Department of Computer Science
Technical University of Cluj-Napoca

Software engineering

Student Management System

Name: Borz Robert-Ionuț

Group: 30434 / 1

Email: borzrobert@yahoo.com

Team: Balas Andrei Ioan, Baias Ioan Nicolae, Borz Robert-Ionuț

Teaching Assistant: Eneia Nicolae Todoran
eneia.todoran@cs.utcluj.ro

Contents

Chapter 1

Abstract

The **Student Management System** is a web-based application engineered to streamline and automate the core administrative and management tasks within educational institutions. This system is a comprehensive solution designed to handle a wide array of functionalities crucial for effective student and campus administration.

At its core, the system leverages the robustness and versatility of **Java**, integrating advanced features and modern practices in software development. Utilizing the **Spring Boot framework**, the application achieves high efficiency and scalability, ensuring seamless handling of concurrent user sessions and extensive data processing. The choice of **Spring Boot**, renowned for its dependency injection and an abundance of ready-to-use modules, facilitates rapid development and easy maintenance.

The application architecture adopts the **Model-View-Controller (MVC)** design pattern, promoting separation of concerns, which enhances code readability and maintainability.

A **key feature** of the system is its comprehensive management of student credentials and personal data, offering robust functionalities for secure data handling and privacy compliance. This includes intricate processes for authentication and authorization, underpinned by **Spring Security**, to ensure that user access is appropriately controlled and sensitive information remains protected.

In addition to student data management, the system is equipped with modules for **managing academic groups and roles**, enabling administrators to assign and manage roles within the educational framework effectively. This feature is crucial for tailoring access and functionalities according to the diverse needs of various user categories, such as students, faculty, and administrative staff.

The **Student Management System** is also designed with extendibility in mind, allowing for future enhancements and integration with other educational tools and technologies. This foresight in design underscores the system's readiness to adapt to evolving educational needs and technological advancements, making it a future-proof solution for educational institutions.

In summary, the **Student Management System** stands as a testament to advanced software engineering practices, offering a robust, scalable, and user-friendly platform for managing the dynamic and complex needs of modern educational institutions. Its technical sophistication and thoughtful design make it an invaluable tool for enhancing administrative efficiency and academic management.

Chapter 2

Design

2.1 Use Case Diagram

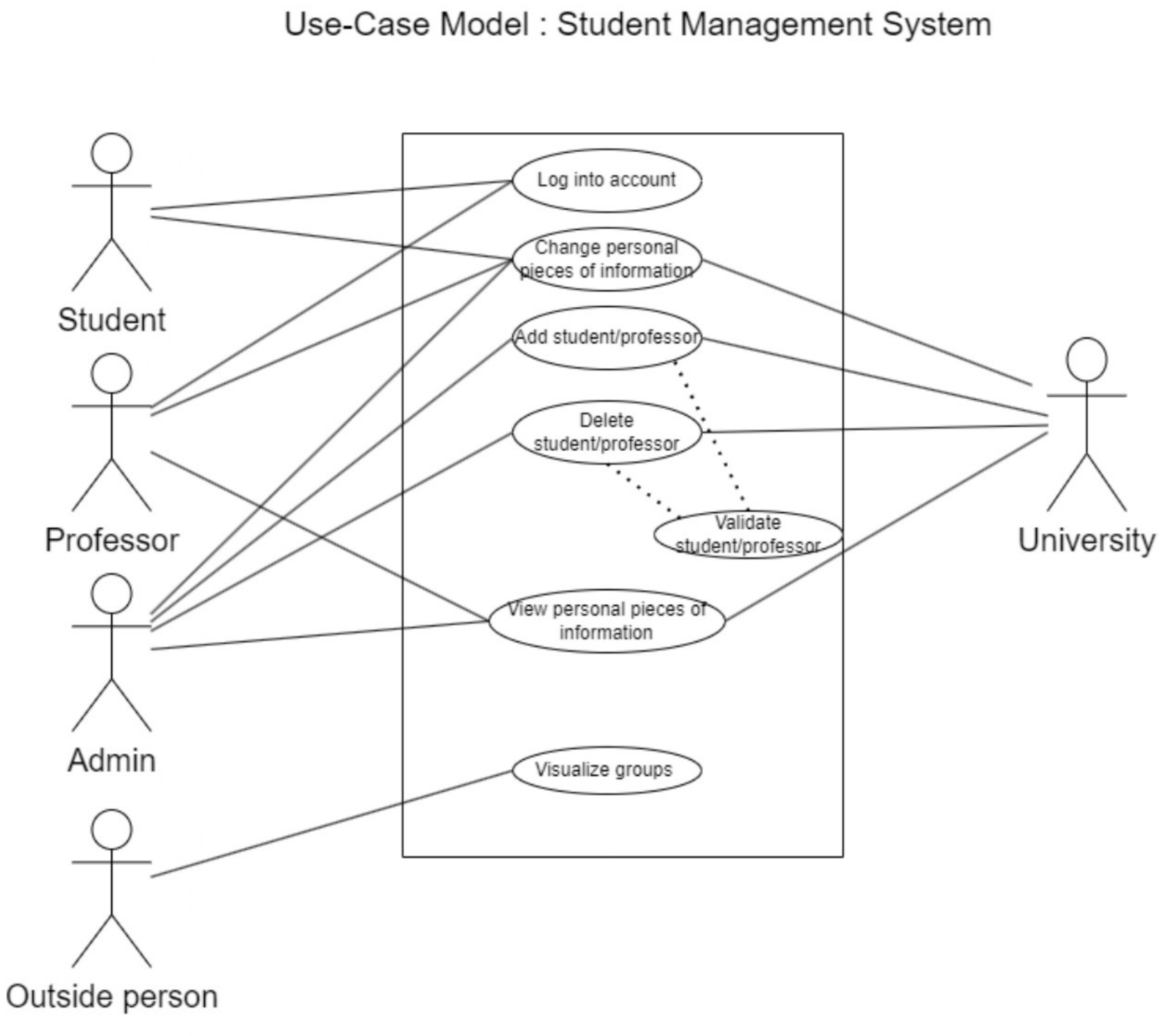

Use-Case Model : Student Management System
Student
Professor
Admin
Outside person
University
Log into account
Change personal pieces of information
Add student/professor
Delete student/professor
Validate student/professor
View personal pieces of information
Visualize groups

Figure 2.1: Use Case Diagram

This diagram outlines the interactions between users and the system. Here is a description of the diagram:

Actors:

- **Student:** A user who can log into their account, change personal information, view personal pieces of information, and may have other use cases not fully visible in the provided diagram.

- **Professor:** Can perform actions such as adding student/professor details, validating student/professor information, and possibly other actions that are not fully visible.

- **Admin:** Responsible for administrative tasks such as deleting student/professor records, visualizing groups, and other.

- **Outside Person:** An external user who can visualize the existing groups of students.

- **University:** It represents a system with overarching access or control.

Main Use Cases:

- **Log into Account:** Allowing users to access the system with their credentials.

- **Add Student/Professor:** Admins can add new student or professor records to the system.

- **Delete Student/Professor:** Admins can remove student or professor records from the system.

- **View Personal Pieces of Information:** Students can view their personal information stored in the system.

- **Validate Student/Professor:** Admins can validate the details of students or professors.

2.2 Class Diagram

The class diagram is a structural representation of a system that shows the system's classes, their attributes, methods, and the relationships between the classes.

Taking into consideration the fact that the number of classes is quite large, we will provide the class diagram for each important package(some diagrams won't be shown here, but could be found in the project's files): **controller, entity, mapper, service, service/Impl, repo**

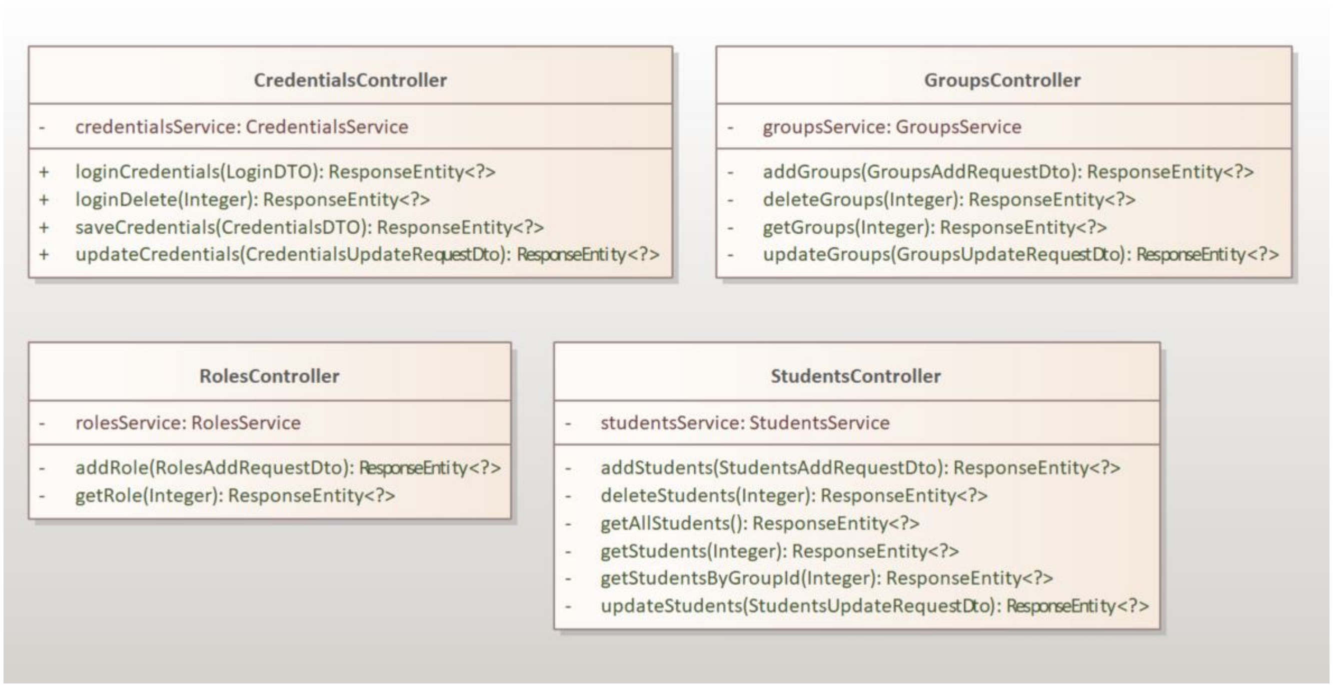

CredentialsController
- credentialsService: CredentialsService
+ loginCredentials(LoginDTO): ResponseEntity<?>
+ loginDelete(Integer): ResponseEntity<?>
+ saveCredentials(CredentialsDTO): ResponseEntity<?>
+ updateCredentials(CredentialsUpdateRequestDto): ResponseEntity<?>
GroupsController
- groupsService: GroupsService
- addGroups(GroupsAddRequestDto): ResponseEntity<?>
- deleteGroups(Integer): ResponseEntity<?>
- getGroups(Integer): ResponseEntity<?>
- updateGroups(GroupsUpdateRequestDto): ResponseEntity<?>
RolesController
- rolesService: RolesService
- addRole(RolesAddRequestDto): ResponseEntity<?>
- getRole(Integer): ResponseEntity<?>
StudentsController
- studentsService: StudentsService
- addStudents(StudentsAddRequestDto): ResponseEntity<?>
- deleteStudents(Integer): ResponseEntity<?>
- getAllStudents(): ResponseEntity<?>
- getStudents(Integer): ResponseEntity<?>
- getStudentsByGroupId(Integer): ResponseEntity<?>
- updateStudents(StudentsUpdateRequestDto): ResponseEntity<?>

Figure 2.2: Class Diagram for **controller** package

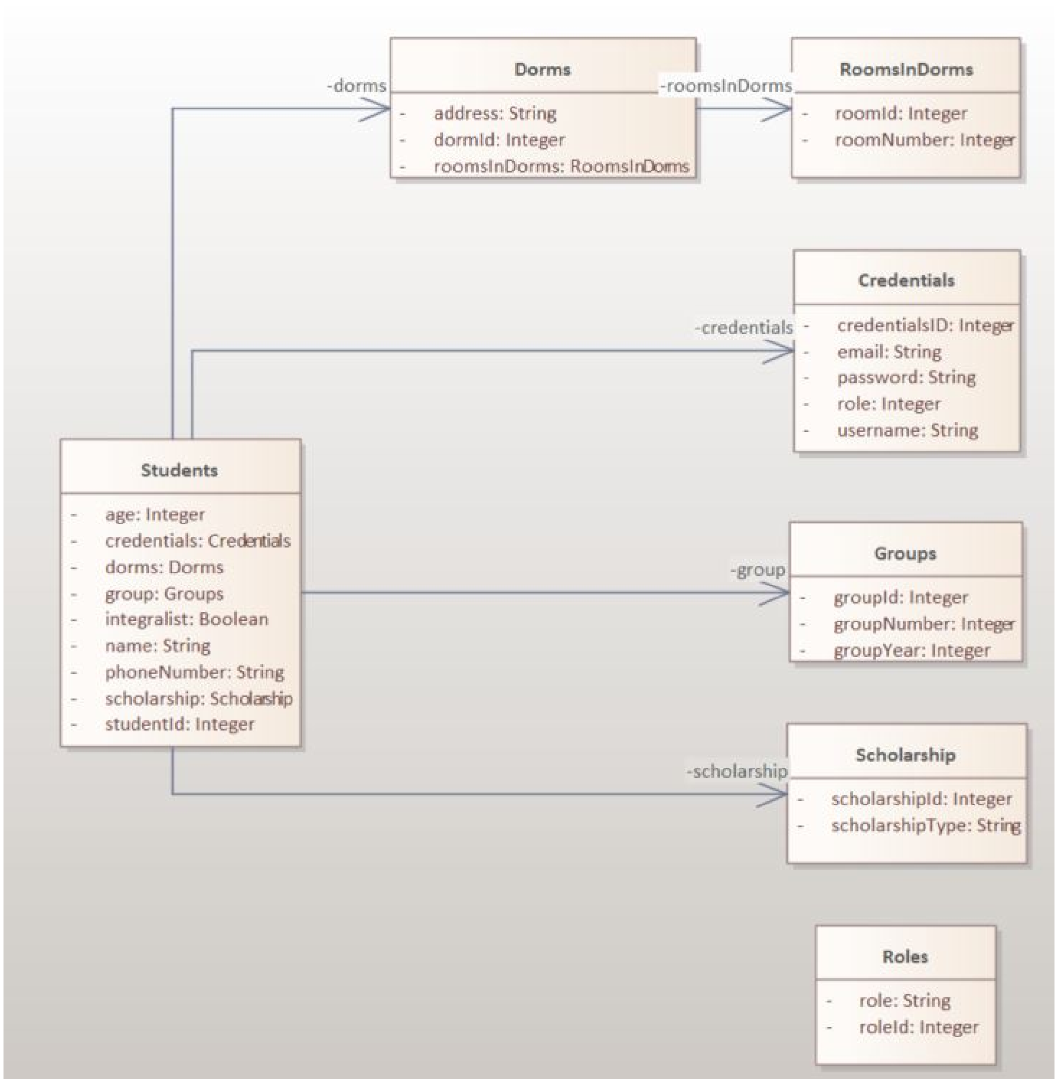


Figure 2.3: Class Diagram for **entity** package

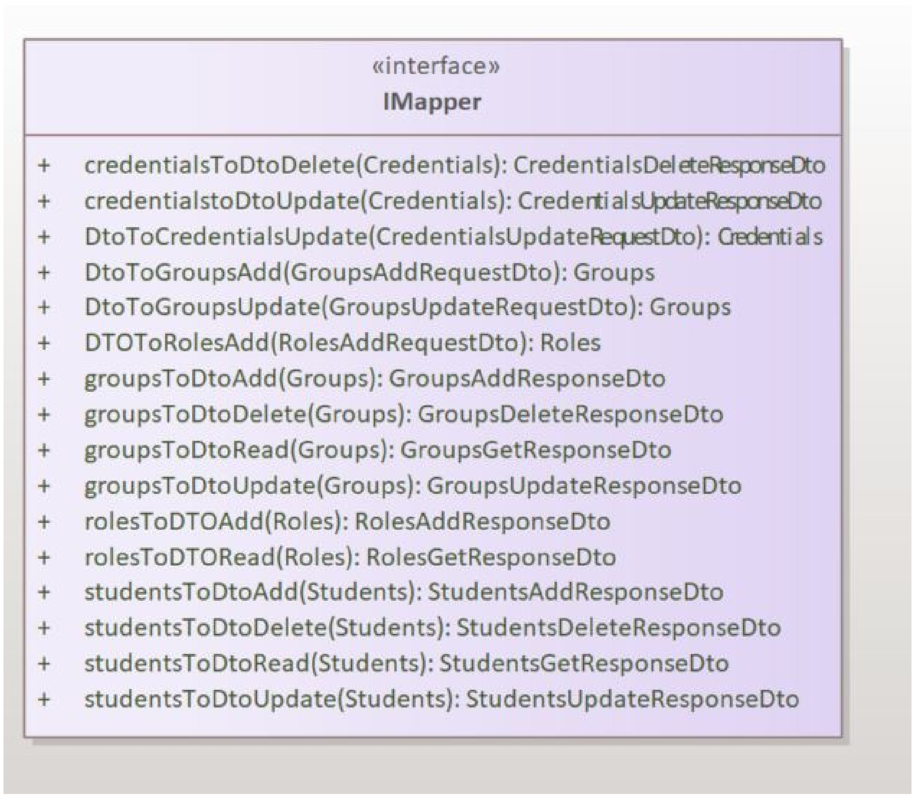

«interface»
IMapper
+ credentialsToDtoDelete(Credentials): CredentialsDeleteResponseDto
+ credentialstoDtoUpdate(Credentials): CredentialsUpdateResponseDto
+ DtoToCredentialsUpdate(CredentialsUpdateRequestDto): Credentials
+ DtoToGroupsAdd(GroupsAddRequestDto): Groups
+ DtoToGroupsUpdate(GroupsUpdateRequestDto): Groups
+ DTOToRolesAdd(RolesAddRequestDto): Roles
+ groupsToDtoAdd(Groups): GroupsAddResponseDto
+ groupsToDtoDelete(Groups): GroupsDeleteResponseDto
+ groupsToDtoRead(Groups): GroupsGetResponseDto
+ groupsToDtoUpdate(Groups): GroupsUpdateResponseDto
+ rolesToDTOAdd(Roles): RolesAddResponseDto
+ rolesToDTORead(Roles): RolesGetResponseDto
+ studentsToDtoAdd(Students): StudentsAddResponseDto
+ studentsToDtoDelete(Students): StudentsDeleteResponseDto
+ studentsToDtoRead(Students): StudentsGetResponseDto
+ studentsToDtoUpdate(Students): StudentsUpdateResponseDto

Figure 2.4: Class Diagram for **mapper** package

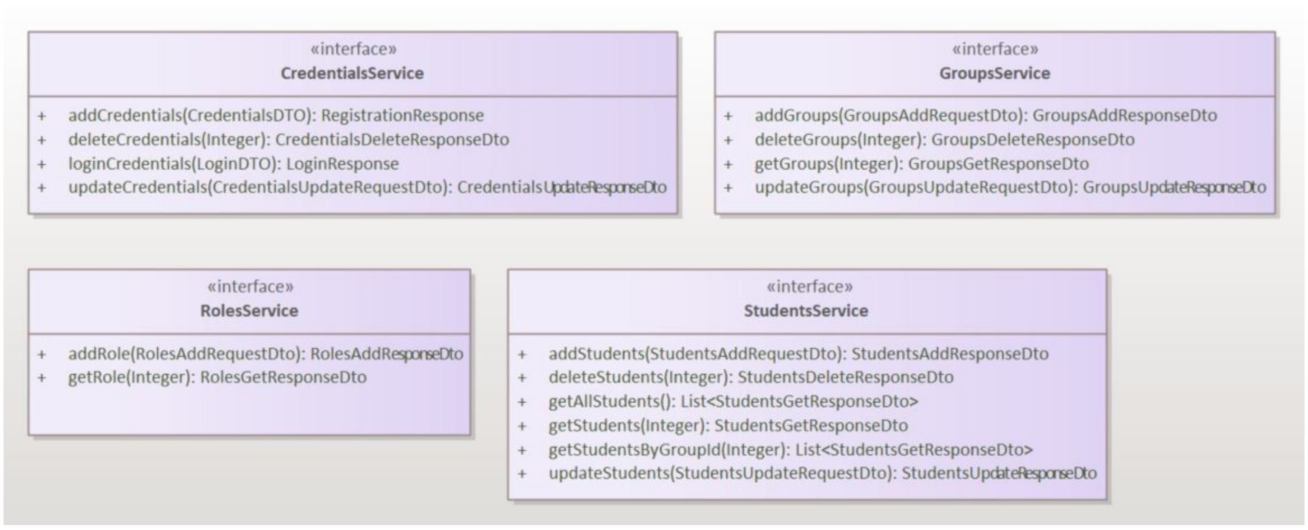

«interface»
CredentialsService
+ addCredentials(CredentialsDTO): RegistrationResponse
+ deleteCredentials(Integer): CredentialsDeleteResponseDto
+ loginCredentials(LoginDTO): LoginResponse
+ updateCredentials(CredentialsUpdateRequestDto): CredentialsUpdateResponseDto
«interface»
GroupsService
+ addGroups(GroupsAddRequestDto): GroupsAddResponseDto
+ deleteGroups(Integer): GroupsDeleteResponseDto
+ getGroups(Integer): GroupsGetResponseDto
+ updateGroups(GroupsUpdateRequestDto): GroupsUpdateResponseDto
«interface»
RolesService
+ addRole(RolesAddRequestDto): RolesAddResponseDto
+ getRole(Integer): RolesGetResponseDto
«interface»
StudentsService
+ addStudents(StudentsAddRequestDto): StudentsAddResponseDto
+ deleteStudents(Integer): StudentsDeleteResponseDto
+ getAllStudents(): List<StudentsGetResponseDto>
+ getStudents(Integer): StudentsGetResponseDto
+ getStudentsByGroupId(Integer): List<StudentsGetResponseDto>
+ updateStudents(StudentsUpdateRequestDto): StudentsUpdateResponseDto

Figure 2.5: Class Diagram for **service** package

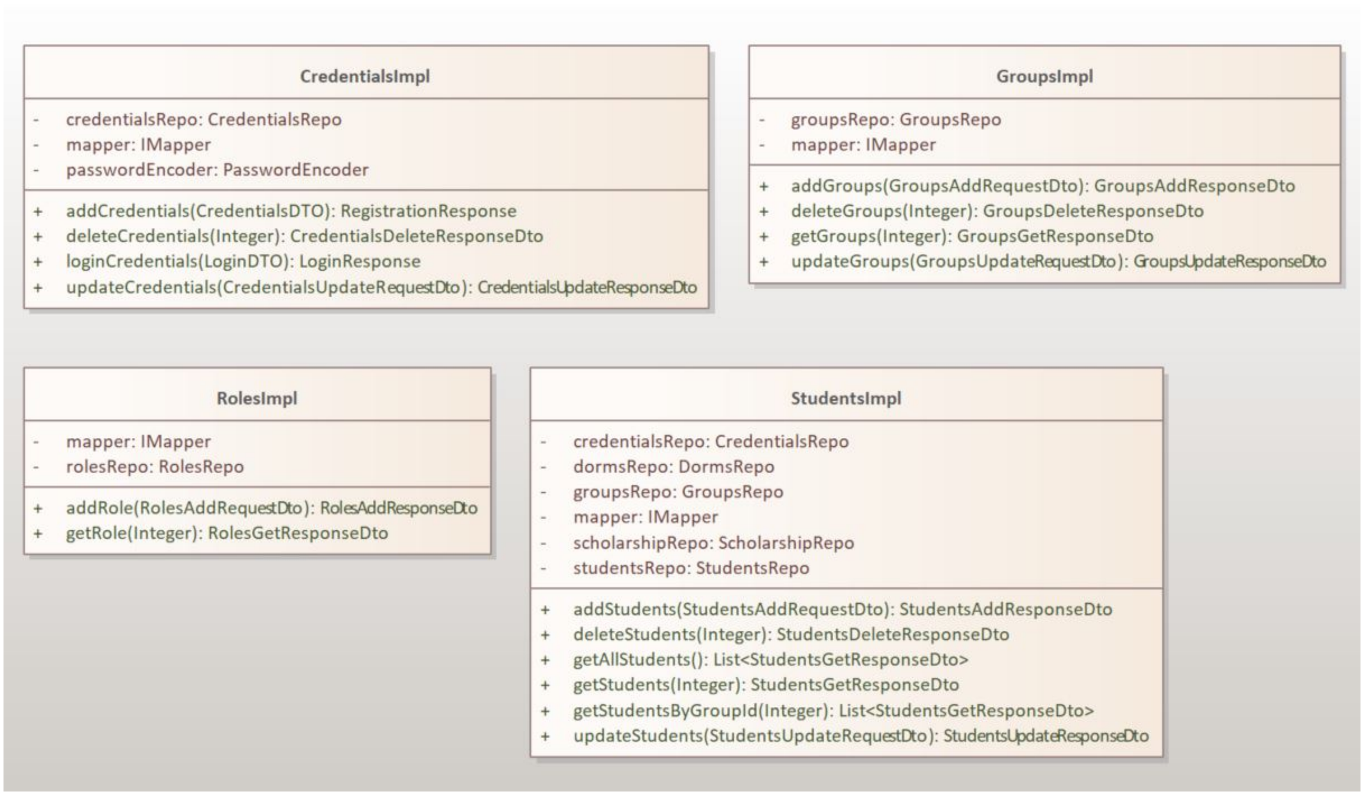

CredentialsImpl
- credentialsRepo: CredentialsRepo
- mapper: IMapper
- passwordEncoder: PasswordEncoder
+ addCredentials(CredentialsDTO): RegistrationResponse
+ deleteCredentials(Integer): CredentialsDeleteResponseDto
+ loginCredentials(LoginDTO): LoginResponse
+ updateCredentials(CredentialsUpdateRequestDto): CredentialsUpdateResponseDto
GroupsImpl
- groupsRepo: GroupsRepo
- mapper: IMapper
+ addGroups(GroupsAddRequestDto): GroupsAddResponseDto
+ deleteGroups(Integer): GroupsDeleteResponseDto
+ getGroups(Integer): GroupsGetResponseDto
+ updateGroups(GroupsUpdateRequestDto): GroupsUpdateResponseDto
RolesImpl
- mapper: IMapper
- rolesRepo: RolesRepo
+ addRole(RolesAddRequestDto): RolesAddResponseDto
+ getRole(Integer): RolesGetResponseDto
StudentsImpl
- credentialsRepo: CredentialsRepo
- dormsRepo: DormsRepo
- groupsRepo: GroupsRepo
- mapper: IMapper
- scholarshipRepo: ScholarshipRepo
- studentsRepo: StudentsRepo
+ addStudents(StudentsAddRequestDto): StudentsAddResponseDto
+ deleteStudents(Integer): StudentsDeleteResponseDto
+ getAllStudents(): List<StudentsGetResponseDto>
+ getStudents(Integer): StudentsGetResponseDto
+ getStudentsByGroupId(Integer): List<StudentsGetResponseDto>
+ updateStudents(StudentsUpdateRequestDto): StudentsUpdateResponseDto

Figure 2.6: Class Diagram for **service/Impl** package

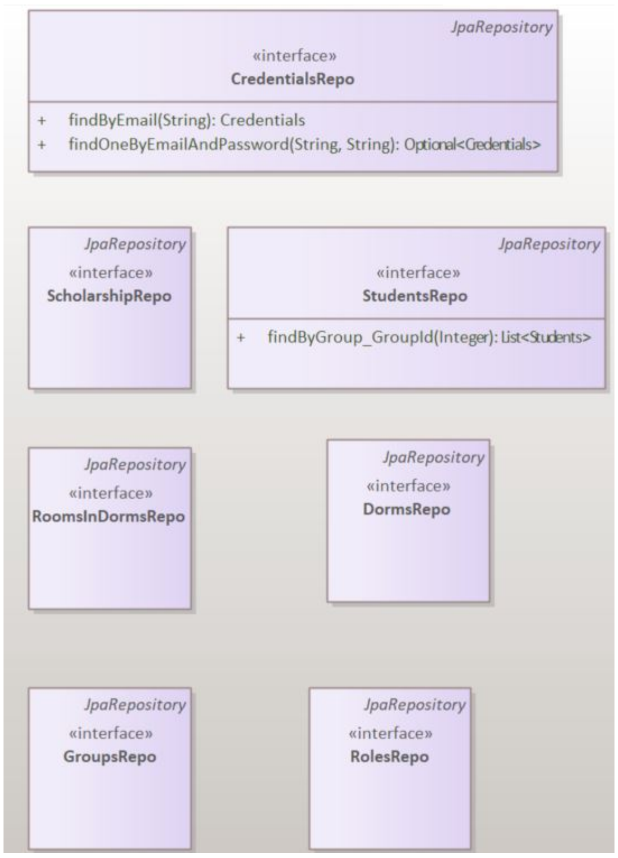


Figure 2.7: Class Diagram for **repo** package

2.3 Deployment Diagram

A deployment diagram in the context of a software system is a type of UML diagram that shows the physical deployment of artifacts (software components, libraries, and executables) on nodes (hardware).

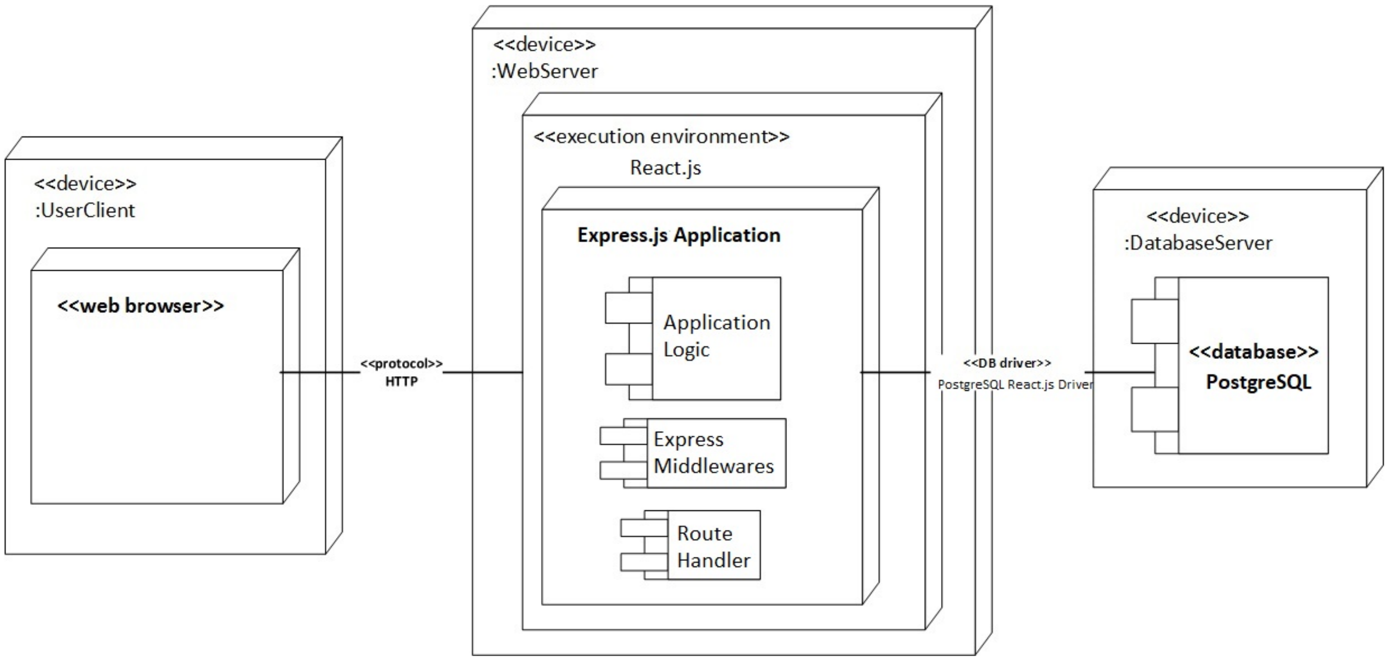


Figure 2.8: Deployment Diagram

Components:

- **UserClient Device:** This represents the client-side device, which is typically the user's computer or a mobile device. It contains a web browser, through which the user interacts with the application.

- **WebServer Device:** This device hosts the server-side application. It is running an "Express.js Application" within a "React.js" execution environment, suggesting that the server delivers a web application built with React for the frontend.

- **DatabaseServer Device:** This represents a separate server for the database, which in this case is a PostgreSQL database.

The connections between these components are also important:

- **HTTP:** The protocol used between the UserClient and the WebServer, indicating that the user's browser communicates with the server over HTTP.

- **DB driver:** The connection between the WebServer and the DatabaseServer, indicating that the Express.js application communicates with the database using a database driver.

Chapter 3

Implementation(personal work)

3.1 Introduction

For this specific project I have chosen to study the **Client-Server communication using HTPP protocol** and to apply the assimilated knowledge in order to develop together with my team, a robust and useful web application.

The tool that I have mainly used in this scope is **Postman**, a tool used to write functional tests, integration tests, regression tests, and more.

3.2 Understanding and Implementing HTTP Protocols

As part of my responsibilities in the team, a significant focus was placed on understanding and effectively implementing HTTP protocols. This was crucial for ensuring robust client-server communication and the efficient functioning of our student management system.

3.3 Core HTTP Methods Used

1. GET Method:

I utilized the **GET** method to retrieve data from the server. This was particularly relevant for endpoints where information needed to be fetched without affecting the server state. For instance, retrieving student details.

In testing these endpoints, I ensured that the **GET** requests were idempotent, meaning multiple requests would yield the same result without additional side effects.

2. POST Method:

The **POST** method was used for creating new resources on the server. This included tasks like registering new students or creating new groups.

3. PUT Method:

I employed the **PUT** method for updating existing resources. This was crucial in scenarios like modifying student information or updating group details.

A key part of my testing here was to validate that the **PUT** requests not only updated resources successfully but also adhered to idempotency, similar to GET requests. I verified that subsequent **PUT** requests with the same data did not lead to unexpected behaviors.

4. DELETE Method:

The **DELETE** method was implemented to remove resources from the server. This action was necessary for functionalities like removing students from the system or deleting groups.

My responsibility included testing these endpoints to ensure that they not only performed the intended deletion but also handled errors gracefully, such as attempting to delete non-existent resources.

3.4 Conclusions

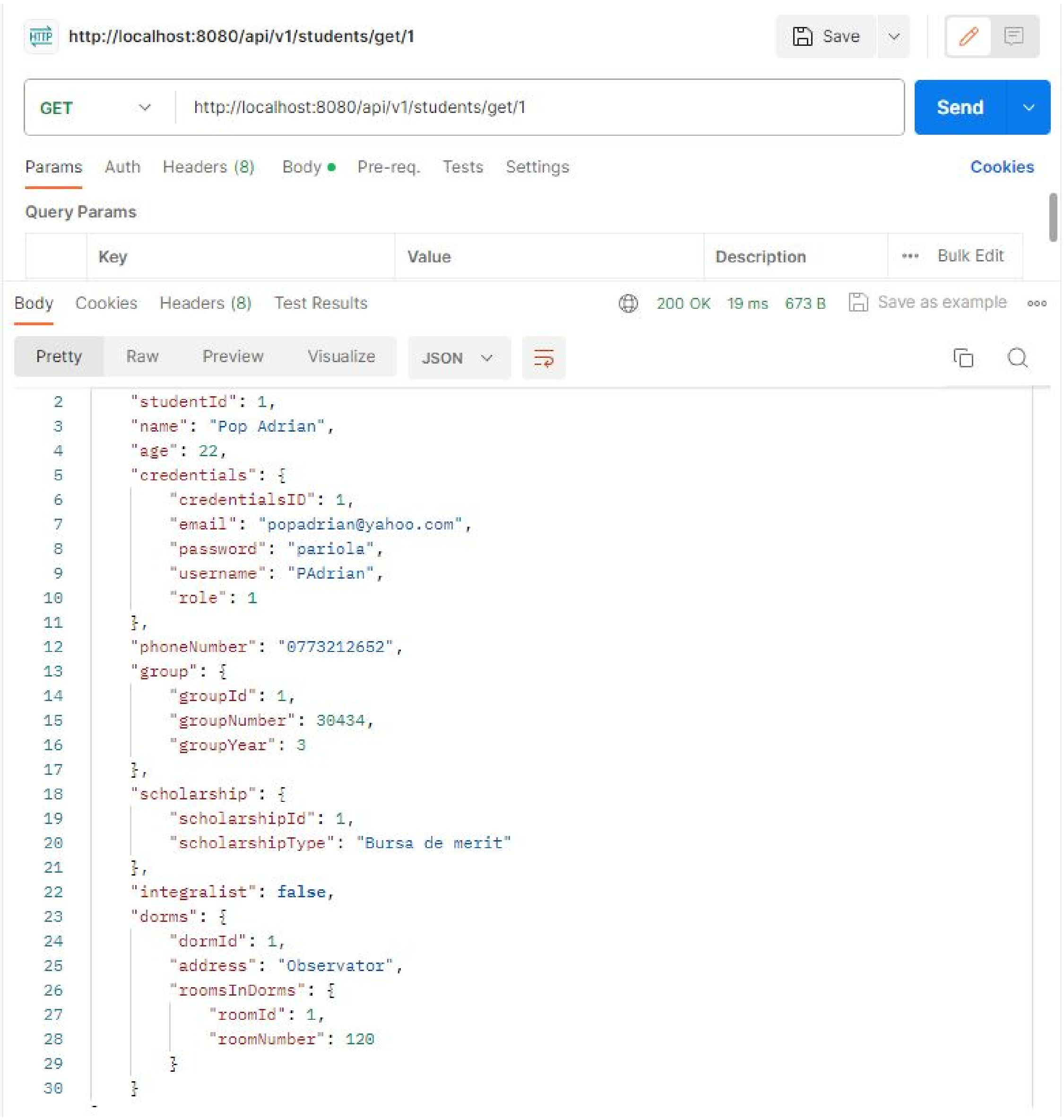
Through hands-on experience with these HTTP methods, I gained a deeper understanding of RESTful services and how they operate within a client-server architecture.

Testing these methods in various scenarios enabled me to appreciate the nuances of each and the importance of correctly implementing them to ensure the system's reliability and efficiency.

I also learned to appreciate the importance of detailed documentation, as it played a pivotal role in understanding each endpoint's expected behavior and in communicating effectively with my team.

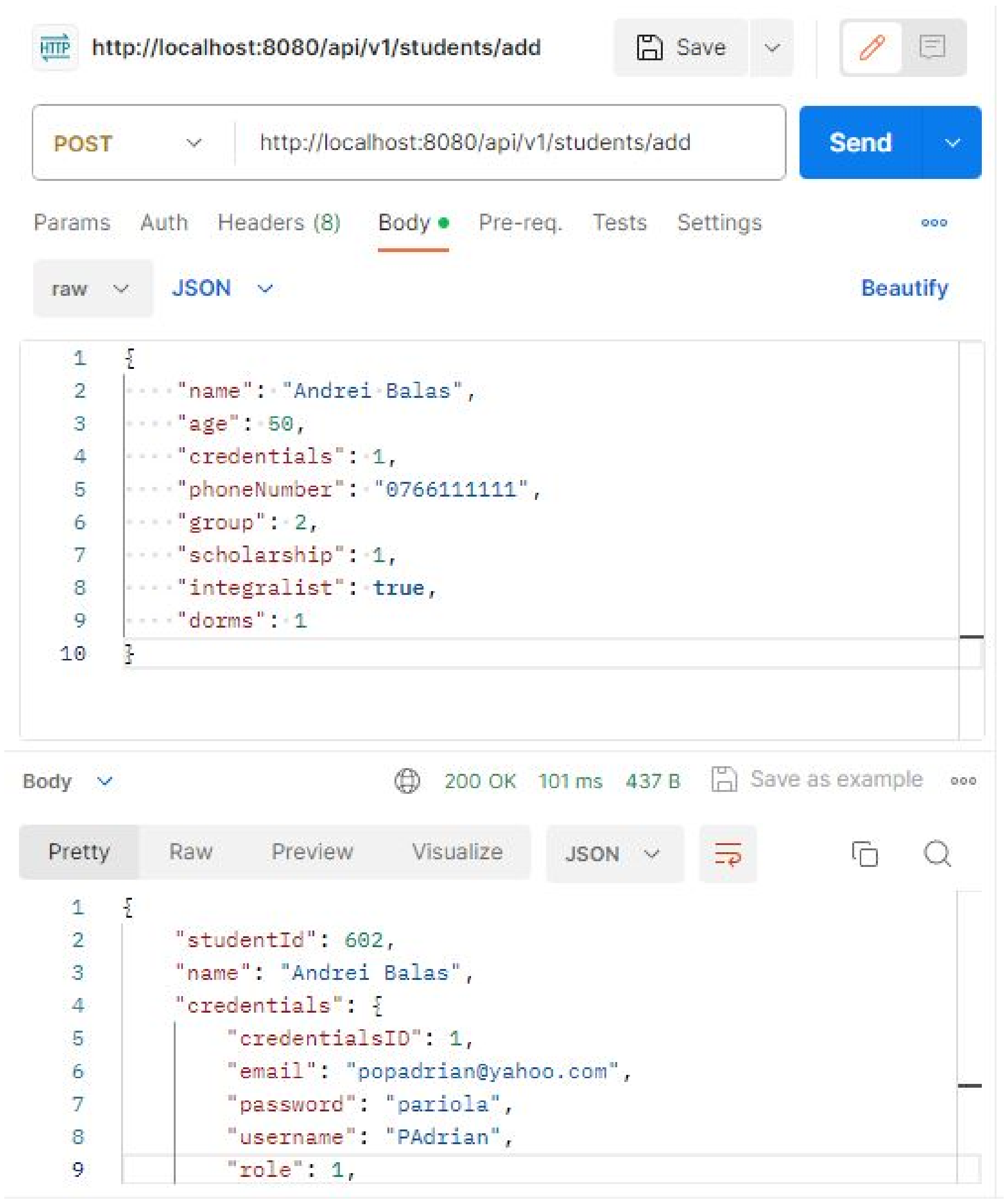
3.5 Examples

More examples could be provided if necessary



```
    "studentId": 1,
    "name": "Pop Adrian",
    "age": 22,
    "credentials": {
        "credentialsID": 1,
        "email": "popadrian@yahoo.com",
        "password": "pariola",
        "username": "PAdrian",
        "role": 1
    },
    "phoneNumber": "0773212652",
    "group": {
        "groupId": 1,
        "groupNumber": 30434,
        "groupYear": 3
    },
    "scholarship": {
        "scholarshipId": 1,
        "scholarshipType": "Bursa de merit"
    },
    "integralist": false,
    "dorms": {
        "dormId": 1,
        "address": "Observator",
        "roomsInDorms": {
            "roomId": 1,
            "roomNumber": 120
        }
    }
```

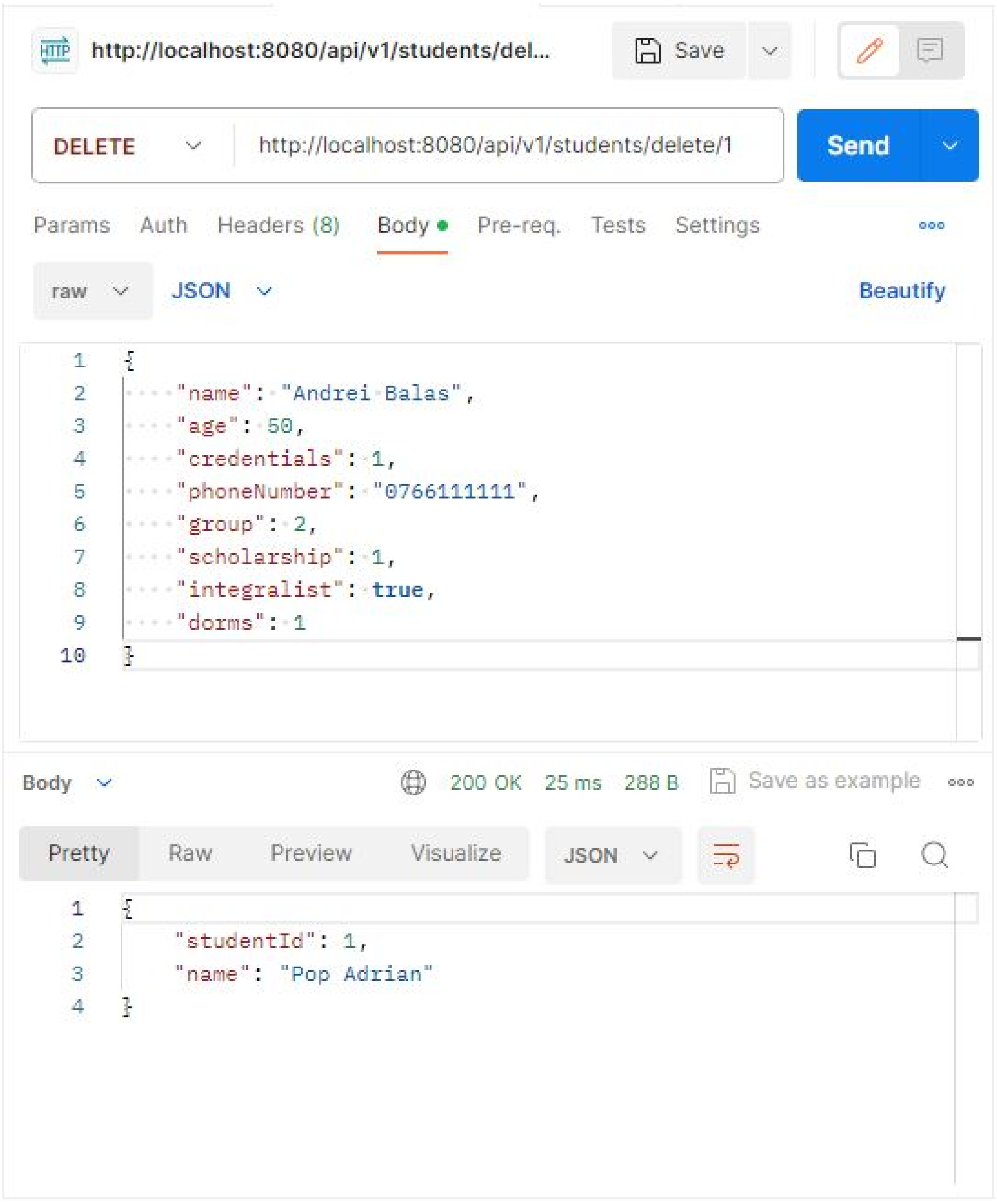
Figure 3.1: **GET** example



```
{
    "name": "Andrei Balas",
    "age": 50,
    "credentials": 1,
    "phoneNumber": "0766111111",
    "group": 2,
    "scholarship": 1,
    "integralist": true,
    "dorms": 1
}
```

```
{
    "studentId": 602,
    "name": "Andrei Balas",
    "credentials": {
        "credentialsID": 1,
        "email": "popadrian@yahoo.com",
        "password": "pariola",
        "username": "PAdrian",
        "role": 1,
```

Figure 3.2: **POST** example



DELETE http://localhost:8080/api/v1/students/delete/1

```
{
    "name": "Andrei Balas",
    "age": 50,
    "credentials": 1,
    "phoneNumber": "0766111111",
    "group": 2,
    "scholarship": 1,
    "integralist": true,
    "dorms": 1
}
```

200 OK 25 ms 288 B

```
{
    "studentId": 1,
    "name": "Pop Adrian"
}
```

Figure 3.3: **DELETE** example

Bibliography

Software Engineering Course/Laboratory, Fall, 2023-2024

https://stackoverflow.com/

https://www.geeksforgeeks.org/

https://coq.inria.fr/

https://spring.io/

Appendix A

Mini project(code source only)

•controller package:

MessageController.java

```java
+import ...

@Controller
public class MessageController {

    MessageRepo repo;

    public MessageController(MessageRepo entityRepository) {
        this.repo = entityRepository;
    }

    @GetMapping("/hello")
    public String helloWorld(Model model) {

        Optional<Message> optionalEntity = repo.findById(1L);

        if(optionalEntity.isPresent()){
            Message entity = optionalEntity.get();
            model.addAttribute("message",entity.getMessage());
            return "helloPage";
        }
        else{
            model.addAttribute("message","Failed to read from database!")
                ;
            return "helloPage";
        }

    }

    @GetMapping("/idea")
    public String Idea(Model model) {
        return "idea";
    }

}
```

PersonController.java

```java
+import ...

@Controller
public class PersonController {

    PersonRepo repo;

    public PersonController(PersonRepo entityRepository) {
        this.repo = entityRepository;
    }

    @GetMapping("/team")
    public String Team(Model model) {
        int index=1;
        ArrayList<Person> members = (ArrayList<Person>) repo.findAll();

        if(!members.isEmpty()){
            for (Person member: members
                 ) {
                model.addAttribute("message"+index, member.getFirstName()
                    + " " +member.getLastName());
                index++;
            }
            return "team";
        }
        else{
            model.addAttribute("message","Failed to read from database!")
                ;
            return "team";
        }
    }
}
```

●model package:

Message.java

```java
+import ...



@Data
@Entity
@Table(name="message")
@ToString
public class Message {
    @Id
    private Long id;
    private String message;
}
```

Person.java

```java
+import ...

@Data
@Entity
@Table(name="person")
@ToString
public class Person {
    @Id
    private Long id;
    private String firstName;
    private String lastName;
}
```

•repo package:

MessageRepo.java

```java

@Repository
public interface MessageRepo extends JpaRepository<Message, Long> {
}
```

PersonRepo.java

```java
+import ...

@Repository
public interface PersonRepo extends JpaRepository<Person, Long> {
}
```

•main package:

DemoApplication.java

```java

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

The **View** part of the Mini project consists of three HTML files, with basic code in order to show a message taken from database.

The source code for the **Final Project** will be provided by the team.