Creating Enterprise Reactive Applications

Part One

Overview

In this lab you'll implement a reactive application that reads and writes data in a relational database. For simplicity, you'll use an in-memory H2 database that is created and destroyed automatically every time your Spring application starts and ends.

In a "traditional" Java application, you'd use JDBC to access the relational database. However, JDBC is synchronous; the only way to achieve asynchrony is by spinning up more threads from the thread pool, and these are a finite resource that don't scale up well (especially if you're using Docker containerization).

In recent times, the Spring Data project has added reactive support for NoSQL databases such as Mongo, Redis, and ElasticSearch. Until recently, Spring Data didn't have any reactive support for RDBMS, due to the blocking nature of JDBC and JPA. That's starting to change now, with the advent of the first stable release of R2DBC – Reactive Relational Database Connectivity.

R2DBC is a specification initiative that declares a reactive API to be implemented by driver vendors to access their databases. The Spring Data project supports R2DBC via Spring Data Repository interfaces in the **`org.springframework.data.repository.reactive`** package:

- **`ReactiveCrudRepository<T,ID>`** – Reactive API for RDBMS
- **`ReactiveSortingRepository<T,ID>`** – Extends the above, with sorting

Have a look at the JavaDocs for these two interfaces online. They are very similar to the **`ReactiveMongoRepository<T,ID>`** interface, as discussed during the chapter. They define a dozen-or-so core CRUD operations to create, read, update, and delete data in a RDMBS. All of these methods are reactive, i.e., they return a **`Mono<T>`** or a **`Flux<T>`**.

R2DBC is a large technology, similar in scope and capability to JDBC but in a reactive kinda way. We'll explore the key concept during this lab, focusing on how to use it effectively in a Spring Boot application. For full details about how to use R2DBC in Spring Boot, and the R2DBC technology itself, see:

https://docs.spring.io/spring-data/r2dbc/docs/current-SNAPSHOT/reference/html/#reference

https://r2dbc.io/

Note: In the next lab (after this one) you'll also implement a web front end for the application. Specifically, you'll implement a reactive REST service using Spring WebFlux. We'll give full details in the next lab doc.

IntelliJ starter project

You will create a new starter project during the lab. See Exercise 1 for details 😊.

IntelliJ solution project

The solution project for this lab is located here:

- **`solutions\solution-webflux1`**

Roadmap

There are 7 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Creating a project in IntelliJ for the reactive application
2. Defining a data class for object-relational mapping
3. Defining a Spring Data reactive repository interface
4. Seeding the database
5. Defining a reactive service layer
6. Testing the reactive service layer
7. (If time permits) Additional suggestions

Exercise 1: Creating a project in IntelliJ for the reactive application

In IntelliJ, create a new Spring Boot project named **`student-webflux`**. Add the following Spring Boot dependencies:

- Developer Tools: Lombok
- Web : Spring Reactive Web, ready for the next lab (after this one)
- SQL: Spring Data R2DBC
- SQL: H2 Database

Take a look at the pom file and note the following dependencies in particular:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Exercise 2: Defining a data class for object-relational mapping

The Spring Boot Data project for R2DBC enables you to define a data class that maps to a table in the database. This is similar to defining entity classes in JPA. It doesn't really matter what kind of data we work with in this lab, so for simplicity let's work with employees. Define an **`Employee`** class to hold the following info about an employee:

- Unique id (**`Long`**) – this will map to the primary key (PK) in the database table
- Name
- Salary
- Region where the employee works

The only special thing you have to do in the data class is to annotate the PK field with **`@Id`**.

Exercise 3: Defining a Spring Data reactive repository interface

Define a Spring Data reactive repository interface named **`EmployeeRepository`**, to enable the application to create, read, update, and delete employees in the database. The interface must extend **`ReactiveCrudRepository<Employee,Long>`**.

You can declare additional query methods inside the interface, such as the following:

```
int countByRegion(String region);
List<Employee> findByRegion(String region);
```

Exercise 4: Seeding the database

Define a class named **`SeedDb`**, to seed the H2 in-memory database every time the application runs. The class will be similar to the **`SeedDb`** class that we showed for MongoDB in the chapter, but with some important differences…

If you've ever worked with Spring Data JPA, you'll know that Spring Data JPA can infer the structure of database tables just by looking at the entity classes, and it will automatically create and drop database tables when the application starts and ends. No such luck with Spring Data R2DBC – it won't create the database tables automatically, you have to do it yourself. Here's how you do it:

- In your **`SeedDb`** class, autowire a **`DatabaseClient`** bean. The **`DatabaseClient`** class is defined in the **`org.springframework.r2dbc.core`** package, and has a **`sql()`** method that allows you to specify the SQL statement to execute against the database (reactively, of course).
- Define an **`onApplicationEvent()`** method in your **`SeedDb`** class, similar to in the chapter. Add the following code to create the **`employee`** table in the database on application startup (where **`client`** is the name of your **`DatabaseClient`** bean):

```
client.sql(
        "create table employee (" +
          "id int auto_increment primary key, " +
          "name varchar, " +
          "salary number, " +
          "region varchar);")
      .then()
      .subscribe(e -> log.info("Created"));
```

- Implement the rest of the **`onApplicationEvent()`** method in a similar way to the code you saw during the chapter for the MongoDB example, to create some sample **`Employee`** objects and insert them into the database via the repository's **`save()`** method. Also (as per the MongoDB example), query all the employees afterwards and display them on the console.

Run your application. All being well, you should see the employees have been successfully inserted into the database. Obviously, the whole database is torn down at the end of the application, but that's just because we're using an H2 in-memory database.

Exercise 5: Defining a reactive service layer

Define a class named **`EmployeeService`** and implement reactive business methods, similar to the **`TxService`** class that we showed for MongoDB during the chapter.

The R2DBC reactive repository interface is very similar to the MongoDB reactive repository interface, so this should be a relatively straightforward exercise. You'll just need to pay special attention to the **`create()`** and **`update()`** methods, because obviously the data needed to create/update an **`Employee`** is different to a **`Tx`**.

Exercise 6: Testing the reactive service layer

Define two test classes named **`EmployeeServiceModificationsTest`** and **`EmployeeServiceQueryTest`**. These test classes should be similar to the corresponding test classes in the MongoDB application, but with some important differences:

- In the MongoDB example, we annotated the test classes with **`@DataMongoTest`** to define a test slice for MongoDB, i.e. to tell Spring Boot to create the beans needed by MongoDB. The equivalent annotation for R2DBC is **`@DataR2dbcTest`**.
- In the MongoDB example, there was no need to define a database schema when running the tests (hence the name, NoSQL). However, in R2DBC the database tables must exist. Furthermore, Spring Data R2DBC doesn't automatically create these tables for you, so you must add code somewhere in your tests to create the **`employees`** table if it doesn't already exist.

Implement a suite of tests in a similar fashion to the MongoDB example. Run each test method as soon as you've written it, to validate your progress as you go.

Exercise 7 (If time permits): Additional suggestions

Take a look at the Spring Data R2DBC documentation here:

https://docs.spring.io/spring-data/r2dbc/docs/current-SNAPSHOT/reference/html/#reference

Have a go at some of the techniques described in the documentation.