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

```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

```
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

```
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

```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

```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

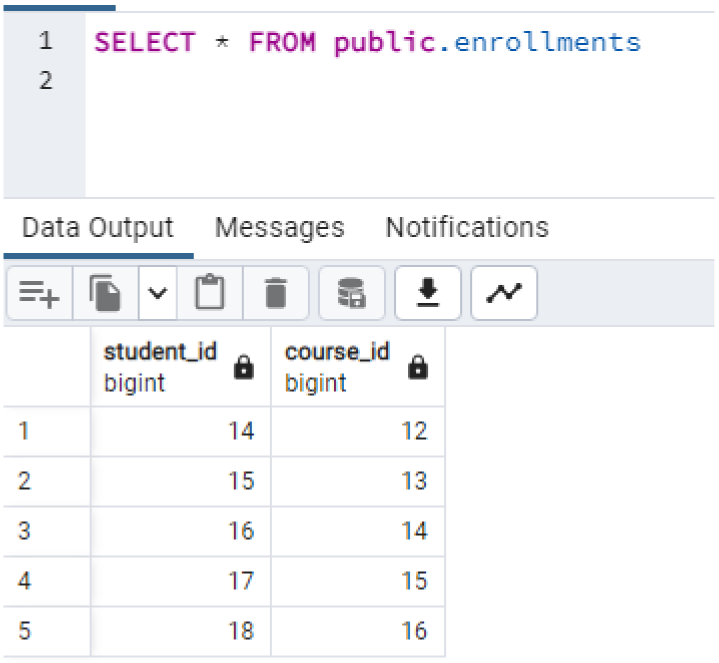
```html
...

<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

```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"