do not require it

The database schema has been created with the use of surrogate keys in a number of situations to avoid concurrency issues – for example, exams for a subject are persisted in the database with a surrogate key. This means there is no need for concurrency support for multiple instructors when creating an exam for the same subject.
Given the relational database of the system has been normalised, this can naturally support concurrency in certain situations and so does not require a locking strategy.
Given business requirements stipulate there is to be only one administrator, no concurrency patterns have been implemented for administrator use cases.

5.1.2.3 Principle 3: Deadlocks must be avoided or managed

The application prevents deadlocks by ensuring that all locks belonging to a session are released at the beginning of any new business transaction. Additionally, by throwing exceptions when a lock cannot be acquired, rather than waiting for locks to become available, the application reduces the risk of processes indefinitely waiting for a lock. Lastly, when a business transaction requires multiple locks, all locks are acquired at the beginning of the transaction, and if any one lock cannot be acquired: all locks are released. This prevents deadlocks, where two processes cannot progress until the other releases a lock.

5.1.3 Concurrency patterns

The Examination Application has implemented control mechanism patterns for dealing with the issues described above. The implemented solutions discussed also carry their own problems, which are discussed below.

5.1.3.1 Offline concurrency control

The database chosen for the project, PostgreSQL, has inbuilt concurrency control that prevents inconsistent reads and lost updates on shared data. However, in the project application, there are several instances where business transactions span multiple database transactions. Since the database is unaware of anything beyond the scope of a database transaction, in these cases, the application itself needs to handle conflicts between competing business transactions.
The project team considered changing the scope of system transactions to leverage the concurrency control in the underlying database. It could do this by implementing:

1. Late transactions; and
2. Long transactions

Long transactions involve keeping a database transaction open across the larger business transaction. In doing so, the application can leverage the concurrency control measures in the underlying database. A pitfall of this approach, however, is that it severely limits the scalability of an application. Typically, databases do not handle long transactions well, and the number of database connections that can remain open simultaneously is limited. In the case of the Heroku-deployed project application, only 20 database connections are permitted at one time. Considering that some business transactions, such as creating an exam, are likely to take a long time, the limitations of using a long transaction were deemed undesirable.
Late transactions are an approach which again combine multiple system transactions into one single system transaction. In this case, however, system transactions are delayed until the end of the business transactions, when they are committed in one batch. This prevents the scalability constraints of having multiple transactions open at once. However, it leaves the application susceptible to inconsistent reads.

Both long transactions and late transactions present significant limitations which were considered undesirable for this project. It was decided that either implementation was insufficient to meet the business requirements of the system. Offline concurrency measures were necessary to achieve more appropriate level of concurrency and data integrity, detailed below.

5.1.3.2 Optimistic Offline Locking

Optimistic locking is an approach which detects conflicts and rolls back business transactions when they occur. It essentially checks whether a business transaction is acting upon the most up-to-date version of data before committing a transaction. If it is not, the update is abandoned. This avoids lost updates – a business transaction will not succeed in updating the data, if the data has been updated since they last retrieved it. However, it does not prevent inconsistent reads. It also results in lost work whenever a conflict occurs.
Therefore, optimistic locking should not be used if conflicts are expected to occur frequently, or if the impact of abandoning a business transaction is significant.

5.1.3.3 Pessimistic Offline Locking

Pessimistic locking works to mitigate the impact of conflicts between concurrent business transactions by avoiding them altogether. Only one business transaction is permitted to access the shared data at a time, ensuring that conflicts will not occur when the transaction is finally committed.
The side effects of pessimistic locking include reduced liveliness in the system. Naturally, if a system enforces some level of exclusive access to a shared resource, then concurrent business transactions may be forced to wait. In this way, progress towards their goals will be slowed. How significantly pessimistic locking will effect liveliness depends on the type of lock chosen – i.e. whether a business transaction needs to acquire a lock to read, or just update data. Implementing pessimistic locking is also a non-trivial task and is prone to developer error. In short, it should be avoided where it is unnecessary.
In essence, optimistic locking favours liveliness over data consistency and integrity, whilst pessimistic locking favours the opposite. In order to reach an appropriate balance between these features, the project team leveraged both approaches. Our specific implementation for each use case is detailed below.

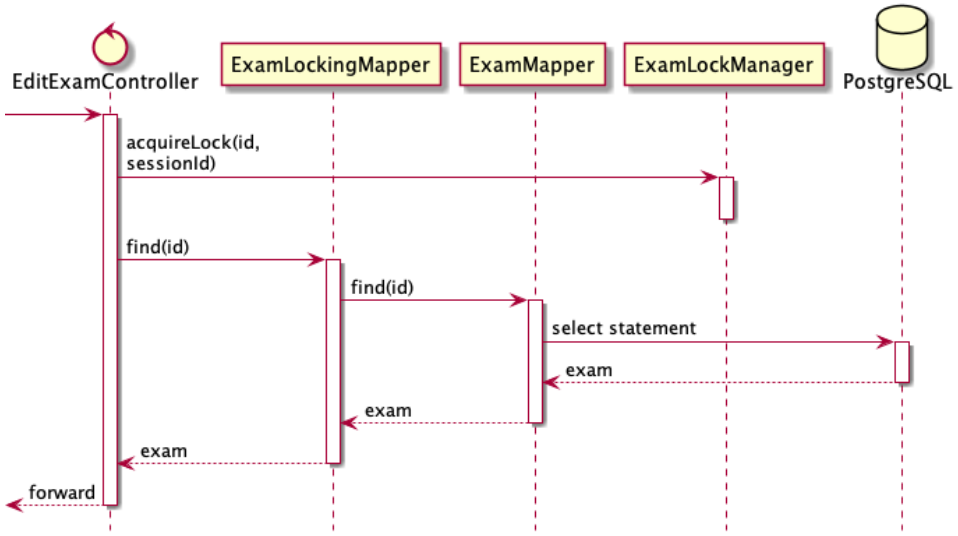
5.1.3.4 Implicit Lock Pattern

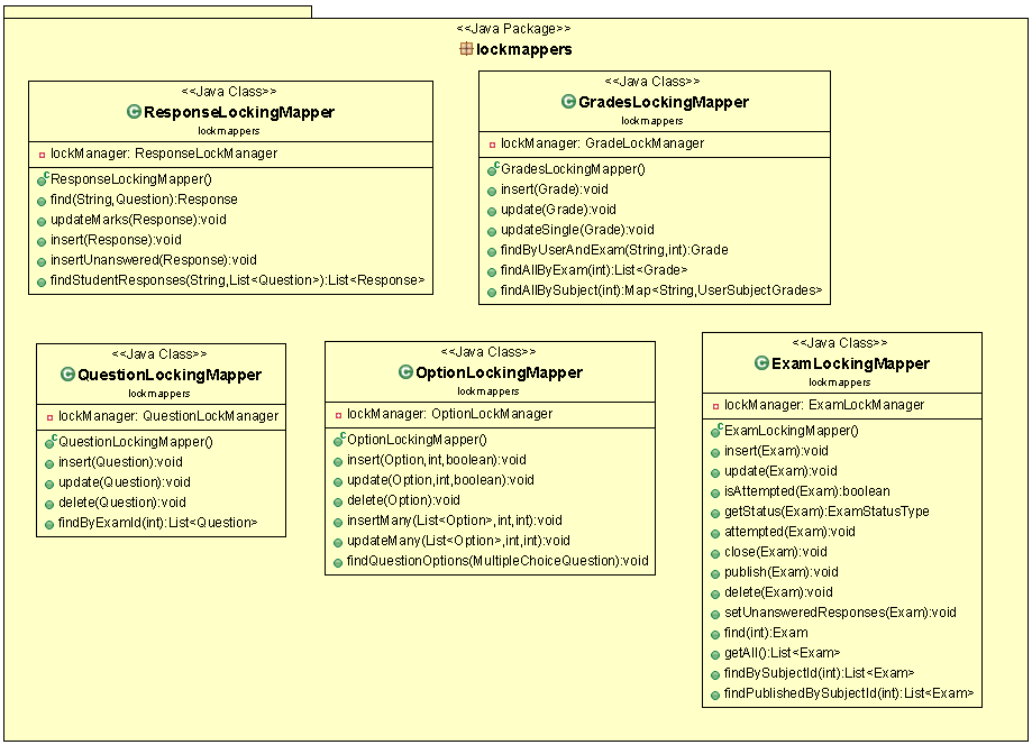
In order to adhere to the principles of low coupling and high cohesion discussed as architectural goals of the Examination Application, an implicit lock pattern has been utilised for handling the locking logic in support of concurrency.
The use of optimistic and pessimistic offline locks introduces several problems:

- The locking mechanism is tightly coupled with the domain logic;
- The locking logic is spread across the domain layer; and
- Locking objects relies on developers to implement – developers may forget to do it or implement it in many different ways. Forgetting to acquire a lock could result in data integrity issues while forgetting to release the lock could adversely impact the liveliness of the Examination Application.

To ensure data integrity and increase application liveliness, the Examination Application has utilised the implicit lock pattern. The use of an implicit lock pattern supports low coupling and high cohesion, as the locking mapper decouples the domain logic from the locking mechanism and concentrates the locking logic in a single place (rather than being spread across the application layers).

The logic of offline locks has been abstracted into a *lockmappers* package which consists of classes for wrapping the associated data mappers – this introduces a mapper layer, which sits between the domain and data source layers. The mapper layer ensures that a lock is correctly acquired and released whenever a lock is used to access data.
The use of an implicit lock minimises the responsibility of transactions when acquiring and releasing locks, which in turn reduces the responsibility of developers to correctly handle locking whenever data is accessed. This ensures that the Examination Application correctly handles instances that could cause data integrity or liveliness issues.
The use of implicit lock pattern can be illustrated by the below sequence diagram – using an example of an instructor who wishes to make updates to an exam in the system:





Class diagram for the implementation of the implicit lock pattern

5.1.4 Concurrency implementation

Locks are managed using a concurrent hash map, which facilitates thread safe acquisition and release of its lock objects. A separate lock manager was created for every shared resource in the application, since having one single lock table would create a single point of contention for all locks, and may cause a performance bottle neck as the system scales. Any data structure that enforces synchronised access will impact performance in an application, however keeping lock managers separate will at least keep this impact minimal.

The application uses the ID of the *HttpSession* object to identify processes and distinguish who is requesting, and who has access to, particular locks. In order to make the session ID available across all layers rather than just our controllers, and to avoid repeatedly sending this data around, two classes *AppSession* and *AppSessionManager* were created. *AppSession* simply stores the ID of the current session, and *AppSessionManager* stores *AppSession* in a ThreadLocal() which is available only to the calling thread. Since each servlet request executes in a different thread, a request simply sets a new 'current' thread in *AppSessionManager* holding an *AppSession* object, and this is then available for any subsequent code that is executed for this request.

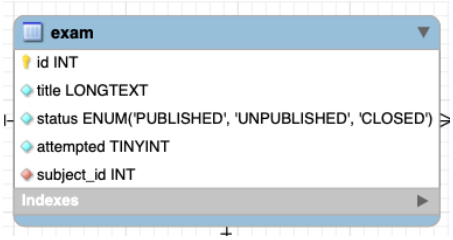
Lockable objects are identified using their unique database IDs. In the case of objects such as *responses* objects, rows have a composite key over *exam_id* and *user_id*. Since the key is unique over these two attributes, the lock table uses the concatenation of these two values as strings, to form a single unique key.

5.1.4.1 Multiple instructors can create exams for the same subject simultaneously

As per the project specification, a concurrency conflict could arise where more than one instructor attempts to create an exam at the same time for the same subject.

Implementation

A database table for exams has been created which uses, as a primary key, a surrogate key:



The surrogate key is an integer created by the database at the time of insertion into the exams database table. This mitigates any concurrency issues and means the Examination Application can handle as many instructors creating exams for a subject simultaneously.

Design rationale

This method was chosen for a number of reasons:

1. The cost of conflict would be high – if an optimistic locking strategy had been used, it would create bad user experience where an instructor spends a significant amount of time entering data without it being persisted having to repeat entry from the beginning.

2. The chance of conflict would likely have been low so would have had required a substantial amount of work implementing a pessimistic locking strategy in order to avoid the cost of conflict for something with marginal probability of occurring.
3. It supports much greater liveliness than any other alternative.
4. This method fully supports the business requirements stipulating that multiple instructors can create an exam for the same subject simultaneously.

Alternatives considered

It was considered to use a combination of exam title and subject ID as the primary key of an exam. However, this was decided against as it would require greater key mapping from the domain level to the database and would have required conflict management where an instructor edits the title of an exam. Managing concurrency through the use of a database-generated surrogate key is a simpler and more intuitive approach.

5.1.4.2 Multiple instructors can update the same exam simultaneously

As per the project specification, a concurrency conflict could arise where more than one instructor attempts to update the same exam at the same time.

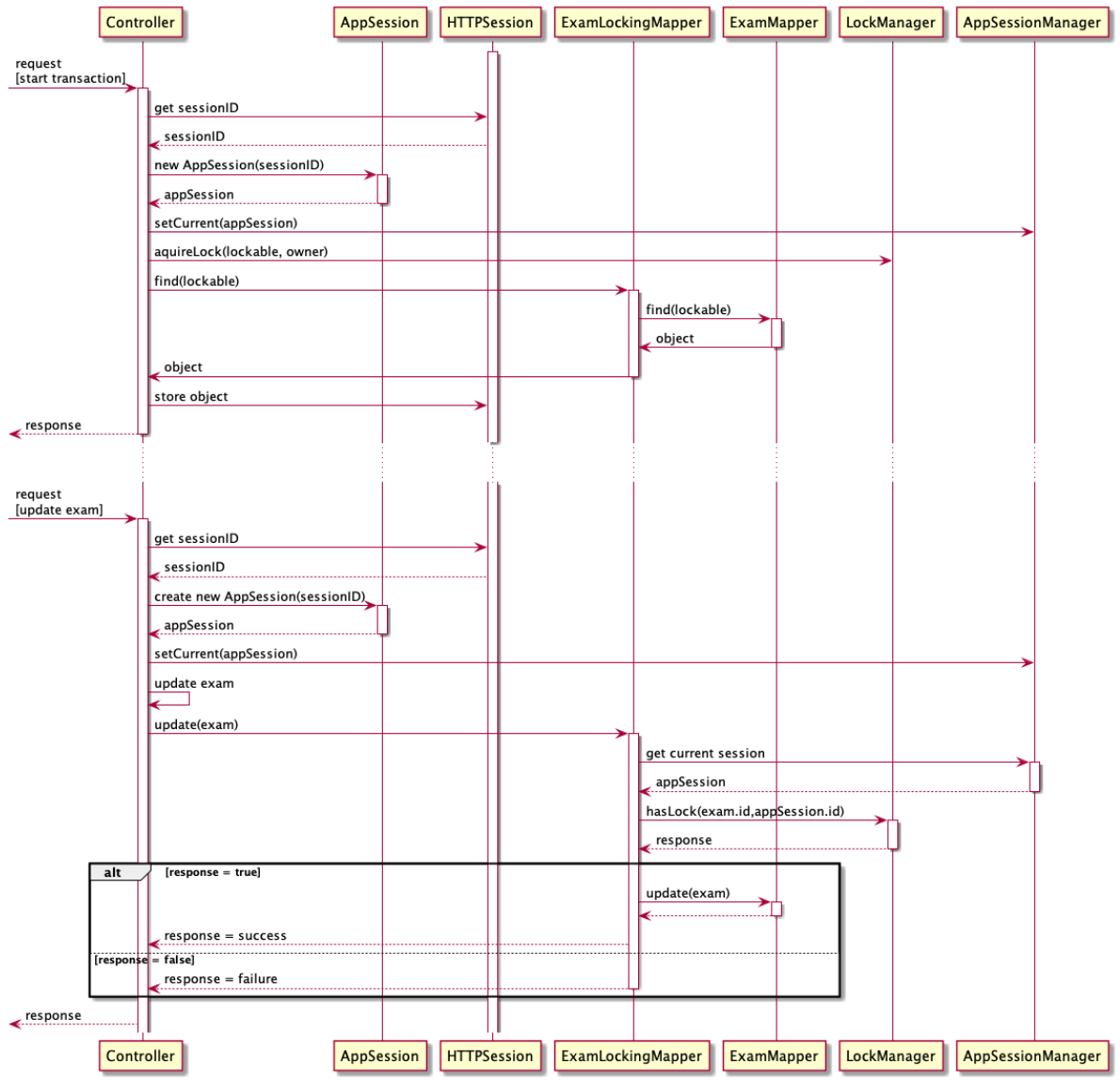
Exam updates have been handled using pessimistic offline write locks.

Implementation

When an instructor selects *Edit* on an exam, the system attempts to obtain an exclusive write lock over the exam record, as well as any question records and option records that it contains. If it fails to obtain any of these locks, then the transaction fails and any locks acquired by the transaction thus far are released. An exception is raised and the system alerts the user that they cannot update the exam at this time as it is currently being modified by another user. It was decided that throwing an exception was more appropriate than having transactions wait for the lock to become available, since updating an exam may take a long time. This also better avoids deadlocks and having to handle timeouts within the system.

The exclusive write lock permits other instructors or students to read the exam freely, but does not permit them to update it. A compromise of the improved liveliness afforded by a write lock, is that it does not prevent inconsistent reads. This meant that occasionally, the data in the view may be out of date. This was considered acceptable, so long as users were not performing actions on that data, unaware of recent changes. To prevent this latter situation, but still maintain appropriate liveliness, the project team implemented an optimistic check to make sure that data had not become stale before trying to perform actions on it. . Furthermore, there is only a limited window where students can see exams that might be updated – between its publishing, and the first time it is attempted. It will be reasonably rare that data read would become stale.

The following sequence diagram demonstrates the flow of events when a business transaction attempts to update an exam:



Design rationale

The use of pessimistic locking here restricts access to the 'Edit' exam option to one instructor at a time, as to prevent the situation where two instructors are updating an exam simultaneously and one loses work – either through lost updates or a last-minute rollback of their transaction. Since editing an exam will usually require significant user input, discarding this is deemed unacceptable.

The reduced liveliness that is inherent to pessimistic locking was considered a worthwhile trade off here. While it may be inconvenient for an instructor to have to wait for another instructor to finish updating an exam, this is more desirable than losing all updates to an exam. It is not detrimental to the goals of the system to allow only one instructor can update a single exam at a time.

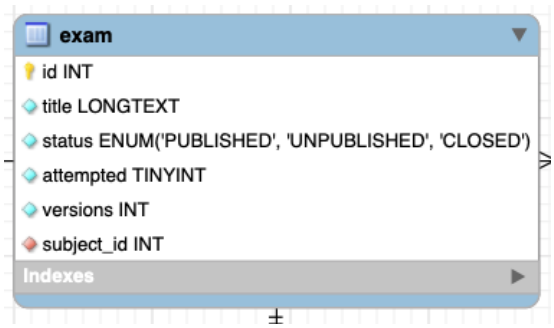
In order to minimise the impact of the pessimistic lock on liveliness in the system, a write lock was chosen. This meant that data could be freely read, and a lock was only required when a business transaction tried to update the data. This was important since Exam records are a heavily accessed resource in the application by all users. Requiring an exclusive read

lock would limit our application to the extent that it would act more like a single user system than a concurrent one. A read / write lock would afford better liveliness, however, it was still considered unnecessary and would add a huge amount of contention to the lock table.

The use of a write lock could introduce non-repeatable reads of the exam object in the following scenario:

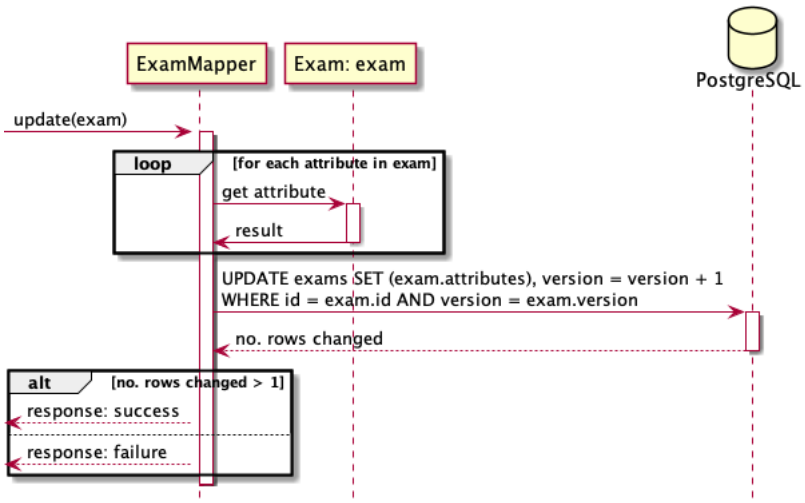
1. Instructor 1 or student views the exams table of a subject at which point the Examination Application loads these exams into in-app memory;
2. Instructor 2 edits an exam of that subject that instructor 1/student are viewing; and
3. Instructor 1/student attempts to edit the same exam Instructor 2 just updated.

Any action Instructor 1 or student attempt to take is an action on an outdated exam resulting in a non-repeatable read – this could be a significant issue for the Examination Application, as it could result in students attempting to take outdated exams. In order to avoid this scenario, the Examination Application persists the version number of exams in the database and any update to an exam iterates the version number:



When a user (in this example, instructor 1 or student) attempt to update an exam, the system compares the version number of the exam loaded into in-app memory with the version number of that same exam in the database. If these version numbers differ, the system rejects the user's attempt to update the exam and warns the user that there has been an update to that exam. It then reloads the new exam from the database and the user is free to try the update operation again.

The following sequence diagram demonstrates this flow of events:



Alternatives considered

The project team considered a number of alternative approaches, including a read or read/write lock on 'exams', and optimistic locking. As previously discussed, the latter option was inappropriate due to the chance of lost work. A read lock was also inappropriate given the frequent use of the 'exam' resource in the application – making reads exclusive would drastically reduce liveliness.

A read/write lock on exams was considered, as this would improve liveliness over a 'read' lock, whilst also preventing inconsistent reads. However, the team decided the chosen implementation achieved similar results in terms of consistency, whilst avoiding the complexity of implementing a read/write lock *and* the performance constraints that may occur if the application needed to manage read accesses to a heavily used resource like exams.

5.1.4.3 Multiple instructors can enter marks for the same student cohort simultaneously (in the table view)

As per the project specification, a concurrency conflict could arise where more than one instructor attempts to update the marks for the same student cohort simultaneously via the table view.

Exam updates have been handled using pessimistic offline write locks.

Implementation

When an instructor tries to edit the grade for a particular student and exam in the table view, the system attempts to obtain an exclusive write lock over the grade object. If it fails to obtain any of these locks, then the transaction fails and the user is not permitted to update the mark. An exception is raised, and the system alerts the user that they cannot update the exam at this time as it is currently being modified by another user.

Failure to obtain a lock will occur either because another instructor is editing the same grade in the grade table, or because another instructor is editing the same exam in the detailed submission view.

Design rationale

Lost work was not a primary concern here, since marks are updated and saved individually in the marks table. However, when an instructor updates individual question marks via the detailed view, they also need to acquire a lock over grades (discussed in the next section). This lock is acquired at the beginning of the transaction.

To meet the needs of both these use cases, a pessimistic write lock was implemented on 'grades'. This did not have a significant impact on liveliness, because the write lock is acquired for a single student/exam grade only. Instructors can still update the grades for any other student/exam pair in the table.

To avoid inconsistent reads, the system loads the singular grade again each time a write lock is obtained, ensuring it is up to date.

Alternatives considered

The project team considered implementing an Optimistic locking technique to improve liveliness and prevent instructor being barred from trying to update the mark for the same student and same exam. Indeed, Optimistic locking is easier to implement and leaves less room for developer error than its pessimistic counterpart. However, it was identified that marking a detailed submission should involve obtaining a lock over a grade at the beginning

of that transaction. For this reason, having a pessimistic lock over grades was considered most appropriate. To mitigate the difficulty of implementing a pessimistic lock, the implicit lock pattern was leveraged.

Additionally, the team considered locking an entire exam column as it is being edited by an instructor. This was primarily considered because the marking table was initially implemented this way: an instructor would chose the 'edit' marks for an exam, at which point that column would become editable, and would then 'save' once all edits were made. Therefore, this implementation would have been easier for the team. However, the team decided that this involved locking an unnecessary number of objects and their corresponding data rows. It would mean that if an instructor was updating any single grade in the grade table, no other instructor could the grade of any other student for that exam, either via the table view or detailed submission. This was considered too detrimental to liveliness in the system.

5.1.4.4 Multiple instructors can enter marks for the same exam and the same student (in the detailed view) simultaneously

As per the project specification, a concurrency conflict could arise where more than one instructor attempts to update the marks for the same student and same exam via the table view.

Exam updates have been handled using pessimistic offline write locks.

Implementation

An instructor can mark an exam submission for a particular student and exam by clicking the 'details' icon from the grade table. At this point, the system attempts to obtain a write lock over the 'responses' for this student and exam, as well as the total 'grade' object. It requires the latter lock because saving any changes to the response marks requires updating the total grade.

If it fails to obtain any of these locks then the transaction fails and the user is not permitted to update the submission marks. An exception is raised and the system alerts the user that they cannot update the exam at this time as it is currently being modified by another user.

Failure to obtain a lock will occur either because another instructor is editing the grade for the same student and same exam in the grade table or in the detailed submission view.

Design rationale

Similarly to updating an exam, this business transaction involves user input that would be undesirable to lose. Repeating work will result in frustration and poor user experience, so it was decided that such conflicts should be prevented altogether.

Again, a write lock was implemented to improve liveliness in the application. Inconsistent reads were not considered a significant problem here.

Alternatives considered

The project team considered an Optimistic offline locking mechanism here, since this would require simpler implementation. However, this was considered undesirable due to lost work.

A read/write lock was also considered, but ultimately deemed unnecessary, for similar reasons as discussed for *exams* above. A read lock here was also considered, and would be a

less problematic choice here than for exams. This was considered a valid alternative, but the team decided it did not add value over a write lock, as occasionally inconsistent data in the view was not a significant problem. Therefore, the team chose the alternative which caused the least restriction on concurrent access and liveliness: the write lock.

6. Development view

6.1 Architectural patterns

6.1.1 Summary of architectural patterns

There were a number of patterns used in this project to support the architectural goals of the system. The patterns used are as follows:

- Authentication;
- Authorisation;
- Secure pipe; and
- Concurrency (discussed in section 5).

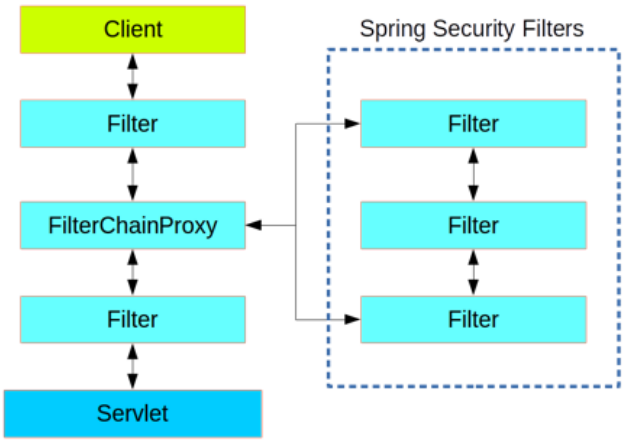
6.1.2 Spring Security

Implementation

Spring Security has been utilised by the Examination Application to implement:

- Authentication; and
- Authorisation (both discussed in detail in the next sections).

At its core, Spring Security acts as filters that sit between the servlets of the Examination Application and the users of the system. Spring Security is a single physical filter but delegates processing to a chain of internal filters:

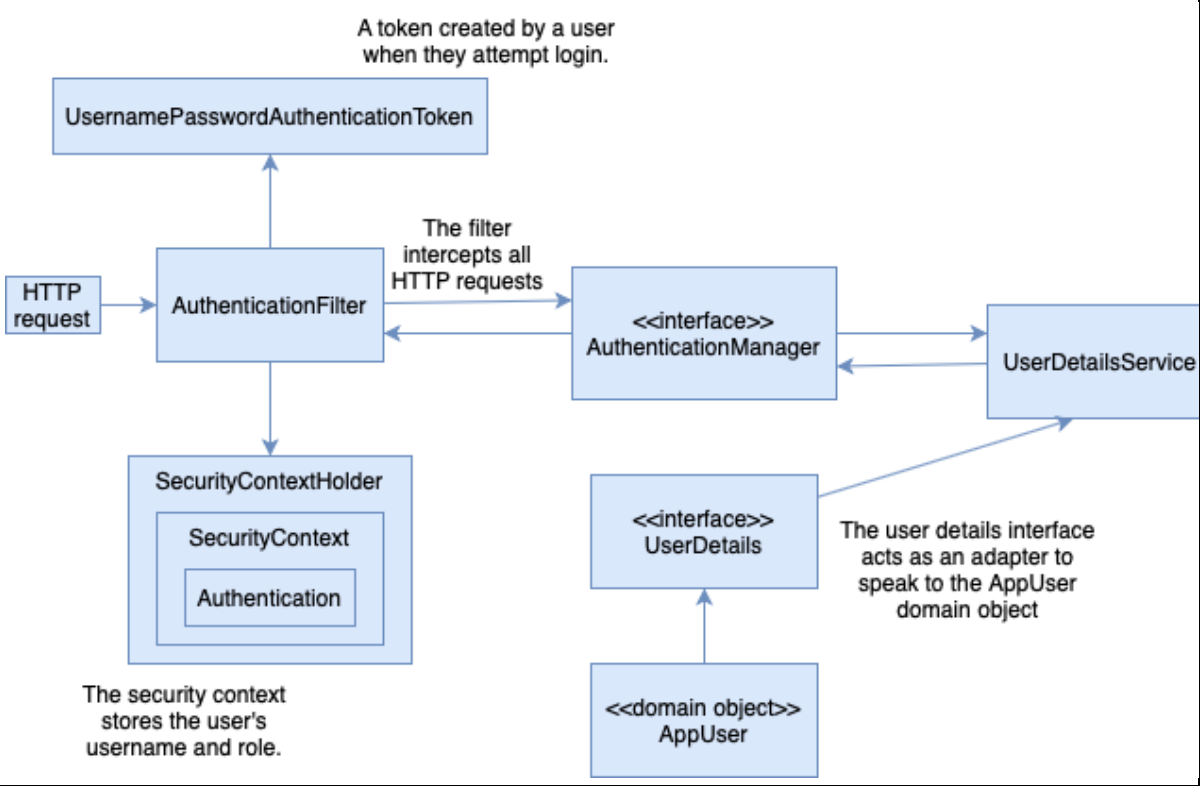


Spring Security filters[4]

These filters intercept any access to the Examination Application to ensure that the user has been authenticated and has the correct authorisation, as specified in the Examination Application's security configuration file. If a user is not logged in, all access to the servlet is filtered and users are redirected to the login page. If a user is logged in but lacks the correct

authorisation, the filter once again denies access to the servlet and presents a 403 forbidden message.

The implementation for authentication and authorisation in the Examination Application can be illustrated as:



Reasons for use

The use of a centralised framework reduced duplicate code and allowed the project team to avoid rewriting authentication logic in the presentation or business layers. The centralisation and lower coupling make it easier to change framework (for example, changing from Spring Security to another provider) – it is far simpler when using a centralised authentication enforcer to swap to another provider as there is less logic to replace throughout the other layers. It also supports high cohesion – each class in the security package has one responsibility and removes the responsibility from other classes to have to manage authentication.

Spring Security is a tried and tested framework – this means it is less likely to introduce vulnerabilities than if the project team had developed their own security implementation.

Alternatives considered

The project team considered several other mainstream security frameworks, such as Apache Shiro and Auth0. Spring Security was chosen because it facilitates both authentication and authorisation, as well as naturally supporting low coupling and ease of extensibility – it requires no further code duplication to extend the domain model or add new classes or methods in any layer. Spring Security is effectively a set of chain-filters that sit between the servlet and user and so naturally supports the n-tier design of the Examination Application. The use of auto-generated login pages also reduced work for the project team.

6.1.2.1 Spring Security authentication

In order to establish that the user of the Examination Application is who they say they are, the system must implement authentication. Authentication is a way for the system to be certain that the user requesting access is a registered and valid user of the system. Without this mechanism, it would be possible for anyone to gain access to the system and view or modify confidential information (for example, exam grading). This is undesirable for almost any application but particularly so for this application given the sensitive nature of data.

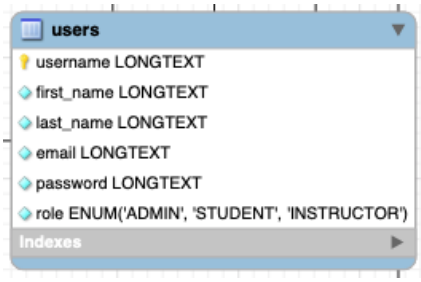
Authentication has been implemented using the Spring Security Framework and form-based authentication. In this implementation of the Examination Application, all users are human users – there are no external systems requiring access meaning that only a process for authenticating human users is required.

Implementation

The authentication mechanism requires two pieces of information from users:

1. A unique, semi-public university username used to establish a claim of the identity of the user; and
2. A private password to verify the claim of identity by the user.

These details are stored in the Examination Application's data store:



Users database table

There are four main scenarios in the Examination Application where Spring Security's framework for authentication has been utilised:

1. User registration;
2. Authentication; and
3. Logout
4. To prevent cross-site request forgeries

These are discussed with examples below.

1. *User registration*

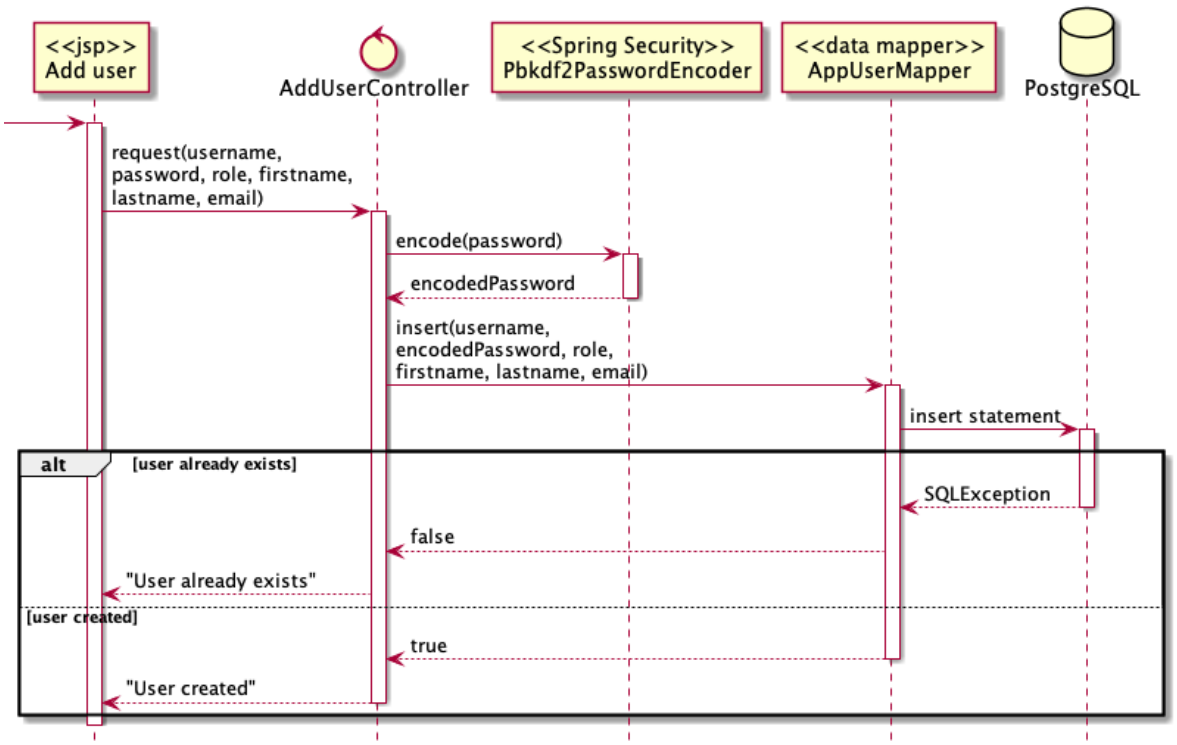
During the user registration process, the administrator sets the user account (including username and password). The password is hashed by the Spring Security framework during registration and stored in the database:

	username [PK] text	first_name text	last_name text	email text	password character varying (100)	role role_type
24	msmith	Mary	Smith	msmith@...	c9afe19818de79e83e6fe6774...	STUDENT
25	cdonaghy	Charlie	Donaghy	cdonagh...	ad5691e7ff97199cc25c8c972...	INSTRUCTOR

Two users in the data store with the same password

This is in line with industry best practices stipulating that passwords should not be stored in plaintext in case the user store is compromised. In order to provide extra safeguards for passwords stored in the database, a salt is used each time a password is hashed. A salt is random data used as an additional input to the Spring Security encoder that hashes passwords. This provides protection for pre-computed hash-attacks and for commonly occurring passwords – it makes the table required for a hash-attack prohibitively large[2]. As an example, two fake users have been created for demonstration purposed – both Mary and Charlie share the same password (*'test'*), however, they have been encoded using different salts and so are represented as different hashes.

It is possible to tune the width of the hash – it has been set to the default width which aims for .5 seconds to validate the user's password upon authentication[3]. The default width provides for sufficient security for the Examination Application.



Sequence diagram for user registration

Initially, the administrator creates a temporary password for the user, however, this password can be changed at any time through the application home page once a user is authenticated. A change password page was created to avoid situations where the administrator knows the passwords of all users.

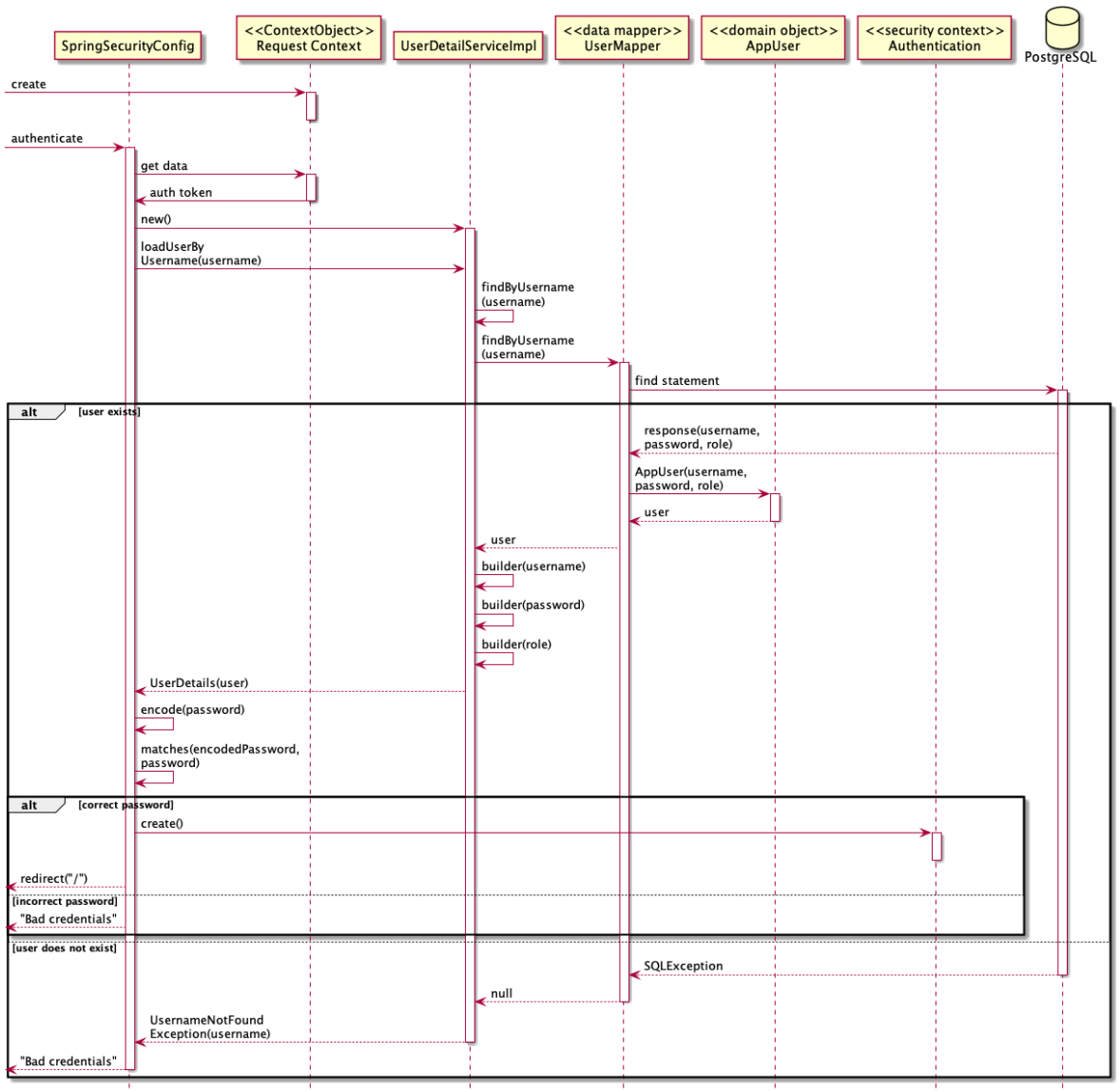
2. Authentication

Form-based authentication is used to authenticate users of the system. From the front page of the Examination Application, users are directed to the login page where they can enter their username and password.

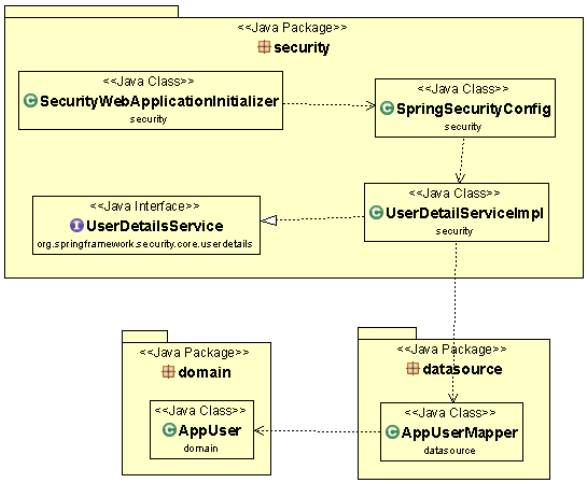
In order to build the authentication of the user, the Spring Security framework uses the *AppUserMapper* to search the database for the username that was entered – if found, the *AppUserMapper* returns an *AppUser* object instantiated with the username and password of the stored user.

Spring Security then encodes the password inputted on the login page and compares the now-encoded password with the already encoded password in the database to determine if they are identical.

If they are identical, the authentication object is built and stored inside the *SecurityContextHolder* with *UserDetails*, which effectively acts as an adapter between the Examination Application's user representation and the Spring Security framework. The *SecurityContextHolder* is used to store the user's username and role and is accessed at later stages to determine authorisation.



Sequence diagram for authentication



Class diagram for the security package with its authentication dependencies

3. *Logout*

At logout, the stored security context Authentication is invalidated.

There has been no session timeout placed on the Examination Application, so the user will remain logged in until the session object is destroyed (once logging out or clearing the user's cache and cookies).

A restriction has been placed on the system that prohibits a user from logging in for a second time if they are already authenticated – the system will redirect all users away from the login page if there is an active session object.

4. *Cross-site request forgeries*

In order for the Examination Application to protect against an authenticated user executing unwanted actions on behalf of a malicious actor, the system must be able to tell the difference between legitimate requests and forged requests from users.

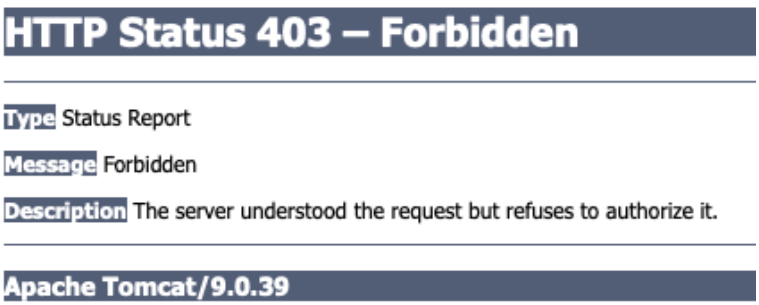
The solution is to ensure that each request requires, in addition to the user's session cookie, a randomly generated token as an HTTP parameter. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail[5].

In the Examination Application, all form requests that transmit confidential data or update system states have been sent using POSTs and have used Spring Security cross-site request forgery (CSRF) tokens in order to guard against cross-site request forgeries:

```
<form name="addSubjectForm" action="AddSubject" method="post">
    <sec:csrfInput />
```

Form implementation with CSRF token

Any POST request that is received by the Examination Application will be rejected if it does not contain a matching CSRF token and the system will present a 403 forbidden message:



Server response for a POST request lacking a CSRF token

6.1.2.2 Spring Security authorisation

Given the sensitive nature of the data stored on the Examination Application, it must be able to discern which users should be shown certain restricted resources (for example, a student should not be able to update marks for an exam). There are three different users of the Examination Application: administrator, student, and instructor. The system must be able to distinguish between the three and restrict or permit access to resources based on their role.

Implementation

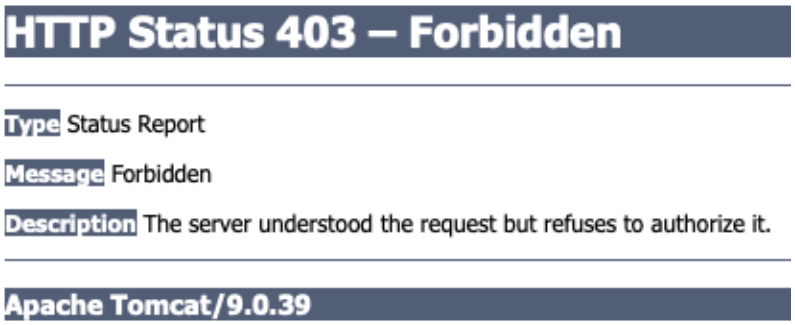
Authorisation has been implemented using the Spring Security framework – as discussed, upon authentication, a *SecurityContext* is created with the username and role of the authenticated user. Throughout the use of the Examination Application, checks are made to the *SecurityContext* under three different scenarios:

1. The *SpringSecurityConfig* class creates a filter chain to prohibit access to servlets based on the role of the authenticated user;
2. From controllers, to determine which method of mapper classes to invoke to return data from the database; and
3. From JSPs, to determine what elements of a page should be rendered for the authenticated user.

Examples of these three scenarios are discussed below.

1. *The SpringSecurityConfig class creates a filter chain to prohibit access to servlets based on the role of the authenticated user*

Users only have access to a subset of pages as defined in the configuration class. Where a user is authenticated but does not have the correct authorisation to view a page (for example, student users cannot gain access to the */AddEnrolment* page), any user attempting to access a view without proper authorisation will be presented with a 403 forbidden error message:



Below is a full list of view access dependant on user role:

URI swen90007-exam-app.herokuapp.com	Authorisation			
	Anonymous	Student	Instructor	Administrator
/	x	x	x	x
/login	x			
/Home		x	x	x
/UpdatePassword		x	x	x
/AddUser				x
/AddEnrolment				x
/AddSubject				x
/ViewUserEnrolments				x
/ViewUsers				x
/ViewExam			x	
/AddExam			x	
/AddQuestion			x	
/TakeExam		x		
/ReadOnlyExam		x	x	
/ViewSubmission		x	x	
/Exams		x	x	
/ViewSubjectEnrolments			x	x
/MarkSubmission			x	
/MarkTable			x	

2. *From controllers, to determine which method of mapper classes to invoke to return data from the database*

There are two situations where the controllers utilise Spring Security's authentication framework to invoke different methods to obtain data from the database:

a) To retrieve data based on authenticated user's username; and
b) To retrieve data based on authenticated user's role

a) To retrieve data based on authenticated user's username

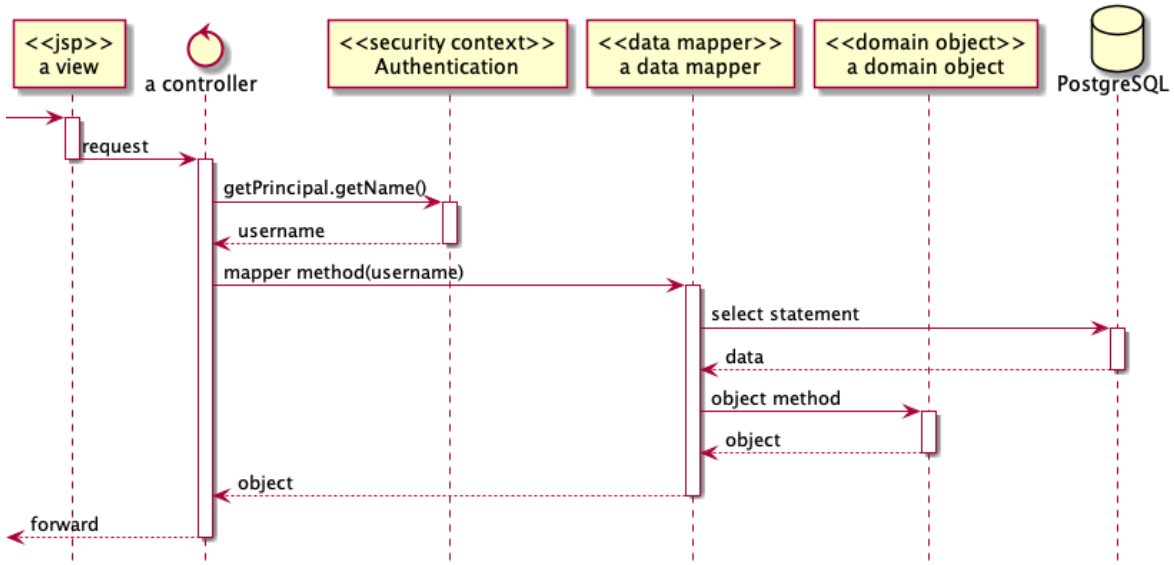
One example of controllers checking for username of the currently authenticated user is in the *ViewSubmissionController*. Authenticated students can navigate to the *viewSubmission* page in order to view their responses and/or grade received for previously submitted exams. Once receiving a request to view submission from the JSP, the controller determines the username of the currently authenticated student in order to determine whose submission it

should obtain from the data store (so as to avoid showing a student the submissions of other students):

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String view = "/viewSubmission.jsp";
    String username = request.getUserPrincipal().getName();
    int examId = Integer.parseInt(request.getParameter("examId"));
    Exam exam = ExamMapper.find(examId);
    Grade grade = GradeMapper.findByUserAndExam(username, examId);
```

The authenticated user's username is used to query the database

This sequence can be demonstrated using generic contexts:

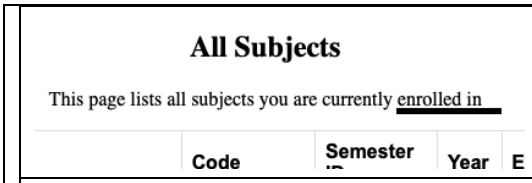


b) To retrieve data based on authenticated user's role

One example of a controller utilising the Spring Security framework to check an authenticated user's role is on the *subjects* view:

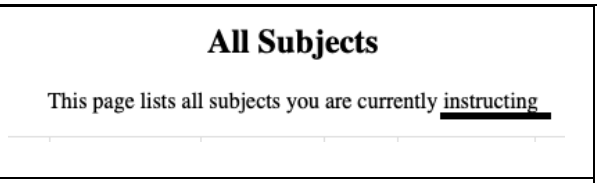
```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String view = "/viewSubjects.jsp";
    List<Subject> listSubjects = null;

    String username = request.getUserPrincipal().getName();
    if (request.isUserInRole("INSTRUCTOR")) {
        listSubjects = SubjectMapper.getInstructorSubjects(username);
    } else if (request.isUserInRole("STUDENT")) {
        listSubjects = SubjectMapper.getStudentSubjects(username);
    } else if (request.isUserInRole("ADMIN")) {
        listSubjects = SubjectMapper.getAll();
    }
```

This changes the resources rendered for each user dependant on role:



Student view of /Exams

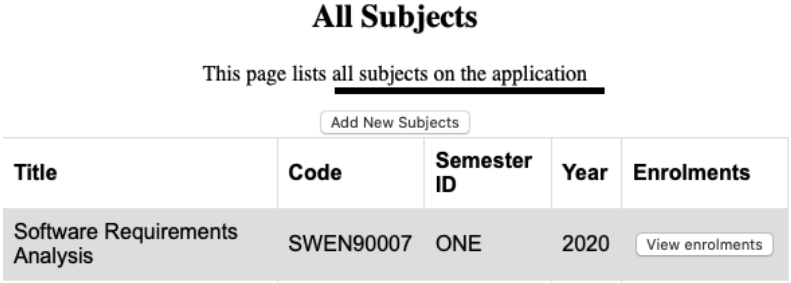
If a user is authenticated as a student, the controller invokes a method in the *SubjectMapper* which obtains all student enrolments for that user in the *student_subject* database table.



Instructor view of /Exams

If the authenticated user is an instructor, it invokes a method that will search the *instructor_subject* database table. The page then displays all subjects the instructor is instructing.

If the authenticated user is an administrator, the Examination Application invokes a method in *SubjectMapper* to return a list of all subjects in the system:



Administrator view of /Exams

In this example (and many others throughout the Examination Application), the view is determined by the current authenticated user's role validated through the *SecurityContext*.

3. *From JSPs, to determine what elements of a page should be rendered for the authenticated user*

One example of conditional rendering of page elements based on role is in the *exams* page. Depending on the role of the authenticated user, the actions the user can perform on an exam differ; if an instructor is authenticated, the instructor should be able to edit exams or change exam states for exams of subjects they instruct:

```
</sec:authorize> <sec:authorize access="hasRole('INSTRUCTOR')">

    <form action="ViewExam" method="get">
        <sec:csrfInput />
        <input type="submit" name="view" value="Edit"
            ${ exam.attempted ? 'disabled="disabled"' : ''} /> <input
            type="hidden" name="examId" value="${exam.id}"> <input
            type="hidden" name="subjectId" size="5"
            value="<c:out value='${subjectId}' />" /> <input type="hidden"
            name="subjectCode" value="<c:out value='${subjectCode}' />" />
    </form>
```

This authorisation check allows the Examination Application to render buttons on the exam view dependant on user's role:

Title	Status	Attempted	Actions
Mid-semester exam	PUBLISHED	false	Edit View Delete Close

Instructor view of /Exams

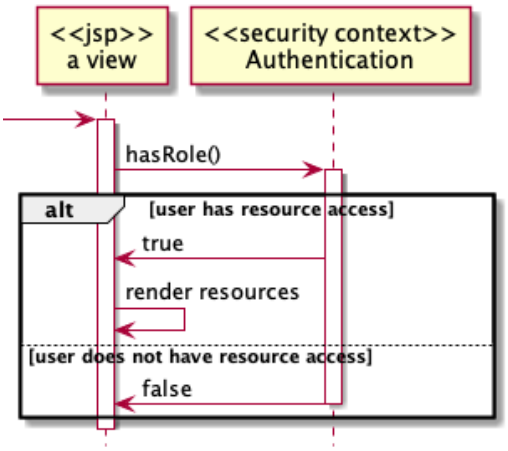
Exam actions rendered for instructor users.

Title	Status	Actions
Mid-semester exam	PUBLISHED	View Exam

Student view of /Exams

Exam actions rendered for student users.

This sequence can be demonstrated using generic contexts:



Below is a full list of object authorities depending on user role:

Object	Authorisation	Student	Instructor	Administrator
Subject	Create			x
	Read	x	x	x
Enrolments	Create			x
	Read		x	x
User	Create			x
	Read			x
	Update	x	x	x
		Note: Users can only update their own passwords.		
Exam	Create		x	
	Read	x	x	

	Update		**x**	
	Delete		**x**	
Response	**Create**	**x**		
	Read	**x**	**x**	
Marks	**Create**		**x**	
	Read	**x**	**x**	
	Update		**x**	
		Note: Instructors and students only have CRUD authorisation for resources of subjects they are currently enrolled in.		

6.1.3 Secure pipe

The Examination Application's trustworthiness is critical to its success – given the sensitive information transmitted to and from the system, students and instructors must have faith that the system guarantees data integrity.

As with all web applications, the Examination Application sends data over a network and is therefore open to security vulnerabilities, including:

- Data privacy, the ability for third-parties to listen in on communications and gain sensitive information; for example, university credentials, exam grades, etc.;
- Data integrity, the ability for an attacker to modify data in transit between users and the Examination Application; and
- Data authenticity, the ability for the client and server to confirm that the data is being sent by the other party (and not a third-party injecting communications).

In order to guarantee data integrity, the Examination Application must address these security vulnerabilities.

Implementation

The Examination Application implements secure pipe pattern through the use of Transport Layer Security (TLS) enabled for herokuapp.com domains. TLS is a cryptographic protocol that provides end-to-end encryption for all requests. Heroku's implementation of TLS manages:

- Data encryption, it hides the data being transferred;
- Data integrity, it verifies that the data has not been forged or tampered with; and
- Authentication through SSL certificates, it ensures that the parties exchanging information are who they claim to be[6].

When a user navigates to the Examination Application domain, Heroku initiates a TLS handshake, which uses public key cryptography to confirm the identities of the server and user. The TLS handshake establishes a cipher suite for each communication session, which is a set of algorithms that specifies details, including shared encryption keys, or session keys, to be used for that particular session in order to encrypt messages between user and server[6].

The encryption of messages for the session ensures that no other party can eavesdrop or tamper with data sent between the user of the Examination Application and the Heroku server and improves the ability of the Examination Application to ensure data privacy, integrity, and authenticity.
Reasons for use
TLS will decrease the likelihood of a violation of data privacy, integrity, and authenticity. TLS is a widely used approach with standardised infrastructure, achieving interoperability with other clients. It also promotes low coupling as it separates the cryptographic algorithms from the business logic – there was no need to implement the secure pipe pattern at the application level as it is all handled by Heroku at deployment.

6.2 Libraries and *Frameworks*

This section describes libraries and *frameworks* used by the Examination Application.

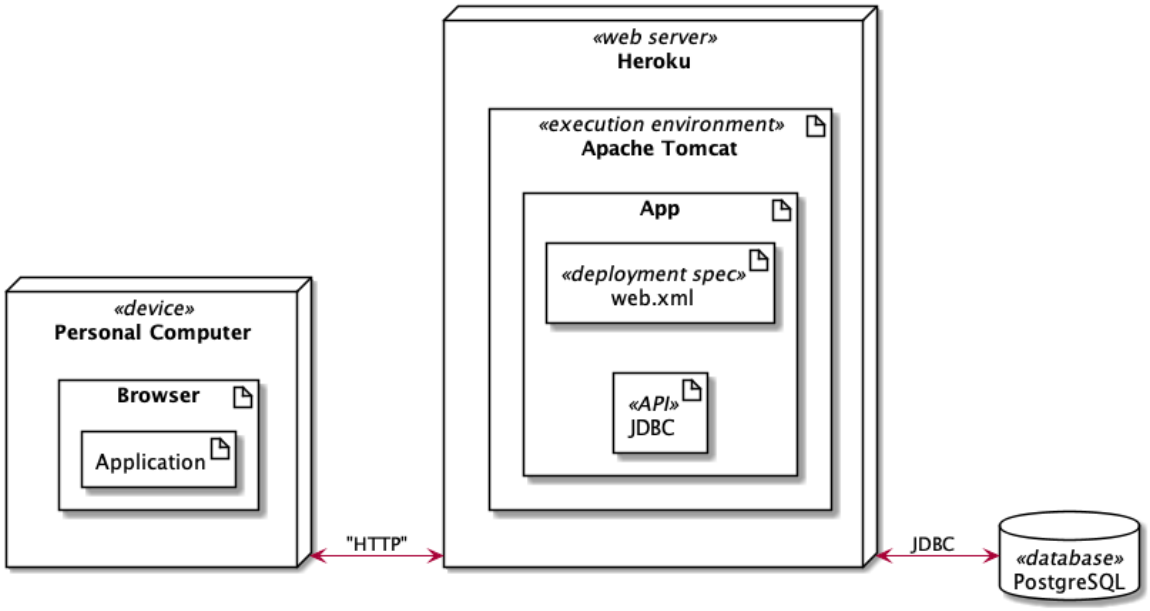
Library / *Framework*	Reason	Version
jQuery	• This library is used to simplify JavaScript code within the application, selected due to team member familiarity.	3.3.1
JavaServer Standard Tag Library (JSTL)	• A set of tag libraries recommended for use by Oracle in its 'JSP Coding Conventions to reduce the need for JSP Scriptlets.	1.2.7
Spring Security tag libraries	• A set of tag libraries that works with the Spring Security framework and allows seamless authorisation and authentication checks on JSPs.	5.4.1

7. Physical View

This section describes the hardware elements of the system and the mapping between them and the software elements.

7.1 Production Environment

This section describes the production environment of the system and the mapping between the software elements and the available hardware. The figure below illustrates the physical view of the production environment.



7.1.1 Hardware

- Heroku PostgreSQL, is a Database as a Service (DaaS) used to persist data.
- Heroku, is a Platform as a Service (PaaS) used to deploy the Examination Application.

7.1.2 Software

- Java Database Connectivity has been used as an Application Programming Interface (API) to define access to the database.
- Apache Tomcat has been used as the execution environment.
- Spring Security has been used as the authentication and authorisation enforcer.
- TLS has been used as the secure pipe pattern.

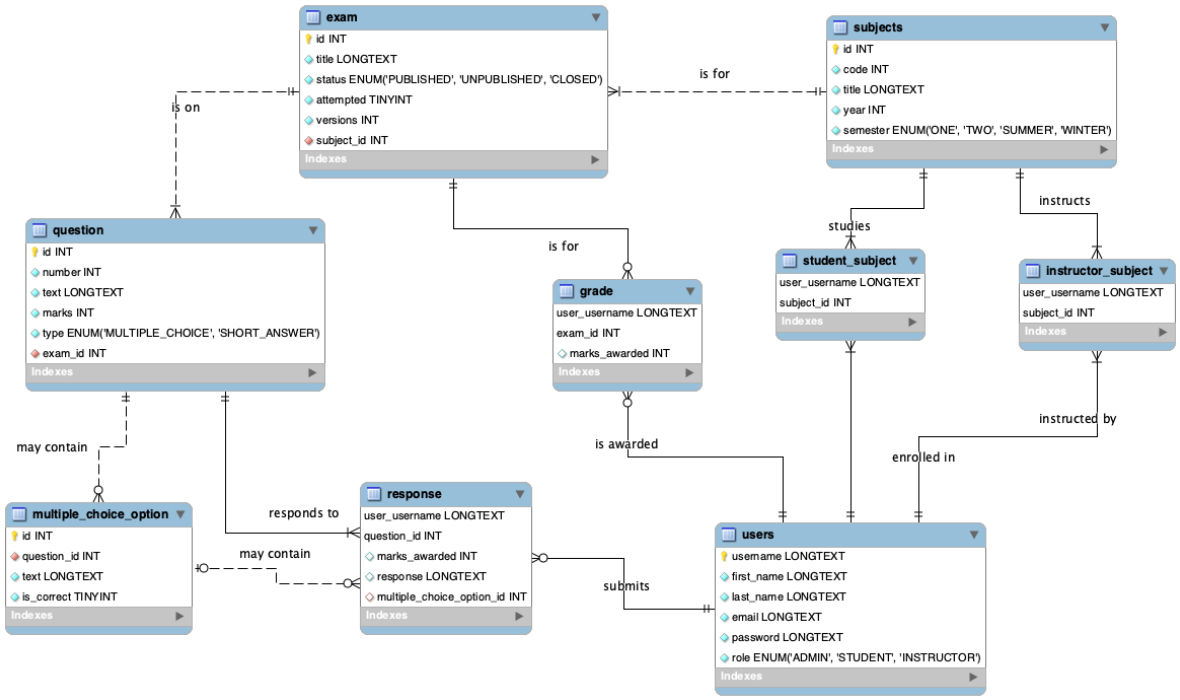
7.2 Development Environment

- Eclipse was primarily used as the Integrated Development Environment (IDE) for the Examination Application.

8. Data View

The database has been designed in order to support the architectural requirements and patterns in this report.

8.1 Database schema



8.2 Test data

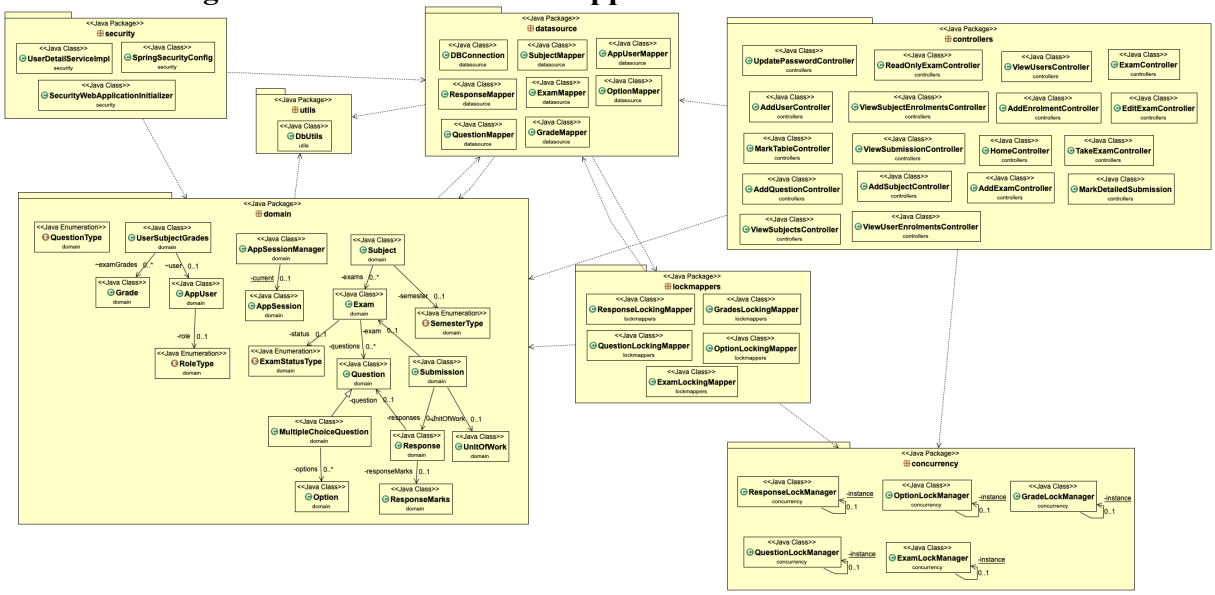
For test cases and data, please see *Part 3 testing document*.
Please see the database script in the appendix below.

9. References

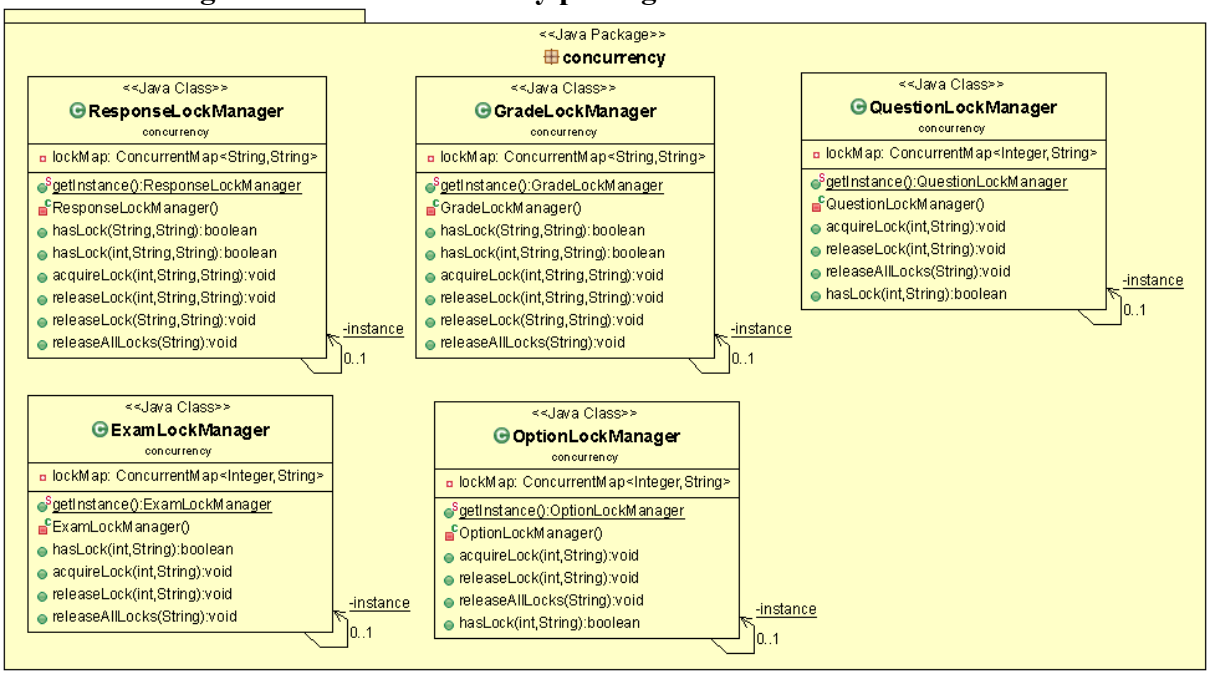
[1] Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE software*, *12*(6), 42-50..
[2] *Salt (cryptography)*. En.wikipedia.org. (2020). Retrieved 27 October 2020, from https://en.wikipedia.org/wiki/Salt_(cryptography).
[3] *Pbkdf2PasswordEncoder (Spring Security 4.2.15.RELEASE API)*. Docs.spring.io. (2020). Retrieved 27 October 2020, from https://docs.spring.io/spring-security/site/docs/4.2.15.RELEASE/apidocs/org/springframework/security/crypto/password/Pbkdf2PasswordEncoder.html#Pbkdf2PasswordEncoder--.
[4] *Spring Security Architecture*. Spring.io. (2020). Retrieved 29 October 2020, from https://spring.io/guides/topicals/spring-security-architecture.
[5] *13. Cross Site Request Forgery (CSRF)*. Docs.spring.io. (2020). Retrieved 29 October 2020, from https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html.
[6] *What is TLS?* (2020). Retrieved 31 October 2020, from https://www.cloudflare.com/en-au/learning/ssl/transport-layer-security-tls/.

10. Appendix

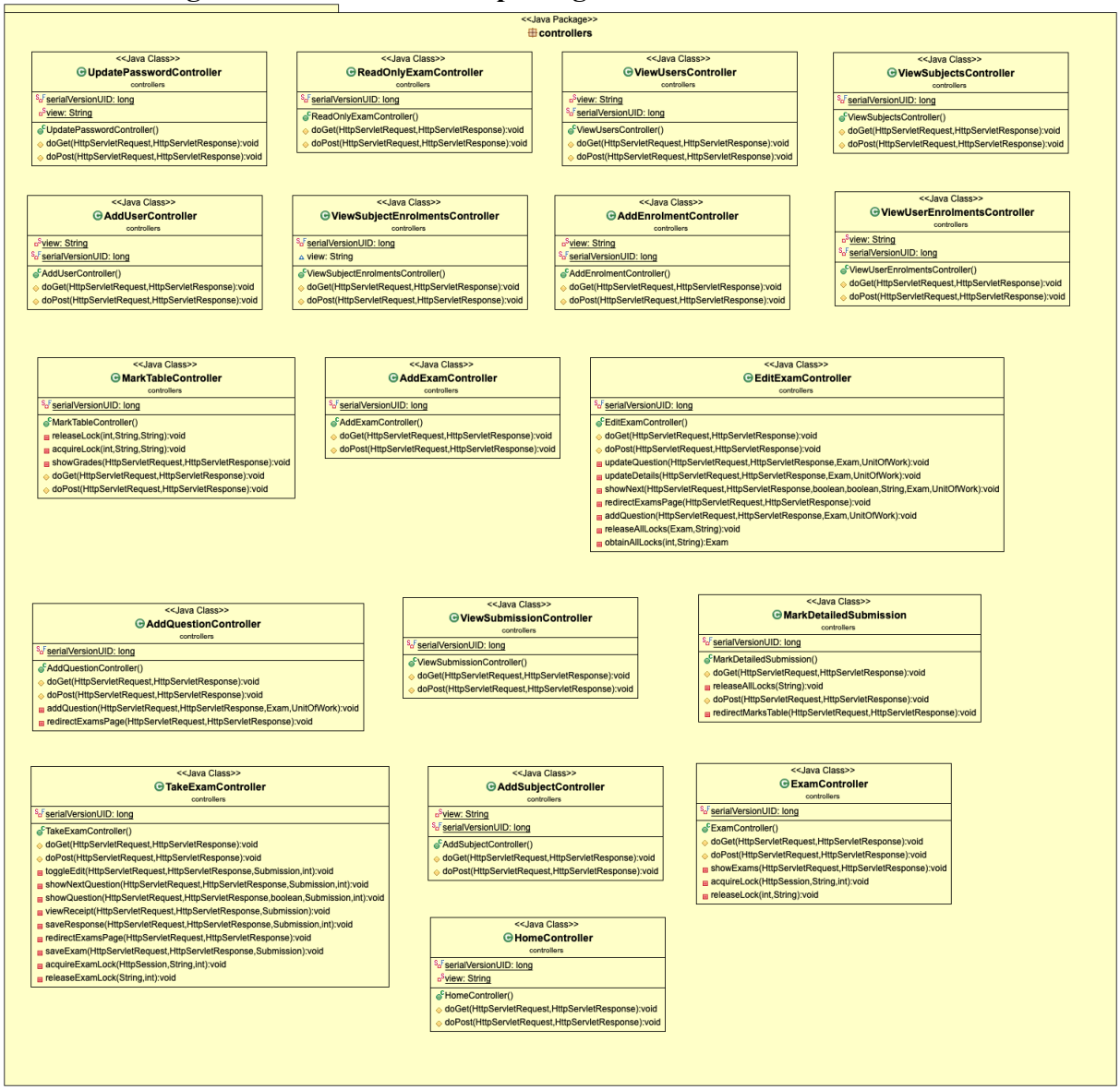
10.1 Class diagram for the Examination Application



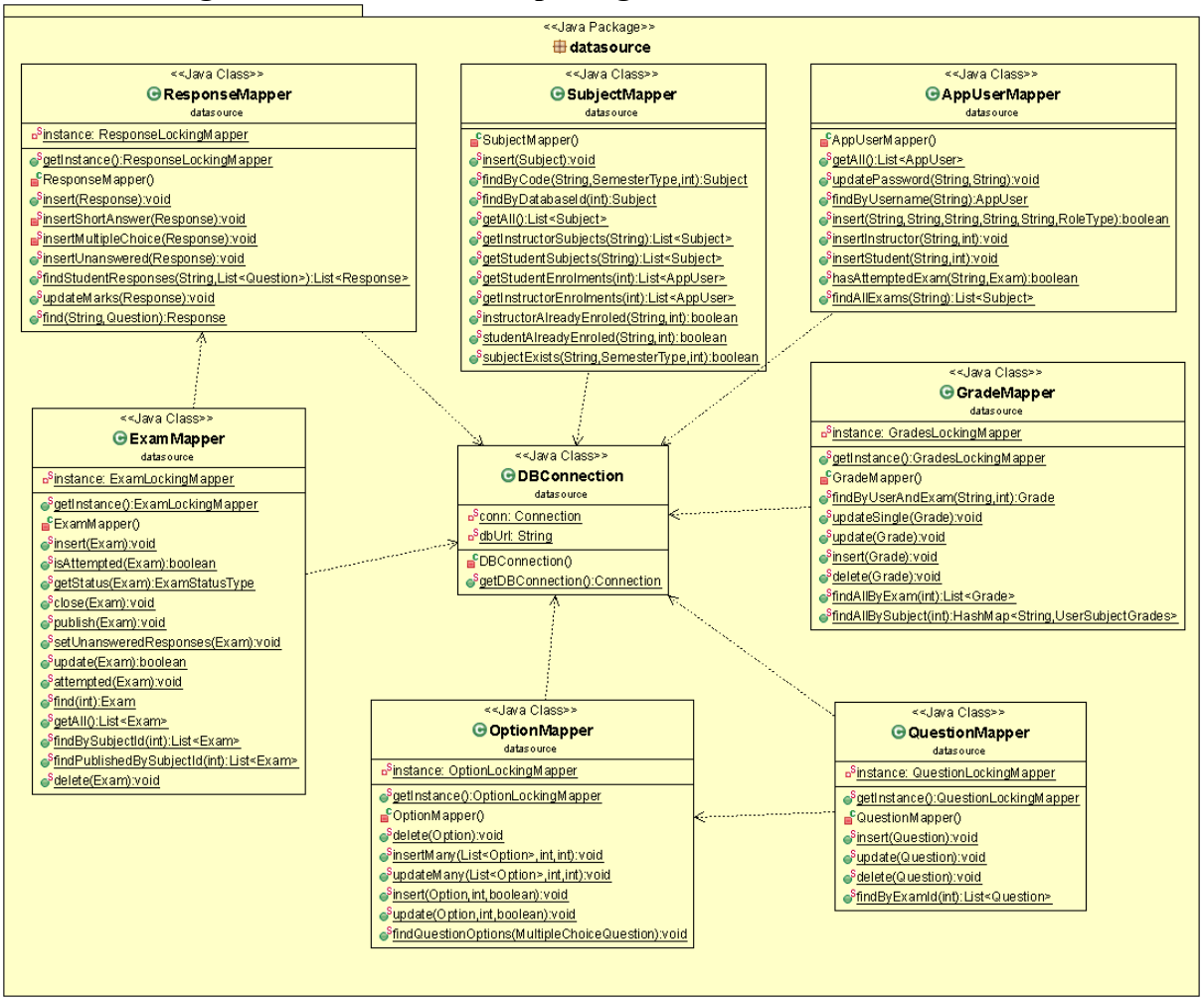
10.2 Class diagram for the concurrency package



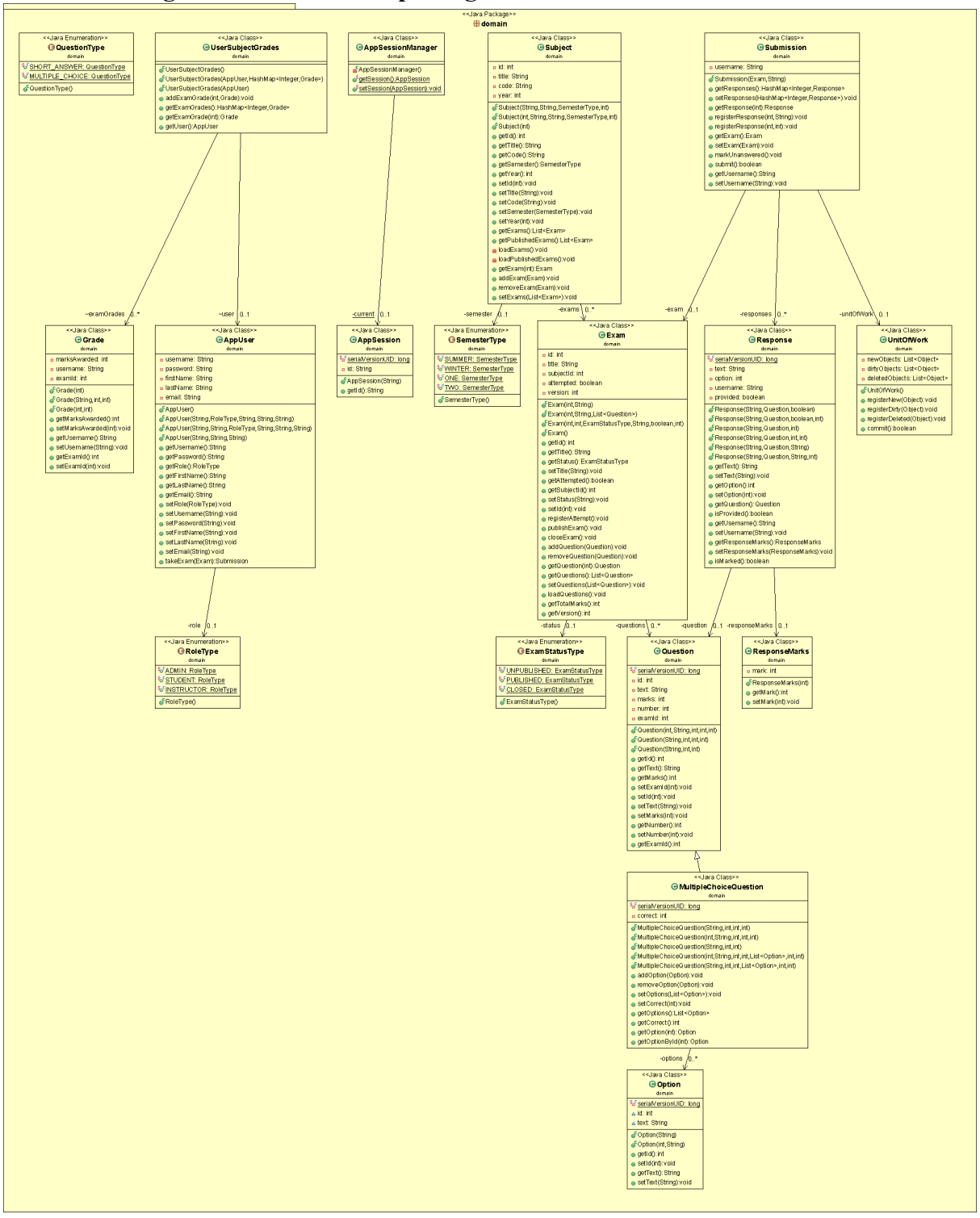
10.3 Class diagram for the controllers package



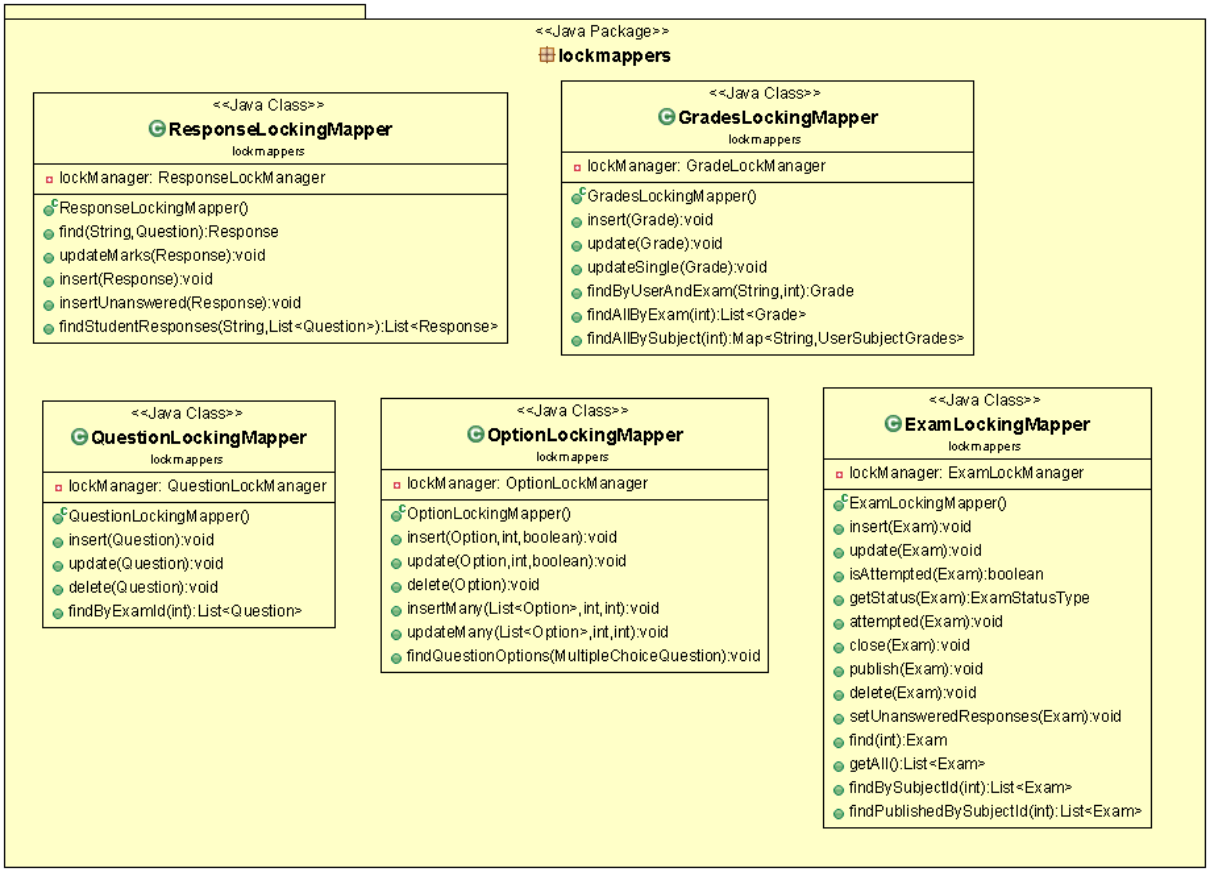
10.4 Class diagram for the datasource package



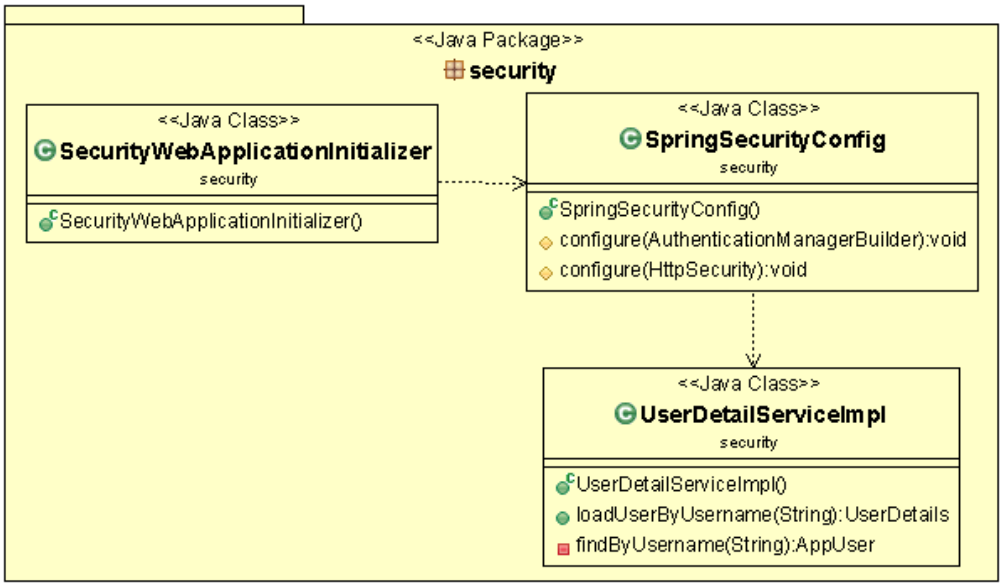
10.5 Class diagram for the domain package



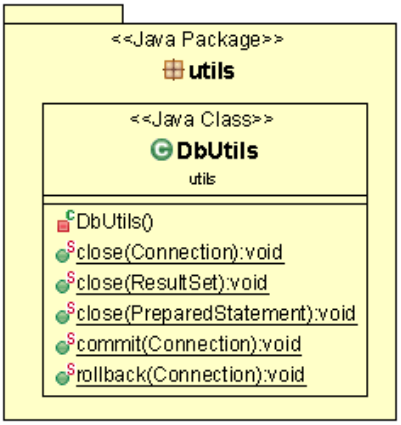
10.6 Class diagram for the lockmapper package



10.7 Class diagram for the security package



10.8 Class diagram for the utils package



10.9 Database script

```
DROP TABLE IF EXISTS grades CASCADE;
DROP TABLE IF EXISTS responses CASCADE;
DROP TABLE IF EXISTS multiple_choice_options CASCADE;
DROP TABLE IF EXISTS questions CASCADE;
DROP TYPE IF EXISTS QUESTION_TYPE CASCADE;
DROP TABLE IF EXISTS exams CASCADE;
DROP TYPE IF EXISTS EXAM_STATUS_TYPE CASCADE;
DROP TABLE IF EXISTS student_subject CASCADE;
DROP TABLE IF EXISTS instructor_subject CASCADE;
DROP TABLE IF EXISTS subjects CASCADE;
DROP TYPE IF EXISTS SEMESTER_TYPE CASCADE;
DROP TYPE IF EXISTS ROLE_TYPE CASCADE;
DROP TABLE IF EXISTS users CASCADE;

CREATE TYPE ROLE_TYPE AS ENUM('ADMIN', 'STUDENT', 'INSTRUCTOR');

CREATE TABLE IF NOT EXISTS users (
username TEXT,
first_name TEXT NOT NULL,
last_name TEXT NOT NULL,
email TEXT NOT NULL UNIQUE,
password VARCHAR(100) NOT NULL,
role ROLE_TYPE NOT NULL,
PRIMARY KEY (username)
);

CREATE TYPE SEMESTER_TYPE AS ENUM ('ONE', 'TWO', 'SUMMER', 'WINTER');

CREATE TABLE subjects (
id BIGSERIAL,
title TEXT NOT NULL,
code TEXT NOT NULL,
semester SEMESTER_TYPE NOT NULL,
year INT NOT NULL,
PRIMARY KEY(id),
UNIQUE (code, semester, year)
);
```

```
CREATE TABLE IF NOT EXISTS instructor_subject (
user_username TEXT NOT NULL
REFERENCES users(username)
ON UPDATE CASCADE
ON DELETE CASCADE,
subject_id INT NOT NULL
REFERENCES subjects(id)
ON UPDATE CASCADE
ON DELETE CASCADE,
PRIMARY KEY (user_username, subject_id)
);

CREATE TABLE IF NOT EXISTS student_subject (
user_username TEXT NOT NULL
REFERENCES users(username)
ON UPDATE CASCADE
ON DELETE CASCADE,
subject_id INT NOT NULL
REFERENCES subjects(id)
ON UPDATE CASCADE
ON DELETE CASCADE,
  PRIMARY KEY (user_username, subject_id)
);

CREATE TYPE EXAM_STATUS_TYPE AS ENUM('PUBLISHED', 'UNPUBLISHED', 'CLOSED');

CREATE TABLE IF NOT EXISTS exams (
id BIGSERIAL,
subject_id INT NOT NULL
REFERENCES subjects(id)
ON DELETE CASCADE
ON UPDATE CASCADE,
status EXAM_STATUS_TYPE NOT NULL
DEFAULT 'UNPUBLISHED',
title TEXT NOT NULL,
attempted BOOLEAN DEFAULT 'false',
versions BIGINT NOT NULL
DEFAULT 1,
PRIMARY KEY (id)
);

CREATE TYPE QUESTION_TYPE AS ENUM('MULTIPLE_CHOICE', 'SHORT_ANSWER');

CREATE TABLE IF NOT EXISTS questions (
  id BIGSERIAL,
  number INT NOT NULL,
  text TEXT NOT NULL,
  marks INT NOT NULL,
  question_type question_type NOT NULL,
  exam_id INT NOT NULL
  REFERENCES exams(id)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  PRIMARY KEY (id),
```

```
  UNIQUE(id, number)
);

CREATE TABLE IF NOT EXISTS multiple_choice_options (
  id BIGSERIAL,
  question_id INT NOT NULL
  REFERENCES questions(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  text TEXT NOT NULL,
  is_correct BOOLEAN NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS responses (
  user_username TEXT NOT NULL
  REFERENCES users(username)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  question_id INT NOT NULL
  REFERENCES questions(id)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  marks_awarded INT,
  response TEXT,
  multiple_choice_option_id INT
  REFERENCES multiple_choice_options(id)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  PRIMARY KEY (user_username, question_id)
);

CREATE TABLE IF NOT EXISTS grades (
  user_username TEXT NOT NULL
  REFERENCES users(username)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  exam_id INT NOT NULL
  REFERENCES exams(id)
  ON DELETE CASCADE
    ON UPDATE CASCADE,
  marks_awarded INT,
  PRIMARY KEY (user_username, exam_id)
);

INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('crosa',
'Christina', 'Rosa', 'c.rosa@unimelb.edu.au', 'ADMIN', 'DfeRT');

INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('jsmith',
'Jane', 'Smith', 'j.smith@unimelb.edu.au', 'STUDENT', 'g3S2qR');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('aklein',
'Angie', 'Klein', 'a.klein@student.unimelb.edu.au', 'STUDENT', '5ufFqx');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('marterton',
'Matthew', 'Arterton', 'm.arterton@student.unimelb.edu.au', 'STUDENT', 'bFe9qD');
```

```
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('lrosa', 'Luke', 'Rosa', 'l.rosa@student.unimelb.edu.au', 'STUDENT', 'WJq9h9');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('ewilliams', 'Emma', 'Williams', 'e.williams@student.unimelb.edu.au', 'STUDENT', 'kJ3aY5');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('cwilliamson', 'Clair', 'Williamson', 'c.williamson@student.unimelb.edu.au', 'STUDENT', '7nwCPM');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('bpitt', 'Brad', 'Pitt', 'b.pitt@student.unimelb.edu.au', 'STUDENT', 'Q8Rdak');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('jstuart', 'Jennifer', 'Stuart', 'j.stuart@student.unimelb.edu.au', 'STUDENT', 'wvRe6E');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('dmahn', 'Dohber', 'Mahn', 'd.mahn@student.unimelb.edu.au', 'STUDENT', '907GH');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('cwahwah', 'Chi', 'Wahwah', 'c.wahwah@student.unimelb.edu.au', 'STUDENT', 'lO0pd');

INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('cstewart', 'Claire', 'Stewart', 'c.stewart@unimelb.edu.au', 'INSTRUCTOR', 'V5K9xc');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('wbottel', 'Walter', 'Bottel', 'w.bottel@unimelb.edu.au', 'INSTRUCTOR', 'RnefG');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('scairn', 'So-Da', 'Cairn', 's.cairn@unimelb.edu.au', 'INSTRUCTOR', 'Sgf56');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('mbimba', 'Mark', 'Bimba', 'm.bimba@unimelb.edu.au', 'INSTRUCTOR', 'Vcg5Lk');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('acastle', 'Andrew', 'Castle', 'a.castle@unimelb.edu.au', 'INSTRUCTOR', 'H52Mew');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('mridolfi', 'Marcel', 'Ridolfi', 'm.ridolfi@unimelb.edu.au', 'INSTRUCTOR', 'zTT2QY');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('jhamilton', 'Jackson', 'Hamilton', 'j.hamilton@unimelb.edu.au', 'INSTRUCTOR', 'e4mWtE');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('kly', 'Kaylin', 'Ly', 'k.ly@unimelb.edu.au', 'INSTRUCTOR', 'U7NnLa');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('ckhor', 'Carmen', 'Khor', 'c.khor@unimelb.edu.au', 'INSTRUCTOR', '2czGTP');
INSERT INTO users (username, first_name, last_name, email, role, password) VALUES ('cskourletos', 'Christina', 'Skourletos', 'c.skourletos@unimelb.edu.au', 'INSTRUCTOR', 'qVz7Mf');

INSERT INTO subjects(title, code, semester, year) VALUES ('Software Requirements Analysis', 'SWEN90007', 'ONE', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Software Modelling and Design', 'SWEN90010', 'TWO', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Models of Computation', 'COMP30025', 'SUMMER', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Database Systems', 'COMP20010', 'ONE', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Introduction to Databases', 'COMP10010', 'TWO', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Introduction to Mathematics', 'MAST10010', 'TWO', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Algebra 1', 'MAST10030', 'ONE', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Algebra 2', 'MAST20030', 'ONE', 2020);
INSERT INTO subjects(title, code, semester, year) VALUES ('Calculus 1', 'MAST10025', 'ONE', 2020);
```

```
INSERT INTO subjects(title, code, semester, year) VALUES ('Calculus 2', 'MAST20050', 'TWO', 2020);

INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cstewart', 1);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cstewart', 4);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cstewart', 5);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mbimba', 3);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mbimba', 7);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mbimba', 9);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('acastle', 9);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('acastle', 1);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('acastle', 2);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mridolfi', 1);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mridolfi', 2);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('mridolfi', 5);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('jhamilton', 4);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('jhamilton', 10);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('jhamilton', 9);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('kly', 2);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('kly', 7);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('kly', 6);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('ckhor', 7);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('ckhor', 9);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('ckhor', 10);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cskourletos', 6);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cskourletos', 5);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('cskourletos', 4);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('crosa', 8);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('crosa', 1);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('crosa', 9);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('crosa', 6);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('wbottel', 6);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('wbottel', 5);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('wbottel', 4);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('scairn', 10);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('scairn', 7);
INSERT INTO instructor_subject(user_username, subject_id) VALUES ('scairn', 1);

INSERT INTO student_subject(user_username, subject_id) VALUES ('dmahn', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('dmahn', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('dmahn', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('dmahn', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwahwah', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwahwah', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwahwah', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwahwah', 9);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jsmith', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jsmith', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jsmith', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jsmith', 4);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jsmith', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('aklein', 9);
INSERT INTO student_subject(user_username, subject_id) VALUES ('aklein', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('aklein', 7);
```

```
INSERT INTO student_subject(user_username, subject_id) VALUES ('aklein', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('aklein', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('marterton', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('marterton', 4);
INSERT INTO student_subject(user_username, subject_id) VALUES ('marterton', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('lrosa', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('lrosa', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('lrosa', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ewilliams', 4);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ewilliams', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ewilliams', 9);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ewilliams', 10);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwilliamson', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwilliamson', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cwilliamson', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('bpitt', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('bpitt', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('bpitt', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jstuart', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jstuart', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jstuart', 10);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cstewart', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cstewart', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cstewart', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cstewart', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('mbimba', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('mbimba', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('mbimba', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('mbimba', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('acastle', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('acastle', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('acastle', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('acastle', 4);
INSERT INTO student_subject(user_username, subject_id) VALUES ('mridolfi', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jhamilton', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jhamilton', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jhamilton', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('jhamilton', 7);
INSERT INTO student_subject(user_username, subject_id) VALUES ('kly', 9);
INSERT INTO student_subject(user_username, subject_id) VALUES ('kly', 10);
INSERT INTO student_subject(user_username, subject_id) VALUES ('kly', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('kly', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ckhor', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ckhor', 4);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ckhor', 3);
INSERT INTO student_subject(user_username, subject_id) VALUES ('ckhor', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cskourletos', 10);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cskourletos', 8);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cskourletos', 6);
INSERT INTO student_subject(user_username, subject_id) VALUES ('cskourletos', 5);
INSERT INTO student_subject(user_username, subject_id) VALUES ('crosa', 10);
INSERT INTO student_subject(user_username, subject_id) VALUES ('crosa', 1);
INSERT INTO student_subject(user_username, subject_id) VALUES ('crosa', 2);
INSERT INTO student_subject(user_username, subject_id) VALUES ('crosa', 3);
```