Code-Along: JDBC Template and Thymeleaf

Site: Engage
Course: TSG Online Java Badge 4
Book: Code-Along: JDBC Template and Thymeleaf

Printed by: Jennifer Trongard
Date: Saturday, 20 May 2023, 6:53 PM

Description

Table of contents

Step 1: Overview and Project Setup

Overview

We are now at the point that we can build a full web application from top to bottom using JDBC Template to talk to the database and Thymeleaf to display our data. In this code along, we are going to build an advanced Class Roster application. We'll be able to track classes, the students in the classes, and the instructor teaching the classes. We'll have pages for each and be able to perform all CRUD operations on each. With CRUD functionality, we'll see how we can represent different relationships in the web pages as well, taking advantage of drop-down selects and multi-select options. We'll use primarily Thymeleaf but also Bootstrap to make our front-end look a little nicer.

We start by setting up our project.

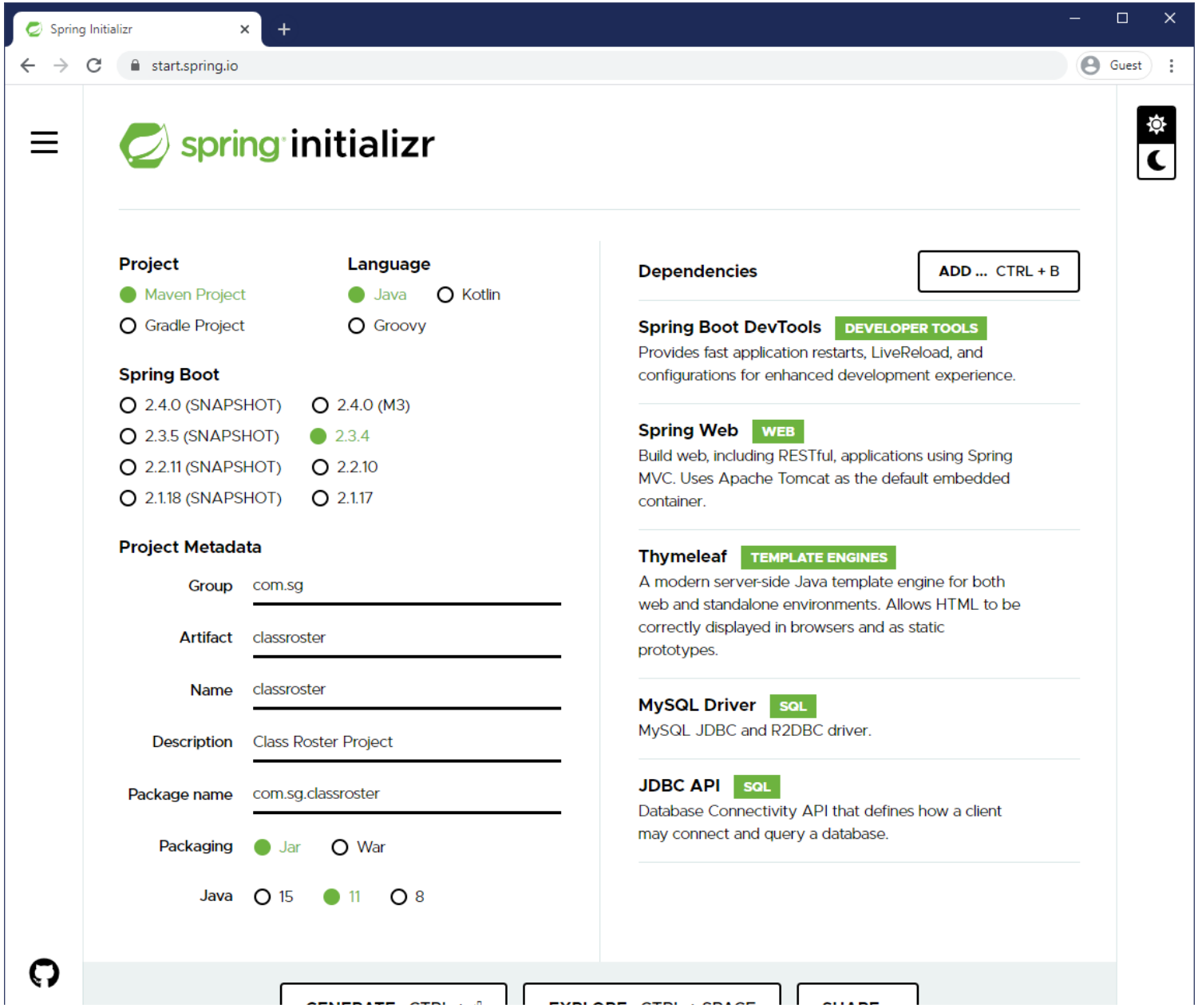
Creating the Project

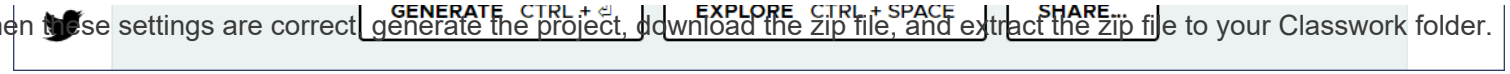
Start at Spring Intializr and create a new project with the following settings:

- Project: Maven Project
- Language: Java
- Spring Boot: The most recent version
- Group: com.sg
- Artifact: classroster
- Name: classroster
- Description: Class Roster Project
- Package Name: com.sg.classroster
- Packaging: Jar
- Java: 11
- Dependencies:

- Spring Boot DevTools
- Spring Web
- Thymeleaf
- MySQL Driver
- JDBC API

The Spring Intializr page should look like:


Spring Initializr
start.spring.io
Guest
spring initializr
Project
Maven Project
Gradle Project
Language
Java
Kotlin
Groovy
Spring Boot
2.4.0 (SNAPSHOT)
2.4.0 (M3)
2.3.5 (SNAPSHOT)
2.3.4
2.2.11 (SNAPSHOT)
2.2.10
2.1.18 (SNAPSHOT)
2.1.17
Project Metadata
Group
com.sg
Artifact
classroster
Name
classroster
Description
Class Roster Project
Package name
com.sg.classroster
Packaging
Jar
War
Java
15
11
8
Dependencies
ADD ... CTRL + B
Spring Boot DevTools
DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
Spring Web
WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Thymeleaf
TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
MySQL Driver
SQL
MySQL JDBC and R2DBC driver.
JDBC API
SQL
Database Connectivity API that defines how a client may connect and query a database.



When these settings are correct, generate the project, download the zip file, and extract the zip file to your Classwork folder.

Open the extracted project in your IDE.

Step 2: Set Up The Database

Next, we'll get our database set up and create our application.properties for both the production and test databases.

Here is the SQL creation script for the production database. Connect to MySQL Server and run this script to set up the database.

```
DROP DATABASE IF EXISTS classRoster;
CREATE DATABASE classRoster;

USE classRoster;

CREATE TABLE teacher(
    id INT PRIMARY KEY AUTO_INCREMENT,
    firstName VARCHAR(30) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    specialty VARCHAR(50)
);

CREATE TABLE student(
    id INT PRIMARY KEY AUTO_INCREMENT,
    firstName VARCHAR(30) NOT NULL,
    lastName VARCHAR(50) NOT NULL
);

CREATE TABLE course(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    description VARCHAR(255),
    teacherId INT NOT NULL,
    FOREIGN KEY (teacherId) REFERENCES teacher(id)
);

CREATE TABLE course_student(
    courseId INT NOT NULL,
    studentId INT NOT NULL,
```

```
    PRIMARY KEY(courseId, studentId),
    FOREIGN KEY (courseId) REFERENCES course(id),
    FOREIGN KEY (studentId) REFERENCES student(id)
);
```

To create the test database, modify the script by changing classRoster to classRosterTest and run it again.

Next, we need to go into project and edit the application.properties to add the connection information. Here is the code, but note that you will need to use the password you set up for MySQL when you installed it on your computer.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/classRoster?serverTimezone=America/Chicago&useSSL=f
alse&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=rootroot
```

We also need to set up the test application.properties. Spring Initializr will not automatically create it, so we have to create the src/test/resources directory and then create an application.properties file there with the following contents. Again, note that you must use your own password.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/classRosterTest?serverTimezone=America/Chicago&useS
SL=false&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=rootroot
```

In both cases, if your MySQL password is different you'll have to update it, but the rest should look just like this.

With the application.properties in place, you should be able to successfully build the project. We're ready to start writing code now.

Step 3: Objects and DAO Interfaces

We should start by creating our entities: the Teacher, Student, and Course objects.

First, we want to create the entities package for the classes to go in. Then we can create the Teacher with these fields:

```
    private int id;
    private String firstName;
    private String lastName;
    private String specialty;
```

Use the "Insert Code…" option to add in getters and setters as well as the equals() and hashCode() methods.

Next, create the Student class:

```
    private int id;
    private String firstName;
    private String lastName;
```

Same as before, use "Insert Code…" to add in getters and setters and the equals() and hashCode() methods.

Finally, we'll add the Course object, which references the Teacher and Student classes:

```
    private int id;
    private String name;
    private String description;
    private Teacher teacher;
    private List<Student> students;
```

Once again, use "Insert Code…" to add in getters and setters and the equals() and hashCode() methods.

The focus of the program is the Courses, so we are letting that class handle the relationships with the other classes.

Now, we'll create the dao package and start creating our DAO interfaces. Let's start with Teacher on this side, too. Here is the interface:

```
public interface TeacherDao {
    Teacher getTeacherById(int id);
    List<Teacher> getAllTeachers();
    Teacher addTeacher(Teacher teacher);
    void updateTeacher(Teacher teacher);
    void deleteTeacherById(int id);
}
```

We only need the five basic CRUD methods for Teacher and nothing else.

Next, we'll create the Student DAO interface:

```
public interface StudentDao {
    Student getStudentById(int id);
    List<Student> getAllStudents();
    Student addStudent(Student student);
    void updateStudent(Student student);
    void deleteStudentById(int id);
}
```

Student also only needs the five basic CRUD methods.

Last, let's make the Course DAO interface, which we'll add a few interface methods to:

```java
public interface CourseDao {
    Course getCourseById(int id);
    List<Course> getAllCourses();
    Course addCourse(Course course);
    void updateCourse(Course course);
    void deleteCourseById(int id);

    List<Course> getCoursesForTeacher(Teacher teacher);
    List<Course> getCoursesForStudent(Student student);
}
```

We start with our five CRUD methods and then add a couple of methods that will be useful for displaying information later: one gets Courses for a Teacher and one gets Courses for a Student.

With the objects and interfaces finished, we can move on to fully implementing the DAOs.

Step 4: Teacher DAO Implementation

We'll start with the Teacher DAO. Create a TeacherDaoDB class that implements the TeacherDao interface, implement the abstract class, annotate it as a @Repository, and autowire in the JdbcTemplate.

```java
@Repository
public class TeacherDaoDB implements TeacherDao{

    @Autowired
    JdbcTemplate jdbc;

    @Override
    public Teacher getTeacherById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public List<Teacher> getAllTeachers() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public Teacher addTeacher(Teacher teacher) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public void updateTeacher(Teacher teacher) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }
```

```
    @Override
    public void deleteTeacherById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }
}
```

We'll also create our TeacherMapper at the end of the file:

```
    public static final class TeacherMapper implements RowMapper<Teacher> {

        @Override
        public Teacher mapRow(ResultSet rs, int index) throws SQLException {
            Teacher teacher = new Teacher();
            teacher.setId(rs.getInt("id"));
            teacher.setFirstName(rs.getString("firstName"));
            teacher.setLastName(rs.getString("lastName"));
            teacher.setSpecialty(rs.getString("specialty"));

            return teacher;
        }
    }
```

With setup complete, we'll implement several methods, starting with getTeacherById:

```
    @Override
    public Teacher getTeacherById(int id) {
        try {
            final String GET_TEACHER_BY_ID = "SELECT * FROM teacher WHERE id = ?";
            return jdbc.queryForObject(GET_TEACHER_BY_ID, new TeacherMapper(), id);
        } catch(DataAccessException ex) {
            return null;
        }
    }
```

- We create our SELECT query string and use it in queryForObject to get the one Teacher we are searching for.
- We surround our code with a try-catch. This will catch the exception thrown if there is no Teacher with that ID, so we can return null in that situation.

Then we have getAllTeachers:

```
    @Override
    public List<Teacher> getAllTeachers() {
        final String GET_ALL_TEACHERS = "SELECT * FROM teacher";
        return jdbc.query(GET_ALL_TEACHERS, new TeacherMapper());
    }
```

- We simply create our SELECT query and use it in the query method to return a list of all Teachers.
- If no Teachers are found, it will return an empty list.

We then move to addTeacher:

```
    @Override
    @Transactional
    public Teacher addTeacher(Teacher teacher) {
        final String INSERT_TEACHER = "INSERT INTO teacher(firstName, lastName, specialty) " +
                "VALUES(?,?,?)";
        jdbc.update(INSERT_TEACHER,
                teacher.getFirstName(),
                teacher.getLastName(),
                teacher.getSpecialty());

        int newId = jdbc.queryForObject("SELECT LAST_INSERT_ID()", Integer.class);
        teacher.setId(newId);
        return teacher;
    }
```

- Our method is @Transactional because we are using the LAST_INSERT_ID query later on.
- We create our INSERT query and use it with the update method and the Teacher data in order.
- We then get the ID for the new Teacher using the LAST_INSERT_ID MySQL function and set it in the Teacher before returning it.

Now we have the updateTeacher method:

```
    @Override
    public void updateTeacher(Teacher teacher) {
        final String UPDATE_TEACHER = "UPDATE teacher SET firstName = ?, lastName = ?, " +
                "specialty = ? WHERE id = ?";
        jdbc.update(UPDATE_TEACHER,
                teacher.getFirstName(),
                teacher.getLastName(),
                teacher.getSpecialty(),
                teacher.getId());
    }
```

- We create our UPDATE query and use it in the update method with the appropriate Teacher data.

Finally, we have the deleteTeacher method:

```
    @Override
    @Transactional
    public void deleteTeacherById(int id) {
        final String DELETE_COURSE_STUDENT = "DELETE cs.* FROM course_student cs "
                + "JOIN course c ON cs.courseId = c.Id WHERE c.teacherId = ?";
        jdbc.update(DELETE_COURSE_STUDENT, id);

        final String DELETE_COURSE = "DELETE FROM course WHERE teacherId = ?";
        jdbc.update(DELETE_COURSE, id);

        final String DELETE_TEACHER = "DELETE FROM teacher WHERE id = ?";
        jdbc.update(DELETE_TEACHER, id);
    }
```

- We make the method @Transactional because we are running multiple queries that modify the database in this method.

- We start by deleting the course_student entries associated with any Course we will be deleting.
- We then delete any Courses associated with the Teacher.
- Finally, we can delete the Teacher itself.
- Order matters here because we can't delete something that is being referenced by another table.

That should finish our Teacher DAO, so we'll now move onto the Student DAO.

Step 5: Student DAO Implementation

The Student DAO will look very similar to the Teacher DAO implementation, with the delete being slightly less complex.

We need to start with our file StudentDaoDB and the basic setup:

```java
@Repository
public class StudentDaoDB implements StudentDao {

    @Autowired
    JdbcTemplate jdbc;

    @Override
    public Student getStudentById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates
    }

    @Override
    public List<Student> getAllStudents() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public Student addStudent(Student student) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public void updateStudent(Student student) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }
```

```
    @Override
    public void deleteStudentById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

}
```

Next, we want to add in our StudentMapper at the end of the file:

```
    public static final class StudentMapper implements RowMapper<Student> {

        @Override
        public Student mapRow(ResultSet rs, int index) throws SQLException {
            Student student = new Student();
            student.setId(rs.getInt("id"));
            student.setFirstName(rs.getString("firstName"));
            student.setLastName(rs.getString("lastName"));

            return student;
        }
    }
```

Now, we can go through the overridden methods, starting with getStudentById:

```
    @Override
    public Student getStudentById(int id) {
        try {
            final String SELECT_STUDENT_BY_ID = "SELECT * FROM student WHERE id = ?";
            return jdbc.queryForObject(SELECT_STUDENT_BY_ID, new StudentMapper(), id);
        } catch (DataAccessException ex) {
            return null;
        }
    }
```

- We create our SELECT query string and use it in queryForObject to get the one Student we are searching for.
- We surround our code with a try-catch that will catch the exception thrown when there is no Student with that ID, so we can return null in that situation.

Next, we go to getAllStudents:

```
    @Override
    public List<Student> getAllStudents() {
        final String SELECT_ALL_STUDENTS = "SELECT * FROM student";
        return jdbc.query(SELECT_ALL_STUDENTS, new StudentMapper());
    }
```

- We simply create our SELECT query and use it in the query method to return a list of all Students.
- If no Students are found, it will return an empty list.

We move to the addStudent method next:

```
    @Override
    @Transactional
    public Student addStudent(Student student) {
        final String INSERT_STUDENT = "INSERT INTO student(firstName, lastName) "
                + "VALUES(?,?)";
        jdbc.update(INSERT_STUDENT,
                student.getFirstName(),
                student.getLastName());

        int newId = jdbc.queryForObject("SELECT LAST_INSERT_ID()", Integer.class);
        student.setId(newId);
        return student;
    }
```

- Our method is @Transactional because we are using the LAST_INSERT_ID query later.
- We create our INSERT query and use it with the update method and the Student data in order.
- We then get the ID for our new Student using the LAST_INSERT_ID MySQL function and set it in the Student before returning it.

We'll look at updateStudent next:

```
    @Override
    public void updateStudent(Student student) {
        final String UPDATE_STUDENT = "UPDATE student SET firstName = ?, lastName = ? "
                + "WHERE id = ?";
        jdbc.update(UPDATE_STUDENT,
                student.getFirstName(),
                student.getLastName(),
                student.getId());
    }
```

- We create the UPDATE query and use it in the update method with the appropriate Student data.

Finally, we'll look at deleteStudentById:

```
    @Override
    @Transactional
    public void deleteStudentById(int id) {
        final String DELETE_COURSE_STUDENT = "DELETE FROM course_student WHERE studentId = ?";
        jdbc.update(DELETE_COURSE_STUDENT, id);

        final String DELETE_STUDENT = "DELETE FROM student WHERE id = ?";
        jdbc.update(DELETE_STUDENT, id);
    }
```

- We'll make the method @Transactional because we are running multiple queries that modify the database in this method.
- We start by deleting the course_student entries associated with the Student.
- We end by deleting the Student itself.
- Order matters here because we can't delete something that is being referenced by another table.

That will finish up the Student DAO. In the next step, we'll look at the Course DAO, which is somewhat more complicated.

Step 6: Course DAO Implementation

The Course DAO is a little more complicated because it handles the relationships in the program and has a few additional public methods. We'll start with our setup in the CourseDaoDB class file you should create:

```java
@Repository
public class CourseDaoDB implements CourseDao {

    @Autowired
    JdbcTemplate jdbc;

    @Override
    public Course getCourseById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public List<Course> getAllCourses() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public Course addCourse(Course course) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public void updateCourse(Course course) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }
```

```
    @Override
    public void deleteCourseById(int id) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public List<Course> getCoursesForTeacher(Teacher teacher) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }

    @Override
    public List<Course> getCoursesForStudent(Student student) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated m
ethods, choose Tools | Templates.
    }
}
```

We also need to add in our CourseMapper:

```
    public static final class CourseMapper implements RowMapper<Course> {

        @Override
        public Course mapRow(ResultSet rs, int index) throws SQLException {
            Course course = new Course();
            course.setId(rs.getInt("id"));
            course.setName(rs.getString("name"));
            course.setDescription(rs.getString("description"));
            return course;
        }
    }
```

We'll make our way through the overridden methods, adding the various private helper methods we need as well.

Here is what we need for getCourseById:

```java
@Override
public Course getCourseById(int id) {
    try {
        final String SELECT_COURSE_BY_ID = "SELECT * FROM course WHERE id = ?";
        Course course = jdbc.queryForObject(SELECT_COURSE_BY_ID, new CourseMapper(), id);
        course.setTeacher(getTeacherForCourse(id));
        course.setStudents(getStudentsForCourse(id));
        return course;
    } catch(DataAccessException ex) {
        return null;
    }
}

private Teacher getTeacherForCourse(int id) {
    final String SELECT_TEACHER_FOR_COURSE = "SELECT t.* FROM teacher t "
            + "JOIN course c ON c.teacherId = t.id WHERE c.id = ?";
    return jdbc.queryForObject(SELECT_TEACHER_FOR_COURSE, new TeacherMapper(), id);
}

private List<Student> getStudentsForCourse(int id) {
    final String SELECT_STUDENTS_FOR_COURSE = "SELECT s.* FROM student s "
            + "JOIN course_student cs ON cs.studentId = s.id WHERE cs.courseId = ?";
    return jdbc.query(SELECT_STUDENTS_FOR_COURSE, new StudentMapper(), id);
}
```

- We wrap the code of the getCourseById method in a try-catch in case the Course does not exist.
- We start with a SELECT query to get the basic Course object.
- We follow up by calling the getTeacherForCourse method, where we JOIN from Teacher to Course to get a Teacher object.

- We then call the getStudentsForCourse method, where we JOIN from Student to course_student to get a list of Student objects for this Course.
- If we catch an exception in getCourseById, we return null because it means the Course does not exist.

Now let's jump to getAllCourses:

```
    @Override
    public List<Course> getAllCourses() {
        final String SELECT_ALL_COURSES = "SELECT * FROM course";
        List<Course> courses = jdbc.query(SELECT_ALL_COURSES, new CourseMapper());
        associateTeacherAndStudents(courses);
        return courses;
    }

    private void associateTeacherAndStudents(List<Course> courses) {
        for (Course course : courses) {
            course.setTeacher(getTeacherForCourse(course.getId()));
            course.setStudents(getStudentsForCourse(course.getId()));
        }
    }
```

- We start by writing a SELECT query and using it to get the list of Courses.
- We pass the list of Courses into associateTeacherAndStudents, where we loop through the list and call our existing Teacher and Students methods to fill in the data for each Course.

We have two other methods that return lists of Courses that we can now fully implement:

```
    @Override
    public List<Course> getCoursesForTeacher(Teacher teacher) {
        final String SELECT_COURSES_FOR_TEACHER = "SELECT * FROM course WHERE teacherId = ?";
        List<Course> courses = jdbc.query(SELECT_COURSES_FOR_TEACHER,
                new CourseMapper(), teacher.getId());
        associateTeacherAndStudents(courses);
        return courses;
    }

    @Override
    public List<Course> getCoursesForStudent(Student student) {
        final String SELECT_COURSES_FOR_STUDENT = "SELECT c.* FROM course c JOIN "
                + "course_student cs ON cs.courseId = c.Id WHERE cs.studentId = ?";
        List<Course> courses = jdbc.query(SELECT_COURSES_FOR_STUDENT,
                new CourseMapper(), student.getId());
        associateTeacherAndStudents(courses);
        return courses;
    }
```

- For getCoursesForTeacher, our query is limited by the teacherId. Once we do the query, we call the associateTeacherAndStudent method again.
- For getCoursesForStudent, we JOIN with course_student so we can limit the query based on the studentId. Once we have the list, we use associateTeacherAndStudent to fill in the rest of the data.

Now we'll move to addCourse:

```
    @Override
    @Transactional
    public Course addCourse(Course course) {
        final String INSERT_COURSE = "INSERT INTO course(name, description, teacherId) "
                + "VALUES(?,?,?)";
        jdbc.update(INSERT_COURSE,
                course.getName(),
                course.getDescription(),
                course.getTeacher().getId());

        int newId = jdbc.queryForObject("SELECT LAST_INSERT_ID()", Integer.class);
        course.setId(newId);
        insertCourseStudent(course);
        return course;
    }

    private void insertCourseStudent(Course course) {
        final String INSERT_COURSE_STUDENT = "INSERT INTO "
                + "course_student(courseId, studentId) VALUES(?,?)";
        for(Student student : course.getStudents()) {
            jdbc.update(INSERT_COURSE_STUDENT,
                    course.getId(),
                    student.getId());
        }
    }
```

- Just like the previous add method, this is @Transactional so we can retrieve the new ID.
- We write our INSERT query and use it to insert the basic Course information into the database.
- We get a new ID and add it to the Course object.

- We can then call our insertCourseStudent helper method, which loops through the list of Students in the Course and adds database entries to course_student for each.
- Once we've done that, we can return the Course from the method.

Next, we have our updateCourse method:

```
    @Override
    @Transactional
    public void updateCourse(Course course) {
        final String UPDATE_COURSE = "UPDATE course SET name = ?, description = ?, "
                + "teacherId = ? WHERE id = ?";
        jdbc.update(UPDATE_COURSE,
                course.getName(),
                course.getDescription(),
                course.getTeacher().getId(),
                course.getId());

        final String DELETE_COURSE_STUDENT = "DELETE FROM course_student WHERE courseId = ?";
        jdbc.update(DELETE_COURSE_STUDENT, course.getId());
        insertCourseStudent(course);
    }
```

- We are using @Transactional here because we are making multiple database-modifying queries in the method.
- We write our UPDATE query and use it in the update method with the appropriate data.
- We need to handle the Students by first deleting all the course_student entries and then adding them back in with the call to insertCourseStudent.

Finally, we can look at deleteCourseById:

```java
    @Override
    @Transactional
    public void deleteCourseById(int id) {
        final String DELETE_COURSE_STUDENT = "DELETE FROM course_student WHERE courseId = ?";
        jdbc.update(DELETE_COURSE_STUDENT, id);

        final String DELETE_COURSE = "DELETE FROM course WHERE id = ?";
        jdbc.update(DELETE_COURSE, id);
    }
```

- This is @Transactional because of the multiple database-modifying queries.
- First, we get rid of the course_student entries that reference our Course.
- Then we can delete the course itself.

That finishes our DAOs. Next, we should create tests for these so we can be confident they are working.

Step 7: DAO Testing

Testing here will look the same as the testing we did in previous lessons. Let's start with the Teacher DAO tests. We first need to set up our test file:

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class TeacherDaoDBTest {

    @Autowired
    TeacherDao teacherDao;

    @Autowired
    StudentDao studentDao;

    @Autowired
    CourseDao courseDao;


    public TeacherDaoDBTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        List<Teacher> teachers = teacherDao.getAllTeachers();
        for(Teacher teacher : teachers) {
```

```
            teacherDao.deleteTeacherById(teacher.getId());
        }

        List<Student> students = studentDao.getAllStudents();
        for(Student student : students) {
            studentDao.deleteStudentById(student.getId());
        }

        List<Course> courses = courseDao.getAllCourses();
        for(Course course : courses) {
            courseDao.deleteCourseById(course.getId());
        }
    }

    @After
    public void tearDown() {
    }
```

Now we can write our tests, making sure to create a more complex set of objects for the delete test:

```java
@Test
public void testAddAndGetTeacher() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test First");
    teacher.setLastName("Test Last");
    teacher.setSpecialty("Test Specialty");
    teacher = teacherDao.addTeacher(teacher);

    Teacher fromDao = teacherDao.getTeacherById(teacher.getId());

    assertEquals(teacher, fromDao);
}

@Test
public void testGetAllTeachers() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test First");
    teacher.setLastName("Test Last");
    teacher.setSpecialty("Test Specialty");
    teacher = teacherDao.addTeacher(teacher);

    Teacher teacher2 = new Teacher();
    teacher2.setFirstName("Test First 2");
    teacher2.setLastName("Test Last 2");
    teacher2.setSpecialty("Test Specialty 2");
    teacher2 = teacherDao.addTeacher(teacher2);

    List<Teacher> teachers = teacherDao.getAllTeachers();
```

```
        assertEquals(2, teachers.size());
        assertTrue(teachers.contains(teacher));
        assertTrue(teachers.contains(teacher2));
    }

    @Test
    public void testUpdateTeacher() {
        Teacher teacher = new Teacher();
        teacher.setFirstName("Test First");
        teacher.setLastName("Test Last");
        teacher.setSpecialty("Test Specialty");
        teacher = teacherDao.addTeacher(teacher);

        Teacher fromDao = teacherDao.getTeacherById(teacher.getId());
        assertEquals(teacher, fromDao);

        teacher.setFirstName("New Test First");
        teacherDao.updateTeacher(teacher);

        assertNotEquals(teacher, fromDao);

        fromDao = teacherDao.getTeacherById(teacher.getId());

        assertEquals(teacher, fromDao);
    }

    @Test
    public void testDeleteTeacherById() {
        Teacher teacher = new Teacher();
```

```
        teacher.setFirstName("Test First");
        teacher.setLastName("Test Last");
        teacher.setSpecialty("Test Specialty");
        teacher = teacherDao.addTeacher(teacher);

        Student student = new Student();
        student.setFirstName("Test Student First");
        student.setLastName("Test Student Last");
        student = studentDao.addStudent(student);
        List<Student> students = new ArrayList<>();
        students.add(student);

        Course course = new Course();
        course.setName("Test Course");
        course.setTeacher(teacher);
        course.setStudents(students);
        course = courseDao.addCourse(course);

        Teacher fromDao = teacherDao.getTeacherById(teacher.getId());
        assertEquals(teacher, fromDao);

        teacherDao.deleteTeacherById(teacher.getId());

        fromDao = teacherDao.getTeacherById(teacher.getId());
        assertNull(fromDao);
    }
```

With that, we move to the Student DAO tests, where we set things up similarly:

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class StudentDaoDBTest {

    @Autowired
    TeacherDao teacherDao;

    @Autowired
    StudentDao studentDao;

    @Autowired
    CourseDao courseDao;

    public StudentDaoDBTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        List<Teacher> teachers = teacherDao.getAllTeachers();
        for(Teacher teacher : teachers) {
            teacherDao.deleteTeacherById(teacher.getId());
```

```
        }

        List<Student> students = studentDao.getAllStudents();
        for(Student student : students) {
            studentDao.deleteStudentById(student.getId());
        }

        List<Course> courses = courseDao.getAllCourses();
        for(Course course : courses) {
            courseDao.deleteCourseById(course.getId());
        }
    }

    @After
    public void tearDown() {
    }
```

Next, we fill in the test, making sure to have complex objects for the delete here as well:

```java
@Test
public void testAddAndGetStudent() {
    Student student = new Student();
    student.setFirstName("Test Student First");
    student.setLastName("Test Student Last");
    student = studentDao.addStudent(student);

    Student fromDao = studentDao.getStudentById(student.getId());
    assertEquals(student, fromDao);
}

@Test
public void testGetAllStudents() {
    Student student = new Student();
    student.setFirstName("Test Student First");
    student.setLastName("Test Student Last");
    student = studentDao.addStudent(student);

    Student student2 = new Student();
    student2.setFirstName("Test Student First 2");
    student2.setLastName("Test Student Last 2");
    student2 = studentDao.addStudent(student2);

    List<Student> students = studentDao.getAllStudents();

    assertEquals(2, students.size());
    assertTrue(students.contains(student));
    assertTrue(students.contains(student2));
}
```

```java
@Test
public void testUpateStudent() {
    Student student = new Student();
    student.setFirstName("Test Student First");
    student.setLastName("Test Student Last");
    student = studentDao.addStudent(student);

    Student fromDao = studentDao.getStudentById(student.getId());
    assertEquals(student, fromDao);

    student.setFirstName("New Test Student First");
    studentDao.updateStudent(student);

    assertNotEquals(student, fromDao);

    fromDao = studentDao.getStudentById(student.getId());

    assertEquals(student, fromDao);
}

@Test
public void testDeleteStudentById() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test First");
    teacher.setLastName("Test Last");
    teacher.setSpecialty("Test Specialty");
    teacher = teacherDao.addTeacher(teacher);
```

```
        Student student = new Student();
        student.setFirstName("Test Student First");
        student.setLastName("Test Student Last");
        student = studentDao.addStudent(student);
        List<Student> students = new ArrayList<>();
        students.add(student);

        Course course = new Course();
        course.setName("Test Course");
        course.setTeacher(teacher);
        course.setStudents(students);
        course = courseDao.addCourse(course);

        Student fromDao = studentDao.getStudentById(student.getId());
        assertEquals(student, fromDao);

        studentDao.deleteStudentById(student.getId());

        fromDao = studentDao.getStudentById(student.getId());
        assertNull(fromDao);
    }
```

Lastly, we have tests for our Course DAO. This will be a little more complicated due to the relationships being managed by the Course object. Here is the setup, like the others:

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class CourseDaoDBTest {

    @Autowired
    TeacherDao teacherDao;

    @Autowired
    StudentDao studentDao;

    @Autowired
    CourseDao courseDao;

    public CourseDaoDBTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        List<Teacher> teachers = teacherDao.getAllTeachers();
        for(Teacher teacher : teachers) {
            teacherDao.deleteTeacherById(teacher.getId());
```

```
        }

        List<Student> students = studentDao.getAllStudents();
        for(Student student : students) {
            studentDao.deleteStudentById(student.getId());
        }

        List<Course> courses = courseDao.getAllCourses();
        for(Course course : courses) {
            courseDao.deleteCourseById(course.getId());
        }
    }

    @After
    public void tearDown() {
    }
```

Then we'll write the tests, using full objects everywhere:

```java
@Test
public void testAddAndGetCourse() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test Teacher First");
    teacher.setLastName("Test Teacher Last");
    teacher.setSpecialty("Test Teacher Specialty");
    teacher = teacherDao.addTeacher(teacher);

    Student student = new Student();
    student.setFirstName("Test Student First");
    student.setLastName("Test Student Last");
    student = studentDao.addStudent(student);

    List<Student> students = new ArrayList<>();
    students.add(student);

    Course course = new Course();
    course.setName("Test Course Name");
    course.setTeacher(teacher);
    course.setStudents(students);
    course = courseDao.addCourse(course);

    Course fromDao = courseDao.getCourseById(course.getId());
    assertEquals(course, fromDao);
}

@Test
public void testGetAllCourses() {
    Teacher teacher = new Teacher();
```

```
        teacher.setFirstName("Test Teacher First");
        teacher.setLastName("Test Teacher Last");
        teacher.setSpecialty("Test Teacher Specialty");
        teacher = teacherDao.addTeacher(teacher);

        Student student = new Student();
        student.setFirstName("Test Student First");
        student.setLastName("Test Student Last");
        student = studentDao.addStudent(student);

        List<Student> students = new ArrayList<>();
        students.add(student);

        Course course = new Course();
        course.setName("Test Course Name");
        course.setTeacher(teacher);
        course.setStudents(students);
        course = courseDao.addCourse(course);

        Course course2 = new Course();
        course2.setName("Test Course Name 2");
        course2.setTeacher(teacher);
        course2.setStudents(students);
        course2 = courseDao.addCourse(course2);

        List<Course> courses = courseDao.getAllCourses();
        assertEquals(2, courses.size());
        assertTrue(courses.contains(course));
        assertTrue(courses.contains(course2));
```

```
}

@Test
public void testUpdateCourse() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test Teacher First");
    teacher.setLastName("Test Teacher Last");
    teacher.setSpecialty("Test Teacher Specialty");
    teacher = teacherDao.addTeacher(teacher);

    Student student = new Student();
    student.setFirstName("Test Student First");
    student.setLastName("Test Student Last");
    student = studentDao.addStudent(student);

    List<Student> students = new ArrayList<>();
    students.add(student);

    Course course = new Course();
    course.setName("Test Course Name");
    course.setTeacher(teacher);
    course.setStudents(students);
    course = courseDao.addCourse(course);

    Course fromDao = courseDao.getCourseById(course.getId());
    assertEquals(course, fromDao);

    course.setName("New Test Course Name");
    Student student2 = new Student();
```

```
        student2.setFirstName("Test Student First 2");
        student2.setLastName("Test Student Last 2");
        student2 = studentDao.addStudent(student2);
        students.add(student2);
        course.setStudents(students);

        courseDao.updateCourse(course);

        assertNotEquals(course, fromDao);

        fromDao = courseDao.getCourseById(course.getId());
        assertEquals(course, fromDao);
    }

    @Test
    public void testDeleteCourseById() {
        Teacher teacher = new Teacher();
        teacher.setFirstName("Test Teacher First");
        teacher.setLastName("Test Teacher Last");
        teacher.setSpecialty("Test Teacher Specialty");
        teacher = teacherDao.addTeacher(teacher);

        Student student = new Student();
        student.setFirstName("Test Student First");
        student.setLastName("Test Student Last");
        student = studentDao.addStudent(student);

        List<Student> students = new ArrayList<>();
        students.add(student);
```

```
    Course course = new Course();
    course.setName("Test Course Name");
    course.setTeacher(teacher);
    course.setStudents(students);
    course = courseDao.addCourse(course);

    Course fromDao = courseDao.getCourseById(course.getId());
    assertEquals(course, fromDao);

    courseDao.deleteCourseById(course.getId());

    fromDao = courseDao.getCourseById(course.getId());
    assertNull(fromDao);
}

@Test
public void testGetCoursesForTeacher() {
    Teacher teacher = new Teacher();
    teacher.setFirstName("Test Teacher First");
    teacher.setLastName("Test Teacher Last");
    teacher.setSpecialty("Test Teacher Specialty");
    teacher = teacherDao.addTeacher(teacher);

    Teacher teacher2 = new Teacher();
    teacher2.setFirstName("Test Teacher First 2");
    teacher2.setLastName("Test Teacher Last 2");
    teacher2.setSpecialty("Test Teacher Specialty 2");
    teacher2 = teacherDao.addTeacher(teacher2);
```

```
Student student = new Student();
student.setFirstName("Test Student First");
student.setLastName("Test Student Last");
student = studentDao.addStudent(student);

List<Student> students = new ArrayList<>();
students.add(student);

Course course = new Course();
course.setName("Test Course Name");
course.setTeacher(teacher);
course.setStudents(students);
course = courseDao.addCourse(course);

Course course2 = new Course();
course2.setName("Test Course Name");
course2.setTeacher(teacher2);
course2.setStudents(students);
course2 = courseDao.addCourse(course2);

Course course3 = new Course();
course3.setName("Test Course Name");
course3.setTeacher(teacher);
course3.setStudents(students);
course3 = courseDao.addCourse(course3);

List<Course> courses = courseDao.getCoursesForTeacher(teacher);
assertEquals(2, courses.size());
```

```
        assertTrue(courses.contains(course));
        assertFalse(courses.contains(course2));
        assertTrue(courses.contains(course3));
    }

    @Test
    public void testGetCoursesForStudent() {
        Teacher teacher = new Teacher();
        teacher.setFirstName("Test Teacher First");
        teacher.setLastName("Test Teacher Last");
        teacher.setSpecialty("Test Teacher Specialty");
        teacher = teacherDao.addTeacher(teacher);

        Student student = new Student();
        student.setFirstName("Test Student First");
        student.setLastName("Test Student Last");
        student = studentDao.addStudent(student);

        Student student2 = new Student();
        student2.setFirstName("Test Student First 2");
        student2.setLastName("Test Student Last 2");
        student2 = studentDao.addStudent(student2);

        List<Student> students = new ArrayList<>();
        students.add(student);
        students.add(student2);

        List<Student> students2 = new ArrayList<>();
        students2.add(student2);
```

```
        Course course = new Course();
        course.setName("Test Course Name");
        course.setTeacher(teacher);
        course.setStudents(students);
        course = courseDao.addCourse(course);

        Course course2 = new Course();
        course2.setName("Test Course Name");
        course2.setTeacher(teacher);
        course2.setStudents(students2);
        course2 = courseDao.addCourse(course2);

        Course course3 = new Course();
        course3.setName("Test Course Name");
        course3.setTeacher(teacher);
        course3.setStudents(students);
        course3 = courseDao.addCourse(course3);

        List<Course> courses = courseDao.getCoursesForStudent(student);
        assertEquals(2, courses.size());
        assertTrue(courses.contains(course));
        assertFalse(courses.contains(course2));
        assertTrue(courses.contains(course3));
    }
```

With the tests complete, you should be able to build or run tests and they should all pass if you brought everything in correctly. If anything does not pass, take a look at the error and then investigate and debug the code to find the issue.

Now that we have our DAOs and tests written, we can start looking at our Controllers and the front-end.

Step 8: Index Page

With our back-end complete, we can start looking at the front end. We will build pages that allow us to perform all CRUD actions for each object. We will also have a simple index page that links out to the other pages.

Setup

To start, this is what the basic template of our pages looks like. It brings in Bootstrap 4 and links up with Thymeleaf:

```
<!doctype html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

    <title>Course Manager</title>
  </head>
  <body>
    <!-- Main Body of Page -->

    <!-- End main body -->

    <!-- Optional JavaScript -->
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>
```

```
  </body>
</html>;
```

All the code we will add to the HTML will be in the main body section of the page, though we will also adjust the title in the head of the page.

Let’s start with our index page.

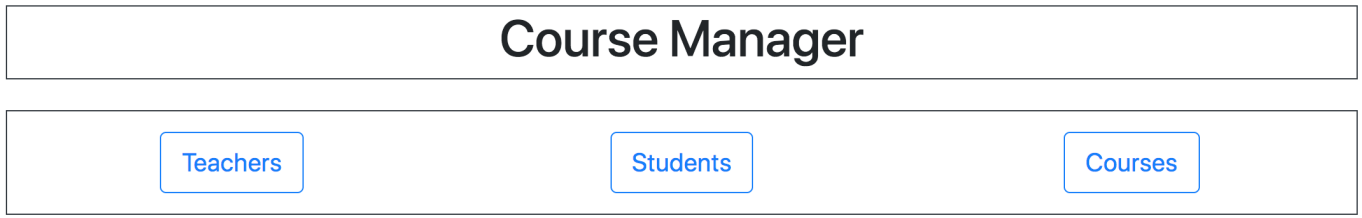
Index Page

The index here is special in that we will not put any dynamic data on this page, so it will simply act as a landing page for when we go to the site. Because of this, we can create our index.html file in the src/main/resources/static directory. We don’t have to combine this file with data from the back-end, so it is considered static.

Here is the main body of our index page, giving us a simple landing page with links to our other sections:

```
<div class="container">
    <div class="row m-4">
        <div class="col text-center border border-dark">
            <h1>Course Manager</h1>
        </div>
    </div>
    <div class="row m-4 border border-dark">
        <div class="col text-center m-3">
            <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
        </div>
        <div class="col text-center m-3">
            <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
        </div>
        <div class="col text-center m-3">
            <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
        </div>
    </div>
</div>
```

And here is how it should look:



It’s been purposely left very simple, so feel free to modify it to add any look and feel you would like. The only requirement is that the button links remain in some way so we can get to those pages.

Next, we should look at the Teachers page.

Step 9: Teachers Page

We will load dynamic data into the Teachers page, so we need to create a controller and mappings for the pages as well as a few template pages to display our data.

We will create multiple controllers, one to manage each section of the site. Start by making a controller package in our project and create a TeacherController file.

We'll start our file with this:

```
@Controller
public class TeacherController {

   @Autowired
   TeacherDao teacherDao;

   @Autowired
   StudentDao studentDao;

   @Autowired
   CourseDao courseDao;
}
```

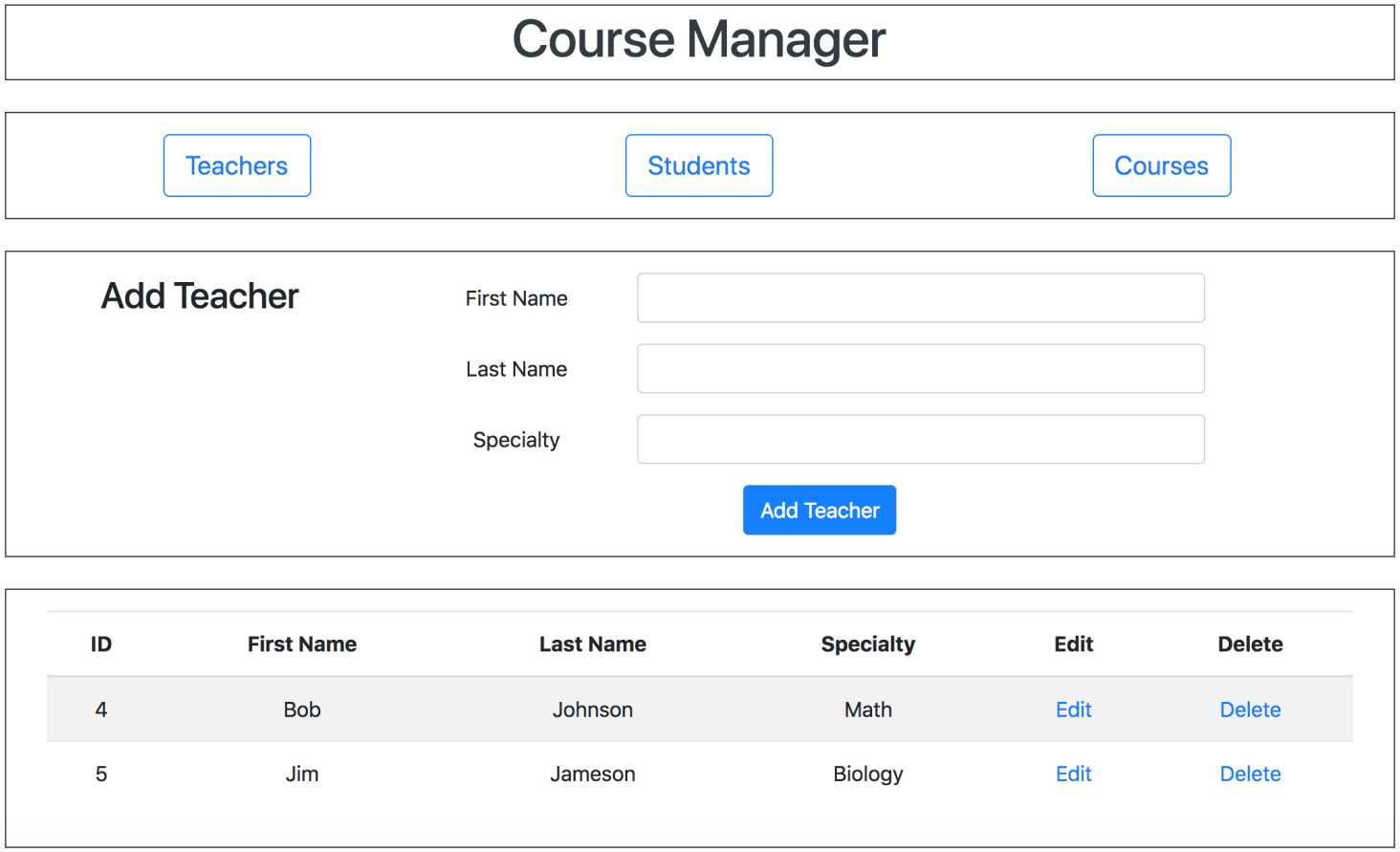
- We annotate it as a @Controller.
- We then autowire in all the DAOs. We may end up not using them all, at which point we can remove the ones we don't need, but to be safe we will start with all of them.

Now we want to create our first mapping: a GET mapping that will display the list of Teachers in our system:

```
@GetMapping("teachers")
public String displayTeachers(Model model) {
    List<Teacher> teachers = teacherDao.getAllTeachers();
    model.addAttribute("teachers", teachers);
    return "teachers";
}
```

- We start with the @GetMapping for teachers.
- We have a Model in our method parameters so that we can send data out to a page.
- We then retrieve the list of Teachers from the DAO and add it to the Model.
- Finally, we return teachers, meaning we will need a teachers.html file to push our data to.

With that in place, we can build our teachers.html page and set it up to display the Teacher information. Create a teachers.html file in the src/main/resources/templates directory and update it to look like the template from above. This is what it will look like when we are finished:



To start, add in the following code:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <table class="table table-striped">
                        <thead>
                            <tr>
                                <th>ID</th>
                                <th>First Name</th>
                                <th>Last Name</th>
                                <th>Specialty</th>
                                <th>Edit</th>
                                <th>Delete</th>
                            </tr>
```

```
                    </thead>
                    <tbody>
                        <tr th:each="teacher : ${teachers}">
                            <td th:text="${teacher.id}">Teacher ID</td>
                            <td th:text="${teacher.firstName}">Teacher First Name</td>
                            <td th:text="${teacher.lastName}">Teacher Last Name</td>
                            <td th:text="${teacher.specialty}">Teacher Specialty</td>
                            <td>Edit</td>
                            <td>Delete</td>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
```

- The top of the page matches the index so we have a way to navigate around our page. This is just one way to manage that - Bootstrap gives us many options for navbars and other utilities to help manage site navigation.
- We set up a table to display our Teachers using the Thymeleaf th:each attribute to generate rows for each Teacher.
- In each row, we include an Edit and Delete cell that we will use later to help us perform those actions on the Teacher objects.

We can't see anything in this list at the moment because we have no way to add Teachers to our system, so putting an Add Teacher form in place will be our next step. For this one, let's build the HTML first.

We will add some code into the middle of our main body, between the section that contains the links and the section that contains the table, so our form is above the Teacher listing. The code to add in looks like this:

```
<div class="row m-4 border border-dark">
    <div class="col-3 text-center m-3">
        <span class="h3">Add Teacher</span>
    </div>
    <div class="col-7 text-center m-3">
        <form action="addTeacher" method="POST">
            <div class="form-group row">
                <label for="firstName" class="col-3 col-form-label">
                    First Name</label>
                <div class="col-9">
                    <input type="text" name="firstName"
                           id="firstName" class="form-control"/>
                </div>
            </div>
            <div class="form-group row">
                <label for="lastName" class="col-3 col-form-label">
                    Last Name</label>
                <div class="col-9">
                    <input type="text" name="lastName"
                           id="lastName" class="form-control"/>
                </div>
            </div>
            <div class="form-group row">
                <label for="specialty" class="col-3 col-form-label">
                    Specialty</label>
                <div class="col-9">
                    <input type="text" name="specialty"
                           id="specialty" class="form-control"/>
                </div>
```

```
                    </div>
                    <button type="submit" class="btn btn-primary">Add Teacher</button>
                </form>
            </div>
        </div>
```

This will create a form that submits a POST to addTeacher. The data will come from the input fields in our form and the important attribute there to pick up the data is the name attribute. When we work on the back-end, we use the name attribute to grab the data from the fields.

Let's move to the back-end now and create our addTeacher POST mapping:

```
@PostMapping("addTeacher")
public String addTeacher(HttpServletRequest request) {
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String specialty = request.getParameter("specialty");

    Teacher teacher = new Teacher();
    teacher.setFirstName(firstName);
    teacher.setLastName(lastName);
    teacher.setSpecialty(specialty);

    teacherDao.addTeacher(teacher);

    return "redirect:/teachers";
}
```

- We have our @PostMapping for addTeacher at the beginning.
- The parameter we need for this is a HttpServletRequest.

- Once in the method, we use the HttpServletRequest to retrieve the fields from the form based on the name we set in the HTML for each input.
- Once we have the data, we create the Teacher object and fill it with the data.
- Once we've finished creating the Teacher, we use our teacherDao to save it to the database.
- Finally, we redirect our browser back to the Teachers page.

This should allow us to create Teacher objects. After they are added to the database, the page should refresh so we can see them in the listing.

Now that we can create Teachers, we need to be able to delete them. We'll look at deleting them before updating them because updating is a little more complicated.

For deleting, we have created a piece of placeholder text in each row of the table that we will turn into a link. The link will trigger the deletion and then take us back to this page.

To create the link, replace the Delete text in the table with this:

```
<a href="#" th:href="@{/deleteTeacher(id=${teacher.id})}">Delete</a>
```

- The attribute th:href is from Thymeleaf and allows us to put dynamic data into links.
- The contents of the th:href - @{/deleteTeacher(id=${teacher.id})} - will generate a URL like /deleteTeacher?id=1. That format is how we add links with dynamic data to our HTML.
- All a tags need an href attribute by itself, so we simply set its value to # and the th:href will replace that when it is processed.

With that link in place, we can now create our mapping in the TeacherController:

```
    @GetMapping("deleteTeacher")
    public String deleteTeacher(HttpServletRequest request) {
        int id = Integer.parseInt(request.getParameter("id"));
        teacherDao.deleteTeacherById(id);

        return "redirect:/teachers";
    }
```

- Because this is working off of a link, we need it to be a @GetMapping for deleteTeacher.
- To pull in the ID from the URL, we use the HttpServletRequest to get that parameter.
- Once we have the ID, we use the DAO to delete the Teacher.
- Finally, we redirect the browser back to the main Teachers page.

The last step for the Teachers page is to allow the user to edit a Teacher. To this end, we'll create a new page that will focus purely on editing the Teacher. To start, however, we need to modify the Edit placeholder in the listing to look like this:

```
<a href="#" th:href="@{/editTeacher(id=${teacher.id})}">Edit</a>
```

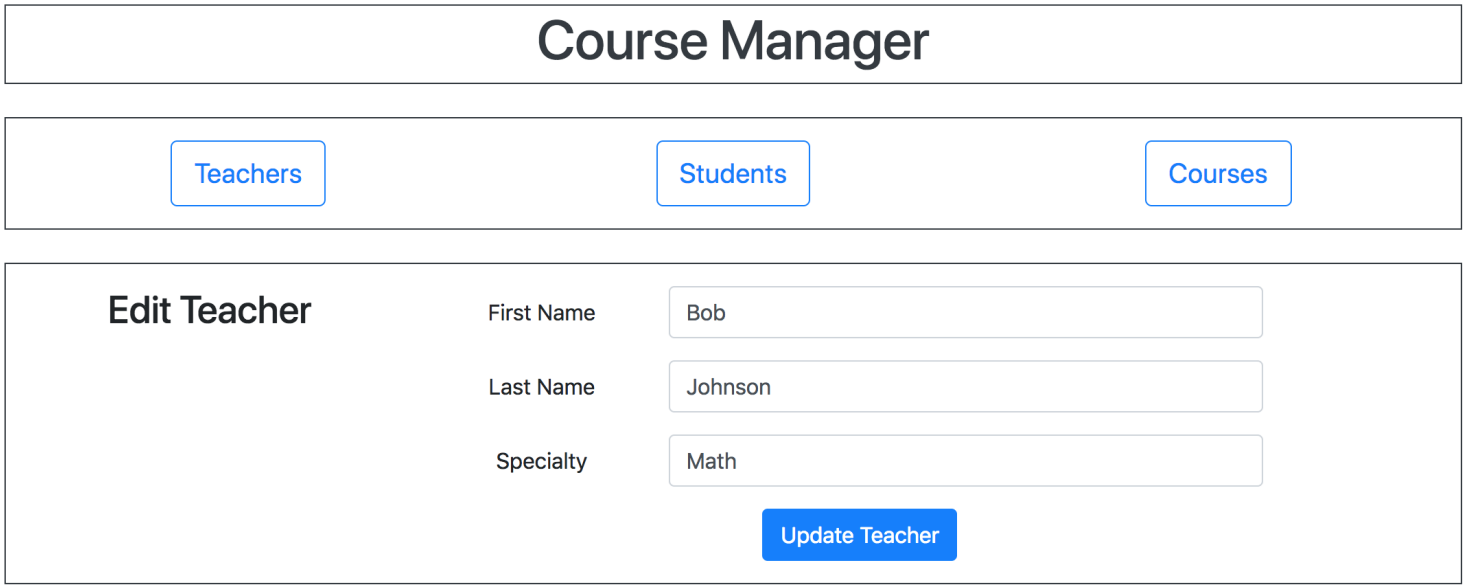
- Similar to the previous link, we use the Thymeleaf th:href attribute to construct our link.
- The value @{/editTeacher(id=${teacher.id})} generates a URL that looks like /editTeacher?id=1 so that we can tell the Edit Teacher page which Teacher to edit.

Now that the link is set up, we can create the GET mapping to display our Edit Teacher page:

```
@GetMapping("editTeacher")
public String editTeacher(HttpServletRequest request, Model model) {
    int id = Integer.parseInt(request.getParameter("id"));
    Teacher teacher = teacherDao.getTeacherById(id);

    model.addAttribute("teacher", teacher);
    return "editTeacher";
}
```

- It's displaying a page, so this will be a @GetMapping for editTeacher.
- We need to bring in the HttpServletRequest to get the ID as well as the Model to send the Teacher object we are editing to the page.
- We retrieve our ID from the HttpServletRequest, retrieve the Teacher for that ID, and put it into the Model.
- Finally, we return editTeacher, which will point us to an editTeacher.html page.

With the GET in place, we can create our editTeacher.html template. This page will display the current data for our Teacher in a form and let us change it, then submit the changes to the database. Here is what the body of that page looks like:



And here is what the HTML for that page looks like:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col-3 text-center m-3">
                    <span class="h3">Edit Teacher</span>
                </div>
                <div class="col-7 text-center m-3">
                    <form action="editTeacher" method="POST">
                        <div class="form-group row">
                            <label for="firstName" class="col-3 col-form-label">
                                First Name</label>
                            <div class="col-9">
                                <input type="text" name="firstName" id="firstName"
                                       class="form-control" th:value="${teacher.firstName}"/>
```

```
                    </div>
                </div>
                <div class="form-group row">
                    <label for="lastName" class="col-3 col-form-label">
                        Last Name</label>
                    <div class="col-9">
                        <input type="text" name="lastName" id="lastName"
                               class="form-control"  th:value="${teacher.lastName}"/>
                    </div>
                </div>
                <div class="form-group row">
                    <label for="specialty" class="col-3 col-form-label">
                        Specialty</label>
                    <div class="col-9">
                        <input type="text" name="specialty" id="specialty"
                               class="form-control" th:value="${teacher.specialty}">
                    </div>
                </div>
                <input type="hidden" name="id" th:value="${teacher.id}"/>
                <button type="submit" class="btn btn-primary">
                    Update Teacher</button>
            </form>
        </div>
    </div>
</div>
```

- We now point our form at editTeacher, a POST mapping we will add in to the TeacherController in a moment.
- To display the existing data, we use the Thymeleaf attribute th:value in each input. For these, we specify which object we are pulling data from and the name of the field we are getting; for example, ${teacher.firstName}.

- We also include a hidden input that will hold the ID of the object we are editing so we can update the correct one when we submit changes.

Now, we can move to our editTeacher POST mapping. Similar to when we did REST, we can have two mappings with the same name as long as the method is different; for example, GET vs. POST. Here is the code:

```
@PostMapping("editTeacher")
public String performEditTeacher(HttpServletRequest request) {
    int id = Integer.parseInt(request.getParameter("id"));
    Teacher teacher = teacherDao.getTeacherById(id);

    teacher.setFirstName(request.getParameter("firstName"));
    teacher.setLastName(request.getParameter("lastName"));
    teacher.setSpecialty(request.getParameter("specialty"));

    teacherDao.updateTeacher(teacher);

    return "redirect:/teachers";
}
```

- This is a POST mapping for editTeacher.
- We only need to take in the HttpServletRequest so we can get the data out of the form.
- We start by getting the hidden ID and pulling in the original version of the object.
- We then get all the new data out of the form and set it into the Teacher object.
- Once we have all the data, we make a call to our DAO update method.
- We can then redirect back to the main Teachers page.

We should now have a fully functional Teachers page with the ability to view, add, delete, and edit any of our Teachers. Next, we'll move on to the Students page. This will look and act similar to the Teachers, but we'll look at a different way to pull in data from our forms.

Step 10: Students Page

Our Students page will look very much like the Teachers page with a form to add a Student, a list of the current Students, and the ability to edit and delete Students, but we will handle a few things behind the scenes in slightly different ways.

Let's start by creating our StudentController class and setting it up:

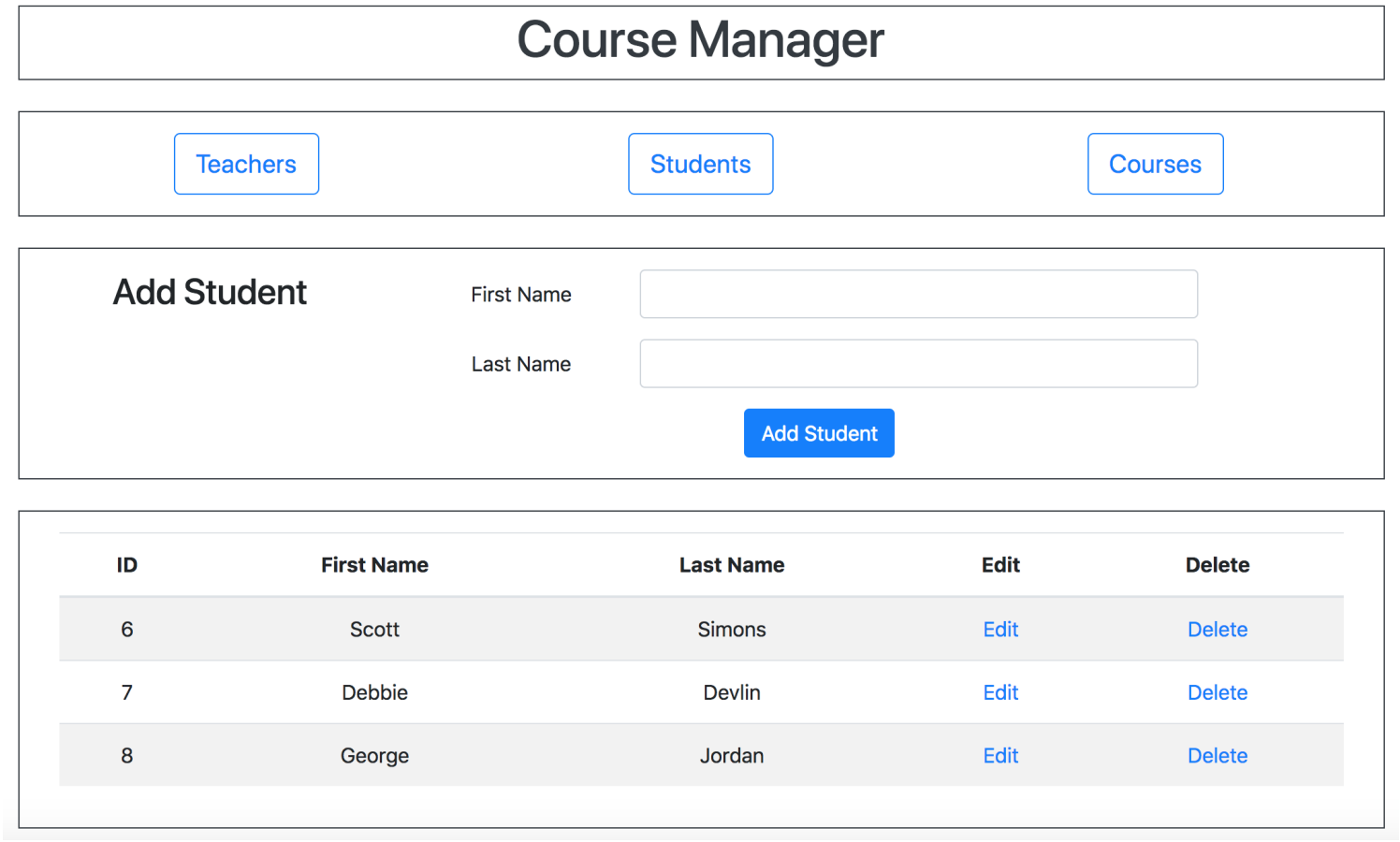
```
@Controller
public class StudentController {

   @Autowired
   TeacherDao teacherDao;

   @Autowired
   StudentDao studentDao;

   @Autowired
   CourseDao courseDao;
}
```

Then let's create a base GET mapping for Students that will display the main Students page:

```
    @GetMapping("students")
    public String displayStudents(Model model) {
        List<Student> students = studentDao.getAllStudents();
        model.addAttribute("students", students);
        return "students";
    }
```

- We start by making a GET mapping for students.
- We bring in the Model so we can send data to the page.
- We then pick up our list of Students from the DAO and put it into the Model.
- Lastly, we return students, so it combines the data with the students.html template we are about to create.

Now we'll make our students.html template that will display this information based on the page template we developed earlier. Here is what the page will look like when we are finished:



This is the HTML of the main body of the page:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
                        <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <table class="table table-striped">
                        <thead>
                            <tr>
                                <th>ID</th>
                                <th>First Name</th>
                                <th>Last Name</th>
                                <th>Edit</th>
                                <th>Delete</th>
                            </tr>
                        </thead>
```

```
                    <tbody>
                        <tr th:each="student : ${students}">
                            <td th:text="${student.id}">Teacher ID</td>
                            <td th:text="${student.firstName}">Teacher First Name</td>
                            <td th:text="${student.lastName}">Teacher Last Name</td>
                            <td>Edit</td>
                            <td>Delete</td>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
```

- We start with the same top navigation, allowing us to move around the site.
- We then have a table that will use the th:each to generate table rows with data for each Student in our system.
- We have placeholders for Edit and Delete there as well.

Now we'll add in our add Student form. The HTML isn't going to look much different, but the back-end will act a little differently compared to the Teachers page. First, let's create our form above the list and below the navigation in our page. Here is the code:

```
            <div class="row m-4 border border-dark">
                <div class="col-3 text-center m-3">
                    <span class="h3">Add Student</span>
                </div>
                <div class="col-7 text-center m-3">
                    <form action="addStudent" method="POST">
                        <div class="form-group row">
                            <label for="firstName" class="col-3 col-form-label">
                                First Name</label>
                            <div class="col-9">
                                <input type="text" name="firstName"
                                       id="firstName" class="form-control"/>
                            </div>
                        </div>
                        <div class="form-group row">
                            <label for="lastName" class="col-3 col-form-label">
                                Last Name</label>
                            <div class="col-9">
                                <input type="text" name="lastName"
                                       id="lastName" class="form-control"/>
                            </div>
                        </div>
                        <button type="submit" class="btn btn-primary">Add Student</button>
                    </form>
                </div>
            </div>
```

- Our form is a POST with the action addStudent.
- We create inputs for the two fields we need to pull in, first name and last name.

- In each input, we have a name attribute that we will use to pick up the data on the back-end.
- Finally, we have a submit button to send the data to our Controller.

Now we can build our addStudent POST mapping in the StudentController:

```
    @PostMapping("addStudent")
    public String addStudent(String firstName, String lastName) {
        Student student = new Student();
        student.setFirstName(firstName);
        student.setLastName(lastName);
        studentDao.addStudent(student);

        return "redirect:/students";
    }
```

- We start by declaring our @PostMapping of Students.
- Instead of pulling in the HttpServletRequest and then pulling out the data, we are bringing the data in directly as parameters of the method. To do this, the name of the parameter has to match the name in the form exactly. Here, firstName in the form is String firstName in the method parameters.
- With the data pulled in, we create our Student object and save it.
- We then redirect back to the main Students page.

Pulling data directly by its name in the form can be useful because it eliminates a step in the process of handling the form data. One potential disadvantage is if you have many form fields to read in, your method signature can get very long. If that's the case, there is yet a third way we can pull in that data that we'll look at for editing Students.

For now, let's focus on deleting. First, we need to replace the Delete placeholder in the listing with the following link:

```
<a href="#" th:href="@{/deleteStudent(id=${student.id})}">Delete</a>
```

- Like the previous links, we use th:href so we can include dynamic data in the link, in this case the Student ID in the URL.

- The Thymeleaf code should generate links for us in the format of /deleteStudent?id=1.

Now we'll create the mapping for deleteStudent:

```
    @GetMapping("deleteStudent")
    public String deleteStudent(Integer id) {
        studentDao.deleteStudentById(id);
        return "redirect:/students";
    }
```

- Because we are using a link to trigger this action, it is a GET mapping for deleteStudent.
- We don't use HttpServletRequest here, either. Instead, we can pull in the data from the URL using the key, id in this case.
- We know it should be a number, so we pull it in as an integer. We use Integer instead of int because an Integer can accept a null value, so if no id was sent we can deal with it in the method if we want.
- Once we have the ID, we call our DAO delete method and redirect back to the main Students page.

Lastly, we need to put the edit in place. Let's start with the link on the Students page, so modify the Edit placeholder to look like this:

```
<a href="#" th:href="@{/editStudent(id=${student.id})}">Edit</a>
```

- We use the th:href to create the link that will have the format /editStudent?id=1.

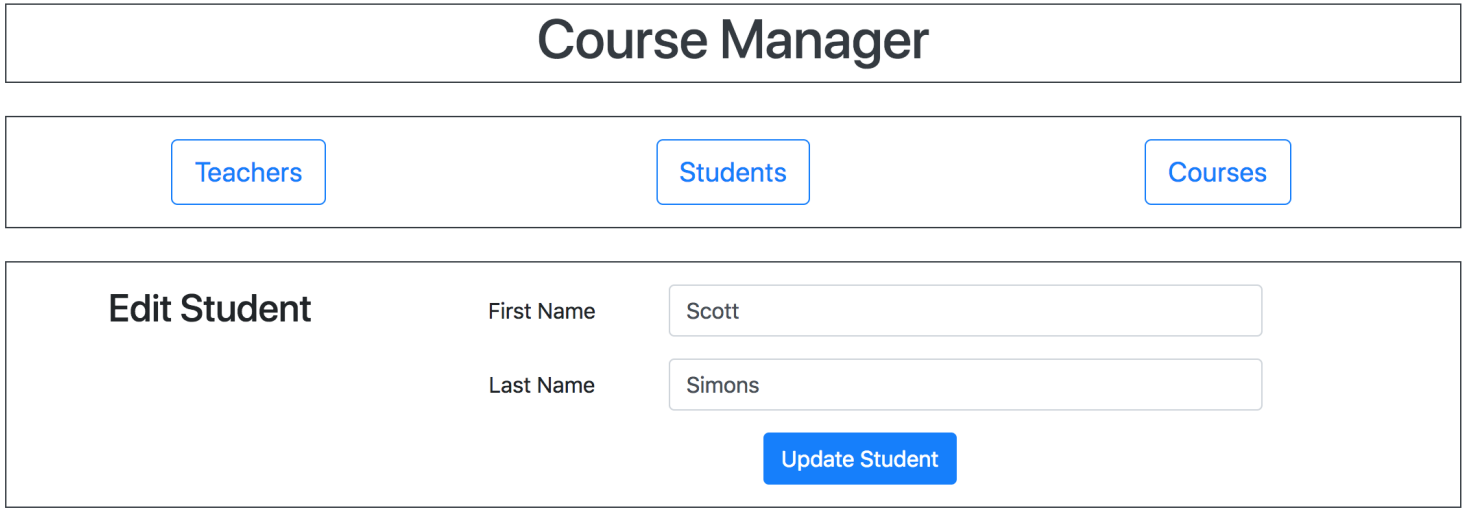
Next, we'll put in place the GET mapping for our Edit Student page:

```
    @GetMapping("editStudent")
    public String editStudent(Integer id, Model model) {
        Student student = studentDao.getStudentById(id);
        model.addAttribute("student", student);
        return "editStudent";
    }
```

- We declare our @GetMapping for editStudent.

- We take the ID directly from the URL and bring in a Model so we can send data to the page.
- We use the ID to get the Student and add it to the Model.
- Finally, we return to editStudent, which will send us to an editStudent.html template that we will create next.

With the mapping in place, let's create our editStudent.html template based on the template example from before. Here is what that page will look like:



And here is the HTML for the body of that page:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col-3 text-center m-3">
                    <span class="h3">Edit Student</span>
                </div>
                <div class="col-7 text-center m-3">
                    <form action="editStudent" method="POST">
                        <div class="form-group row">
                            <label for="firstName" class="col-3 col-form-label">
                                First Name</label>
                            <div class="col-9">
                                <input type="text" name="firstName" id="firstName"
                                       class="form-control" th:value="${student.firstName}"/>
```

```
                    </div>
                </div>
                <div class="form-group row">
                    <label for="lastName" class="col-3 col-form-label">
                        Last Name</label>
                    <div class="col-9">
                        <input type="text" name="lastName" id="lastName"
                               class="form-control"  th:value="${student.lastName}"/>
                    </div>
                </div>
                <input type="hidden" name="id" th:value="${student.id}"/>
                <button type="submit" class="btn btn-primary">Update Student</button>
            </form>
        </div>
    </div>
</div>
```

- We point our form at editStudent, a POST mapping we will add in to the StudentController.
- To display the existing data, we use the Thymeleaf attribute th:value in each input with the name of the field in the object we want, like ${student.firstName}.
- We also need to include a hidden input that will hold the ID of the object we are editing so the correct object is updated in the database when we submit.

We can now build our editStudent POST mapping in the back-end, and we can see the third way of bringing in data from a form:

```
@PostMapping("editStudent")
public String performEditStudent(Student student) {
    studentDao.updateStudent(student);
    return "redirect:/students";
}
```

- We create a @PostMapping for editStudent.
- Because the fields in our form match up exactly with the fields in our Student object, we can pull the entire form in as a Student object. This only works for input fields where the name matches the name of a field in the Student class. Any input where the name doesn't match up will have to be pulled in separately, either using its name as a parameter or through the HttpServletRequest. If any fields are missing from the form, those will be left blank in the object you pull in.
- We have the full object right away, so we can call our DAO update method and redirect back to the main Students page.

With that in place, our Students page should be complete.

We now know three different ways to pull in form data: through the HttpServletRequest, as individual method parameters, or as a full object. The HttpServletRequest can always be used as a fallback if the others won't work because it will always pull in the data as a string and let you modify it from there. The other two need a little more care to work with, but can be more efficient. It's up to you to decide which one is appropriate for the work you are doing.

Next, we'll move to the Courses page, where we will introduce drop-downs and multi-selects in order to manage our relationships.

Step 11: Courses Page

The Courses page will look a little different than the previous two. Because a Course manages the relationships with its Teacher and its list of Students, the page needs to represent those in some way. The way we will do this is by using drop-down menus and multi-select inputs. We also need to take special consideration with how we read these into the back-end.

Before we get to all that though, let's start by creating our Controller:

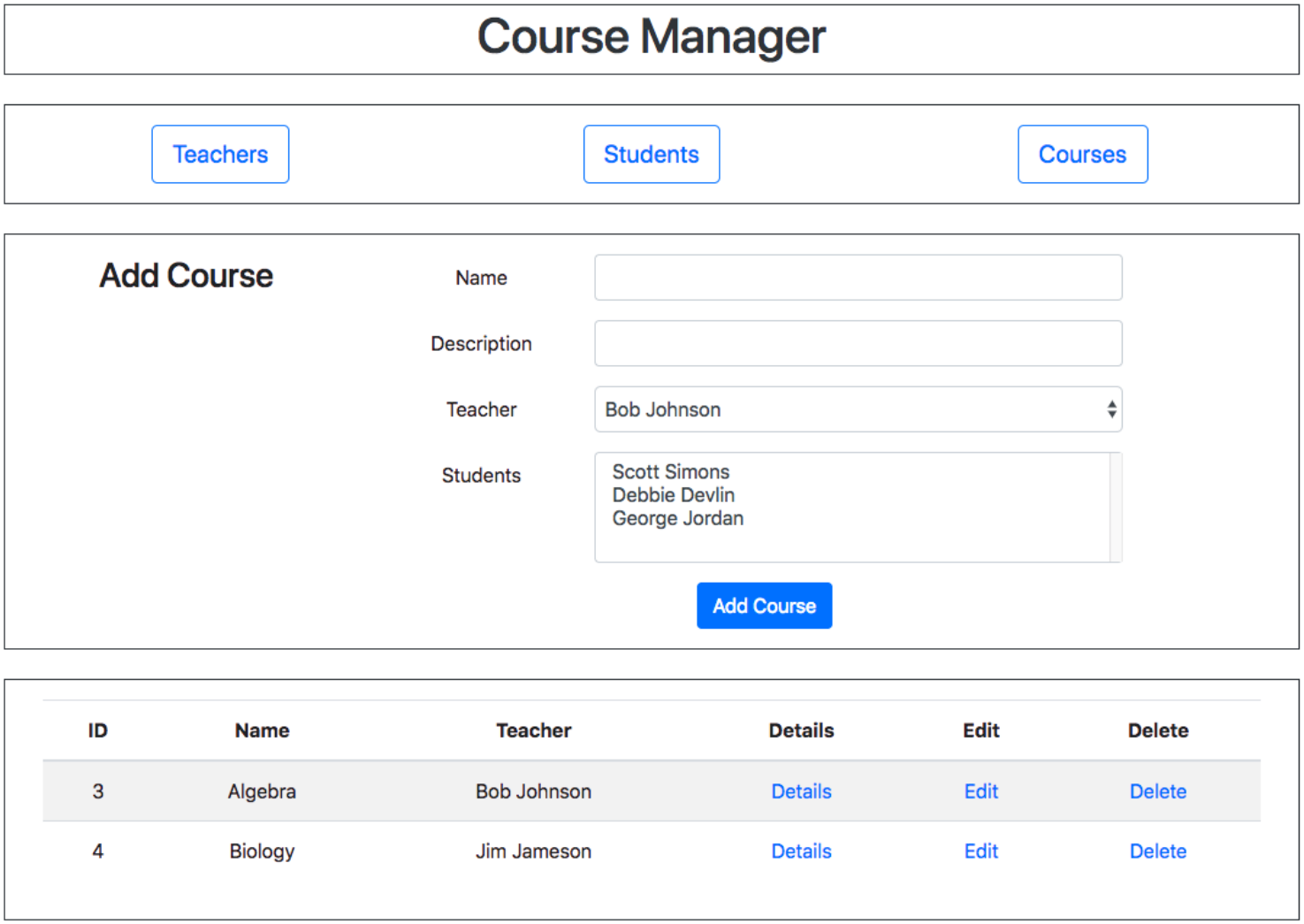
```
@Controller
public class CourseController {

   @Autowired
   TeacherDao teacherDao;

   @Autowired
   StudentDao studentDao;

   @Autowired
   CourseDao courseDao;
}
```

Now, we'll add in our base GET mapping for the Courses page:

```
    @GetMapping("courses")
    public String displayCourses(Model model) {
        List<Course> courses = courseDao.getAllCourses();
        List<Teacher> teachers = teacherDao.getAllTeachers();
        List<Student> students = studentDao.getAllStudents();
        model.addAttribute("courses", courses);
        model.addAttribute("teachers", teachers);
        model.addAttribute("students", students);
        return "courses";
    }
```

- We start by making our @GetMapping for courses.
- We then bring in the Model to send data to the page.
- Next, we pull in our list of all Courses and add it to the Model.
- We also pull in our list of all Students and Teachers and add them to the Model to support the Add function we will create later.
- Finally, we return courses, which will point us at the courses.html page we'll create next.

Now we'll make our courses.html template, based on the page template we developed earlier, to display this information. Here is what it will look like when we are finished:



Here is what the main body of the page should look like to start:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <table class="table table-striped">
                        <thead>
                            <tr>
                                <th>ID</th>
                                <th>Name</th>
                                <th>Teacher</th>
                                <th>Details</th>
                                <th>Edit</th>
                                <th>Delete</th>
                            </tr>
```

```
                    </thead>
                    <tbody>
                        <tr th:each="course : ${courses}">
                            <td th:text="${course.id}">Course ID</td>
                            <td th:text='${course.name}'>Course Name</td>
                            <td th:text="${course.teacher.firstName + ' '
                                + course.teacher.lastName}">Teacher  Name</td>
                            <td>>Details</td>
                            <td>Edit</td>
                            <td>Delete</td>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
```

- We start with the same top navigation, allowing us to move around the site.
- We then have our table that will use the th:each attribute to generate table rows with data for each Course in our system.
- We only display the course ID, course name, and teacher name on this page.
- We add a placeholder for Details, which will be a link to a page that will list the full info for a Course, including description and students.
- We have our placeholders for Edit and Delete as well.

Next, we'll create the form to add a Course. For this form, we need more than just text inputs, so we'll have a drop-down that will let us select a single Teacher and a multi-select to select as many Students as we want. Here is the code for that:

```
<div class="row m-4 border border-dark">
    <div class="col-3 text-center m-3">
        <span class="h3">Add Course</span>
    </div>
    <div class="col-7 text-center m-3">
        <form action="addCourse" method="POST">
            <div class="form-group row">
                <label for="name" class="col-3 col-form-label">
                    Name</label>
                <div class="col-9">
                    <input type="text" name="name"
                           id="name" class="form-control"/>
                </div>
            </div>
            <div class="form-group row">
                <label for="description" class="col-3 col-form-label">
                    Description</label>
                <div class="col-9">
                    <input type="text" name="description"
                           id="description" class="form-control"/>
                </div>
            </div>
            <div class="form-group row">
                <label for="teacher" class="col-3 col-form-label">
                    Teacher</label>
                <div class="col-9">
                    <select id="teacher" name="teacherId" class="form-control" >
                        <option th:each="teacher : ${teachers}"
                                th:value="${teacher.id}"
```

```
                                th:text="${teacher.firstName + ' '
                                    + teacher.lastName}">
                                Teacher Name</option>
                        </select>
                    </div>
                </div>
                <div class="form-group row">
                    <label for="students" class="col-3 col-form-label">
                        Students</label>
                    <div class="col-9">
                        <select multiple id="students" name="studentId"
                                class="form-control" >
                            <option th:each="student : ${students}"
                                th:value="${student.id}"
                                th:text="${student.firstName + ' '
                                    + student.lastName}">
                                Student Name</option>
                        </select>
                    </div>
                </div>
                <button type="submit" class="btn btn-primary">Add Course</button>
            </form>
        </div>
    </div>
```

- We start by taking in the name and description like we normally would, with input text fields.
- When we get to Teacher, we use a select tag. With select, we can't send the entire object, so we give the select tag the name teacherId. We send only the ID of the Teacher we want for this Course to the back-end.
- Inside the select tag, we put a th:each attribute inside an option tag so that we create an option for each Teacher.

- Inside the option tag, we use th:value to indicate what data is sent to the back-end and th:text to indicate what is displayed on the screen.
- When we get to Students, we do the same thing, except we indicate that the select tag is a multiple select, meaning more than one option can be chosen. When we pick up the studentId in the back-end, we will get an array of the IDs that were chosen.
- We fill in the option tags the same way we did with the Teacher.
- Lastly, we create our submit button.

The important takeaways here are how we represent the relationships on the page. The one-to-many relationship of Courses with Teachers can be easily represented as a drop-down menu. For the many-to-many relationship of Courses with Students, we need the multi-select, letting us pick more than one Student from the list. Another common option is checkboxes. With either one, you can select multiple Students to be part of the Course.

Let's look at how we pick up the information from the Add Course form by writing the POST mapping for addCourse:

```
@PostMapping("addCourse")
public String addCourse(Course course, HttpServletRequest request) {
    String teacherId = request.getParameter("teacherId");
    String[] studentIds = request.getParameterValues("studentId");

    course.setTeacher(teacherDao.getTeacherById(Integer.parseInt(teacherId)));

    List<Student> students = new ArrayList<>();
    for(String studentId : studentIds) {
        students.add(studentDao.getStudentById(Integer.parseInt(studentId)));
    }
    course.setStudents(students);
    courseDao.addCourse(course);

    return "redirect:/courses";
}
```

- We start with our @PostMapping for addCourse.
- For this one, we will take in a Course object that captures the name and description fields and an HttpServletRequest object that we use to capture the teacherId and studentIds.
- We pull out the teacherId like we normally pull data from the HttpServletRequest.
- We use the getParameterValues method to get a string array of studentIds.
- We can then use the teacherId to get the Teacher out of the Teacher DAO and set it in the Course.
- Next, we create an empty list of Students, loop through the studentIds, retrieve each Student, and add it to the list.
- Once we have all the students, we set that list into the Course.
- Finally, we call addCourse in our Course DAO to put it into the database.
- We finish by redirecting back to the main Courses page.

We are combining two different ways of retrieving data in this method. We can pull in the name and description using a Course object and because the field names line up, the data goes directly in. We retrieve the teacherId and studentId fields with our HttpServletRequest. They do not go into the Course object because they do not line up with the name of any field in the Course.

Next, we are going to build our Course Details page. First, we want to modify the main Courses page and change the Detail placeholder to:

```
<a href="#" th:href="@{/courseDetail(id=${course.id})}">Details</a>
```

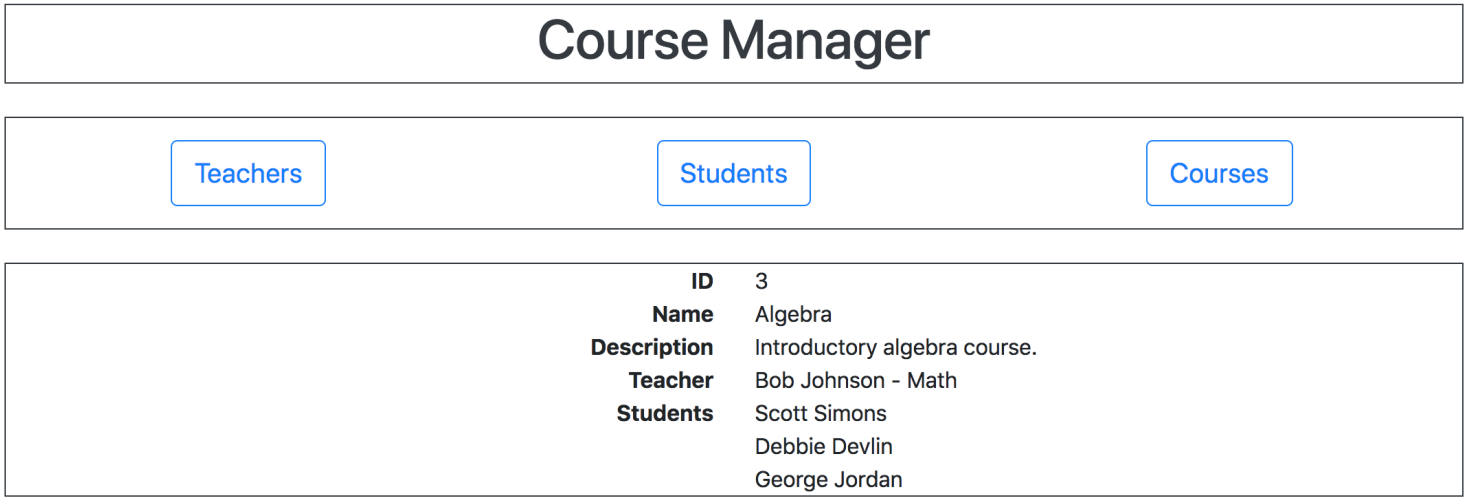
- We create an anchor tag with a th:href attribute where the contents will look like /courseDetail?id=1.

Now we can make the GET mapping for courseDetail:

```
    @GetMapping("courseDetail")
    public String courseDetail(Integer id, Model model) {
        Course course = courseDao.getCourseById(id);
        model.addAttribute("course", course);
        return "courseDetail";
    }
```

- We start by making our @GetMapping for courseDetail.
- We take in the ID from the URL and a Model so we can send data to the page.
- We use the ID to get the Course and add it to the Model.
- Finally, we return courseDetail, which will send us to the courseDetail.html page.

We should now create a courseDetail.html template based on the previously-defined page template. The page will look like this:



And the code for the main body of the page looks like this:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col-6 text-right"><strong>ID</strong></div>
                <div class="col-6 text-left">
                    <span th:text="${course.id}">display id</span>
                </div>

                <div class="col-6 text-right"><strong>Name</strong></div>
                <div class="col-6 text-left">
                    <span th:text="${course.name}">display name</span>
                </div>

                <div class="col-6 text-right"><strong>Description</strong></div>
```

```
            <div class="col-6 text-left">
                <span th:text="${course.description}">display description</span>
            </div>
            <div class="col-6 text-right"><strong>Teacher</strong></div>
            <div class="col-6 text-left">
                <span th:text="${course.teacher.firstName + ' ' +
                      course.teacher.lastName + ' - ' +
                      course.teacher.specialty}">display teacher</span>
            </div>
            <div class="col-6 text-right"><strong>Students</strong></div>
            <div class="col-6 text-left">
                <span th:each="student : ${course.students}">
                    <span th:text="${student.firstName + ' ' + student.lastName}">
                        display student</span><br/>
                </span>
            </div>
        </div>
    </div>
```

- We have our normal navigation at the top.
- We then get into the details, with labels on the left and data on the right.
- For Teacher, we pull out the first name, last name, and specialty to display.
- For Students, we use a th:each attribute to loop through all the Students and display the first name and last name of each.

Our full Course object is sent to the page, so we can access and display all the data we put in it.

Now, let's go back to the main Courses page and take care of the delete. We must first replace the Delete placeholder with the following:

```
<a href="#" th:href="@{/deleteCourse(id=${course.id})}">Delete</a>
```

We'll now add the GET mapping for deleteCourse:

```
    @GetMapping("deleteCourse")
    public String deleteCourse(Integer id) {
        courseDao.deleteCourseById(id);
        return "redirect:/courses";
    }
```

- We start with the @GetMapping of deleteCourse.
- We take in the ID from the URL.
- Using the ID, we call deleteCourseById in the Course DAO.
- Finally, we redirect back to the main Courses page.

With the delete finished, we can finally move on to the edit. Let's start by replacing the Edit placeholder on the main Courses page with this:

```
<a href="#" th:href="@{/editCourse(id=${course.id})}">Edit</a>
```

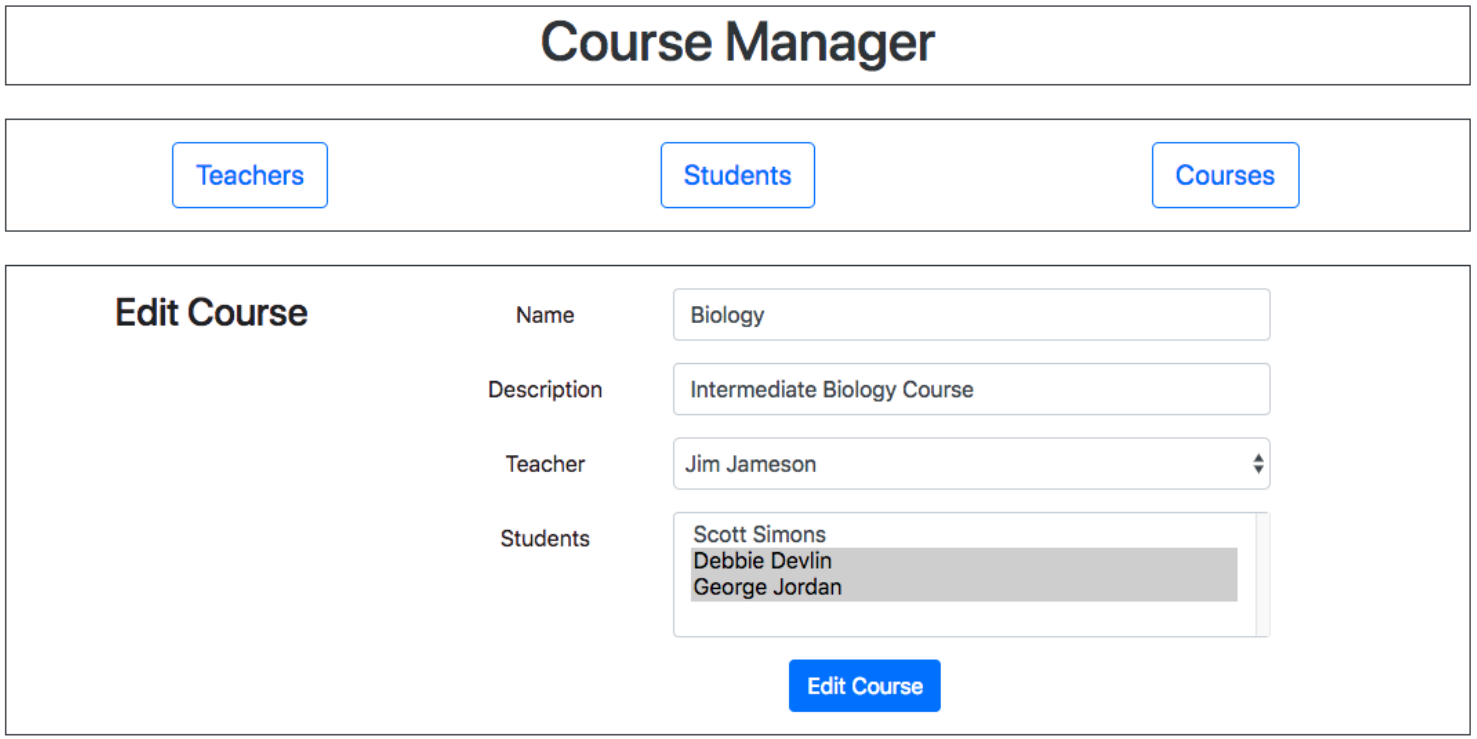
Now we can create the GET mapping to display the Edit Course page:

```
    @GetMapping("editCourse")
    public String editCourse(Integer id, Model model) {
        Course course = courseDao.getCourseById(id);
        List<Student> students = studentDao.getAllStudents();
        List<Teacher> teachers = teacherDao.getAllTeachers();
        model.addAttribute("course", course);
        model.addAttribute("students", students);
        model.addAttribute("teachers", teachers);
        return "editCourse";
    }
```

- We start with the @GetMapping for editCourse.
- Similar to the Course Details page, we take in both the ID and the Model.
- We get our Course as well as the lists of Students and Teachers so we can display all of them for the Edit.
- We put those three variables into the Model.
- Finally, we return editCourse, taking us to the editCourse.html template we are about to create.

Similar to the Add Course form, our Edit Course form will require all the Teachers and all the Students so we can potentially change both. Our Edit Course page will let us choose a new Teacher and change the Students that are selected.

Here is what the Edit Course page will look like:



And here is the main body of the editCourse.html template that you should create from the previous template:

```
        <div class="container">
            <div class="row m-4">
                <div class="col text-center border border-dark">
                    <h1><a href="/" class="text-dark">Course Manager</a></h1>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col text-center m-3">
                    <a href="teachers" class="btn btn-outline-primary btn-lg">Teachers</a>
                </div>
                <div class="col text-center m-3">
                    <a href="students" class="btn btn-outline-primary btn-lg">Students</a>
                </div>
                <div class="col text-center m-3">
                    <a href="courses" class="btn btn-outline-primary btn-lg">Courses</a>
                </div>
            </div>
            <div class="row m-4 border border-dark">
                <div class="col-3 text-center m-3">
                    <span class="h3">Edit Course</span>
                </div>
                <div class="col-7 text-center m-3">
                    <form action="editCourse" method="POST">
                        <div class="form-group row">
                            <label for="name" class="col-3 col-form-label">Name</label>
                            <div class="col-9">
                                <input type="text" name="name" id="name" class="form-control"
                                       th:value="${course.name}"/>
                            </div>
```

```
</div>
<div class="form-group row">
    <label for="description" class="col-3 col-form-label">
        Description</label>
    <div class="col-9">
        <input type="text" name="description"
               id="description" class="form-control"
               th:value="${course.description}"/>
    </div>
</div>
<div class="form-group row">
    <label for="teacher" class="col-3 col-form-label">Teacher</label>
    <div class="col-9">
        <select id="teacher" name="teacherId" class="form-control" >
            <option th:each="teacher : ${teachers}"
                    th:value="${teacher.id}"
                    th:text="${teacher.firstName + ' '
                        + teacher.lastName}"
                    th:selected="${course.teacher.id == teacher.id}">
                Teacher Name</option>
        </select>
    </div>
</div>
<div class="form-group row">
    <label for="students" class="col-3 col-form-label">Students</label>
    <div class="col-9">
        <select multiple id="students" name="studentId"
                class="form-control">
            <option th:each="student : ${students}"
```

```
                                    th:value="${student.id}"
                                    th:text="${student.firstName + ' '
                                        + student.lastName}"
                                    th:selected="${course.students.contains(student)}">
                                Student Name</option>
                        </select>
                    </div>
                </div>
                <input type="hidden" name="id" th:value="${course.id}"/>
                <button type="submit" class="btn btn-primary">Edit Course</button>
            </form>
        </div>
    </div>
</div>
```

- We have our normal navigation at the top.
- We then start the editCourse POST form.
- We can display our Course name and description normally.
- With Teacher, we display the full list of teachers, but we add: th:selected="${course.teacher.id == teacher.id}. This adds the 'selected' as true or false, depending on whether the current Teacher matches the course Teacher. This should make the current Teacher show up as the selected option in the drop-down.
- Next, we display the full list of Students in a multi-select box and add: th:selected="${course.students.contains(student)}. This checks if the current Student is in the list of Students in the Course. If it is, the Student will be highlighted on the page.
- At the end, we add the hidden input with the Course ID so we can properly identify which Course to update. We finally put in our submit button to send the data to the back-end.

The big difference is with editing the relationships. Here, we aren't just displaying text data, so we have to make sure to correctly select which item in the drop-down or items in the multi-select are already a part of the Course. The data on this page will be sent as-is to update the Course, so any change made will go through.

Let's see how we handle it on the back-end. It ends up looking a lot like our Add Course method:

```
@PostMapping("editCourse")
public String performEditCourse(Course course, HttpServletRequest request) {
    String teacherId = request.getParameter("teacherId");
    String[] studentIds = request.getParameterValues("studentId");

    course.setTeacher(teacherDao.getTeacherById(Integer.parseInt(teacherId)));

    List<Student> students = new ArrayList<>();
    for(String studentId : studentIds) {
        students.add(studentDao.getStudentById(Integer.parseInt(studentId)));
    }
    course.setStudents(students);
    courseDao.updateCourse(course);

    return "redirect:/courses";
}
```

- We make our @PostMapping for editCourse.
- We pull in a Course object for the simple data (ID, name, description) and the HttpServletRequest for the more complicated data (Teacher, Students).
- We get our teacherId and array of studentIds out of the HttpServletRequest.
- We first set the Teacher using the teacherId to retrieve the Teacher from its DAO.
- We loop through the studentIds and fill a list of Students that we retrieve from the Student DAO.
- We then set our Students with the list we filled in the loop.
- Next, we call our updateCourse method in the Course DAO.
- Finally, we redirect back to the main Courses page.

Once again, the main thing to remember is how we can use the HttpServletRequest to get the IDs out of the form. As long as we have an ID, we can get the full object to add it to the Course.

Step 12: Summary

Summary

Our Course Manager application is now complete. You should have a fully-built DAO with solid tests. You should be able to view, add, update, and remove Teachers, Students, and Courses from the front-end. The complex relationships between the different objects should also be represented and maintained correctly throughout the application. While building this application, you should have noticed that there are patterns to how the various operations should be coded. Recognizing those patterns can help you develop code like this more quickly.

At this point, we don’t have any way to verify that the data being entered is valid, so that will be the focus of the next lesson, Input Validation.