Exercises for IDATA2306 Application Development: Week #10

All the exercises this week will extend the library application from previous weeks.

You can either work with your own project, or extend this one: https://github.com/strazdinsg/app-dev/tree/main/exercise-solutions/ex05.4-crudrest

The main idea of this week is to appreciate the advantages of using Object-Relational Mapping (ORM) and Java Persistence API (API), instead of raw SQL access with JDBC.

Alternatives: if you use language and/or framework other than Java Spring Boot, find the proper way to do corresponding things there!

EXERCISE 10.1 – SET UP LOGGING FOR THE LIBRARY PROJECT, FUNDAMENTAL [0.5H]

Experiment with logging for a Spring Boot project. You can use any project really, for example, the library project (used in all other exercises this week).

The task: create at least one instance of "proper logging", using the Logback library.

1. You don't need to add any other libraries to your project, Logback is included in all Spring Boot applications by default.
2. Create instance of a logger. For example, if you do that inside BookController class, define field
   ```
   private static final Logger logger =
         LoggerFactory.getLogger(BookController.class);
   ```
   Hint: there are several options, but you probably want to import the `Logger` and `LoggerFactory` classes from the `org.slf4j` package.
3. Now print a log message. For example, inside the `HTTP GET /books` handler method you can have `logger.info("Getting all books");`

Exercise 10.2 – Set up MySQL connection with Spring-data-JPA, Fundamental [0.5h]

Last week we looked at setting up a connection to a SQL database in the "raw way", using JDBC. The task in this exercise is to do it differently, in the "Spring Data JPA way".

Alternatives: if you use language and/or framework other than Java Spring Boot, find the proper way to set up a database connection there!

Steps:

1. Import the spring-boot-starter-data-jpa dependency:

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

2. Add the MySQL connector dependency (if your project does not already have it):

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

3. If you use a database other than MySQL – find out which dependencies need to import for your database!
4. Now you need to configure the database access parameters in the file `src/main/resources/application.properties`:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=yourSqlUsername

spring.datasource.password=yourSqlUserPassword

# replace localhost with the database host, 3306 with port number, library
with the database name you use

spring.datasource.url=jdbc:mysql://localhost:3306/library


# If you want to see the SQL queries executed by the framework, uncomment the
next line:

# spring.jpa.show-sql: true
```

It is not good to store usernames and passwords in the source code. We will solve this issue later. For now – just use a password which you don't use in other systems!

5. Run the application.

6. In case you get an error: *"Unable to create a Configuration, because no Jakarta Bean Validation provider could be found. Add a provider like Hibernate Validator (RI) to your classpath."* You can solve this by adding the following dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId> spring-boot-starter-validation</artifactId>
</dependency>
```

   This ensures that hibernate-validator package is added (which is necessary for Spring Data JPA).

If everything went according to the plan, you should have a connection to your SQL database established. If you don't get any error messages when you run the application, you are good to go. We have not written any database logic yet. We will do that in the next exercises.

EXERCISE 10.3 – ENTITY CLASSES AND AUTO-GENERATED SQL TABLES [FUNDAMENTAL, 0.5H]

The task of this exercise is to mark your data-model classes (`Book` and `Author`) with necessary annotations so that Spring Data JPA understands that these classes should be stored in SQL (as tables). Spring Data JPA can then auto-generate the necessary tables. It will also be able to automatically convert a `Book` instance to a row in a book-table, etc.

Steps:

1. You must tell Spring Data JPA which classes describe your data model. These will be called "entity classes" in JPA terminology. To mark a class as an entity, add `@Entity` annotation to it (import it from `jakarta.persistence` package). Add the `@Entity` annotation to both `Book` and `Author` classes.
2. In case your `Book` an `Author` are records, change them to classes.
3. In case you don't have a default constructor for the entity classes, the IDE will show you an error. Create a default constructor for `Book` and `Author`. Example: `public Book() {}`
4. You will also get an error "*Add Id attribute*". Each entity-class must have one field, which is unique and can be used as a primary key in the SQL tables. In our case both `Book` and `Author` have an `id` field. Annotate the `id` field with `@Id` annotation. This will tell Spring Data JPA, that this field can be used as a primary key.
5. Also, you probably don't want to bother to generate unique ID values yourself. It is better to let Spring do it for you. To instruct Spring to auto-generate unique ID values, add an `@GeneratedValue` annotation to `id` field in both classes. By default, this will generate extra tables in your MySQL database holding the next value for the ID fields. To use the built-in AUTO_INCREMENT feature of MySQL, extend the annotation as follows: `@GeneratedValue(strategy = GenerationType.IDENTITY)`
6. For a reference, your `Book` class might look something like this now (note: the @Schema annotations are for Swagger documentation, from the previous exercises):

```
@Schema(description = "A book in our library", title = "Ei bok")
@Entity
public class Book {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  private String title;

  @Schema(description = "The year of publication, for example, 1984")
  private int year;

  private int numberOfPages;

  public Book() { }

  public Book(int id, String title, int year, int numberOfPages) {
    this.id = id;
    this.title = title;
    this.year = year;
    this.numberOfPages = numberOfPages;
  }
```

7. You should also tell Spring that you want the SQL tables to be checked and updated every time you restart the application. To do that, add the following line to the file `src/main/resources/application.properties`:

   ```
   spring.jpa.hibernate.ddl-auto=update
   ```

   This will tell Spring JPA to auto-generate SQL tables according to your entity-class structure. Also, if you edit the classes and run the application again, Spring Data JPA will compare your classes with current SQL table structure. If the SQL tables are outdated, it will automatically update SQL table structure so that it matches your classes in the Java code. Isn't that cool?
8. Run the application. It should run smoothly, without errors.
9. Go to your database browsing tool and observe the tables in your database. You should have (at least) 2 tables: `book`, and `author`. Look at the table structure. The `book` and `author` tables will store your book and author objects (while `book_seq` and `author_seq` store next value for the auto-generated `id` fields, in case you don't use the IDENTITY id strategy). This is Spring Data JPAs way to ensure unique IDs (AUTO_INCREMENT fields may not be supported by all SQL databases; Spring ensures that there is a way which will work independently of SQL database choice).

Now you have marked your entity classes with necessary annotations (no relations yet). Let's go to the next exercise!

EXERCISE 10.4 – DATABASE ACCESS WITH REPOSITORY CLASSES [FUNDAMENTAL, 1H]

In this exercise we will finally implement the data storage and retrieval in SQL.

Steps:

1. Create a `BookRepository` interface, let it extend interface `CrudRepository<Book, Integer>` . Import the `CrudRepository` interface from the package `org.springframework.data.repository`.
   What does this mean? We are saying that our `BookRepository` will be an SQL repository which will implement CRUD operations on the SQL database, and the mapping between the book table and `Book` class. We also say that the primary key for `Book` objects is of type integer. Behind the scenes Spring Data JPA will be able to analyze the `Book` class and auto-generate an anonymous class which implements the `BookRepository` interface and provides many useful methods for us, for example:
   a. `findById` – select one row in book table by ID, convert it to `Book` object
   b. `findAll` – select all books from the book table
   c. `save` – insert a book object in the book table (or update it if one exists already)
   d. `delete` – delete a book

   Spring Data JPA can do more. We can define some functions inside the interface and Spring will understand what we mean and auto-generate code for it. For example, we could define a method with the following signature inside BookRepository:
   ```
   Iterable<Book> findByYearGreaterThan(int minYear);
   ```
   This method will select all the books having publication year greater than the given `minYear` value. I.e., the following SQL query will be executed to select the necessary books from the database: `SELECT * FROM book WHERE year > minYear`
   Let's use these methods now in the following steps:
2. Implement `BookController.getAll()` method. It must call `bookRepository.findAll()`. This will return `Iterable<Book>` which is an acceptable return value for our controller method.
3. You can delete the else statement in the `BookController.getAll()` method, because `bookRepository.findAll()` never returns null.
4. Now implement `BookController.getOne()`. It must call `bookRepository.findById()`. This will return `Optional<Book>`. To check if a book was found, call `.isPresent()` method for the returned optional value.
5. Now implement the rest of the methods for the book controller: adding, deleting and updating books in the database. Methods which may be handy in the `bookRepository`: `save` and `delete`.
6. Run the application. It should work without errors.
7. You should be also able to run all the Postman tests and they should succeed. Perhaps one test will fail – the one expecting a positive number of books in the database.
8. Insert a book or two manually to have some data in the database by default.
9. Insert some authors manually as well.

See how much code we had to write with plain JDBC database access, and how little we write now, with `CrudRepository` interface and Spring Data JPA auto-generated code? Nice, right?

EXERCISE 10.5 – MANY-TO-ONE RELATIONSHIPS WITH JPA [FUNDAMENTAL, 0.5H]

Now let's connect the books and authors together. A database is not a real relational database unless there are some relations between the tables.

1. Let's start small. Let's assume that each book can have only one author. What would this mean if we think about classes? How about adding a field `author` of type `Author` inside the `Book` class:

```
class Book {
  // …
  private Author author;
}
```

2. You probably get an error, something like "Basic attribute type should not be 'Persistence entity'". You need to explicitly tell JPA that this is a Many-to-one relationship. Fortunately, it is relatively simple to do it, just add the `@ManyToOne` annotation to the field. It should look like this now:

```
class Book {
  // …
  @ManyToOne
  private Author author;
}
```

3. Add the necessary getters and setters.
4. Run the application.
5. Check your database in DBeaver. You should see an `author_id` column inside the `book` table?
6. Insert some data manually in the database: some authors, some books, assign authors to books.
7. Send an HTTP GET request to the `/books` endpoint (using Postman). You should see the books and the corresponding author objects.

Exercise 10.6 - Many-to-many relationship modelling with JPA [Fundamental, 0.5h]

One book having a single author - that is, of course, unrealistic. Each book can have several authors. And each author can have authored many books. Which means that we want to implement a many-to-many (M:N) relationship between books and authors. This part is a bit tricky in JPA. Doable, but tricky. Let's do this once to avoid some surprises later in your project!

Why is it tricky? Well, why is it tricky in SQL? Because you need a junction table (AKA join table) to describe an M:N relationship. For books and authors you would store something like author_book, with two foreign keys author_id and book_id pointing to author.id and book.id respectively. We need to describe this with annotations in a way understandable to Spring Data JPA.

Steps for creating the book-author relationship:

1. Remove the `author` field from Book class and the corresponding getter and setter.
2. Create a field `` `authors` `` inside the `Book` class:
   `private final Set<Author> authors = new HashSet<>();`
   It can also be of other types, such as List. But set may be better if we want to check if the authors contain a specific author.
3. Add `@ManyToMany` annotation to this field, with the following syntax:
   ```
   @ManyToMany(mappedBy = "books")
   private Set<Author> authors = new HashSet<>();
   ```
4. Create getter and setter methods for the authors (your IDE can probably auto-generate those). These methods are needed for JPA to work correctly.
5. Now we also need to define the relationship from the "author's perspective" inside the `Author` class. Add a field `` `books` `` to the `Author`, it will also be a set.
6. Mark it with `@ManyToMany` annotation and `@JoinTable` annotation with the following syntax:
   ```
   @ManyToMany
   @JoinTable(
       name="author_book",
       joinColumns = @JoinColumn(name="author_id"),
       inverseJoinColumns = @JoinColumn(name = "book_id")
   )
   private Set<Book> books = new HashSet<>();
   ```
   This gives extra information to JPA about the junction table (join table).
7. Create getter and setter for it.
8. Re-run your application. Go to your database browser (DBeaver) and refresh the tables. Do you see author_book junction table?
9. Insert some data in your database manually (through DBeaver).
10. Now try to access `HTTP GET /books` . What happens? Did you get an error message? Can you guess what is happening? Try to understand it before you look for hints and a solution on the next page!

Let's think about the last problem for a while. What happens when you have a book that has an author? The `Book` object has a reference to the `Author` object and vice versa. What happens when you try to get a JSON representation of the book? JSON encoder sees reference to the `Author` object and decides – OK, I will recursively parse the `Author` object and include its value in the book object. Then it starts to parse the `Author` object and sees "Wow, there is a reference to a `Book` object. Nice, I will parse it recursively." And the party goes on. What happens when you call a function recursively all too many times? Have you heard about the term "stack overflow"? ;)

There is one simple solution to this problem: add `@JsonIgnore` annotation to the `books` field inside the `Author` class. This effectively says: "ignore the `books` value when you convert an `Author` object to JSON".

When using `@JsonIgnore` you need to always think about which end of the relation will you apply it? In this case the dilemma is – should we ignore books for an author or authors for a book. In this business case it is probably more natural to say that when we select a book we are also interested in the authors, but not necessarily in the opposite direction. But you need to always make this decision "manually", Spring can't do it for you.

EXERCISE 10.7 – EXTRA CHALLENGE – ADD TAGS [OPTIONAL, 1H]

Extra challenge – remember the "tags" idea we discussed last week? Each book can have several attached tags which are plain "strings". Implement this using JPA. No instruction, here you are on your own, commander!

PROJECT ACTIVITIES

You have sketched the structure of your database. Start creating classes which represent this structure and start implementing database connectivity for your project. For example:

1. Connect your Spring application to a MySQL database (or other SQL database of your choice).
2. Annotate the model classes (the data) with necessary annotations.
3. Run the application, see that SQL tables are created.
4. Create necessary repository classes.
5. Start creating some REST API controllers (either those you will really need, or just some examples for testing).
6. Connect the controllers with your repositories – make them call repository methods to get the necessary data.
7. In cases where you have some more business logic (not just adding or inserting data), you may want to create a service-class which is a middle-man between the controllers and repositories. Move the logic there if necessary.