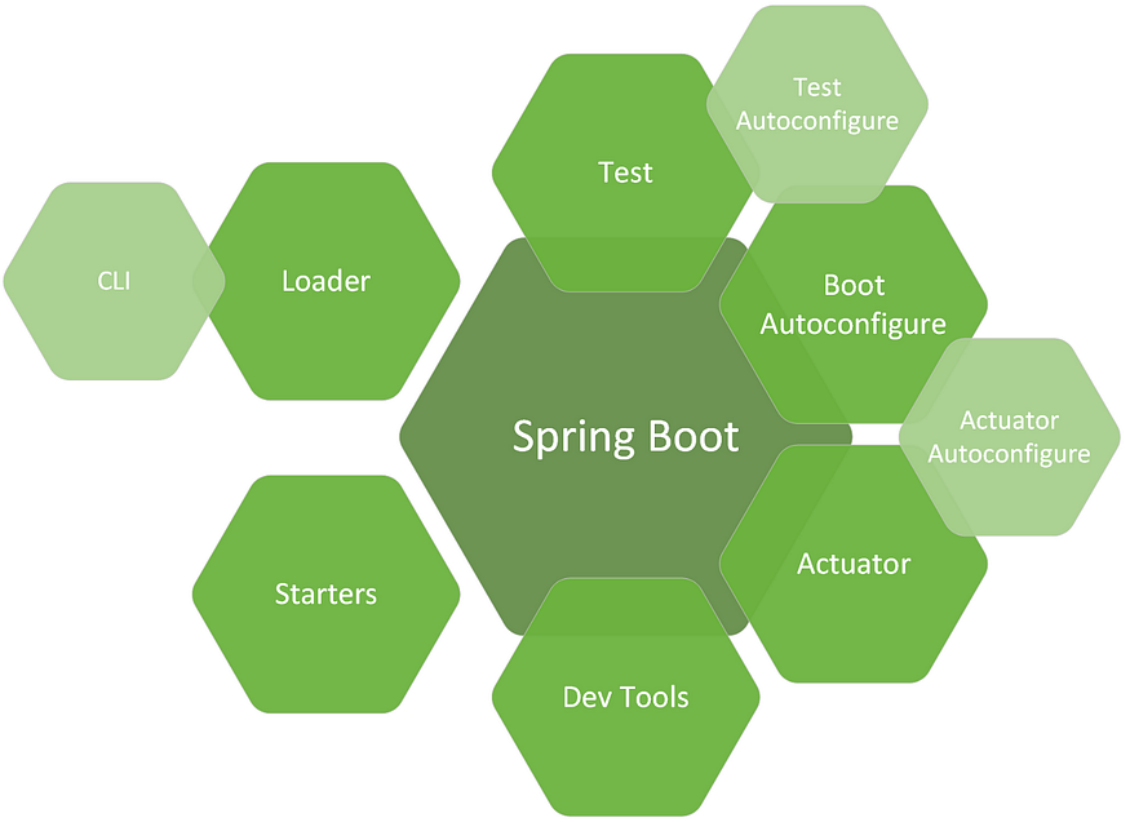
Student Grading System

Developed with Sockets, Servlet and Spring



Outlines:

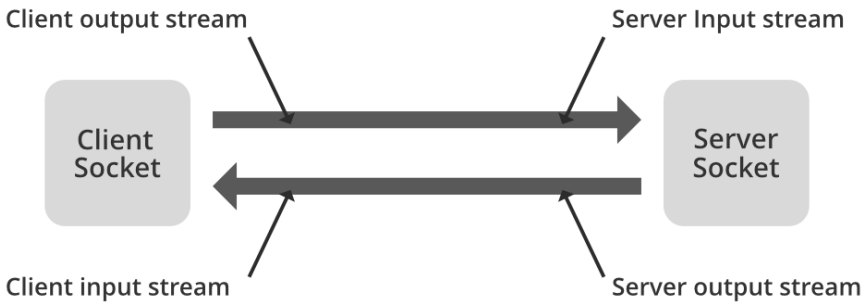
- **Introduction**
- **Design**
- **Implementation**
- **Challenges and Solution**
- **Security Considerations**
- **Testing**

Introduction

First let us consider what we will build for each technology:

- **Socket Programming**
  - Which is the initial Student Grading System with a **command-line** interface, using **sockets** and **JDBC** for backend communication with a **MySQL** database to store and manage student data, courses, and grades

  - We will consider the multithreading in this part (in the other parts, the frameworks themselves will be built-in multithreaded.



  - Also, we will use the socket input and output stream to print the data (**responses**) from the server.

- **Servlet & JSP**
  - Then we will upgrade the Student Grading System to a web application using **MVC** architecture with **Servlets** and **JSPs**, enabling user login, mark visualization, display**,** while ensuring **secure database interactions via JDBC.**

  - We will use **Tomcat** as the container for the Servlet and JSP to run them, and we will use **JDBC** (docker container for the **first two parts**).

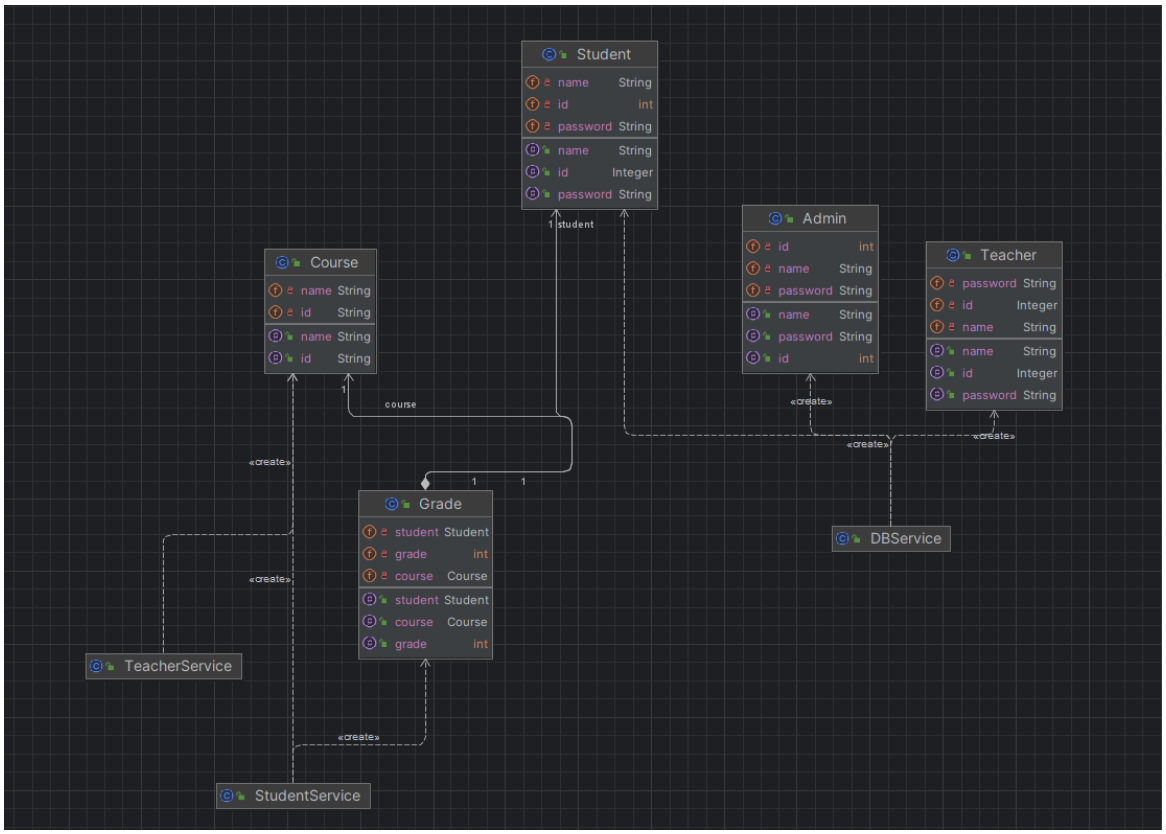
- **Spring MVC & Spring REST**
  - Now, we will use **Spring MVC** and **Spring REST**, replacing traditional Servlets with Spring **controllers**, implementing **Spring Security** for user authentication, and utilizing JPA for database interactions.

Note: We will be using **Docker** so we can build **MySQL** container using the **mysql** image.

Design

For each one of the technologies, we will have a design for it, so let us talk about each one:

Socket Programming:

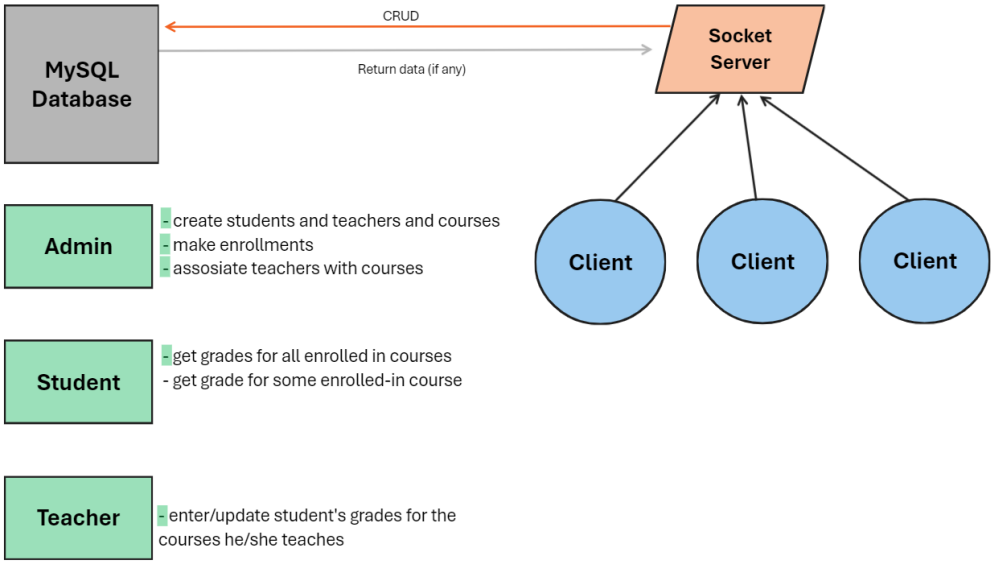


- The **DBService** class will be responsible for all the database operations for all the tables, and in this part we will implement all the methods (but later we will just use JPA like in Spring), and the methods of this class takes a connection (**MySQL** Connection) and then make a query to the database, for example, one of the queries is **enrollStudentInCourse** which will take the **course ID** and the **student Id**, and just make an enrollment as the following:

```
public static boolean enrollStudentInCourse(Connection conn, String studentID, String courseID) throws SQLException{
    String query = "INSERT INTO grades (student_id, course_id) VALUES(?, ?)";

    PreparedStatement statement = conn.prepareStatement(query);

    statement.setString( parameterIndex: 1, studentID);
    statement.setString( parameterIndex: 2, courseID);

    int rs = statement.executeUpdate();
    return rs > 0;
}
```

- In the above method, we will make a **String query**, and then using the connection, we will make this query to be **statement** (an executable query), and then say **statement.exexute** (depends on the query sometimes when we execute a **POST** query will call the **executeUpdat** and when we execute a **GET**, we will use **executeQuery**.

- The **Grade** class will have a composite key (coursID, studentID), means that this pair will not be repeated and we will only insert it one time.



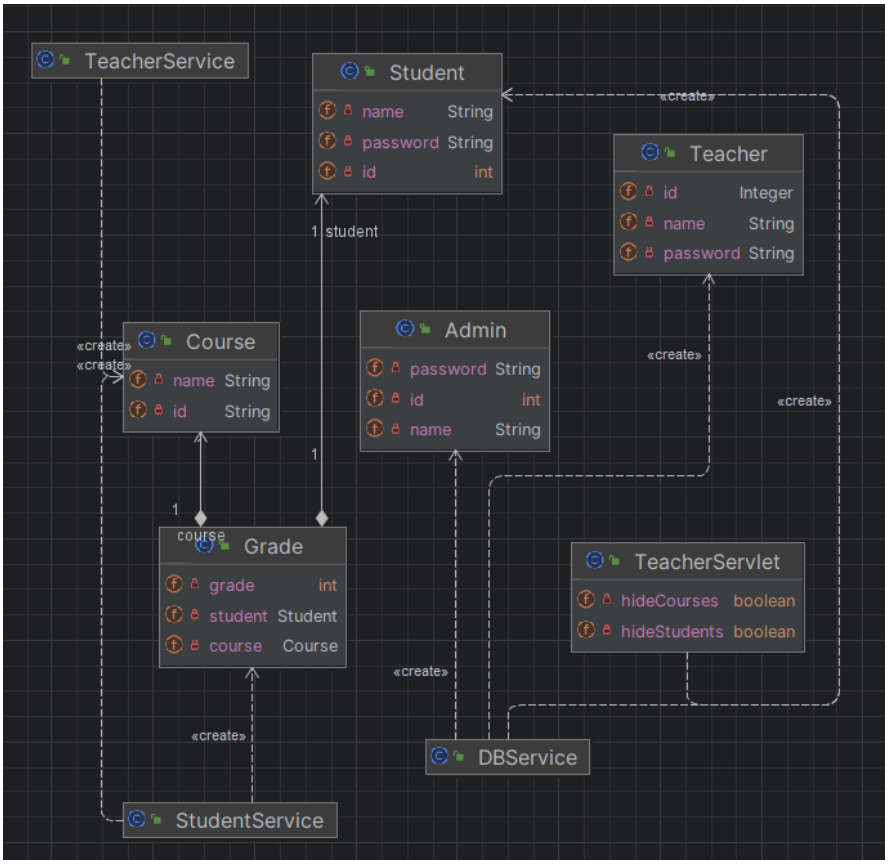
- Now lets talk about the **Sockets**, the socket is (in short) an endpoint of some connection, so as we can see, the clients (which may be an **Admin**, **Student** or **Teacher**) will talk to the server (have a connection with the server) using a socket, the server will have a **Server Socket**, which will be a special socket that will accept the other sockets, which make it a server socket, and the client socket.

- Now after the **connection** between the server and the client is made (using the **sockets**), the server and the client can both send requests and responses to each other.

- We will also have a **socket** input and output stream, so that when the response come from the server (or the client, but almost the **server** that will give the services to the **client**) will be shown to the client, and that is what we will see in the implementation section, how they are talk, how the client makes the action (like creating a new **Course** if the client is **Admin**!)

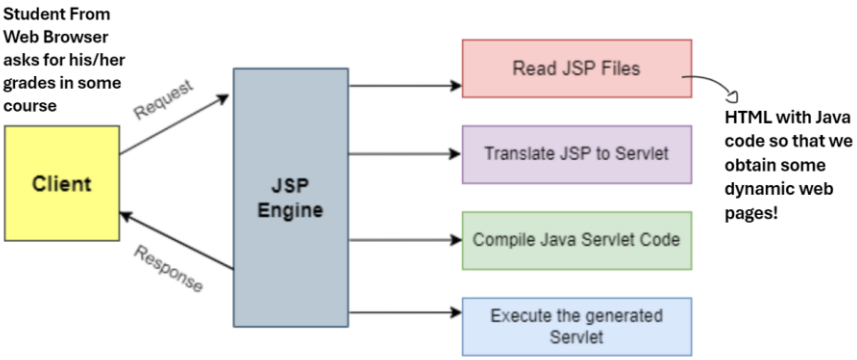
Servlet & JSP:

Now, let us look at the servlet architecture and flow of interaction:



- As we can see, the backend architecture is almost the same as the backend architecture of the Socket part, but here, we will handle the requests and the responses in another (**much easier**) way.

- First, **JSP** is just like an **HTML** document, but with java code embedded in it, allowing us to make dynamic web pages based on some conditions and actions.



- But in here, each entity will have its own **Servlet** that handles the requests and send responses.

- Also, we will have the same **DBService**, which will serve the same as the previous part, but here, we will just use it from **Utility** class called Creator.

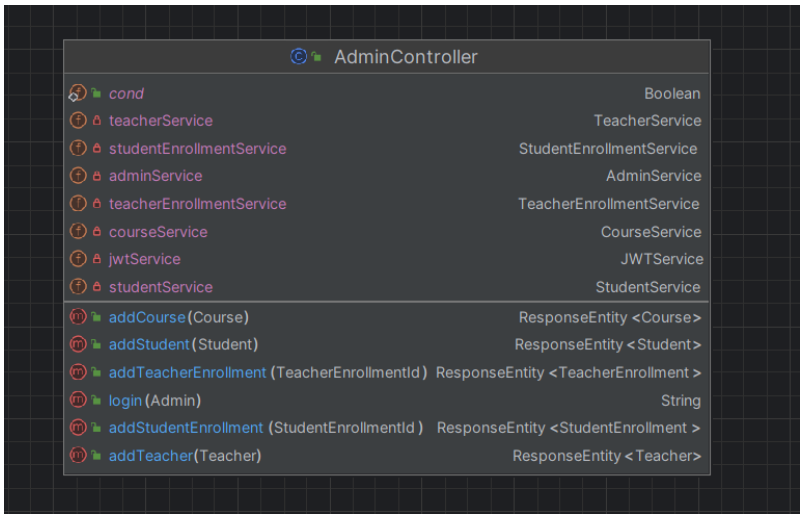
- For each entity (like student), we will have a **doGet** and **doPost** that may handle one or more action inside it.

- We will also have something called "**session**", which is the way that the user will access the services (only the allows service for each particular user) **without keep verifying** at each login time, so we will manage this session using **HttpSession.**

Spring:

- At this level, we will show the **MVC** (Model-View-Controller) design pattern, which is commonly used for developing user interfaces, which separates an application into three interconnected components.

- The architecture for this part goes as the following:
    - **Controller:**
      Which is the intermediary between the model and the view, it receives users from the view (**model**) then update it and finally pick the wanted view.



The **Admin Controller** for example, here we have methods like **addCourse**, **AddStudent** or **addTeacher**).

These controllers will be called from the frontend

- **Model:**
  Which is the classes and then entities that acts like a table inside the database, so for example, the **Admin** Class;

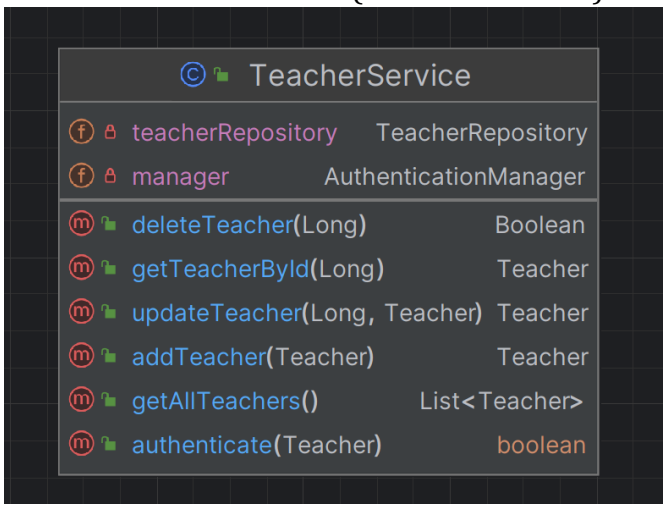
```java
public class Admin {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String password;

    public Admin(String name, String password) {  no usages
        this.name = name;
        this.password = password;
    }

}
```

  Because I'm using **JPA**, the classes that represent entities will be converted to real database tables!

- **Repository**
  We will use the Spring data **JPA** as I said before, so that it can be easy and very fast to act with the database and make operations on it, retrieve data from it, updating and deleting as well.

- **Security**
  Which will be used to secure our application using **JWT** tokens, which are very powerful and very efficient in terms of the security

- **Services**
  These are the layers that will deal with the database, and the controllers will use them (**autowired** them)



Implementation

Next, we will explain the implementation of each part:

Socket Programming:

- **Database**:

  After we've run the docker container that will just create the **MySQL** Database,

  We will create the database layer that will handle the interaction with the database,

  But first, let use create the **MySQL** connection class, that will give us the connection when needed:

```
conn = DriverManager.getConnection(
        url: "jdbc:mysql://localhost:3306/grading_system",
        user: "root",
        password: "1234578"
);
```

  This conn (of type **Connection**) will get the connection to the database or return an error if the provided data is invalid, or if the database isn't on.

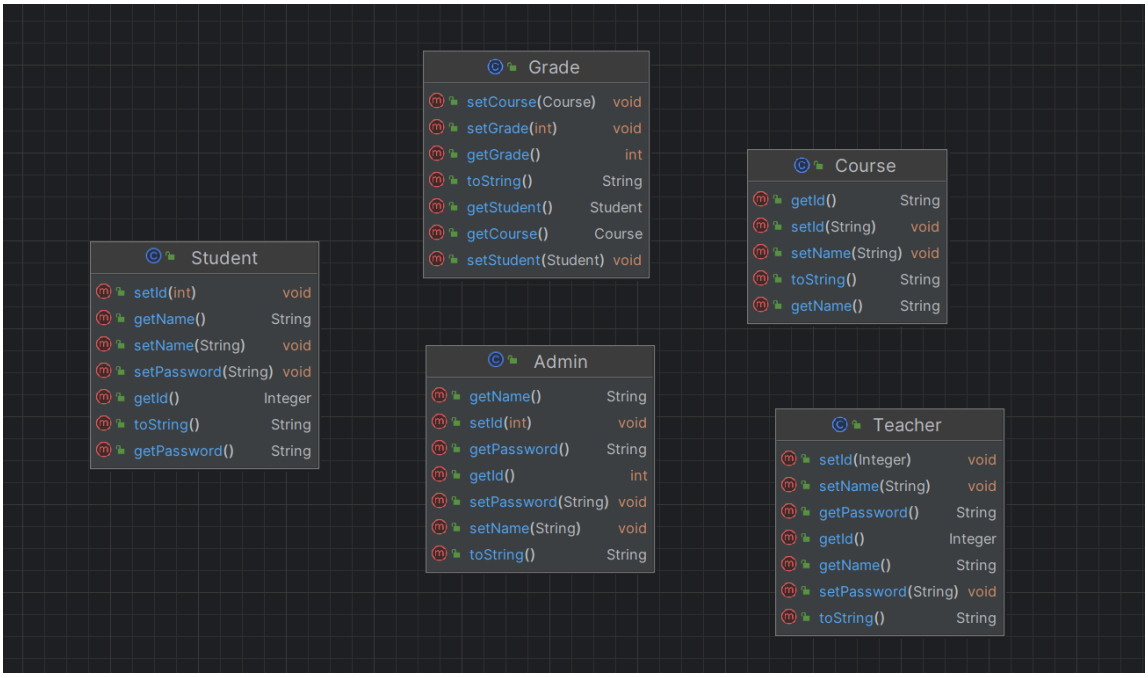
  After that, we will make **DBService** class that will interact with the database, and this class will have all the **needed functions** in our application, means that it will contain the Admin actions on the database, and the teacher actions, and the student and finally the super admin.

```
public static boolean createEntity(Connection conn, String table,  3 usages
                                   String name, String password) throws SQLException {
    String nameAttribute = table + "_name";
    String passwordAttribute = table + "_password";
    String query = "INSERT INTO " + table + " (" + nameAttribute + ", " + passwordAttribute + ") VALUES(?, ?)";

    PreparedStatement statement = conn.prepareStatement(query);

    statement.setString( parameterIndex: 1, name);
    statement.setString( parameterIndex: 2, password);

    int rs = statement.executeUpdate();
    return rs > 0;
}
```

  For this function, we will create a **Student**, **Teacher** or **Admin**, just by passing the username and the password for this method, and also the name of the table that the entity will be inserted (**created**).

  The query here changes as the value of the table variable.

- **Model:**



which will contains in this part the three entities and the **Course** entity and **Grade** entity (which is the student **enrollment** in a course, but with **grade** for each object)

- **Service**

- Authentication service**:**

```
public static Object login(BufferedReader reader, PrintWriter writer, Connection conn, String entity)
    String name = Readers.readUsername(reader, writer, entity);
    String password = Readers.readPassword(reader, writer, entity);
    if (Objects.equals(entity, b: "admin")) {
        return AuthService.checkAdminExistence(writer, conn, name, password);
    } else if (Objects.equals(entity, b: "teacher")) {
        return AuthService.checkTeacherExistence(writer, conn, name, password);
    } else if (Objects.equals(entity, b: "student")) {
        return AuthService.checkStudentExistence(writer, conn, name, password);
    }
    writer.println("Invalid entity name!");
    return null;
}
```

this will take the entity, and ask the user (the **client socket**) to enter a username and password, and then try to check if these are exists in the required table or not, if yes then it will return the entity itself, otherwise will return null.

- **Admin Service:**
  Which will be responsible for **create**, **update delete** a student or teacher or course or student enrollment in a course or teacher enrollment in a course, as the following:

```java
public static boolean createStudent(Connection conn, String name, String password) throws SQLException {  1 usage
    return DBService.createEntity(conn,  table: "student", name, password);
}

public static boolean createTeacher(Connection conn, String name, String password) throws SQLException {  1 usage
    return DBService.createEntity(conn,  table: "teacher", name, password);
}

public static boolean createCourse(Connection conn, String id, String name) throws SQLException {  1 usage
    return DBService.createCourse(conn, id, name);
}
```

  And it will go to the database and make the creation using **createEntity**

- **Student Service:**
  The student already has one action, which is reviewing the **grades** of his/her courses:

```java
String studentID = student.getId().toString();
ResultSet courses = DBService.getStudentEnrollmentsByStudentID(conn, studentID);
while (courses.next()) {
    String courseID = courses.getString( columnLabel: "course_id");
    ResultSet res = DBService.getCourses(conn, courseID);
    res.next();
    String courseName = res.getString( columnLabel: "course_name");
    int courseGrade = courses.getInt( columnLabel: "grade");
    Course course = new Course(courseID, courseName);
    Grade grade = new Grade(student, course, courseGrade);
    grades.add(grade);
}
```

  This will return to us a **List<Grade>** of some course that the user enrolled (with exception handling)

- **Teacher**
  The teacher has two actions, which are reviewing the courses that he/she **enrolled**, or update the marks of some course for all the student,

```java
while (courses.next()) {
    String courseID = courses.getString( columnLabel: "course_id");
    ResultSet res = DBService.getCourses(conn, courseID);
    res.next();
    String courseName = res.getString( columnLabel: "course_name");
    Course course = new Course(courseID, courseName);
    allCourses.add(course);
}
```

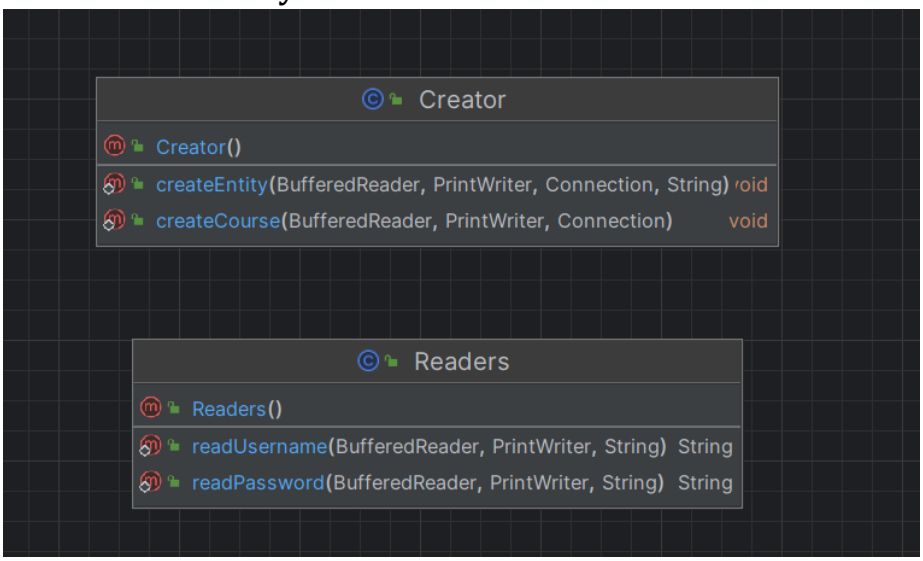
  So here we can see all the relative courses to some teacher (by **teacher id**), And here we can see the update grades operation

```
if (DBService.checkTeacherAssociation(conn, teacherID, courseID)){
    ResultSet students = DBService.getStudentEnrollmentsByCourseID(conn, courseID);
    while (students.next()) {
        String studentID = students.getString( columnLabel: "student_id");
        ResultSet student = DBService.getEntities(conn,  table: "student", studentID);
        student.next();
        String studentName = student.getString( columnLabel: "student_name");
        writer.println("Enter the grade for the student: " + studentName);
        int grade = Integer.parseInt(reader.readLine());

        if (DBService.updateStudentGrade(conn, studentID, courseID, grade)){
            writer.println("Successfully updated!");
        }
        else {
            writer.println("Failed to update student grade!");
        }
    }
} else {
    writer.println("The course " + courseID + " doesn't associated with this teacher or doesn't exist!");
}
```

With some exception handling of course.

And for the super admin service, it will just create a new admin after the **authentication** is done.

- **Utility classes:**
  Here are some **utility** classes that we could benefit from:



For the creator, will just take the **input** and **output** streams of the **socket** and create the called entity, or it will create a new course (which will a little bit differ in the process).

And for the reader:

```
public static String readPassword(BufferedReader reader, PrintWriter writer, String entity)
    writer.println("Enter " + entity + " password: ");
    return reader.readLine();
}
```

It will just read **username** and **password** and other needed things like above

- **Sockets:**
  For the client and **server sockets** (which is the most important part, we will create them as the following:

```
try (Socket socket = new Socket( host: "localhost",  port: 5000)) {
    System.out.println("[ " + socket.getLocalPort() + " socket ] : connected");

    Scanner scanner = new Scanner(System.in);
    BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    PrintWriter writer = new PrintWriter(socket.getOutputStream(),  autoFlush: true);

    while(true){
        readData(socket, reader);
        writeData(socket, writer, scanner);
    }


} catch (IOException e) {
    e.printStackTrace();
}
```

  Client socket will be created and will connect to the **server socket**, and after that:

```
try (ServerSocket serverSocket = new ServerSocket( port: 5000)) {
    System.out.println("Server is listening on port 5000");
    while(!serverSocket.isClosed()){
        Socket socket = serverSocket.accept();
        System.out.println("[ SERVER ] : New connection from " + socket);

        Thread client = new ClientHandler(socket);
        client.start();
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

  Server will create a new **Thread** for this socket and pass it to the **ClientHandler**, which will act as a connection between the server and the client.

  As we can see the class **ClientHandler** is a **Thread** class, so the run of this class will be:

```
@Override
public void run() {
    try {
        writer.println("Client is running on the thread: " + Thread.currentThread());
        ClientController.makeAction(reader, writer, conn);
    } catch (IOException | SQLException e) {
        closeResources(conn, reader, writer);
        e.printStackTrace();
    }
}
```

Servlet with JSP:

The only thing that we will change is the front end, so we will see the **servlet** and **JSPs**:

- **Authentication:**

```
public boolean checkAuth(HttpServletRequest req, String table, String name, String pass)
    Connection conn = MySQLDatabase.getConnection();
    Object entity = AuthService.login(conn, table.toLowerCase(), name, pass);
    System.out.println(entity);
    System.out.println(name + " " + pass);
    HttpSession session = req.getSession();
    System.out.println(entity);
    session.setAttribute(table.toLowerCase(), entity);
    return entity != null;
}
```

We will first check if the are **authenticated** or not using the **Http Servlet Requests** and **Responses**.
If the user try to enter the admin or student or the teacher, automatically the user will be **redirected** to the **authentication** phase and then go to the service that he/she wants:

- **Admin:**

So for example, if the user trying to go the endpoint of the ("**/**") **admin**, then the request will be redirected to the **authentication** servlet, which will render and **html** to login (using the name and password):

```
Boolean auth = (Boolean) session.getAttribute( s: "adminAuth");
if (auth == null || !auth){
    res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
    session.setAttribute( s: "adminAuth",  o: false);
    session.setAttribute( s: "authType",  o: "Admin");
    res.sendRedirect( s: "/main/auth");
    return;
}
```

After the login success, a session for this user will be created to prevent login **each time requesting a service**.

In the doPost of the **Admin**, it will use the function createEntity to create an entity that the user asks (the admin) as the following: (next page)

```java
@Override  no usages
protected void doPost(HttpServletRequest req, HttpServletResponse res)
    String action = req.getParameter( s: "action");

    if (action != null){
        try {
            createEntity(req, res);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    doGet(req, res);
}
```

- **Student:**

  Same as the admin, the student will be **redirected** to the **authentication** if he/she **has no session currently**, and after that the student html will be rendered to do the student actions like show the grades:

```java
String action = req.getParameter( s: "action");
if (action != null) {
    Student student = (Student) session.getAttribute( s: "student");
    try {
        List<Grade> grades = StudentService.getStudentGrades(student);
        session.setAttribute( s: "grades", grades);
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    hide = !hide;
    session.setAttribute( s: "hide", hide);
}
```

- **Teacher:**

  For the teacher also the same, the teacher will be redirected to the **authentication** if he/she has no session, and after that the teacher html will be rendered to do the student actions like show the grades:

```java
public List<Course> getAllCourses(String teacherID) throws SQLException {
    return TeacherService.getAssociatedCourses(teacherID);
}
```

  Also, the teacher can **update** the grades as the following:

```
List<Student> allStudents = new ArrayList<>();
ResultSet students = DBService.getStudentEnrollmentsByCourseID(conn, courseID);
while (students.next()) {
    int studentID = students.getInt( columnLabel: "student_id");
    ResultSet result = DBService.getEntities(conn,  table: "student", Integer.toString(studentID));
    result.next();
    String studentName = result.getString( columnLabel: "student_name");
    Student student = new Student(studentID, studentName,  password: null);
    allStudents.add(student);
}
```

- **Super Admin:**
  The authentication of the **super admin** will be also considered but just by entering the password of the super admin (there will be **one super admin**), and then an html page will be rendered to create an admin in the following login:

```
public void createAdmin(HttpServletRequest req, HttpServletResponse res) throws SQLException {  1 usage
    HttpSession session = req.getSession();
    String name = req.getParameter( s: "name");
    String password = req.getParameter( s: "password");
    System.out.println("Name: " + name);
    System.out.println("Password: " + password);
    Connection conn = MySQLDatabase.getConnection();
    session.setAttribute( s: "created",  o: false);
    if (SuperAdminService.createAdmin(conn, name, password)) {
        res.setStatus(200);
        session.setAttribute( s: "created",  o: true);
    } else {
        res.setStatus(403);
    }
}
```

Now, let us talk about the **JSPs** in the application:

- For all the **JSPs** of the **admin**, **student**, **teacher**, **super admin** and **authentication**, they are making **dynamic web pages**, for example if the admin hits the create button for the student or the teacher or course, the following jsp will be applied:

```
<%
    String action = request.getParameter("create");
    String type = "password";

    if (action != null) {
        if (!action.equals("createTeacher") && !action.equals("createStudent")) {
            type = "text";
        }
%>
```

Which returns an **html** if the condition is true:

```
<form action="admin" method="post">
    <br>
    <label><%= placeName %>:
        <input type="text" name="first" placeholder="<%= placeName %>" required>
    </label>
    <br><br>
    <label><%= placePassword %>:
        <input type="<%=type%>" name="second" placeholder="<%= placePassword %>" required>
    </label>
    <br><br>
    <button type="submit" name="action" value="create">Submit Admin Details</button>
</form>
```

Challenges and Solutions

Here are some of the challenges that I've faced during the development:

- **Multithreading in sockets:**
  The issue was in the **output** and the **input of the socket**, they are in the **same thread**, and the process will not work, what I've made is to **create a new thread** for **reading data** for the client (so that it will not wait the other to write and keep wait and wait):

```
public static void readData(Socket socket, BufferedReader reader){  1 usage
    new Thread(() -> {
        String data;

        while(socket.isConnected()){
            try{
                data = reader.readLine();
                System.out.println(data);
            } catch (IOException e){
                e.printStackTrace();
            }
        }
    }).start();
}
```

- **Spring Security:**
  Actually, I spent a lot of time on the **spring security** because of this issue, which is the following, what if we want to create the security configuration class and it has an AuthenticationProvidor that will make the following:

```
@Bean
public AuthenticationProvider authenticationProvider(){
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
    provider.setUserDetailsService(entityDetailsService);
    return provider;
}
```

  The **entityDetailsService** is implementing the **UserDetailsService**, but the implementation of the **entityDetailsService** was focused on the admin, and after I tried to make the **authentication** for the other users, I said how can I pass the Details? Like how we can specify what **details** we want (knowing that when the configuration file executed, we still don't want what is the current user, is it admin or other?) so I've just came up with this idea, let us make a **static variable for each controller** of the entities, called **cond** that is a **boolean**, when the current request is user, we will set the cond of the user to be **true**, and the other to be **false**:

```
@PostMapping("/login")
public String login(@RequestBody Admin admin) {
    StudentController.cond = false;
    TeacherController.cond = false;
    cond = true;
```

above is an example of the **admin** login, when the admin **requested** to login, the cond value will be true for admin, false for other, and **actually this will case an error if we think of the multithreading**, but at least solves the problem of the User Details.

Security Considerations

We will consider the **authentication** in each action of our **application**, so we will do that by a simple **login** and simple **authentication** that will connect to the database and check the user **existence**.

But now, what if the user **logged i**n, but it asks for **another action** (which will happen in the normal flow), we will consider making a session to the user so that it makes an action (for a **period**).

For the **Servlet**, we will make a **session** to the user, when logged in, and that is so easy using **HttpSession**.

So for the student servlet, we will do the following:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res
    HttpSession session = req.getSession();
    session.setAttribute( s: "hide",  o: false);

    Boolean auth = (Boolean) session.getAttribute( s: "studentAuth");
    if (auth == null || !auth){
        res.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        session.setAttribute( s: "studentAuth",  o: false);
```

req.**getSession**() will check if the current user has a session, if yes it will **return i**t, otherwise it will **create** a new one.

For the Spring security, I've used **JWT token** instead of creating a session, which is more **secure** and **saves no data** in the browser, so we could use it.

It takes some steps to **create** it, but at all, it is more **efficient** and well implemented.

Next, let us test the application for each part

Testing

- **Socket:**
  First, we will run the server, and then we can run clients (**sockets**) that are multithreaded:

```
  .m2\repository\com\google\protobuf\protobuf-j
Server is listening on port 5000
```

  Run **client** will make a connection:

```
[ SERVER ] : New connection from Socket[addr=/127.0.0.1,port=63541,localport=5000]
```

  And on the **client** side we have:

```
[ 63541 socket ] : connected
Client is running on the thread: Thread[#21,Thread-0,5,main]
Pick number from the following list:
1. Login to Super Admin
2. Login to Admin
3. Login to Teacher
4. Login to Student
5. Exit
```

  Here if we **pick 1**:

```
1
Enter super admin password:
1234578
Pick number from the following list:
1. Create a new Admin
2. Exit
```

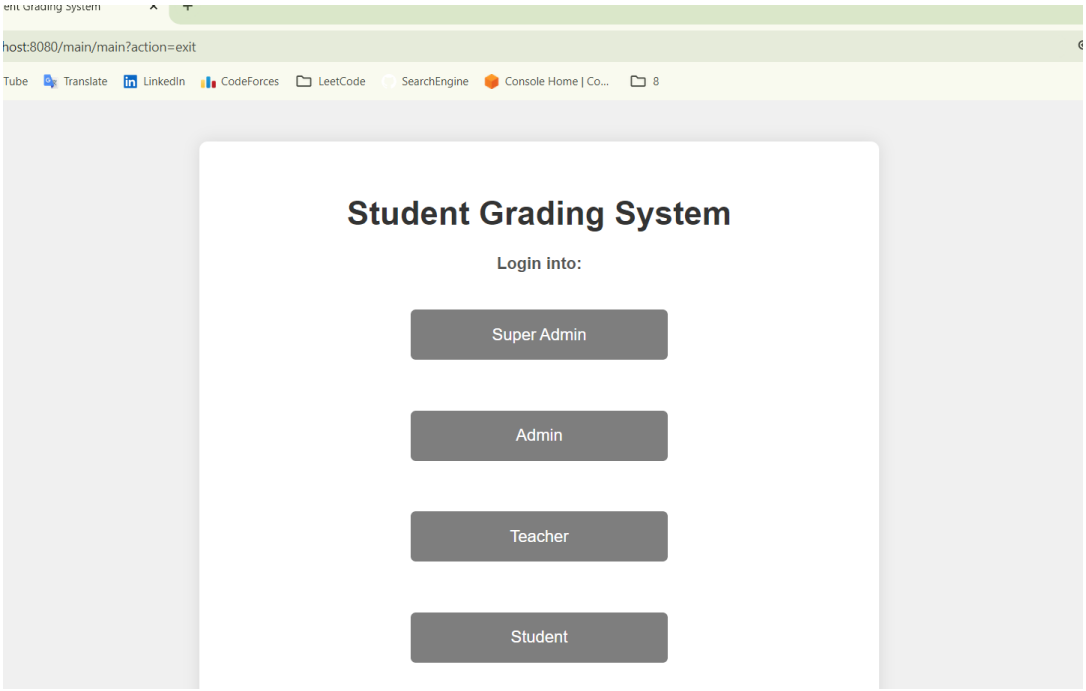
  Pick 1 again:

```
1
Enter admin name:
Mohammad
Enter admin password:
00000
```

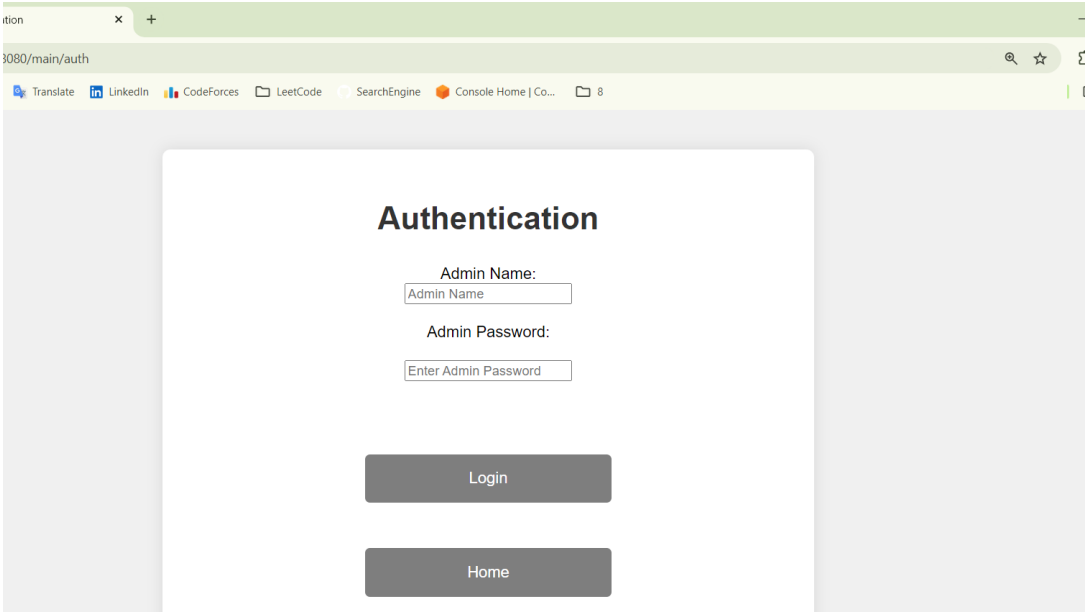
  And enter, the admin **created**, and so for the other services, we can also run multiple admins with multiple threads.

- **Servlet:**

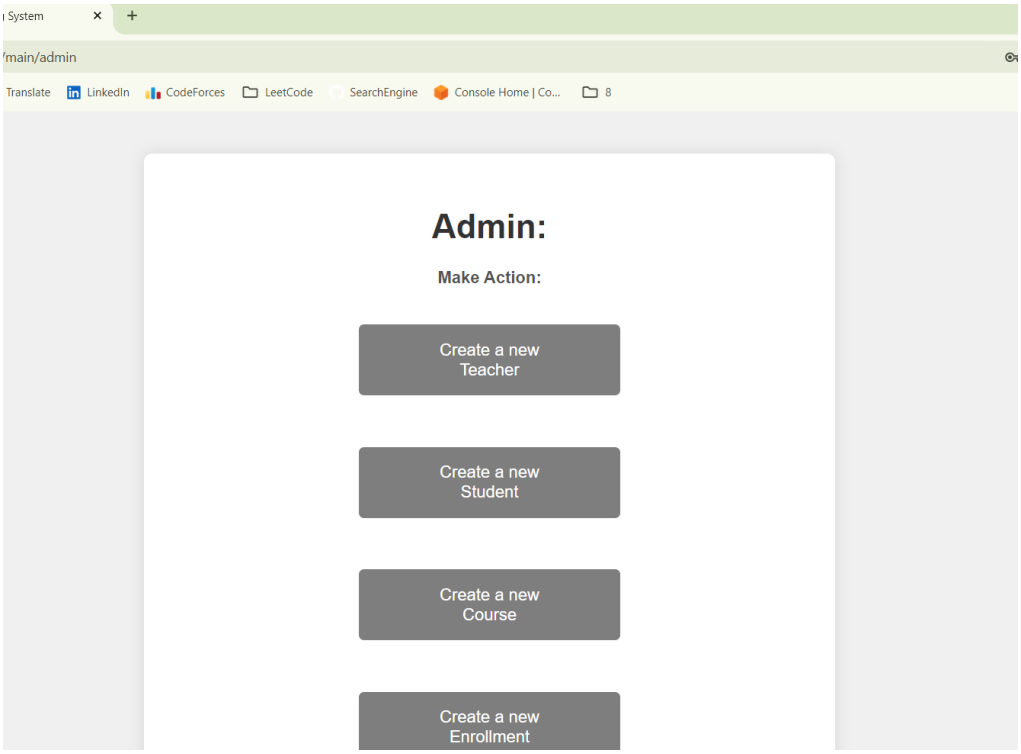
  If we run the **servlet**:



  If we click any entity (**admin** for example):



  The **authentication** is required first, after that:



We are able to create the **actions** until the **session expires**, and the same thing for the **Spring b**oot, but we will be dealing with **JWT tokens**.

Thanks

Ahmad Nabeel Al-Jaber