JPA and PostgreSQL (Grade Submission Portal)

Overview of JPA and PostgreSQL

JPA

- Java Persistence API (JPA) is a specification that provides a standard for managing relational data in Java applications.
- JPA allows developers to **map Java objects to database tables** and manage their relationships.

Installing required dependencies

To get started, we need to install the required dependencies in our Spring Boot project:

1. Spring Boot Starter Data JPA: A starter dependency for using JPA with Hibernate as the default implementation.
2. PostgreSQL Driver: The JDBC driver to connect and interact with the PostgreSQL database.

To add these dependencies, open your `pom.xml` file and add the following lines inside the `<dependencies>` section:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Configuring the application properties

Now that we have the necessary dependencies, we need to configure our Spring Boot application to use PostgreSQL as the database.

Open the `application.properties` file located in the `src/main/resources` folder and add the following lines:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/grade_submission_portal
spring.datasource.username=postgres
spring.datasource.password=your_postgres_password

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true
```

Replace `your_postgres_password` with the password for your PostgreSQL user.

These properties configure the connection to the PostgreSQL database and set up Hibernate to manage the database schema.

Now, your Spring Boot application is ready to use JPA with PostgreSQL.

In the next part of the tutorial, we will create the entity classes and define their relationships using JPA annotations.

4.3 Object Relational Mapper

Introduction to ORM

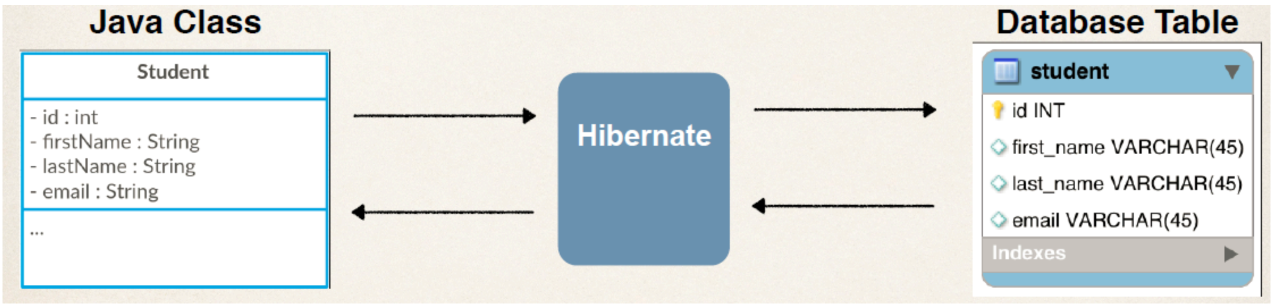
- An Object Relational Mapper (ORM) is a tool that simplifies the **interaction between an object-oriented programming language and a relational database**.

ORM

- An ORM maps objects and their properties to database tables and columns, handling the conversion between the two.

How JPA works as an ORM

- JPA provides a **set of annotations and interfaces that developers can use** to define the mapping between Java objects and database tables.
- **Hibernate** is a popular implementation of the JPA specification and is the default implementation used by Spring Boot.



Hibernate

JPA Entities

- In JPA, **entities** are Java classes that represent the **tables** in the database. Entities are annotated with `@Entity` and should have a primary key annotated with `@Id`.
- The **properties** of the entity class map to the **columns** in the database table.

JPA Repositories

- JPA **repositories** are interfaces that extend `JpaRepository` and provide methods for performing CRUD operations on the entities.

Spring Boot + ORM + PostgreSQL

Spring Boot provides built-in support for various ORM frameworks such as **Hibernate**, which can be used to **interact with the PostgreSQL** database.

To use PostgreSQL with Spring Boot, you'll need to add the necessary dependencies in your project's build file such as `pom.xml` for Maven and configure the database connection properties in the `application.properties` file.

Here's an example of adding the PostgreSQL dependency in a Maven-based Spring Boot project:

```
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>42.3.1</version>
</dependency>
```

Once you have added the dependency, you can create a database connection using the following configuration in the application.properties file:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mydb
spring.datasource.username=myusername
spring.datasource.password=mypassword
spring.datasource.driver-class-name=org.postgresql.Driver
```

With the database connection configured, you can now use an ORM framework such as Hibernate to interact with the PostgreSQL database. You can define your entity classes and use annotations to map them to database tables. Here's an example of a simple entity class for a "user" table:

```
@Entity
@Table(name = "users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(nullable = false)
   private String name;

   @Column(nullable = false)
   private String email;

   // getters and setters
}
```

With the entity class defined, you can now use Hibernate to perform database operations such as saving and retrieving data. For example, you can use a JpaRepository interface to provide **CRUD (Create, Read, Update, Delete)** operations for the "user" entity:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
   // custom methods can be added here
}
```

Overall, using a PostgreSQL database and ORM in a Spring Boot MVC application is straightforward and can be accomplished with just a few configuration steps.

4.4 Student Entity (from ArrayList to PostgreSQL)

Migrating from ArrayList to PostgreSQL in Spring Boot

1) Change the Student to an @Entity

- Annotate the class with `@Entity` to indicate that it represents a table in the database.
- Define a primary key using the `@Id` annotation and generate a value for it using the `@GeneratedValue` annotation.

Transform your Student class into a JPA entity:

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "First name cannot be blank")
    @Column(name = "first_name", nullable = false)
    private String firstName;

    @NotBlank(message = "Last name cannot be blank")
    @Column(name = "last_name", nullable = false)
    private String lastName;

    // Getters and setters
}
```

Key changes:

- Added `@Entity` annotation
- Defined primary key with `@Id` and `@GeneratedValue`
- Added column mappings with `@Column`
- Removed UUID-based studentId generation

The creation of the `student` table in the database is triggered by the `@Entity` annotation in the `Student` class and the Hibernate framework. When your Spring Boot application starts, it scans for classes annotated with `@Entity`.

Grade Entity Updates:

```
public class Grade {
    // ... other fields ...

    @NotNull(message = "You must select a student!")
    private Long studentId;  // Changed from String to Long

    private Student student; // reference to the student object

    // Updated setter and getter
    public void setStudentId(Long studentId) {
        this.studentId = studentId;
    }

    public Long getStudentId() {
        return this.studentId;
    }
}
```

2) StudentRepository - Create JPA Repository

- The `JpaRepository` is a generic interface that provides basic CRUD (Create, Read, Update, Delete) operations for a specific entity class (Student) and its primary key type (`Long` in our example).

An **interface** in Java is a list of actions (methods) for a class to follow. It sets the rules without providing implementation details.

Replace the ArrayList-based repository with JPA:

StudentRepository.java

```java
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
}

```

This single interface provides:

- CRUD operations
- Pagination
- Sorting capabilities

3) StudentService - Service Layer Refactoring

Basic CRUD Operations:

```
@Service
@Transactional
public class StudentService {
    private final StudentRepository studentRepository;

    public StudentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    public Student getStudentById(Long studentId) {
        return studentRepository.findById(studentId)
            .orElseThrow(() -> new StudentNotFoundException("Student
not found with id: " + studentId));
    }

    public List<Student> getAllStudents() {
        return studentRepository.findAll();
    }

    public Student saveStudent(Student student) {
        return studentRepository.save(student);
    }

    public void deleteStudent(Long studentId) {
        Student student = getStudentById(studentId);
        studentRepository.delete(student);
    }
}
```

Key changes:

- Added `@Transactional` annotation
- Replaced ArrayList operations with JPA methods
- Improved error handling with custom exceptions
- Removed manual index tracking

`@Transactional` ensures database operations are atomic - they either all succeed or all fail together. If an exception occurs during a transaction, all changes are automatically rolled back, maintaining data consistency.

4) Adapting the StudentController

```
@Controller
public class StudentController {
    private final StudentService studentService;

    @PostMapping("/studentSubmit")
    public String submitStudent(@Valid Student student, BindingResult
bindingResult) {
        if (bindingResult.hasErrors()) {
            return "add_student";
        }
        studentService.saveStudent(student);
        return "redirect:/students";
    }

    @GetMapping("/addstudent")
    public String getStudentForm(Model model, @RequestParam(required =
false) Long id) {
        Student student = (id != null)
            ? studentService.getStudentById(id)
            : new Student();
        model.addAttribute("student", student);
        return "add_student";
    }

    @GetMapping("/delete-student")
    public String deleteStudent(@RequestParam Long studentId) {
        studentService.deleteStudent(studentId);
        return "redirect:/students";
    }
}
```

Key changes:

- Updated parameter types from String to Long for IDs
- Improved null handling
- Simplified student creation/update logic

Grade Controller Adjustments:

```
@Controller
public class GradeController {
    @GetMapping("/")
    public String getForm(Model model) {
        model.addAttribute("grade", new Grade());
        // Updated to use new service method name
        model.addAttribute("students",
studentService.getAllStudents());
        return "form";
    }

    @PostMapping("/handleSubmit")
    public String handleSubmit(@Valid Grade grade) {
        // Updated to use new student service method
        Student student =
studentService.getStudentById(grade.getStudentId());
        grade.setStudent(student);
        // ...
    }
}
```

5) Update Thymeleaf Templates

```
<!-- add_student.html -->
<form method="post" th:object="${student}"
th:action="@{/studentSubmit}">
    <input type="text" placeholder="First Name" th:field="*
{firstName}">
    <p style="color:red;" th:errors="*{firstName}"></p>
    <input type="text" placeholder="Last Name" th:field="*{lastName}">
    <p style="color:red;" th:errors="*{lastName}"></p>
    <input type="hidden" th:field="*{id}">
    <br><br>
    <input type="submit" value="Submit">
</form>

<!-- students.html -->
<tr th:each="student: ${students}">
    <td th:text="${student.firstName}"></td>
    <td th:text="${student.lastName}"></td>
    <td><a role="button" class="update"
th:href="@{/addstudent(id=${student.id})}">Update</a></td>
    <td><a role="button" class="update" th:href="@{/delete-
student(studentId=${student.id})}">Delete</a></td>
</tr>
```

Key changes:

- Updated form fields to match entity structure
- Changed ID references from studentId to id
- Simplified form structure

Grades form.html

```
<select name="studentId" th:field="*{studentId}">
    <option value="" hidden>Select a student</option>
    <!-- Updated to use student.id instead of student.studentId -->
    <option th:each="student : ${students}"
            th:value="${student.id}"
            th:text="${student.firstName} + ' ' + ${student.lastName}">
    </option>
</select>
```

6) Custom Exception

```java
// exception/StudentNotFoundException.java
public class StudentNotFoundException extends RuntimeException {
    public StudentNotFoundException(String message) {
        super(message);
    }
}
```

`StudentNotFoundException` extends `RuntimeException` to create a specific error type.

- Spring detects the `StudentNotFoundException`
- Routes it to the matching `@ExceptionHandler` method
- Handler creates a clean error response

Without custom exception handling, when a student isn't found:

- A generic 500 Internal Server Error is returned
- User sees a technical stack trace
- No clear message explaining what went wrong

7) Testing the Migration

1. Start PostgreSQL server
2. Create the database: `grade_submission_portal`
3. Run the application
4. Verify table creation
5. Test CRUD operations

4.5 Grade Entity & One-to-Many

Use a new **git branch** "**feature-grade**" from **development**

In this section, we will update the existing `Grade` entity and `GradeController` to establish a One-to-Many relationship between Student and Grade. This will allow us to store the grades of each student and view them on a separate page.

1. Creating the **Grade entity** & adding a **Many-to-One** relationship between Grade and Student
2. Creating the **GradeRepository**
3. Adding a **One-to-Many** relationship between Student and Grade
4. Updating the **repositories and controllers**
5. Update the **details.html** view

1. Creating the Grade entity

Update the existing Grade class to include the **Many-to-One** relationship with the Student entity using the `@ManyToOne` annotation. Replace the existing code in the `Grade.java` file with the following code:

Grade.java

```
@Entity
@Table(name = "grades")
public class Grade {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String subject;
    private Double score;

    //Each grade is associated with one and only one student
    //FetchType.LAZY is used to optimize performance
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "student_id", nullable = false)
    private Student student;

    // Constructor
    public Grade() {

    }

    // getters and setters

}
```

In the code above, we have defined a new entity called Grade with three attributes: id, subject, and score. We have also defined a **Many-to-One** relationship with the Student entity using the **@ManyToOne** annotation:

This relationship specifies that: **"each grade is associated with one and only one student."**

`fetch = FetchType.LAZY` is used to optimize performance and reduce unnecessary data retrieval by only fetching related entities when they are actually needed.

2. Creating the GradeRepository

Create a new interface called `GradeRepository.java`. Add the following code to the **GradeRepository** interface:

GradeRepository.java

```java
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface GradeRepository extends JpaRepository<Grade, Long>
{
    // Method to retrieve all grades for a particular student
    List<Grade> findByStudentId(Long studentId);
}
```

In the code above, we have defined a new repository called GradeRepository. We have also defined a new method called `findByStudentId`, which retrieves all grades for a particular student.

The method name follows the Spring Data JPA naming convention, which allows you to specify the **property name** and a **query keyword** to generate the query.

In this case, the property name is `studentId` and the query keyword is `findBy`, which generates a query to retrieve all `Grade` entities where the `studentId` property matches the specified `studentId`.

"Behind the scenes, Spring Data JPA generates the SQL query for you, based on the method name and the entity mappings. So when you call `gradeRepository.findByStudentId(id)`, it executes the SQL query to retrieve all grades for the specified student and returns the result as a `List<Grade>`."

3. Adding a One-to-Many relationship

Open the `Student` class and add the following code:

Student.java

```
...

//Each student can have many grades.
@OneToMany(mappedBy = "student", cascade = CascadeType.REMOVE,
orphanRemoval = true)
private List<Grade> grades = new ArrayList<>();

...


public List<Grade> getGrades() {
    return this.grades;
}

public void setGrades(List<Grade> grades) {
    this.grades = grades;
}
```

In the code above, we have defined a **One-to-Many** relationship with the Grade entity using the **@OneToMany** annotation.

This relationship specifies that: **"Each student can have many grades."**

- The **`mappedBy`** attribute indicates that the relationship is mapped by the `student` field in the `Grade` class.
- The **`cascade`** attribute specifies that all `Grade` entities associated with a `Student` should be removed when the `Student` is removed.
- The **`orphanRemoval`** attribute specifies that any `Grade` entities that are no longer associated with a `Student` should be removed.

4. Update GradeService.java

```
package lu.cnfpcjavadev.grade_submission.service;

import java.util.List;

import org.springframework.stereotype.Service;

import jakarta.transaction.Transactional;
import
lu.cnfpcjavadev.grade_submission.exception.GradeNotFoundException;
import lu.cnfpcjavadev.grade_submission.model.Grade;
import lu.cnfpcjavadev.grade_submission.model.Student;
import lu.cnfpcjavadev.grade_submission.repository.GradeRepository;

@Service
@Transactional
public class GradeService {
    private final GradeRepository gradeRepository;
    private final StudentService studentService;

    public GradeService(GradeRepository gradeRepository, StudentService
studentService) {
        this.gradeRepository = gradeRepository;
        this.studentService = studentService;
    }

    public Grade getGradeById(Long id) {
        return gradeRepository.findById(id)
                .orElseThrow(() -> new GradeNotFoundException("Grade
not found with id: " + id));
    }

    public List<Grade> getAllGrades() {
        return gradeRepository.findAll();
    }

    public List<Grade> getGradesByStudentId(Long studentId) {
        // First verify student exists
        studentService.getStudentById(studentId);
        return gradeRepository.findByStudentId(studentId);
    }

    public Grade saveGrade(Grade grade) {
        // Verify student exists before saving grade
        Student student =
studentService.getStudentById(grade.getStudent().getId());
        grade.setStudent(student);
        return gradeRepository.save(grade);
    }
    public Grade updateGrade(Long id, Grade gradeDetails) {
        Grade grade = getGradeById(id);
        grade.setSubject(gradeDetails.getSubject());
        grade.setScore(gradeDetails.getScore());

        // If student is being changed, verify new student exists
        if
(!grade.getStudent().getId().equals(gradeDetails.getStudent().getId()))
{
            Student newStudent =
studentService.getStudentById(gradeDetails.getStudent().getId());
            grade.setStudent(newStudent);
        }

        return gradeRepository.save(grade);
    }

    public void deleteGrade(Long id) {
        Grade grade = getGradeById(id);
        gradeRepository.delete(grade);
    }
}
```

5. Update GradeController.java

```
package lu.cnfpcjavadev.grade_submission.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

import jakarta.validation.Valid;
import lu.cnfpcjavadev.grade_submission.model.Grade;
import lu.cnfpcjavadev.grade_submission.model.Student;
import lu.cnfpcjavadev.grade_submission.service.GradeService;
import lu.cnfpcjavadev.grade_submission.service.StudentService;
import lu.cnfpcjavadev.grade_submission.service.SubjectService;

@Controller
public class GradeController {

    private GradeService gradeService;
    private StudentService studentService;
    private SubjectService subjectService;

    public GradeController(GradeService gradeService, StudentService
studentService, SubjectService subjectService) {
        this.gradeService = gradeService;
        this.studentService = studentService;
        this.subjectService = subjectService;
    }

    @GetMapping("/")
    public String getForm(Model model, @RequestParam(required = false)
Long id) {

        // Create new grade or get existing one
        Grade grade = (id != null)
                ? gradeService.getGradeById(id)
                : new Grade();

        // pass the new or existing student grade object
        model.addAttribute("grade", grade);
        // pass all students to the view
        model.addAttribute("students",
studentService.getAllStudents());
        // pass the fetched list of subjects (from external API)
        model.addAttribute("subjects", subjectService.getSubjects());
        return "form";
    }
    @GetMapping("/grades")
    public String getGrades(Model model) {
        // add data to the model
        model.addAttribute("grades", gradeService.getAllGrades());
        return "grades";
    }

    @PostMapping("/handleSubmit")
    public String submitGrade(@Valid Grade grade, BindingResult 
bindingResult, Model model) {
        if (bindingResult.hasErrors()) {
            // Re-add the list of students and subjects to the model if 
there are errors
            model.addAttribute("students", 
studentService.getAllStudents()); // Fetch list of students
            model.addAttribute("subjects", 
subjectService.getSubjects()); // Fetch list of subjects
            return "form";
        }

        // assigning the student object reference
        Student student = grade.getStudent();
        grade.setStudent(student);

        // calling gradeService to submit the new or updated grade.
        gradeService.saveGrade(grade);

        return "redirect:/grades";
    }

    @GetMapping("/delete-grade")
    public String deleteGrade(@RequestParam Long id) {
        gradeService.deleteGrade(id);
        return "redirect:/grades";
    }

}
```

6. Custom Exception

```
package lu.cnfpcjavadev.grade_submission.exception;

public class GradeNotFoundException extends RuntimeException {
    public GradeNotFoundException(String message) {
        super(message);
    }
}
```

7. Update the View form.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Form</title>
    <link rel="stylesheet" th:href="@{/form.css}">
  </head>
  <body>
    <div class="topnav">
      <a th:href="@{/}">Add Grade</a>
      <a th:href="@{/grades}">Grades</a>
      <a th:href="@{/addstudent}">Add Student</a>
      <a th:href="@{/students}">Students</a>
      <a th:href="@{/courses}">Course Details</a> <!-- New link to
course details -->
    </div>
    <div class="container">
      <h2> Grade Portal </h2>
      <form method="post" th:object="${grade}"
th:action="@{/handleSubmit}">

        <select name="student" id="student" th:field="*{student}">
          <option value="" hidden>Select a student</option>
          <option th:each="student : ${students}"
th:value="${student.id}"
          th:text="${student.firstName}+' '+${student.LastName}">
</option>
        </select>
        <p style="color:red;" th:errors="*{student}"></p>

        <input type="text" placeholder="Score" th:field="*{score}">
        <p style="color:red;" th:errors="*{score}"></p>

        <select name="subject" id="subject" th:field="*{subject}">
          <option value="" hidden>Select a subject</option>
          <option th:each="subject : ${subjects}" th:value="${subject}"
          th:text="${subject}"></option>
        </select>

        <p style="color:red;" th:errors="*{subject}"></p>
        <input type="hidden" th:field="*{id}">
        <br><br>
        <input type="submit" value="Submit">
      </form>
    </div>
</body>
</html>
```

4.8 Exercise: GradeController & Display All Grades

Exercise: GradeController & Display All Grades

In this exercise, you will create a new controller called `GradeController` that displays all grades for all students in the system. You will also reuse the existing `grades.html` Thymeleaf template.

Follow the steps below to complete this exercise:

1. Create a new `GradeController` class in the `lu.cnfpc.gradesubmission` package. This controller should be annotated with `@Controller` and should have a request mapping for `/grades`.
2. Autowire the `GradeRepository` and `StudentRepository` beans in the `GradeController`.
3. Create a new handler method called `listGrades` that returns a view name of `grades` and adds a model attribute called `grades`. The `grades` model attribute should be populated with all grades in the system. You can get all grades by calling `gradeRepository.findAll()`.
4. Update the `grades.html` Thymeleaf template to display all grades returned by the `listGrades` handler method. You will need to update the Thymeleaf expression that iterates over grades to use the `grades` model attribute.

Solution

1. Create a new GradeController class in the lu.cnfpc.gradesubmission package. This controller should be annotated with @Controller.

```
@Controller
public class GradeController {
   // code for the controller
}
```

2. Autowire the GradeRepository and StudentRepository beans in the GradeController.

```
@Controller
public class GradeController {
    @Autowired
    private GradeRepository gradeRepository;

    @Autowired
    private StudentRepository studentRepository;

   // code for the controller
}
```

3. Create a new handler method called listGrades that returns a view name of grades and adds a model attribute called grades. The grades model attribute should be populated with all grades in the system. You can get all grades by calling gradeRepository.findAll().

```
@GetMapping("/")
public String listGrades(Model model) {
    List<Grade> grades = gradeRepository.findAll();
    model.addAttribute("grades", grades);
    return "grades";
}
```

4. Update the grades.html Thymeleaf template to display all grades returned by the listGrades handler method. You will need to update the Thymeleaf expression that iterates over grades to use the grades model attribute.

```
<!DOCTYPE>
<html>
  <head>
    <title>Grades</title>
    <link rel="stylesheet" th:href="@{grade-stylesheets.css}">
  </head>
  <body>

    <div class="topnav">
      <a th:href="@{/}">Form</a>
      <a th:href="@{/grades}">Grades</a>
    </div>

    <div class= "container">
      <h2> Grades </h2>
      <table id="table">
        <tr>
          <th>Student</th>
          <th>Subject</th>
          <th>Score</th>
        </tr>
        <tr th:each="grade: ${grades}">
          <td th:text="${grade.student.name}"></td>
          <td th:text="${grade.subject}"></td>
          <td th:text="${grade.score}"></td>
          <td> <a class="update" role="button" th:href="@{/(id =
${grade.id})}">Update</a> </td>
        </tr>
      </table>
    </div>

  </body>
</html>
```

This should display a table of all the grades in the system when you navigate to http://localhost:8080/grades ↗.

4.10 Course Entity

1. Creating the Course entity
2. Creating the CourseRepository
3. Adding a One-to-Many relationship between Course and Grades
4. Updating the repositories and controllers
5. Creating the view Template

In this section, we will create a new entity called `Course` and add a one-to-many relationship between `Course` and `Grade`.

1. Creating the Course entity

First, let's create the `Course` entity. This entity will have an `id` field (which will be the primary key), a `name` field, and a `description` field.

Course.java

```java
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;

    // Getter and Setter methods
}
```

2. Creating the CourseRepository

Next, let's create the `CourseRepository` interface, which will extend the `JpaRepository` interface. This interface will provide basic CRUD operations for the `Course` entity.

CourseRepository.java

```java
public interface CourseRepository extends JpaRepository<Course, Long> {
}
```

3. Adding a One-to-Many relationship between Course and Grades

Now, let's add a one-to-many relationship between `Course` and `Grade`. Each course can have multiple grades.

Course.java

```java
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;

    //Each course can have multiple grades
    @OneToMany(mappedBy = "course", cascade = CascadeType.REMOVE)
    private List<Grade> grades = new ArrayList<>();

    // Getter and Setter methods
}
```

- we have added a new `grades` field to the `Course` entity.
- field is annotated with `@OneToMany`, which indicates that each course can have multiple grades. `mappedBy` attribute specifies the name of the field in the `Grade` entity that maps to the `Course` entity.
- `cascade = CascadeType.REMOVE` , which means that if a `Course` entity is deleted, all associated `Grade` entities will be deleted as well.

4. Updating the repositories and controllers

Now that we have added a new entity and a relationship between `Course` and `Grade`, we need to update the repositories and controllers to handle the new entity.

In the `Grade` entity, we have added a new `course` field, which maps to the `Course` entity. We need to update the `GradeRepository` to include a new method that returns all grades for a given course:

GradeRepository.java

```java
public interface GradeRepository extends JpaRepository<Grade, Long> {
    List<Grade> findByCourseId(Long courseId);
}
```

In the `CourseController`, we can add a new handler method that displays all courses:

CourseController.java

```java
//handler method that displays all courses
@GetMapping("/courses")
public String listCourses(Model model) {
    List<Course> courses = courseRepository.findAll();
    model.addAttribute("courses", courses);
    return "courses";
}
```

In the `GradeController`, we can add a new handler method that displays all grades for a given course:

GradeController.java

```java
//handler method that displays all grades for a given course
@GetMapping("/courses/{courseId}/grades")
public String listGradesForCourse(@PathVariable Long courseId, Model
model) {
    Course course = courseRepository.findById(courseId).orElse(null);
    if (course == null) {
        // case where the course doesn't exist
        return "redirect:/courses";
    }

    List<Grade> grades = gradeRepository.findByCourseId(courseId);
    model.addAttribute("course", course);
    model.addAttribute("grades", grades);
    return "grades";
}
```

In this handler method, we first look up the `Course` object with the given ID in the `CourseRepository`. If the course doesn't exist, we redirect the user back to the list of courses. Otherwise, we use the `findByCourseId` method in the `GradeRepository` to get all grades associated with the course. We add the Course and List objects to the model and return the `grades.html` Thymeleaf template to display the grades for the given course.

5. `courses.html` view template

Let's create a Thymeleaf template called `courses.html` in the `src/main/resources/templates` directory to display a list of all courses. Here's an example of what the template might look like:

courses.html

```
...
<table>
<tr>
    <th>ID</th>
    <th>Name</th>
    <th>Description</th>
    <th>Grades</th>
</tr>
<tr th:each="course : ${courses}">
    <td th:text="${course.id}"></td>
    <td th:text="${course.name}"></td>
    <td th:text="${course.description}"></td>
    <td><a th:href="@{/courses/{id}/grades(id=${course.id})}">View
Grades</a></td>
</tr>
</table>

...
```

4.9 Exercise: Add and Update Grade for Existing Student

1. Create a new **handler method in** `GradeController` that handles GET requests to `/grades/new`. This method should return a view name of `/grades/new` and should add model attributes called `students` and `courses` that contains all of the students and courses in the database.
2. Create a new **Thymeleaf template** called `/grades/new.html`. This template should display a form that allows the user to enter a new grade for an existing student.

   The form should have fields for the student's name, the course of the grade, and the score.

   - Use Thymeleaf to populate a **dropdown lists:** *(https://www.w3schools.com/tags/tag_select.asp ↗)* with the names of all the students and names of all the courses (which you added to the model attribute in step 1).
3. Create a new **handler method in** `GradeController` that handles **POST** requests to `/grades/new`. It should look up the student with the given id in the `StudentRepository` and the corresponding course with the given id in the courseRepository. Finally, create a new `Grade` object with the student and the given course and score.

   It should then save the **new** `Grade object` to the `GradeRepository` and redirect the user to the `/grades/list` page.
4. Create a new **handler method in** `GradeController` that handles **GET** requests to `/grades/{id}/edit`, where `{id}` is the ID of the grade to be edited. This method should return a view name of `/grades/edit` `.`
5. Create a new **Thymeleaf template called** `/grades/edit.html`. This template should display a form that allows the user to edit an existing grade.
   - The form should have fields for the student's name, the course's name of the grade, and the score.
   - You should use Thymeleaf to populate a **dropdown lists** with the names of all the students and courses. The form should be pre-populated with the existing values for the grade.

6. Create a new **handler method in `GradeController`** that handles **POST requests** to **`/grades/{id}/edit`**, where **`{id}`** is the ID of the grade to be edited.
   - It should look up the grade, the student and the course and update the `Grade` object with the given ID to have the new student, subject, and score.
   - It should then save the updated `Grade` object to the `GradeRepository` and redirect the user to the `/grades/list` page.

Congrats! You should now have a fully-functional system for entering and updating grades for existing students.

Solution Exercise 4.9

1. Create a **new handler method** in **GradeController** that handles **GET requests** to `/grades/new`. This method should return a view name of **grades/new** and should add the model attributes called students and courses that contains all of the students and all the courses in the database.

GradeController.java

```java
...


// Handler method handling GET requests to get the add-grade FORM
@GetMapping("/grades/new")
public String showAddGradeForm(Model model) {
    List<Student> students = studentRepository.findAll();
    List<Course> courses = courseRepository.findAll();
    model.addAttribute("students", students);
    model.addAttribute("courses", courses);
    return "grades/new";
}

...
```

2. Create a new **Thymeleaf template** called **new.html**. This template should display a form that allows the user to enter a new grade for an existing student and existing course. You should use Thymeleaf to populate two dropdowns lists with the names of all the students and the names of all the courses (which you added to the model attribute in step 1).

grades/new.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form</title>
    <link rel="stylesheet" th:href="@{/form-stylesheets.css}" />
  </head>
  <body>
    <div class="topnav">
      <a th:href="@{/}">Home</a>
      <a th:href="@{/students/list}">Students</a>
      <a th:href="@{/students/new}">New Student</a>
      <a th:href="@{/grades/list}">Grades</a>
      <a th:href="@{/grades/new}">New Grade</a>
      <a th:href="@{/courses/list}">Courses</a>
      <a th:href="@{/courses/new}">New Course</a>
    </div>
    <div class="container">
      <h2>Grade Portal</h2>
      <form method="post" th:action="@{/grades/add}">
        <select id="studentId" name="studentId">
          <option value="" selected disabled>Select a student</option>
          <option
            th:each="student : ${students}"
            th:value="${student.id}"
            th:text="${student.name}"
          ></option>
        </select>
        <select id="courseId" name="courseId">
          <option value="" selected disabled>Select a course</option>
          <option
            th:each="course : ${courses}"
            th:value="${course.id}"
            th:text="${course.name}"
          ></option>
        </select>
        <input
          type="number"
          placeholder="Enter a score"
          id="score"
          name="score"
          step="0.1"
        />
        <br /><br />
        <br /><br />
        <input type="submit" value="Submit" />
      </form>
    </div>
  </body>
```

```
</html>
```

the `select` element uses the `student.id` as the value and `student.name` as the displayed text. By doing so, when the form is submitted, the `studentId` parameter in the `addGrade` handler method of `GradeController` will receive the selected student ID instead of the name.

You can then use this ID to look up the student in the `StudentRepository`.

3. Create a new **handler method** in GradeController that handles **POST requests** to **`/grades/new`**. This method should take the parameters **studentId, courseId, and score**. It should look up the student with the given id in the StudentRepository and the course with the given id in the courseRepository. Finally, it should create a new Grade object with the student and the given course and score. It should then save the new Grade object to the GradeRepository and redirect the user to the /grades page.

GradeController.java

```
...

//Handler method handling POST requests in order to add new grades
@PostMapping("/grades/new")
public String addGrade(@RequestParam Long studentId, @RequestParam Long
courseId, @RequestParam double score) {
    // find corresponding student
    Student student =
studentRepository.findById(studentId).orElse(null);
    if (student == null) {
        // case where the student doesn't exist redirect to grades
        return "redirect:/grades/list/";
    }

    // find correspondant course
    Course course = courseRepository.findById(courseId).orElse(null);
    if (course == null) {
        // case where the course doesn't exist redirect to grades
        return "redirect:/grades/list";
    }

    // prepare the new grade object
    Grade grade = new Grade();
    grade.setStudent(student);
    grade.setCourse(course);
    grade.setScore(score);
    // save it to the database
    gradeRepository.save(grade);

    return "redirect:/grades/list";
}

...
```

4. Create a **new handler method** in **`GradeController`** that handles **GET requests** to **`/grades/{id}/edit`**, where **`{id}`** is the ID of the grade to be edited. This method should return a view name of **`grades/edit`** and should add a model attribute called `grade` that contains the `Grade` object with the given ID.

GradeController.java

```java
...

// Handler method to get the form to edit a grade
@GetMapping("/grades/{id}/edit")
public String editGrade(@PathVariable("id") Long id, Model model) {
    Grade grade = gradeRepository.findById(id).orElse(null);
    if (grade == null) {
        return "redirect:/grades/list";
    }

    List<Student> students = studentRepository.findAll();
    List<Course> courses = courseRepository.findAll();
    model.addAttribute("grade", grade);
    model.addAttribute("students", students);
    model.addAttribute("couses", courses);

    return "grades/edit";
}

...
```

In this method, we first get the `Grade` object with the given ID from the `GradeRepository`. If the `Grade` doesn't exist, we redirect the user to the `/grades/list` page. If it does exist, we add the `Grade` object and a list of all students to the model, and return the `grades/edit` view.

5. Create a new **Thymeleaf template** called **`edit.html`**. This template should display a form that allows the user to edit an existing grade. The form should have fields for the student's name, the course of the grade, and the score. You should use Thymeleaf to populate a dropdowns lists. The form should be pre-populated with the existing values for the grade.

grades/edit.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form</title>
    <link rel="stylesheet" th:href="@{/form-stylesheets.css}" />
  </head>
  <body>
    <div class="topnav">
      <a th:href="@{/}">Home</a>
      <a th:href="@{/students/list}">Students</a>
      <a th:href="@{/students/new}">New Student</a>
      <a th:href="@{/grades/list}">Grades</a>
      <a th:href="@{/grades/new}">New Grade</a>
      <a th:href="@{/courses/list}">Courses</a>
      <a th:href="@{/courses/new}">New Course</a>
    </div>
    <div class="container">
      <h2>Grade Portal</h2>
      <form method="post"
th:action="@{/grades/{id}/edit(id=${grade.id})}">
        <select id="studentId" name="studentId">
          <option th:each="student : ${students}"
            th:value="${student.id}"
            th:text="${student.name}"
            th:selected="${student.name == grade.student.name}">
          </option>
        </select>
        <select id="courseId" name="courseId">
          <option th:each="course : ${courses}"
            th:value="${course.id}"
            th:text="${course.name}"
            th:selected="${course.name == grade.course.name}"></option>
          ></option>
        </select>
        <input
          type="number"
          placeholder="Enter a score"
          id="score"
          name="score"
          step="0.1"
          th:value="${grade.score}"
        />
        <br /><br />
        <br /><br />
        <input type="submit" value="Save" />
      </form>
    </div>
  </body>
```

```
</html>
```

In this template, we display a form that allows the user to edit an existing grade.

6. Create a **new handler method in GradeController** that handles **POST requests** to **`/grades/{id}/edit`**, where **`{id}`** is the ID of the grade to be edited. It should look up the student with the given id in the `StudentRepository` and update the `Grade` object with the given ID to have the new student, course, and score. It should then save the updated `Grade` object to the `GradeRepository` and redirect the user to the **`/grades/list`** page.

```
// Handler method handling POST requests in order to edit grades
@PostMapping("/grades/{id}/edit")
public String editGrade(@PathVariable("id") Long id, @RequestParam Long
studentId, @RequestParam Long courseId, @RequestParam double score) {
    // Find the grade to be edited
    Grade grade = gradeRepository.findById(id).orElse(null);

    // Find the student by id
    Student student =
studentRepository.findById(studentId).orElse(null);

    // Find the student by id
    Course course = courseRepository.findById(courseId).orElse(null);

    // Update the grade with the new data
    grade.setStudent(student);
    grade.setCourse(course);
    grade.setScore(score);

    // Save the updated grade
    gradeRepository.save(grade);

    // Redirect to the grades page
    return "redirect:/grades/list";
}
```

In this method, we first find the grade to be edited by its ID. Then we find the student by id in the `StudentRepository`. We update the grade object with the new student, course, and score. Finally, we save the updated grade to the GradeRepository and redirect the user to the /grades/list page.

4.11 Exercise: Add and Update Courses

In this exercise, we will add the ability to add and update courses:

1. Create a **new handler method in** `CourseController` that handles `GET` requests to `/courses/new`. This method should return a view name of `/courses/new` .
2. Create a new **Thymeleaf template called** `/courses/new.html`**.** This template should display a form that allows the user to enter a new course. The form should have fields for the course name and description.
3. Create a **new handler method in** `CourseController` that handles `POST` requests to `/courses/new`. It should create a new Course object with the given name and description and save it to the `CourseRepository`. It should then redirect the user to the `/courses/list` page.
4. Create a new handler method in `CourseController` that handles `GET` requests to `/courses/{id}/edit`, where `{id}` is the ID of the course to be edited. This method should return a view name of `/courses/edit` and should add the needed model attributes.
5. Create a new **Thymeleaf template called** `/courses/edit.html`. This template should display a form that allows the user to edit an existing course. The form should have fields for the course name and description. The form should be pre-populated with the existing values for the course.
6. Create a **new handler method in** `CourseController` that handles `POST` requests to `/courses/{id}/edit`, where `{id}` is the ID of the course to be edited. It should look up the Course object and update it to have the new name and description. It should then save the updated Course object to the `CourseRepository` and redirect the user to the `/courses/list` page.

Congratulations! You should now have a fully-functional application for adding and updating courses.

Solution Exercice 4.11

1. Create a new handler method in `CourseController` that handles `GET requests` to `/courses/new`. This method should return a view name of `/courses/new`.

CourseController.java

```java
...

@GetMapping("/courses/new")
public String addCourse(Model model) {
    model.addAttribute("course", new Course());
    return "/courses/new";
}

...
```

2. Create a new **Thymeleaf template** called `new.html`. This template should display a form that allows the user to enter a new course. The form should have fields for the course's name and description.

courses/new.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form</title>
    <link rel="stylesheet" th:href="@{/form-stylesheets.css}" />
  </head>
  <body>
    <div class="topnav">
      <a th:href="@{/}">Home</a>
      <a th:href="@{/students/list}">Students</a>
      <a th:href="@{/students/new}">New Student</a>
      <a th:href="@{/grades/list}">Grades</a>
      <a th:href="@{/grades/new}">New Grade</a>
      <a th:href="@{/courses/list}">Courses</a>
      <a th:href="@{/courses/new}">New Course</a>
    </div>
    <div class="container">
      <h2>Grade Portal</h2>
      <form method="post" th:action="@{/courses/new}"
th:object="${course}">
        <input
          type="text"
          id="name"
          placeholder="Course name"
          name="name"
          th:field="*{name}"
        /><br />
        <textarea
          id="description"
          placeholder="Course description"
          name="description"
          th:field="*{description}"
        ></textarea>
        <br />
        <input type="submit" value="Add Course" />
      </form>
    </div>
  </body>
</html>
```

3. Create a **new handler method** in `CourseController` that handles **POST requests** to `/courses/new`. It should create a new Course object with the given name and description, and then save it to the `CourseRepository`. It should then redirect the user to the `/courses/list` page.

CourseController.java

```
...

//handler method handling POST request with new course to add
@PostMapping("/courses/new")
public String saveCourse(Course course) {
    courseRepository.save(course);
    return "redirect:/courses/list";
}

...
```

4. Create a **new handler method** in `CourseController` that handles `GET requests` to /`courses/{id}/edit`, where `{id}` is the ID of the course to be edited. This method should return a view name of `/courses/edit` and should add a model attribute called course that contains the Course object with the given ID.

CourseController.java

```
//handler method handling GET requests to get the edit form page
@GetMapping("/courses/{id}/edit")
public String editCourse(@PathVariable Long id, Model model) {
    Course course = courseRepository.findById(id).orElse(null);
    model.addAttribute("course", course);
    return "/courses/edit";
}
```

5. Create a **new Thymeleaf template** called `/courses/edit.html`. This template should display a form that allows the user to edit an existing course. The form should have fields for the course's name and description. The form should be pre-populated with the existing values for the course.

edit-course.html

```
...

<h2>Edit Course</h2>
<form action="#" th:action="@{/courses/{id}}" th:object="${course}"
method="post">
    <input type="hidden" id="id" name="id" th:field="*{id}">

    <label for="name">Name:</label>
    <input type="text" id="name" name="name" th:field="*{name}"><br>

    <label for="description">Description:</label>
    <textarea id="description" name="description" th:field="*
{description}"></textarea><br>

    <input type="submit" value="Update Course">
</form>
...
```

6. Create a **new handler method** in `CourseController` that handles **`POST requests`** to /courses/{id}/edit, where {id} is the ID of the course to be edited. This method should take the parameters name and description. It should look up the Course object with the given ID in the CourseRepository, update the name and description, and save the updated Course object to the CourseRepository. Finally, it should redirect the user to the /courses/list page.

CourseController.java

```java
...

@PostMapping("/courses/{id}/edit")
public String updateCourse(@PathVariable Long id, @RequestParam String
name, @RequestParam String description) {
    Course course = courseRepository.findById(id).orElse(null);
    if (course == null) {
        // case where the course doesn't exist
        return "redirect:/courses/list";
    }
    course.setName(name);
    course.setDescription(description);
    courseRepository.save(course);
    return "redirect:/courses/list";
}

...
```

4.12 Many-to-Many Relationship

- Adding a Many-to-Many relationship between Students and Courses
- Updating the repositories and controllers
- Creating views for managing Students and Courses

In this section, we will add a many-to-many relationship between the `Student` and `Course` entities. **Each student can enroll in multiple courses, and each course can have multiple students.**

1. Adding a Many-to-Many relationship between Students and Courses

To create a many-to-many relationship between two entities, we need to use a join table that maps the relationships between the two entities. In our case, we need to create a join table that maps the relationship between `Student` and `Course`.

We can create the join table using the `@JoinTable` annotation in the `Student` entity:

Student.java

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    @ManyToMany
    @JoinTable(name = "enrollments",
            joinColumns = @JoinColumn(name = "student_id"),
            inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses = new ArrayList<>();

    // Getter and Setter methods
}
```

In the `@JoinTable` annotation, we specify the name of the join table (`enrollments`) and the names of the columns that map to the `Student` and `Course` entities.

We also need to update the `Course` entity to map the many-to-many relationship:

Course.java

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;

    //mappedBy attribute indicates that the Student entity owns the
relationship
    @ManyToMany( mappedBy="courses", cascade = {CascadeType.PERSIST,
CascadeType.MERGE})
    private List<Student> students = new ArrayList<>();

    // Getter and Setter methods
}
```

In the `@ManyToMany` annotation, we specify the `mappedBy` attribute to indicate that the `Student` entity owns the relationship.

2. Updating the repositories and controllers

Now that we have added a many-to-many relationship between `Student` and `Course`, we need to update the repositories and controllers to handle the new relationship.

- In the `CourseRepository`: `findCoursesByStudentsId` to retrieve all courses associated with multiple students based on the provided `StudentsId`.

CourseRepository.java

```java
public interface CourseRepository extends JpaRepository<Course, Long> {
    // retrieve all courses associated with students based on the
provided StudentsId.
    List<Course> findCoursesByStudentsId(Long studentId);
}
```

- In the `StudentRepository`: `findStudentsByCoursesId` to retrieve all students associated with multiple courses based on the provided `CoursesId`.

StudentRepository

```java
public interface StudentRepository extends JpaRepository<Student, Long>
{
    // retrieve all students associated to courses based on the
provided CoursesId.
    List<Student> findStudentsByCoursesId(Long courseId);
}
```

We can now update the `StudentController` and `CourseController` to handle the new relationship.

In the **`StudentController`**, we can add a **new handler method that displays all courses for a given student:**

StudentController.java

```
...

//handler method that displays all courses for a given student
@GetMapping("/students/{studentId}/courses")
public String listCoursesForStudent(@PathVariable Long studentId, Model
model) {
    Student student =
studentRepository.findById(studentId).orElse(null);
    if (student == null) {
        // case where the student doesn't exist
        return "redirect:/students";
    }

    List<Course> courses =
courseRepository.findCoursesByStudentsId(studentId);
    model.addAttribute("student", student);
    model.addAttribute("courses", courses);
    return "courses/list";
}

...
```

In the `CourseController`, we can add a **new handler method that displays all students for a given course:**

CourseController.java

```
...

//handler method handling POST request with student to enroll in a
course
@PostMapping("/courses/{courseId}/enroll")
@Transactional
public String enrollStudentInCourse(@PathVariable Long courseId,
@RequestParam Long studentId) {
    // Retrieve the course and student objects from their respective
repositories
    Course course = courseRepository.findById(courseId).orElse(null);
    Student student =
studentRepository.findById(studentId).orElse(null);

    if (course != null && student != null) {
        // Add the course to the student's courses list
        student.getCourses().add(course);

        // Add the student to the course's students list
        course.getStudents().add(student);

        // Save the course and student
        studentRepository.save(student);
        courseRepository.save(course);
    }

    return "redirect:/courses/list";
}

...
```

In this handler method, we first look up the `Course` object with the given ID in the `StudentRepository`. If the `course` doesn't exist, we redirect the user back to the list of courses. Otherwise, we use the `findStudentsByCoursesId` method in the `StudentRepository` to get all students associated with the course. We add the `Course` and `List` objects to the model and return the `students/list.html` Thymeleaf template to display the students for the given course.

3. Views for managing Students and Courses

We can now use the Thymeleaf templates to manage students and courses. We can update two existing templates, `students/list.html` and `courses/list.html`, to display the proper title for courses for a given student and the students for a given course.

In the `students/list.html` template, **we can display a table that shows the students for the current course**:

students/list.html

```
...

<h2 th:text="${course} ? ${course.name}+' Students List' : 'Students
List'">

...
```

In the `courses/list.html` template, **we can display a table that shows the courses for the current student:**

courses/list.html

```
...

<h2 th:text="${student} ? ${student.name}+' Courses List' : 'Courses
List'">

...
```

With these templates in place, we can now navigate to `/students/{studentId}/courses` and `/courses/{courseId}/students` to view the courses for a given student and the students for a given course, respectively.

In the `courses/edit.html` template, **we can add a form that enrolls a student to the current course:**

courses/edit.html

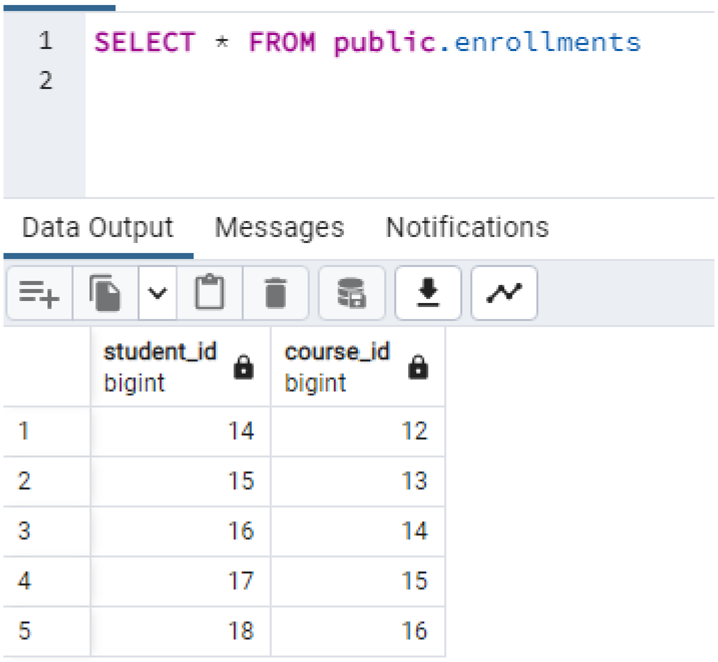
```
...

<h2>Enroll student</h2>

<form method="post"
th:action="@{/courses/{courseId}/enroll(courseId=${course.id})}">
  <!-- Other input fields for course details -->
  <select name="studentId">
      <option th:each="student : ${students}" th:value="${student.id}"
th:text="${student.name}"></option>
  </select>
  <input type="submit" value="Enroll" />
</form>

...
```

It might be necessary to add students to the model in the `CourseController > editCourse`:

CourseController.java

```
model.addAttribute("students", students);
```

If you run some a test, the student should be successfully enrolled in the course.



```
SELECT * FROM public.enrollments
```

	student_id bigint	course_id bigint
1	14	12
2	15	13
3	16	14
4	17	15
5	18	16

JoinTable "enrollments"