Web App

In this report, I will talk about my grading system web application. This system has three versions, one is made using a command-line interface, the second is using JSPs and Servlets, and the third one is made using Spring framework.

I will discuss how web application development has developed over the years by showing the differences between each version. The report will discuss a topic at a time, while each topic clarifies the difference between all three versions in relation to the topic being discussed.

Topics to be discussed:

1. System features.
2. Code architecture and data model.
3. Database design.
4. Error handling.
5. Security measures.
6. Future work.

1. System features.

The system is designed to have three roles (types) of users, students, instructors, and administrators. The following details specify the capabilities of each role.

Student:

1- Can check all his grades.
2- Can check the average of his grades.

Instructor:

1- Can see all the courses he is teaching.
2- Can edit each grade in any specific course.

Admin:

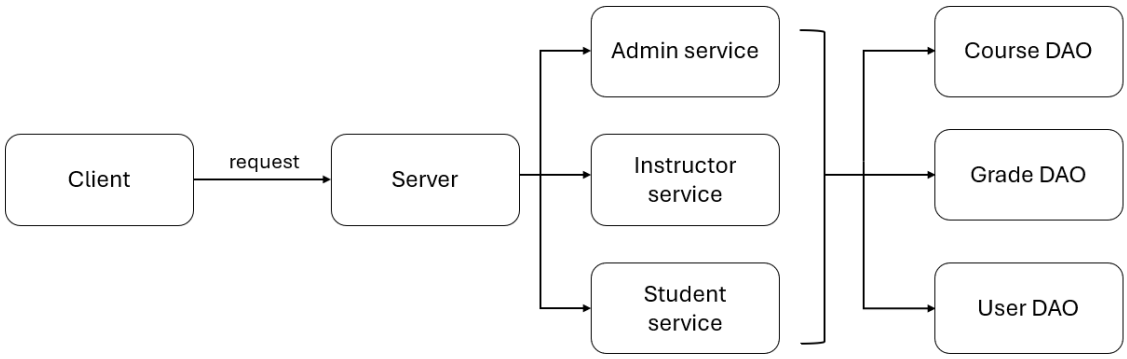
1- Can see all the users.
2- Can see all the courses.

3- Can create new users.
4- Can create new courses.
5- Can add students to courses.

2. Code architecture and data model.

Regarding data model, it must be nearly the same between all versions since it does not depend on the framework. Since we have users and a login system in the application, then we have a user model with an id, name email, password and role determining whether he is a student, instructor, or an admin. Also, we have a course model having an id, a name describing it, and an instructor id. The last model is the grade; it contains a student id, a course id, the student name, the course name, and the grade itself.

Now regarding code architecture, the figure below describes how the communication between components is established.

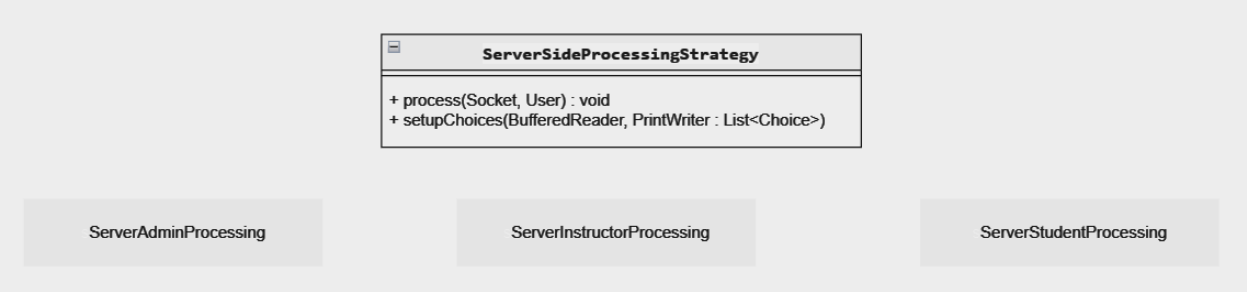


The client and server components contain the main differences between all three versions. In the first version, the client is represented by a client class containing a main method. This class establishes a socket connection with the server. Speaking of the server, it also has a server class with a main method that listens to any clients in a multithreading fashion. These two classes handle the login process before deciding the role (type) of the user.

Command-line:

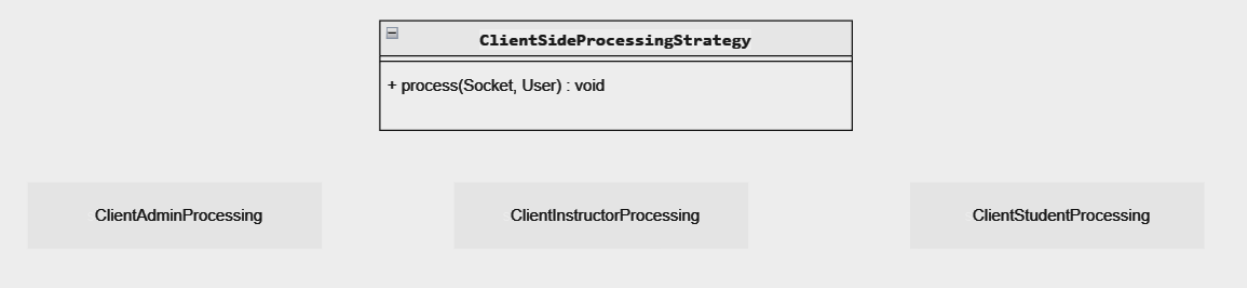
The client is prompted to answer questions or choose from a list of choices, and the server responds accordingly. The communication is done by utilizing socket programming in Java. The server utilizes **multithreading** to be able to serve multiple clients at once. A cached thread pool is used to create a dynamic pool of threads.

Moreover, the client and the server components are implemented in a dynamic way that dynamically responds to the role of the user. Since socket programming mainly depends on input and output streams, then each role will have its own set of choices and questions. For that reason, I have created the following architecture.



The user class in the first version has a ServerSideProcessingStrategy field that is injected manually at runtime after the user logs in. If the logged-in user is an admin, then a ServerAdminProcessing instance is injected into the field, and so on. This is useful because the process method decides how many read operations and write operations the server must execute. Because we know that each write operation to a socket from the server side must correspond to a read operation from the socket on the client side and so on. Also, the second method of the interface defines the list of choices offered from the server to the client, since an admin can do different things than a student.

The same goes for the client side:



The main difference is that the client cannot decide the list of choices he/she has. This must be defined from the server side since the server is the one who writes the rules.

Now let's discuss the services provided to the server. First, I should answer why it is important to add a service layer before accessing the database. You can answer this question by asking yourself, what if we change the way we access databases? What are the parts of code that would be affected in both ways of implementing? Obviously, handling the changes in the service is way better than handling them in the server (controller) itself since we reduce the number of changes that need to be made. Now back to our existing services, after convincing you why we need them. For each role there is a service, since each type of user demands different requests depending on the business (features) mentioned in the first topic. Although the implementation of all services is not different from the one in the services of the next two versions of the app, each type of the services has an interface the developer must stick to. This interface is prone to changes when the business is changing. Let's discuss each service interface:

```
public interface AdminService {
    1 usage   1 implementation
    List<User> getUsers();
    1 usage   1 implementation
    List<Course> getCourses();
    1 usage   1 implementation
    boolean createCourse(Course course);
    1 usage   1 implementation
    boolean createUser(User user);
    1 usage   1 implementation
    boolean addStudentToCourse(int studentId, int courseId);
}
```

The admin service can get all users, get all courses, create a course, create a user, and add a student to a course. Not surprisingly, it is exactly like described in the first topic.

```java
public interface InstructorService {
    1 usage  1 implementation
    List<Course> getCourses(int instructorId);
    1 usage  1 implementation
    List<Grade> getCourseGrades(int courseID);
    1 usage  1 implementation
    boolean editGrade(GradeDTO grade);
}
```

```java
public interface StudentService {
    1 usage  1 implementation
    List<Grade> getGrades(int studentId);
    no usages  1 implementation
    double getAverage(List<Integer> grades);
}
```

Same goes for instructor service and student service. Also, we have a login service that has a single method called authenticate that returns the user if the authentication process is successful, throws an exception otherwise.

It is the turn to discuss the data access objects of our project, which must be the same for all versions of the application, except for the spring version which uses the same interface for sure, but with minor changes to the implementation since it uses JDBC template. We have a data access object for each table.

```java
public interface UsersDao {
    2 usages  1 implementation
    User findById(int id) throws NoSuchElementException, SQLException;
    1 usage  1 implementation
    User findByEmail(String email) throws NoSuchElementException, SQLException;
    1 usage  1 implementation
    List<User> findAll() throws SQLException;
    1 usage  1 implementation
    boolean insertUser(User user) throws SQLException;
}
```

We can get the users from the database either using their id or email (which is unique). Also, we can get all users at once, and create a new user.

```java
public interface CoursesDao {
    2 usages  1 implementation
    Course findById(int id) throws NoSuchElementException, SQLException;
    1 usage  1 implementation
    List<Course> findByInstructorId(int id) throws NoSuchElementException, SQLException;
    1 usage  1 implementation
    List<Course> findAll() throws SQLException;
    1 usage  1 implementation
    boolean insertCourse(Course course) throws SQLException;
}
```

We can find courses by either course id or instructor id, or we can get them all, or create a new course.

```java
public interface GradesDao {
    1 usage  1 implementation
    List<GradeDTO> getGradesForCourse(int courseId) throws SQLException;
    1 usage  1 implementation
    List<GradeDTO> getGradesForStudent(int studentId) throws SQLException;
    1 usage  1 implementation
    boolean insertGrade(GradeDTO grade) throws SQLException;
    1 usage  1 implementation
    boolean editGrade(GradeDTO grade) throws SQLException;
}
```

Here we are introduced to something called data transfer objects represented by the GradeDTO class, it contains the same columns of the grade table in the database. I needed to have both Grade class and GradeDTO class because the first one can be represented to the user since it contains the names of both the student and the course, while the DTO is less readable since it contains the IDs. But the DTO is still needed because it is the one containing the unique information of the grade.

Servlet MVC:

After discussing the architecture part for the command-line version, I will discuss it for the Servlet version. Regarding the DAO interfaces and the services, they are the same. The main difference is the client and the server. Now we have a whole user interface instead of the command-line interface. Utilizing JSPs (Java Server Pages) we can render html code with java embedded inside it. I can feel a heavy weight going off my shoulders now, since

dealing with modern UI is way easier than dealing with command-line. Command-line is hard against catching errors and handling them, for example, you must handle the case where the user needs to choose between three options in the command-line, what if he enters a number larger than the number of choices, and so on.

We have 6 JSPs:

1- Admin dashboard.
2- Instructor dashboard.
3- Student dashboard.
4- Login.
5- Course details.
6- Error.

You can still style each JSP with CSS as if it is indeed regular HTML code with Java embedded in it.

This was for the front-end of the application. Regarding the backend, I used Java Servlets. Which replaces the server in the first version of the application. A servlet is basically an API (Application Programming Interface) that is called from a remote source usually by sending a request to it, then this API responds to that request with a response.

A servlet has a path (URL) that can be called from the browser or anywhere else to call the APIs of that servlet. The servlet is a controller inside the server that uses its services and resources. We have a servlet for each role and for the login process. To be mentioned, the piece of xml code below is written inside the web.xml file. It specifies the name of the API that must be called when we request the base URL of our application, which is the login page.

```
<welcome-file-list>
    <welcome-file>login</welcome-file>
</welcome-file-list>
```

Below you can find the URL mappings for the second version, knowing that the base URL is http://localhost:8080/ServletMVC

URL	JSP mapped to	HTTP methods
/login	login.jsp	GET, POST
/admin_dashboard/*	admin_dashboard.jsp	GET, POST
/instructor_dashboard/*	instructor_dashboard.jsp	GET, POST

/student_dashboard/*	student_dashboard.jsp	GET
/instructor_daashboard/courses/*	course_details.jsp	GET, POST

Spring MVC:

Again, the front-end and the backend parts are different than the previous versions. In Spring, we use regular HTML code with an included templating engine called ThymeLeaf. ThymeLeaf allows you to do java-similar functionality without embedding java code into HTML. This improvement removes the ugly embedding of java code within HTML (one step forward).

Regarding the backend, Spring supports SOAP APIs and REST APIs by providing two types of controllers (instead of servlets). One big advantage of Spring over Servlet MVC is dependency injection. In Spring, there is a new word called bean. A bean is basically an object, it is an annotation provided by Spring framework that allows Spring to know about any objects we would like to inject. So, it is a helper annotation for the framework to make it easy to inject dependencies.

Another advantage of Spring is the organization of the project, you have a file called application.properties, this file can change the behavior of your application by changing a single line in the properties file.

```
spring.datasource.url=jdbc:mysql://localhost:3306/grading_system
spring.datasource.username=root
spring.datasource.password=Password123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
server.port=8085
```

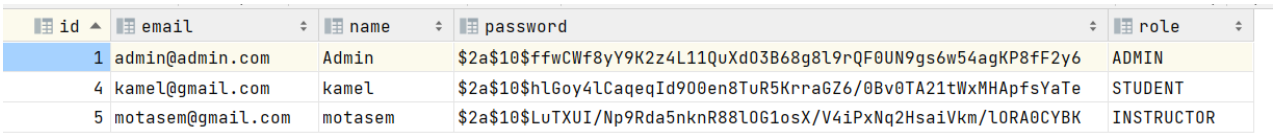
As we can see it contains information about the database and the running port of the application. Below you can find the URL mappings for the second version, knowing that the base URL is http://localhost:8080/

URL	HTML mapped to	HTTP methods	REST?
/dashboard	Used to redirect to one of the below dashboards.	GET	No
/admin_dashboard/id	Admin_dashboard.html	GET	No
/admin_dashboard/create_user	Admin_dashboard.html	POST	Yes
/admin_dashboard/create_course	Admin_dashboard.html	POST	Yes

/admin_dashboard/add_student_to_course	Admin_dashboard.html	POST	Yes
/instructor_dashboard/id	instructor_dashboard.html	GET	No
/instructor_dashboard/courses	instructor_dashboard.html	POST	No
/instructor_dashboard/courses/id	course_details.html	GET	No
/instructor_dashboard/edit_grade	course_details.html	POST	Yes
/student_dashboard/id	student_dashboard.html	GET	No

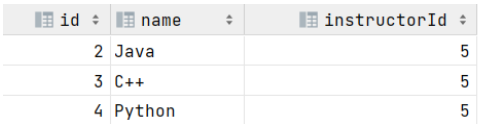
3. Database design.

All three versions share exactly the same database schema, I have a database called “grading_system”. The database has 3 tables as defined below.



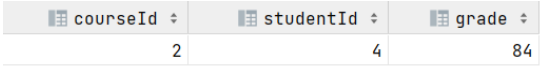
id	email	name	password	role
1	admin@admin.com	Admin	$2a$10$ffwCWf8yY9K2z4L11QuXdO3B68g8l9rQF0UN9gs6w54agKP8fF2y6	ADMIN
4	kamel@gmail.com	kamel	$2a$10$hlGoy4lCaqeqId9O0en8TuR5KrraGZ6/0Bv0TA21tWxMHApfsYaTe	STUDENT
5	motasem@gmail.com	motasem	$2a$10$LuTXUI/Np9Rda5nknR88lOG1osX/V4iPxNq2HsaiVkm/lORA0CYBK	INSTRUCTOR

This is the users table, it has the id as the primary key, the email as a unique identifier, and a name and a password, role defined as not null. I chose to combine all users in one database instead of creating a table for each role, since it is going to fit better with Spring Security in version 3. Also, we can notice that the passwords are not stored as plain text, they are hashed using BCrypt.



id	name	instructorId
2	Java	5
3	C++	5
4	Python	5

The courses table has the id as the primary key and the name of the course, also it contains a foreign key of the instructor id that is teaching this course.



courseId	studentId	grade
2	4	84

The grades table references the course id and the student id in addition to the grade itself.

I would like to mention some extra details about the database, I added some conditions and triggers on the that side to prevent some non-desired behavior. For example, a trigger is created to throw an SQL error whenever there is an attempt to add an admin to a course (instead of a student). Or whenever the admin tries to assign a student as the instructor of a new course. These details are related to the next topic, error handling.

3. Error handling.

Error handling is a little bit different between the first version and the other two versions. Since the other versions use a user interface to interact with the application.

Command-line:

The command-line interaction can be messed up very easily since you don't click on something or choose from multiple choices by jumping between them using arrows for instance. So, it is more error-prone, hence, it needs more error-handling. For error handling, either throwing exceptions or catching them is utilized, sometimes returning a Boolean identifying the result of the operation. Below we can find some possible errors and the way they are handled.

- Getting a list from the database (the list of users): handled by returning an empty list.
- Creating a user or course: handled by showing a feedback message whether the operation was successful or failed.
- Login failure: handled by showing a "Wrong credentials" message.
- Choosing a wrong choice from the list of choices: handled by ignoring the choice and prompting the user again.

Servlet MVC and Spring:

- The same goes for getting a list in the first version.
- A browser popup shows up instead of the feedback messages in the first version, so if creating a user fails, a popup says failed.
- Login failed in the Servlet MVC version refreshes the login page, while It is handled by Spring Security in the third version.
- An error page is shown whenever other exceptions are thrown.

4. Security measures.

Command-line:

Since the user cannot enter a URL or contact a specific server, then we don't need any kind of session management and authentication between the pages. But for sure the passwords are encrypted before being saved to the databases; they are hashed using bcrypt. Also, the SQL commands are safe against SQL injection attacks since pre-compiled statements are used from the JDBC API.

Servlet MVC:

Passwords are also encrypted before being saved into the database. But in this version and the next one, some session management must be done since we are dealing with a browser now, or some software like Postman. Therefore, the user logs in with a login page, after being authenticated by the database I utilize the session created by Java servlets. The role and the id of the user are saved in the mentioned session. Then when every page is opened, it is checked whether there is an ID saved in the session, if so, it is compared to the current user ID to make sure each user can only access the allowed resources. In addition, the role of the logged-in user is compared to the role saved in the session to make sure students can't access the admin page for example. This session data is cleared whenever the login page is accessed again.

Spring:

In Spring, it is different from the previous versions. We can find the power of this framework and how web development improved from the socket to the servlets and up to here. Spring framework provides the Spring Security module that handles all security-related stuff that needed to be done manually in the previous version. Spring Security handles the session management and logging in the user. Also, it comes with a default login page. This is all done by creating a bean for a SecurityFilterChain object that handles the roles for each user and the URLs they can open. Also, it is easy to integrate Spring Security with databases either using JDBC template or JPA. I used JDBC template as required in the assignment.

Future work.

- Using JWT tokens instead of the session-saved attributes in the Servlet MVC version, also adding the token to Spring Security configuration.
- Adding more detailed error handling.
- Studying the best practices for the communication between the views and the controllers.