Creating Enterprise Reactive Apps Part Two

1. Understanding Spring WebFlux
2. Implementing a REST controller endpoint
3. Implementing a REST handler endpoint
4. Testing Spring WebFlux endpoints

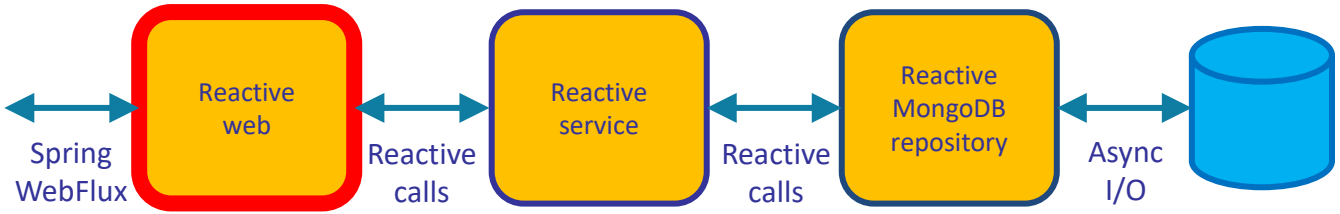
1. Understanding Spring WebFlux

- Overview
- About Spring Web MVC
- About Spring WebFlux

Overview

- In the previous chapter we implemented a reactive data repository and reactive service
  - The reactive service returned a publisher (i.e. `Mono` or `Flux`)

- In this chapter we'll implement a reactive web layer
  - Runs on Spring WebFlux



- We'll also see how to test the Reactive web layer

About Spring Web MVC

- Spring Web MVC was the original web framework in Spring
  - Purpose-built for the Servlet API and Servlet containers
- Spring Web MVC is a blocking API
  - Servlet containers assume calls will block the current thread
  - Servlet containers therefore use a large thread pool, to absorb potential blocking during request handling
- To implement a REST service using Spring Web MVC:
  - Define a blocking MVC controller class with synchronous logic

About Spring WebFlux

- Spring WebFlux was introduced in Spring Framework 5.0
  - Purpose-built for the Reactive Streams API

- Spring WebFlux is a non-blocking API
  - Reactive servers assume calls will not block the current thread
  - Reactive servers therefore use a small, fixed-size large thread pool, to handle the request event loop

- To implement a REST service using Spring WebFlux:
  - Either implement an MVC-like non-blocking controller class
  - Or implement a handler class using a functional programming style

- We'll examine both techniques in the following sections
  - For the full demo code, see the `demo.webflux.rest` package

2. Implementing a REST Controller Endpoint

- Overview
- Defining a reactive controller class
- Implementing reactive REST methods
- Pinging the reactive REST controller

Overview

- You can implement a Spring WebFlux REST endpoint using familiar REST controller techniques and annotations
  - `org.springframework.web.bind.annotation.*`
- E.g. you can use familiar annotations such as:
  - `@Controller`, `@RestController`
  - `@CrossOrigin`, `@RequestMapping`
  - `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`
  - `@PathVariable`, `@RequestParam`, `@RequestBody`, `@ResponseBody`, etc.
- So what's different?
  - Your methods will be called reactively by the server, e.g. Netty
  - So your methods must return a publisher (`Mono` or `Flux`)

Defining a Reactive Controller Class

- To define a reactive controller class, the class definition itself is just like a traditional Spring Web MVC controller

```
@Log4j2
@RestController
@RequestMapping(value="/tx", produces=MediaType.APPLICATION_JSON_VALUE)
@Profile("controller-style-endpoint")
class TxRestController {

    private final TxService service;

    TxRestController(TxService service) {
        this.service = service;
    }

    // Define reactive methods here, using familiar MVC techniques…
}
```

TxRestController.java

- Note the `@Profile` annotation in our example
  - This is because our demo has two REST services…
  - We define a different profile for each, to avoid interference
  - We set the active profile in `application.properties`

Implementing Reactive REST Methods (1 of 2)

- In a reactive REST controller, endpoints must return a publisher
  - So the reactive server can offer all the benefits of reactive APIs…
  - Asynchrony, back pressure, etc.

- Here are some examples of reactive REST methods
  - Their inputs are just like Spring Web MVC REST methods
  - But their outputs are always a publisher

```
@GetMapping
Publisher<Tx> getAll() {
    return this.service.getAll();
}

@GetMapping("/{id}")
Publisher<Tx> getById(@PathVariable("id") String id) {
    return this.service.getById(id);
}
```

TxRestController.java

Implementing Reactive REST Methods (2 of 2)

- Here are some more interesting reactive REST methods
  - Discuss!

```
@PostMapping
Publisher<ResponseEntity<Tx>> create(@RequestBody Tx tx) {
    return this.service
               .create(tx.getAmount())
               .map(t -> ResponseEntity.created(URI.create("/tx/" + t.getId()))
                                       .contentType(MediaType.APPLICATION_JSON)
                                       .build());
}

@PutMapping("/{id}")
Publisher<ResponseEntity<Tx>> update(@PathVariable String id, @RequestBody Tx tx) {
    return this.service
               .update(id, tx.getAmount(), tx.getWhen())
               .map(t -> ResponseEntity.ok()
                                       .contentType(MediaType.APPLICATION_JSON)
                                       .build());
}

@DeleteMapping("/{id}")
Publisher<Tx> delete(@PathVariable String id) {
    return this.service.delete(id);
}
```

TxRestController.java

Pinging the Reactive REST Controller (1 of 2)

- Set the active profile in `application.properties`

```
spring.profiles.active=onlyForDemoPurposes,controller-style-endpoint
```

- Then run the `Application` class
  - The application starts the Netty server, by default

```
INFO 16472 --- [           main] org.mongodb.driver.cluster               : Cluster created with settings {hosts=[localhost:27017], mode=SINGLE, req
INFO 16472 --- [localhost:27017] org.mongodb.driver.connection            : Opened connection [connectionId{localValue:2, serverValue:195}] to local
INFO 16472 --- [localhost:27017] org.mongodb.driver.cluster               : Monitor thread successfully connected to server with description ServerD
INFO 16472 --- [           main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port(s): 8080
INFO 16472 --- [           main] demo.webflux.DemowebfluxApplication      : Started DemowebfluxApplication in 3.344 seconds (JVM running for 4.878)
```

- You can also switch to Tomcat or Jetty
  - Spring WebFlow uses their support for Servlet 3.1 non-blocking I/O
- You can also switch to use Undertow
  - Spring WebFlow uses Undertow APIs directly (not the Servlet API)

Pinging the Reactive REST Controller (2 of 2)

- You can ping the REST controller as normal
  - E.g. browse to `http://localhost:8080/tx`

3. Implementing a REST Handler Endpoint

- Overview
- Defining a reactive handler class
- Accessing HTTP request / response Info
- Implementing reactive REST methods
- Configuring a routing table
- Pinging the reactive REST handler

Overview

- As stated earlier, Spring WebFlux offers two ways to create a reactive REST endpoint…
- As an MVC-like non-blocking controller class
  - Convenient if you want to port a synchronous Spring Web MVC controller to the reactive world
- As a non-blocking handler class
  - Not coupled to `DispatcherServlet` or centralized routing
  - You can define dynamic route tables for agility and flexibility
  - You can leverage lambdas, FP, and DSL programming techniques

Defining a Reactive Handler Class

- You define a reactive handler class as a Spring component, i.e. using `@Component`
  - Not as a controller class via `@Controller`/ `@RestController`

- Example:

```
@Log4j2
@Component
@Profile("handler-style-endpoint")
public class TxRestHandler {

    private final TxService service;

    TxRestHandler(TxService service) {
        this.service = service;
    }

   // Define reactive methods here, using DSL functional programming techniques
   …
}
```

TxRestHandler.java

Accessing HTTP Request / Response Info

- When you define a reactive handler class, it is not hooked up to a `DispatcherServlet`
- No `DisptcherServlet` to detect inbound HTTP info!
  - Instead, your method always receives a `ServerRequest`
  - Gives reactive access to incoming HTTP info
- No `DisptcherServlet` to set outbound HTTP info!
  - Instead, your method always returns a `Mono<ServerRequest>`
  - Publishes reactive result to client
- So you'll need the following imports:

```
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
```

Implementing Reactive REST Methods (1 of 2)

- Here are some examples of reactive REST methods in a Spring WebFlux handler class
  - Use a very contemporary DSL builder programming style

```
Mono<ServerResponse> getAll(ServerRequest req) {

    Flux<Tx> gotten = this.service.getAll();

    return ServerResponse.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(gotten, Tx.class);
}

Mono<ServerResponse> getById(ServerRequest req) {

    String id = req.pathVariable("id");

    Mono<Tx> gotten = this.service.getById(id);

    return ServerResponse.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(gotten, Tx.class);
}
```

TxRestHandler.java

Implementing Reactive REST Methods (2 of 2)

```
Mono<ServerResponse> create(ServerRequest req) {

    Flux<Tx> created = req.bodyToFlux(Tx.class)
                         .flatMap(tx -> this.service.create(tx.getAmount()));

    return Mono.from(created)
               .flatMap(tx -> ServerResponse.created(URI.create("/tx/" + tx.getId()))
                                            .contentType(MediaType.APPLICATION_JSON)
                                            .build());
}

Mono<ServerResponse> update(ServerRequest req) {
    String id = req.pathVariable("id");
    Flux<Tx> updated = req.bodyToFlux(Tx.class)
                         .flatMap(tx -> this.service.update(
                                               id, tx.getAmount(), tx.getWhen()));

    return ServerResponse.ok()
                         .contentType(MediaType.APPLICATION_JSON)
                         .body(updated, Tx.class);
}

Mono<ServerResponse> delete(ServerRequest req) {
    String id = req.pathVariable("id");
    Mono<Tx> deleted = this.service.delete(id);

    return ServerResponse.ok()
                         .contentType(MediaType.APPLICATION_JSON)
                         .body(deleted, Tx.class);
}
```

TxRestHandler.java

Configuring a Routing Table (1 of 2)

- Recall the handler class is just a `@Component`
  - It's not a `@Controller` / `@RestController`
  - So no `DispatcherServlet` to route requests automatically
- Instead you must configure a routing table
  - You must explicitly map URL patterns to handler methods
- This provides a great deal of flexibility
  - You can use sophisticated pattern-matching to map routes
  - You can adjust the routing table dynamically at run time
  - You can implement endpoints as lambdas etc.

Configuring a Routing Table (2 of 2)

- Here's the routing table configuration in our demo

```
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;
import org.springframework.web.reactive.function.server.RouterFunctions;

import static org.springframework.web.reactive.function.server.RequestPredicates.*;
…

@Configuration
@Profile("handler-style-endpoint")
public class TxRestHandlerConfiguration {

    @Bean
    RouterFunction<ServerResponse> routes(TxRestHandler handler) {
        return RouterFunctions.route(GET("/tx"), handler::getAll)
                .andRoute(GET("/tx/{id}"),       handler::getById)
                .andRoute(POST("/tx"),           handler::create)
                .andRoute(PUT("/tx/{id}"),       handler::update)
                .andRoute(DELETE("/tx/{id}"),    handler::delete);
    }
}
```

TxRestHandlerConfiguration.java

Pinging the Reactive REST Handler (1 of 2)

- Set the active profile in `application.properties`

```
spring.profiles.active=onlyForDemoPurposes, handler-style-endpoint
```

- Then run the `DemowebfluxApplication` class
  - The application starts the Netty server, as before
  - Netty supports both controller-based and handler-based styles

```
INFO 16472 --- [           main] org.mongodb.driver.cluster               : Cluster created with settings {hosts=[localhost:27017], mode=SINGLE, req
INFO 16472 --- [localhost:27017] org.mongodb.driver.connection          : Opened connection [connectionId{localValue:2, serverValue:195}] to local
INFO 16472 --- [localhost:27017] org.mongodb.driver.cluster             : Monitor thread successfully connected to server with description ServerD
INFO 16472 --- [           main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port(s): 8080
INFO 16472 --- [           main] demo.webflux.DemowebfluxApplication      : Started DemowebfluxApplication in 3.344 seconds (JVM running for 4.878)
```

Pinging the Reactive REST Handler (2 of 2)

- You can ping the REST controller as before
  - E.g. browse to `http://localhost:8080/tx`

3. Testing Spring WebFlux Endpoints

- Overview
- Test classes in the demo app
- Defining a test superclass
- Injecting test dependencies
- Defining a test method
- Defining test subclasses

Overview

- The `spring-test` module has a `WebTestClient` class
  - You can use this to test Spring WebFlux server endpoints
  - Can be used with or without an actual server
  - Provides a fluent API to build a request and verify a response

- Here are some of its methods to prepare a request:
  - `uri()`, `get()`, `post()`, `put()`, `delete()`
  - `body()`, `contentType()`, `accept()`

- Here are some of its methods to verify a response:
  - `expectStatus()`, `expectHeader()`, `expectBody()`
  - `jsonPath()`

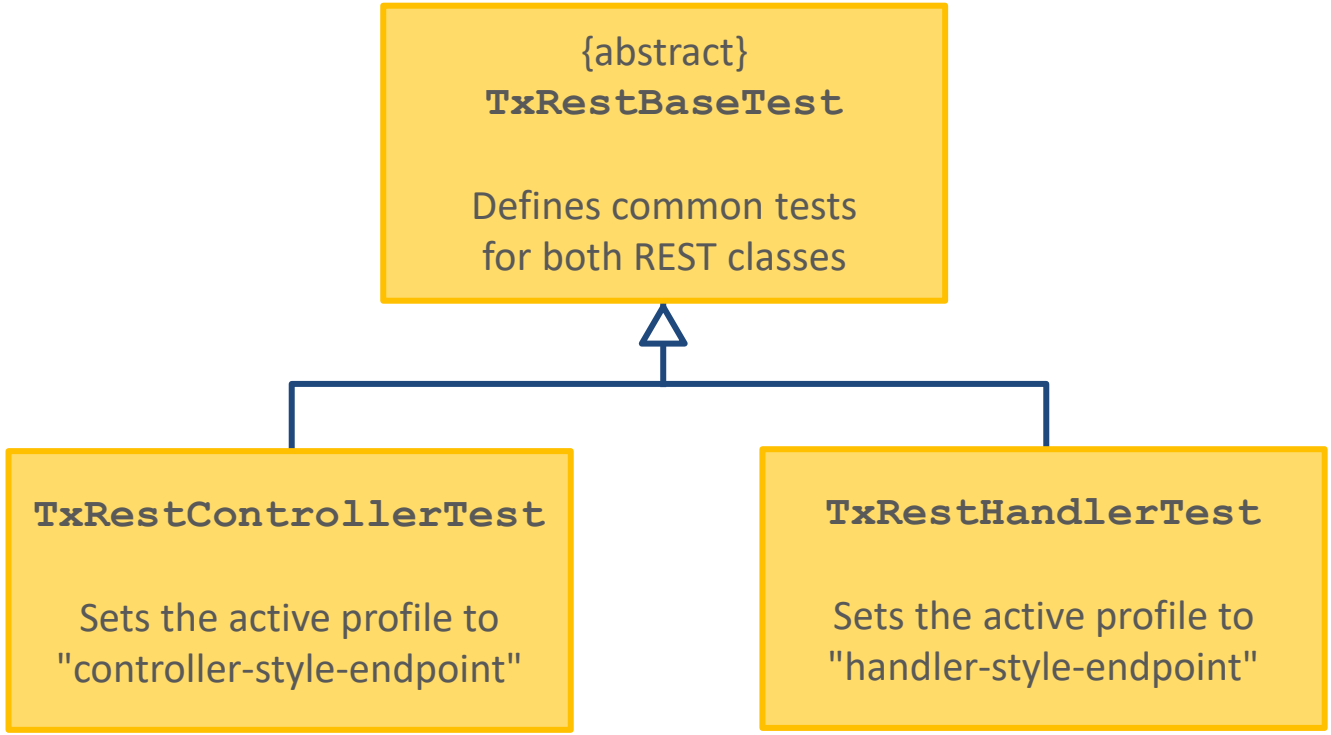
Test Classes in the Demo App (1 of 2)

- The demo app has two separate REST classes:
  - `TxRestController` – Active profile "controller-style-endpoint"
  - `TxRestHandler` – Active profile "handler-style-endpoint"

- These two classes are semantically equivalent, so the tests should be the same too
  - Therefore we've put all the tests in a common superclass
  - We've also defined 2 subclasses that set the correct active profile
  - See following slide…

Test Classes in the Demo App (2 of 2)

- Here's how we've organized our test classes:



Defining a Test Superclass

- Here's the outline of the `TxRestBaseTest` class

```
@Log4j2
@WebFluxTest
public abstract class TxRestBaseTest {
    …
}
```

TxRestBaseTest.java

- `@WebFluxTest` defines a *test slice*, so we only get config relevant to Spring WebFlux tests
  - E.g. `@Controller` (but not `@Service`, `@Repository`, etc.)

Injecting Test Dependencies

- Our test superclass has a couple of dependencies
  - A mock repository (to avoid talking to a real database)
  - A `WebTestClient` (passed in from subclass ctor, see later)

```
@Log4j2
@WebFluxTest
public abstract class TxRestBaseTest {

    @MockBean
    private TxRepository repo;

    private final WebTestClient client;

    public TxRestBaseTest(WebTestClient client) {
        this.client = client;
    }
    …
}
```

TxRestBaseTest.java

Defining a Test Method

- Here's a typical test method, to test the GET handler
  - See demo code for tests for POST, PUT, and DELETE handlers
  - Notice the use of `jsonPath()` to drill into the JSON response

```
@Test
public void getAll() {

    Mockito.when(this.repo.findAll())
           .thenReturn(Flux.just(new Tx("1", 1111, LocalDateTime.now()),
                                 new Tx("2", 2222, LocalDateTime.now())));

    this.client
            .get()
            .uri("/tx")
            .accept(MediaType.APPLICATION_JSON)
            .exchange()
            .expectStatus().isOk()
            .expectHeader().contentType(MediaType.APPLICATION_JSON)
            .expectBody()
            .jsonPath("$.[0].id").isEqualTo("1")
            .jsonPath("$.[0].amount").isEqualTo(1111)
            .jsonPath("$.[1].id").isEqualTo("2")
            .jsonPath("$.[1].amount").isEqualTo(2222);
}
```

TxRestBaseTest.java

Defining Test Subclasses

- Here are the test subclasses to test the REST controller class and the REST handler class, respectively

```
@Log4j2
@ActiveProfiles("controller-style-endpoint")
@Import({TxRestController.class, TxService.class})
public class TxRestControllerTest extends TxRestBaseTest {

    TxRestControllerTest(@Autowired WebTestClient client) {
        super(client);
    }
}
```

TxRestControllerTest.java

```
@Log4j2
@ActiveProfiles("handler-style-endpoint")
@Import({TxRestHandlerConfiguration.class, TxRestHandler.class, TxService.class})
public class TxRestHandlerTest extends TxRestBaseTest {

    TxRestHandlerTest(@Autowired WebTestClient client) {
        super(client);
    }
}
```

TxRestHandlerTest.java

Summary

- Understanding Spring WebFlux
- Implementing a REST controller endpoint
- Implementing a REST handler endpoint
- Testing Spring WebFlux endpoints