Creating Enterprise Reactive Applications

Part Two

Overview

In this lab you'll enhance the "employee management" application that you wrote in the previous lab, to add a reactive REST service to it. You'll implement the reactive REST service in two different ways:

- As a controller class, similar to the way you implement controllers in Spring Web MVC
- As a handler class, with an associated mapping class that maps URLs to handler methods

IntelliJ starter project

If you're happy to continue where you left off in the previous lab, use the following project:

- **`student\student-webflux`**

If you'd prefer a fresh start, you can use the "solution" project from the previous lab:

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

IntelliJ solution project

The solution project for this lab is located here:

- **`solutions\solution-webflux2`**

Roadmap

There are 4 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. Configuring your IntelliJ project to support Spring WebFlux
2. Implementing a reactive REST service as a controller class
3. Implementing a reactive REST service as a handler class
4. (If time permits) Testing the reactive REST services

Exercise 1: Configuring your IntelliJ project to support Spring WebFlux

When you created the "employee management" project in the previous lab, we asked you to include the following Spring Boot dependency:

- Spring Reactive Web

If you did that, you should find the following Maven dependency in your pom file:

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

This is the only Maven dependency you need for Spring WebFlux. Specifically, you *do not* need the Spring Web MVC dependency (Spring Web MVC and Spring WebFlux are completely independent of each other).

Exercise 2: Implementing a reactive REST service as a controller class

Define a new class named **`EmployeeRestController`**, and implement a suite of reactive REST methods to get, post, put, and delete **`Employee`** objects. Your class will be similar in intent and functionality to the **`TxRestController`** class that you saw during the chapter.

Just to make sure you understand how a reactive REST controller class differs from a traditional Spring Web MVC controller class:

- In a reactive REST controller, methods are non-blocking, i.e. they return immediately with reactive responses.
- In a non-reactive REST controller, methods are blocking, i.e. they only return a response when it is completely ready to send.

Run the application and then use a tool such as Postman or Chrome Advanced REST Client to test that the methods return the correct results.

Exercise 3: Implementing a reactive REST service as a handler class

Define a new class named **`EmployeeRestHandler`**, and implement a suite of reactive methods to get, post, put, and delete **`Employee`** objects. Your class will be similar to the **`TxRestHandler`** class that you saw during the chapter.

You will also need to define a "handler configuration" class, which defines a routing table that maps the URLs to methods in your handler class.

Run the application again and test it using Postman or Chrome Advanced REST Client, to ensure all the handler methods are invoked correctly and that they return the expected results.

Exercise 4 (If time permits): Testing the reactive REST services

Write some JUnit tests for your two reactive REST classes. You can follow the examples shown in the chapter for guidance if you like. Feel free to use Mockito to mock any interactions with a real H2 database.

Run your tests, and make sure they all pass.