Reactive Spring

Ken Kousen
ken.kousen@kousenit.com
2020-08-12

Table of Contents

Exercises

1. Building a REST client
2. Asynchronous Access
3. Reactive Spring Data
4. Spring WebFlux with Annotated Controllers
5. Spring WebFlux with Functional Endpoints

Exercises

1. Building a REST client

This exercise uses the `RestTemplate` class to synchronously access a RESTful web service. The template is used to convert the response into an object for the rest of the system. Later the new Spring 5 `WebClient` class will be used to do the same asynchronously.

1. Create a new Spring Boot project (either by using the Initializr at http://start.spring.io or using your IDE) called `restclient`. Add both the Web and the Reactive Web dependencies.
2. Create a service class called `JokeService` in a `com.nfjs.restclient.services` package under `src/main/java`
3. Add the annotation `@Service` to the class (from the `org.springframework.stereotype` package, so you'll need an `import` statement)
4. Add a private attribute to `JokeService` of type `RestTemplate` called `template`
5. Add a constructor to `JokeService` that takes a single argument of type `RestTemplateBuilder`.

> **NOTE** Because there are so many possible configuration options, Spring does not automatically provide a `RestTemplate`. It does, however, provide a `RestTemplateBuilder`, which can be used to configure and create the `RestTemplate`

6. Inside the constructor, invoke the `build()` method on the `RestTemplateBuilder` and assign the result to the `template` attribute.

> **NOTE** If you provide only a single constructor in a class, you do not need to add the `@Autowired` annotation to it. Spring will inject the arguments anyway

7. The site providing the joke API is http://icndb.com, the Internet Chuck Norris Database. The site exposes the jokes through the URL http://api.icndb.com. The API supports a few properties that will be useful here: the client can specify the hero's first and last names and the joke category.
8. Add a `public` method to the service called `getJokeSync` that takes two `String` arguments, `first` and `last` and returns a `String`
9. Inside the method, create a local variable of type `String` called `base` and assign it to the URL "http://api.icndb.com/jokes/random?limitTo=[nerdy]".

10. Then create the full URL for the API:

```java
String url = String.format("%s&firstName=%s&lastName=%s", base, first, last);
```

11. The `RestTemplate` class has a `getForObject` method that takes two arguments: the URL and the class to instantiate with the resulting JSON response. The documentation on the services says that the resulting JSON takes the form:

```javascript
{
  "type": "success",
  "value": {
    "id": 268,
    "joke": "Time waits for no man, unless that man is Chuck Norris."
    "categories": [
        "nerdy"
    ]
  }
}
```

12. Since there are only two nested JSON objects, you can create two classes that model them. Create the classes `JokeValue` and `JokeResponse` in the `com.nfjs.restclient.json` package.
13. The code for the classes are shown below. Note how the properties match the keys in the JSON response exactly. You can use annotations from the included Jackson 2 JSON parser to customize the attributes if you like, but in this case it's easy enough to make them the same as the JSON variable names.

JAVA

```java
package com.nfjs.restclient.json;

public class JokeResponse {
    private String type;
    private JokeValue value;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public JokeValue getValue() {
        return value;
    }

    public void setValue(JokeValue value) {
        this.value = value;
    }
}

public class JokeValue {
    private int id;
    private String joke;
    private String[] categories;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getJoke() {
        return joke;
    }

    public void setJoke(String joke) {
        this.joke = joke;
    }

    public String[] getCategories() {
        return categories;
    }

    public void setCategories(String[] categories) {
        this.categories = categories;
    }
}
```

14. The JSON response from the web service can now be converted into an instance of the `JokeResponse` class. Add a line to do that inside the `getJoke` method:

```java
return template.getForObject(url, JokeResponse.class)
    .getValue().getJoke();
```

15. To demonstrate how to use the service, create a test for it. Create a class called `JokeServiceTest` in the `com.nfjs.restclient.services` package under the test hierarchy, `src/test/java`.

16. The source for the test is:

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
public class JokeServiceTest {
    @Autowired
    private JokeService service;

    @Test
    public void getJokeSync() {
        String joke = service.getJokeSync("Craig", "Walls");
        assertTrue(joke.contains("Craig") || joke.contains("Walls"));
    }
}
```

17. Inside the test, add the capability to log the jokes to the console. Spring Boot provides loggers from a variety of sources. In this case, use the one from the SLF4J library by adding an attribute to the `JokeServiceTest` class:

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// ...

private Logger logger = LoggerFactory.getLogger(JokeServiceTest.class);
```

18. Use the logger inside the `getJoke` method to log the joke, either before or after the assertion:

JAVA

```
logger.info(joke);
```

19. Feel free to change the name of the hero to anyone you prefer. (Craig Walls is the author of both *Spring in Action* and *Spring Boot in Action*.)
20. Execute the test and make any needed corrections until it passes.

2. Asynchronous Access

The `webflux` module in Spring 5 allows you to use the Project Reactor types `Flux` and `Mono`. Methods that work synchronously can be converted to asynchronous by changing the return type to one of those types. The `WebClient` class then knows how produce those types, and is now the preferred asynchronous rest client (the class `AsyncRestTemplate` is now deprecated).

1. In the `JokeService` class, add an attribute of type `WebClient` and nitialize it to the base URL in the constructor, using the autowired `WebClient.Builder` class.

JAVA

```
private final WebClient client;

@Autowired
public JokeService(WebClient.Builder builder) {
    client = builder.baseUrl("http://api.icndb.com").build();
}
```

2. Now add a new method called `getJokeAsync` that takes two `String` values as arguments called `first` and `last`. For the return type use `Mono<String>` instead of `String`. The implementation is:

JAVA

```
public Mono<String> getJokeAsync(String first, String last) {
    return client.get()
                .uri(uriBuilder -> uriBuilder.path("/jokes/random")
                        .queryParam("limitTo", "[nerdy]")
                        .queryParam("firstName", first)
                        .queryParam("lastName", last)
                        .build())
                .accept(MediaType.APPLICATION_JSON)
                .retrieve()
                .bodyToMono(JokeResponse.class)
                .map(jokeResponse -> jokeResponse.getValue().getJoke());
}
```

3. The `get` method is used to make an HTTP GET request. the `uri` method takes the path and inserts the variables `first` and `last` into it. The `retrieve` method schedules the retrieval. Then the `bodyToMono` method extracts the body from the HTTP response and converts it to an instance of `JokeResponse` and wraps it in a `Mono`. Finally, the `map` method on `Mono` uses a `Function` to convert the contained `JokeResponse` into a `String` and wraps the result back inside a `Mono`, which is returned to the client.

4. To test this, go back to the `JokeServiceTest` class. There are two ways to test the method. One is to invoke it and block until the request is complete. A test to do that is shown here:

JAVA

```java
@Test
public void getJokeAsync() {
    String joke = service.getJokeAsync("Craig", "Walls")
            .block(Duration.ofSeconds(2));
    logger.info(joke);
    assertTrue(joke.contains("Craig") || joke.contains("Walls"));
}
```

5. As an alternative, the Reactor Test project includes a class called `StepVerifier`, which includes assertion methods. A test using that class is given by:

JAVA

```java
@Test
public void useStepVerifier() {
    StepVerifier.create(service.getJokeAsync("Craig", "Walls"))
            .assertNext(joke -> {
                logger.info(joke);
                assertTrue(joke.contains("Craig") || joke.contains("Walls"));
            })
            .verifyComplete();
}
```

6. Both of the new tests should now pass. The details of the `StepVerifier` class will be discussed during the course.

3. Reactive Spring Data

1. Create a new project called `reactive-officers`. Add in the `Spring WebFlux`, `Reactive MongoDB`, and `Embedded MongoDb` dependencies.
2. Add the following `Officer` and `Rank` classes as entities. Put them in the `com.oreilly.reactiveofficers.entities` package. Note that the primary key type on `Officer` is of type `String`.

JAVA

```
public enum Rank {
    ENSIGN, LIEUTENANT, COMMANDER, CAPTAIN, COMMODORE, ADMIRAL
}
```

JAVA

```
public class Officer {
    private String id;
    private Rank rank;
    private String first;
    private String last;

    public Officer() {}

    public Officer(Rank rank, String first, String last) {
        this.rank = rank;
        this.first = first;
        this.last = last;
    }

    public Officer(String id, Rank rank, String first, String last) {
        this.id = id;
        this.rank = rank;
        this.first = first;
        this.last = last;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Rank getRank() {
        return rank;
    }

    public void setRank(Rank rank) {
        this.rank = rank;
    }

    public String getFirst() {
        return first;
    }

    public void setFirst(String first) {
        this.first = first;
    }

    public String getLast() {
        return last;
    }

    public void setLast(String last) {
        this.last = last;
    }
```

```
    @Override
    public String toString() {
        return "Officer{" +
                "id=" + id +
                ", rank=" + rank +
                ", first='" + first + '\'' +
                ", last='" + last + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Officer)) return false;

        Officer officer = (Officer) o;

        if (!id.equals(officer.id)) return false;
        if (rank != officer.rank) return false;
        if (first != null ? !first.equals(officer.first) : officer.first != null)
return false;
        return last.equals(officer.last);
    }

    @Override
    public int hashCode() {
        int result = id.hashCode();
        result = 31 * result + rank.hashCode();
        result = 31 * result + (first != null ? first.hashCode() : 0);
        result = 31 * result + last.hashCode();
        return result;
    }
}
```

3. Annotate `Officer` with `@Document` from `org.springframework.data.mongodb.core.mapping`.
4. Annotate `id` with `@Id` from `org.springframework.data.annotation`. Note that the data type for the `id` is `String`, since we will be using a MongoDB database.
5. Make a Spring Data interface called `OfficerRepository` that extends `ReactiveMongoRepository<Officer, String>` in the `com.nfjs.reactiveofficers.dao` package.

JAVA

```
package com.oreilly.reactiveofficers.dao;

import com.oreilly.reactiveofficers.entities.Officer;
import com.oreilly.reactiveofficers.entities.Rank;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;

public interface OfficerRepository extends ReactiveMongoRepository<Officer, String> {
}
```

6. Create a test for the repository called `OfficerRepositoryTest`. Add the class annotation `@SpringBootTest`.
7. Autowire in an instance of `OfficerRepository` called `repository`.
8. Provide initialization data in the form of a list of officers:

JAVA

```
private List<Officer> officers = Arrays.asList(
        new Officer(Rank.CAPTAIN, "James", "Kirk"),
        new Officer(Rank.CAPTAIN, "Jean-Luc", "Picard"),
        new Officer(Rank.CAPTAIN, "Benjamin", "Sisko"),
        new Officer(Rank.CAPTAIN, "Kathryn", "Janeway"),
        new Officer(Rank.CAPTAIN, "Jonathan", "Archer"));
```

9. Note that the `id` fields will be null or empty until the officers are saved. To save them, add a method called `setUp` that takes no arguments and returns `void`. Annotated it with `@BeforeEach` from JUnit 5. This method will drop the existing collection (if there is one) and recreate it before each test.
10. The body of the `setUp` method is:

JAVA

```
repository.deleteAll()
          .thenMany(Flux.fromIterable(officers))
          .flatMap(repository::save)
          .then()
          .block();
```

11. To test the `save` method, create an officer and save it, then check that the generated `id` is not an empty string:

JAVA

```
@Test
public void save() {
    Officer lorca = new Officer(Rank.CAPTAIN, "Gabriel", "Lorca");
    StepVerifier.create(repository.save(lorca))
            .expectNextMatches(officer -> !officer.getId().equals(""))
            .verifyComplete();
}
```

12. Test `findAll` by checking that there are five officers in the test collection:

JAVA

```
@Test
public void findAll() {
    StepVerifier.create(repository.findAll())
            .expectNextCount(5)
            .verifyComplete();
}
```

+ To check `findById`, get the generated id's from the officer collection, find each one by id, and verify that a single element is returned each time. For invalid id's verify that no elements are emitted.

JAVA

```
@Test
public void findById() {
    officers.stream()
            .map(Officer::getId)
            .forEach(id ->
                    StepVerifier.create(repository.findById(id))
                            .expectNextCount(1)
                            .verifyComplete());
}

@Test
public void findByIdNotExist() {
    StepVerifier.create(repository.findById("xyz"))
            .verifyComplete();
}
```

13. Check the `count` method by again verifying that there are five officers in the sample collection

JAVA

```
@Test
public void count() {
    StepVerifier.create(repository.count())
            .expectNext(5L)
            .verifyComplete();
}
```

14. The tests should all pass.

4. Spring WebFlux with Annotated Controllers

1. For the actual application, you'll need a running instance of MongoDB. The free community server can be downloaded from https://www.mongodb.com/download-center#community . Note that if you are on OSX, you can do an install via Homebrew, and on Linux you can install with yum.
2. Once MongoDB is installed, start it up at a command prompt with the `mongod` command.
3. Assuming you have a running server, initialize a collection with sample data using a `ApplicationRunner` from Spring. To do so, create a class called `OfficerInit` in the `com.oreilly.reactiveofficers` package, which implements the `ApplicationRunner` interface.
4. Add a `@Component` annotation to the class to make sure Spring detects it on a component scan.
5. Add a constructor that takes an argument of type `OfficerRepository` . Add an attribute of that type and save the constructor argument to the attribute. The presence of a single constructor will automatically autowire in the arguments. Implement the required `run` method using the same approach taken in the test:

JAVA

```
@Component
public class OfficerInit implements ApplicationRunner {
    private OfficerRepository dao;

    public OfficerInit(OfficerRepository dao) {
        this.dao = dao;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        dao.deleteAll()
            .thenMany(Flux.just(new Officer(Rank.CAPTAIN, "James", "Kirk"),
                                new Officer(Rank.CAPTAIN, "Jean-Luc", "Picard"),
                                new Officer(Rank.CAPTAIN, "Benjamin", "Sisko"),
                                new Officer(Rank.CAPTAIN, "Kathryn", "Janeway"),
                                new Officer(Rank.CAPTAIN, "Jonathan", "Archer")))
           .flatMap(dao::save)
           .thenMany(dao.findAll())
           .subscribe(System.out::println);
    }
}
```

6. Add a rest controller:

JAVA

```java
package com.oreilly.reactiveofficers.controller;

import com.nfjs.reactiveofficers.dao.OfficerRepository;
import com.nfjs.reactiveofficers.entities.Officer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping("/officers")
public class OfficerController {
    private OfficerRepository repository;

    public OfficerController(OfficerRepository repository) {
        this.repository = repository;
    }

    @GetMapping
    public Flux<Officer> getAllOfficers() {
        return repository.findAll();
    }

    @GetMapping("{id}")
    public Mono<Officer> getOfficer(@PathVariable String id) {
        return repository.findById(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Mono<Officer> saveOfficer(@RequestBody Officer officer) {
        return repository.save(officer);
    }

    @PutMapping("{id}")
    public Mono<ResponseEntity<Officer>> updateOfficer(@PathVariable(value = "id")
String id,
                                                       @RequestBody Officer officer) {
        return repository.findById(id)
                .flatMap(existingOfficer -> {
                    existingOfficer.setRank(officer.getRank());
                    existingOfficer.setFirst(officer.getFirst());
                    existingOfficer.setLast(officer.getLast());
                    return repository.save(existingOfficer);
                })
                .map(updateOfficer -> new ResponseEntity<>(updateOfficer,
HttpStatus.OK))
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
```

```
    @DeleteMapping("{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<ResponseEntity<Void>> deleteOfficer(@PathVariable(value = "id") String
id) {
        return repository.deleteById(id)
                .then(Mono.just(new ResponseEntity<Void>(HttpStatus.NO_CONTENT)))
                .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @DeleteMapping
    public Mono<Void> deleteAllOfficers() {
        return repository.deleteAll();
    }
}
```

7. Add an integration test that uses `WebTestClient`. Note that the server must be running to execute this test.

JAVA

```java
package com.oreilly.reactiveofficers.controllers;

import com.oreilly.reactiveofficers.dao.OfficerRepository;
import com.oreilly.reactiveofficers.entities.Officer;
import com.oreilly.reactiveofficers.entities.Rank;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;
import java.util.List;

import static org.junit.Assert.*;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class OfficerControllerTest {
//    private WebTestClient client = WebTestClient.bindToServer()
//                                                .baseUrl("http://localhost:8080")
//                                                .build();

    @Autowired
    private WebTestClient client;

    @Autowired
    private OfficerRepository repository;

    private List<Officer> officers = Arrays.asList(
            new Officer(Rank.CAPTAIN, "James", "Kirk"),
            new Officer(Rank.CAPTAIN, "Jean-Luc", "Picard"),
            new Officer(Rank.CAPTAIN, "Benjamin", "Sisko"),
            new Officer(Rank.CAPTAIN, "Kathryn", "Janeway"),
            new Officer(Rank.CAPTAIN, "Jonathan", "Archer"));

    @BeforeEach
    public void setUp() {
        repository.deleteAll()
                  .thenMany(Flux.fromIterable(officers))
                  .flatMap(repository::save)
                  .doOnNext(System.out::println)
                  .then()
                  .block();
    }

    @Test
    public void testGetAllOfficers() {
        client.get().uri("/officers")
```

```
                .accept(MediaType.APPLICATION_JSON_UTF8)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON_UTF8)
                .expectBodyList(Officer.class)
                .hasSize(5)
                .consumeWith(System.out::println);
    }

    @Test
    public void testGetOfficer() {
        client.get().uri("/officers/{id}", officers.get(0).getId())
                .exchange()
                .expectStatus().isOk()
                .expectBody(Officer.class)
                .consumeWith(System.out::println);
    }

    @Test
    public void testCreateOfficer() {
        Officer officer = new Officer(Rank.LIEUTENANT, "Nyota", "Uhuru");

        client.post().uri("/officers")
                .contentType(MediaType.APPLICATION_JSON_UTF8)
                .accept(MediaType.APPLICATION_JSON_UTF8)
                .body(Mono.just(officer), Officer.class)
                .exchange()
                .expectStatus().isCreated()
                .expectHeader().contentType(MediaType.APPLICATION_JSON_UTF8)
                .expectBody()
                .jsonPath("$.id").isNotEmpty()
                .jsonPath("$.first").isEqualTo("Nyota")
                .jsonPath("$.last").isEqualTo("Uhuru")
                .consumeWith(System.out::println);
    }
}
```

5. Spring WebFlux with Functional Endpoints

1. Create an `OfficerHandler` class in the `controllers` package that is just a regular `@Component`, with methods that each take a `ServerRequest` and return a `Mono<ServerResponse>`:

JAVA

```java
package com.oreilly.reactiveofficers.controllers;

import com.oreilly.reactiveofficers.dao.OfficerRepository;
import com.oreilly.reactiveofficers.entities.Officer;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

@Component
public class OfficerHandler {
    private OfficerRepository repository;

    public OfficerHandler(OfficerRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listOfficers(ServerRequest request) {
        return ServerResponse.ok()
                             .contentType(APPLICATION_JSON)
                             .body(repository.findAll(), Officer.class);
    }

    public Mono<ServerResponse> createOfficer(ServerRequest request) {
        Mono<Officer> officerMono = request.bodyToMono(Officer.class);
        return officerMono.flatMap(officer ->
                                           ServerResponse.status(HttpStatus.CREATED)

.contentType(APPLICATION_JSON)

.body(repository.save(officer), Officer.class));
    }

    public Mono<ServerResponse> getOfficer(ServerRequest request) {
        String id = request.pathVariable("id");
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        Mono<Officer> personMono = this.repository.findById(id);
        return personMono
                .flatMap(person -> ServerResponse.ok()
                                                 .contentType(APPLICATION_JSON)
                                                 .body(fromObject(person)))
                .switchIfEmpty(notFound);
    }
}
```

2. Make a class called `RouterConfig` in the `com.oreilly.reactiveofficers.configuration` package. Annotate it with `@Configuration` and add a bean to return a `RouterFunction<ServerResponse>`:

JAVA

```
package com.oreilly.reactiveofficers.configuration;

import com.oreilly.reactiveofficers.controllers.OfficerHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

@Configuration
public class RouterConfig {
    @Bean
    public RouterFunction<ServerResponse> route(OfficerHandler handler) {
        return RouterFunctions
                .route(GET("/route/{id}").and(accept(APPLICATION_JSON)),
                       handler::getOfficer)
                .andRoute(GET("/route").and(accept(APPLICATION_JSON)),
                          handler::listOfficers)
                .andRoute(POST("/route").and(contentType(APPLICATION_JSON)),
                          handler::createOfficer);
    }
}
```

3. Add a test for the router function and handler:

JAVA

```java
package com.oreilly.reactiveofficers.controllers;

import com.oreilly.reactiveofficers.dao.OfficerRepository;
import com.oreilly.reactiveofficers.entities.Officer;
import com.oreilly.reactiveofficers.entities.Rank;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Before;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;
import java.util.List;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class OfficerHandlerAndRouterTests {
    @Autowired
    private WebTestClient client;

    @Autowired
    private OfficerRepository repository;

    private List<Officer> officers = Arrays.asList(
            new Officer(Rank.CAPTAIN, "James", "Kirk"),
            new Officer(Rank.CAPTAIN, "Jean-Luc", "Picard"),
            new Officer(Rank.CAPTAIN, "Benjamin", "Sisko"),
            new Officer(Rank.CAPTAIN, "Kathryn", "Janeway"),
            new Officer(Rank.CAPTAIN, "Jonathan", "Archer"));

    @BeforeEach
    public void setUp() {
        repository.deleteAll()
                  .thenMany(Flux.fromIterable(officers))
                  .flatMap(repository::save)
                  .doOnNext(System.out::println)
                  .then()
                  .block();
    }

    @Test
    public void testCreateOfficer() {
        Officer officer = new Officer(Rank.LIEUTENANT, "Hikaru", "Sulu");
        client.post().uri("/route")
              .contentType(MediaType.APPLICATION_JSON)
              .accept(MediaType.APPLICATION_JSON)
              .body(Mono.just(officer), Officer.class)
              .exchange()
```

```
                .expectStatus().isCreated()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBody()
                .jsonPath("$.id").isNotEmpty()
                .jsonPath("$.first").isEqualTo("Hikaru")
                .jsonPath("$.last").isEqualTo("Sulu");
    }

    @Test
    public void testGetAllOfficers() {
        client.get().uri("/route")
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_JSON)
                .expectBodyList(Officer.class);
    }

    @Test
    public void testGetSingleOfficer() {
        Officer officer = repository.save(new Officer(Rank.ENSIGN, "Wesley",
"Crusher")).block();

        client.get()
                //.uri("/route/{id}", Collections.singletonMap("id", officer.getId()))
                .uri("/route/{id}", officer.getId())
                .exchange()
                .expectStatus().isOk()
                .expectBody()
                .consumeWith(response ->

Assertions.assertThat(response.getResponseBody()).isNotNull());
    }
}
```

4. The tests should all now pass.

Last updated 2020-08-12 17:56:34 EDT