Adding Spring Data JPA Support

In this exercise, we will enhance our existing REST API by integrating JPA (Java Persistence API) to manage data persistence. We will create entities, repositories, and services to handle CRUD operations for our resources.

Step 1: Add JPA Dependencies

1. Open your existing Spring Boot project.

2. Add the following dependencies to your `pom.xml` file:

   - `Spring Data JPA`
   - `H2 Database`
   - `MySQL Driver`

   ```
   <dependency>
       <!-- OTHER DEPENDENCIES -->

       <!-- Spring Data JPA -->
      <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-data-jpa</artifactId>
       </dependency>
       <!-- H2 Database (for in-memory db) -->
       <dependency>
           <groupId>com.h2database</groupId>
           <artifactId>h2</artifactId>
           <scope>runtime</scope>
       </dependency>
       <!-- MySQL Driver -->
       <dependency>
           <groupId>com.mysql</groupId>
           <artifactId>mysql-connector-j</artifactId>
           <scope>runtime</scope>
       </dependency>
   </dependencies>
   ```

   3. Reload your `pom.xml` to download the new dependencies.

Step 2: Configuring In-memory (H2) Database Connection

1. Open `src/main/resources/application.properties` and add the following configuration for the H2 database:

   ```
   # H2 Database Configuration
   spring.datasource.url=jdbc:h2:mem:testdb
   ```

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

# JPA Hibernate Configuration
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

Step 3: Make the `Course` class a JPA Entity

1. Open the `Course` class you created in the previous exercise.
2. Annotate the class with `@Entity` (makes it a JPA entity).
3. Add the `@Id` annotation to the `id` field and use `@GeneratedValue(strategy = GenerationType.IDENTITY)` to auto-generate the ID.
4. Ensure you have a no-args constructor (required by JPA).

```java
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // Other fields, constructors, getters, and setters
}
```

Step 4: Create a JPA Repository

1. Create an interface named `CourseRepository` in the `repository` package that extends `JpaRepository<Course, Long>`.

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

2. Add a derived query method to find courses by name:

   ```java
   List<Course> findByNameContainingIgnoreCase(String name);
   ```

Step 5: Create a Service Layer

1. Create a class named `CourseService` in the `service` package.

   - We use constructor injection to inject the `CourseRepository`.

- Each method in the service layer corresponds to a CRUD operation and interacts with the repository to perform database operations.
- `Optional` is used to handle the possibility of a course not being found.

```java
@Service
public class CourseService {
    private final CourseRepository courseRepository;

    public CourseService(CourseRepository courseRepository) {
        this.courseRepository = courseRepository;
    }

    public List<Course> getAllCourses(String name) {
        // TODO: Implement filtering by name
        return courseRepository.findAll();
    }

    public Course getCourseById(Long id) {
        Optional<Course> course = courseRepository.findById(id);
        if (course.isPresent()) {
            return course.get();
        }
        throw new RuntimeException("Course not found with id: " + id);
    }

    public Course createCourse(Course course) {
        course.setId(null); // Ensure the ID is null for new entities
        return courseRepository.save(course);
    }

    public Course updateCourse(Long id, Course courseDetails) {
        Course course = this.getCourseById(id);
        course.setName(courseDetails.getName());
        course.setDescription(courseDetails.getDescription());
        course.setActive(courseDetails.isActive());
        course.setStartDate(courseDetails.getStartDate());
        return courseRepository.save(course);
    }

    public void deleteCourse(Long id) {
        Course course = this.getCourseById(id);
        if (course == null) {
            throw new RuntimeException("Course not found with id: " +
id);
        }
        courseRepository.delete(course);
    }
}
```

Step 6: Update the Controller to Use the Service

1. Modify the `CourseController` to use the `CourseService` instead of managing the `courses` list directly.
   - We use constructor injection to inject the `CourseService`.
   - Each controller method calls the corresponding service method to perform the required operation.
   - Here is the updated `CourseController`:

```
@RestController
@RequestMapping("/api/courses")
public class CourseController {
    private final CourseService courseService;

    public CourseController(CourseService courseService) {
        this.courseService = courseService;
    }

    @GetMapping
    public ResponseEntity<List<Course>>
getAllCourses(@RequestParam(required = false) String name) {
        List<Course> courses = courseService.getAllCourses(name);
        return ResponseEntity.ok(courses);
    }

    // TODO: Other controller methods updated similarly
}
```

2. Use try-catch blocks to handle exceptions and return appropriate HTTP status codes (e.g., `404 Not Found` for not found resources).

Step 7: Adding sample data

1. Create a new class named `InitData` in the main package (e.g., `dk.ek.<YOUR_STUDENT_ID>.courseapi`).
   - This class will be responsible for initializing the database with some sample data.
   - The `CommandLineRunner` interface allows us to run code at application startup.

```
@Component
public class InitData implements CommandLineRunner {
    private final CourseRepository courseRepository;
    public InitData(CourseRepository courseRepository) {
        this.courseRepository = courseRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        Course course1 = new Course(null, "Java Basics", "Learn the
basics of Java", true, LocalDate.now());
        Course course2 = new Course(null, "Spring Boot", "Introduction
to Spring Boot", true, LocalDate.now());
```

```
            courseRepository.save(course1);
            courseRepository.save(course2);
        }
    }
```

Step 8: Testing the endpoints

1. Run the Spring Boot application.
2. Use **curl** to test the API endpoints:

- Get all courses:

```
curl http://localhost:8080/api/courses
```

- Get a course by ID:

```
curl http://localhost:8080/api/courses/1
```

- Create a new course:

```
curl -X POST http://localhost:8080/api/courses -H "Content-Type:
application/json" -d '{"name":"New Course","description":"Course
Description","active":true,"startDate":"2023-10-01"}'
```

- Update a course:

```
curl -X PUT http://localhost:8080/api/courses/1 -H "Content-Type:
application/json" -d '{"name":"Updated Course","description":"Updated
Description","active":false,"startDate":"2023-10-01"}'
```

- Delete a course:

```
curl -X DELETE http://localhost:8080/api/courses/1
```

Step 9: Using MySQL Database

1. If you want to switch to a MySQL database, update the `application.properties` file with the following configuration:

```
# H2 Database Configuration
#spring.datasource.url=jdbc:h2:mem:testdb
#spring.datasource.driverClassName=org.h2.Driver
#spring.datasource.username=sa
#spring.datasource.password=
#spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
#spring.h2.console.enabled=true
#spring.jpa.hibernate.ddl-auto=create
#spring.jpa.show-sql=true

# MySQL Configuration (Commented out for H2)
spring.datasource.url=jdbc:mysql://localhost:3306/courses_db
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

2. Create a database named `courses_db` - open a terminal and run:

   ```
   mysql -u root -p
   ```

   - `root` is the MySQL root user.
3. Enter your MySQL root password when prompted (if you haven't set one, just press Enter).
4. Create the database by running:

   ```
   CREATE DATABASE courses_db;
   ```

5. While still connected to MySQL, run:

   ```
   CREATE USER 'yourusername'@'localhost' IDENTIFIED BY 'yourpassword';
   GRANT ALL PRIVILEGES ON courses_db.* TO 'yourusername'@'localhost';
   FLUSH PRIVILEGES;
   EXIT;
   ```

6. Then use these credentials in your configuration:

   ```
   spring.datasource.url=jdbc:mysql://localhost:3306/courses_db
   spring.datasource.username=yourusername
   spring.datasource.password=yourpassword
   spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
   spring.jpa.hibernate.ddl-auto=create
   spring.jpa.show-sql=true
   ```

7. Restart your Spring Boot application to connect to the MySQL database.

Step 10: Verify data in MySQL

1. Open a terminal and log in to MySQL:

   ```
   mysql -u yourusername -p
   ```

2. Switch to the `courses_db` database by running (*remember to end with a `;`*):

   ```sql
   USE courses_db;
   ```

3. Verify that the `course` table has been created and contains the sample data:

   ```sql
   SHOW TABLES;
   ```

4. Query the `course` table to see the data:

   ```sql
   SELECT * FROM course;
   ```

5. Exit MySQL by typing:

   ```
   exit
   ```