Lab: Implementing GET

1: Write a Spring Boot Test for the GET endpoint

Just as if we're on a real project, let's use test driven development to implement our first API endpoint.

1. Write the test.

   Let's start by implementing a test using Spring's @SpringBootTest.

   Update `src/test/java/example/cashcard/CashCardApplicationTests.java` with the following:

```
package example.cashcard;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class CashCardApplicationTests {
    @Autowired
    TestRestTemplate restTemplate;

    @Test
    void shouldReturnACashCardWhenDataIsSaved() {
        ResponseEntity<String> response =
restTemplate.getForEntity("/cashcards/99", String.class);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    }
}
```

2. Understand the test.

   Let's understand several important elements in this test.

   `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)``

   This will start our Spring Boot application and make it available for our test to perform requests to it.

```
@Autowired
TestRestTemplate restTemplate;
```

   We've asked Spring to inject a test helper that'll allow us to make HTTP requests to the locally running application.

Note that while @Autowired is a form of Spring dependency injection it's best used only in tests. We'll discuss this in more detail later.

`ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/99", String.class);`

Here we use restTemplate to make an HTTP GET request to our application endpoint /cashcards/99.

restTemplate will return a ResponseEntity, which we've captured in a variable we've named response. ResponseEntity is another helpful Spring object that provides valuable information about what happened with our request. We will use this information throughout our tests in this course.

`assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);`

We can inspect many aspects of the response, including the HTTP Response Status code, which we expect to be 200 OK.

3. Now run the test.

   What do you think will happen when we run the test?

   It will fail, as expected. Why? As we've learned in test-first practice, we describe our expectations before we implement the code that satisfies those expectations.

   Now let's run the test. Note that we will run ./gradlew test for every test run.

   `[~/exercises] $ ./gradlew test`

   It fails! Search the output for the following:

   ```
   CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
   org.opentest4j.AssertionFailedError:
   expected: 200 OK
   but was: 404 NOT_FOUND
   ```

   But why are we getting this specific failure?

4. Understand the test failure.

   As we explained, we expected our test to currently fail.

   Why is it failing due to an unexpected 4048 NOT_FOUND HTTP response code?

   Answer: since we have not instructed Spring Web how to handle GET cashcards/99, Spring Web is automatically responding that the endpoint is NOT_FOUND.

   Thank you for handling that for us, Spring Web!

Next, let's get our application working properly.

2: Create a REST Controller

Spring Web Controllers are designed to handle and respond to HTTP requests.

1. Create the Controller

Create the Controller class in `src/main/java/example/cashcard/CashCardController.java`.

```
package example.cashcard;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

public class CashCardController {
}
```

2. Add the handler method.

Implement a findById() method to handle incoming HTTP requests.

```
public class CashCardController {
   public ResponseEntity<String> findById() {
      return ResponseEntity.ok("{}");
   }
}
```

3. Now rerun the test.

What do we expect to happen when we rerun the tests?

```
expected: 200 OK
 but was: 404 NOT_FOUND
Same result! Why?
```

Despite the name, `CashCardController` is not really a Spring Web Controller; it's just a class with Controller in the name. Thus, it's not "listening" for our HTTP requests. So next we need to tell Spring to make the Controller available as a Web Controller to handle requests to cashcards/* URLs.

3: Add the GET endpoint

1. Update the Controller.

Let's update our `CashCardController` so it's configured to listen for and handle HTTP requests to `/cashcards`.

```
@RestController
@RequestMapping("/cashcards")
public class CashCardController {

   @GetMapping("/{requestedId}")
   public ResponseEntity<String> findById() {
         return ResponseEntity.ok("{}");
   }
}
```

2. Understand the Spring Web annotations.

Let's review our additions.

```
`@RestController`

This tells Spring that this class is a Component of type RestController and capable of handling HTTP requests.

`@RequestMapping("/cashcards")`

This is a companion to @RestController that indicates which address requests must have to access this Controller.

```
@GetMapping("/{requestedId}")
public ResponseEntity<String> findById() {...}
```
```

@GetMapping marks a method as a handler method. GET requests that match cashcards/{requestedID} will be handled by this method.

3. Run the tests.

They now pass!

```
BUILD SUCCESSFUL in 6s
```

Now we have a Controller and handler method that matches the request performed in our test.

4: Complete the GET endpoint

So far our test only asserts that the request succeeded by checking for a 200 OK response status. Now let's test that the response contains the correct values.

1. Update the test.

```
assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

DocumentContext documentContext = JsonPath.parse(response.getBody());
Number id = documentContext.read("$.id");
assertThat(id).isNotNull();
```

2. Understand the additions.

```
DocumentContext documentContext = JsonPath.parse(response.getBody());
```

This converts the response String into a JSON-aware object with lots of helper methods.

```
Number id = documentContext.read("$.id");
assertThat(id).isNotNull();
```

We expect that when we request a Cash Card with id of `99` a JSON object will be returned with something in the id field. For now assert that the id is not null.

3. Run the test and note the failure.

Since we return an empty JSON object `{}` we should not be surprised that the id field is empty.

```
CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
    com.jayway.jsonpath.PathNotFoundException: No results for path: $['id']
```

4. Return a Cash Card from the Controller.

Let's make the test pass, but return something intentionally wrong, such as 1000L. You'll see why later.

Additionally, let's utilize the CashCard data model class that we created in an earlier lesson. Please review it under `src/main/java/example/cashcard/CashCard.java` if needed.

```
@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById() {
   CashCard cashCard = new CashCard(1000L, 0.0);
   return ResponseEntity.ok(cashCard);
}
```

5. Run the test.

It passes! But does it feel correct? Not really. Having the test pass with incorrect data seems wrong.

We asked you to intentionally return an incorrect id of `1000L` to illustrate a point: it is important that tests pass or fail for for the right reason.

6. Update the test

Update the test to assert that the id is correct.

```
DocumentContext documentContext = JsonPath.parse(response.getBody());
Number id = documentContext.read("$.id");
assertThat(id).isEqualTo(99);
```

7. Rerun the tests and note the new failure message.

```
expected: 99
 but was: 1000
```

Now the test is failing for the right reason: we did not return the correct id.

8. Fix `CashCardController`.

Update `CashCardController` to return the correct id.

```
@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById() {
   CashCard cashCard = new CashCard(99L, 0.0);
   return ResponseEntity.ok(cashCard);
}
```

9. Run the test -- woo hoo, it passes!

10. Test the `amount`.

Now let's add an assertion for amount indicated by the JSON contract.

```
DocumentContext documentContext = JsonPath.parse(response.getBody());
Number id = documentContext.read("$.id");
```

```
assertThat(id).isEqualTo(99);

Double amount = documentContext.read("$.amount");
assertThat(amount).isEqualTo(123.45);
```

11. Run the tests and observe the failure.

Sure enough, we don't return the correct amount in the response.

```
expected: 123.45
but was: 0.0
```

12. Return the correct amount.

Let's update the `CashCardController` to return the amount indicated by the JSON contract.

```
@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById() {
   CashCard cashCard = new CashCard(99L, 123.45);
   return ResponseEntity.ok(cashCard);
}
```

13. Rerun the tests.

They pass!

```
BUILD SUCCESSFUL in 6s
```

5: Using the @PathVariable

Thus far we've ignored the requestedId in the Controller handler method. Let's use this path variable in our Controller to make sure we return the correct Cash Card.

1. Add a new test method.

Let's write a new test that expects that we ignore Cash Cards that do not have an id of 99. use 1000 as we have in previous tests.

```
@Test
void shouldNotReturnACashCardWithAnUnknownId() {
  ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/1000",
String.class);

  assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
  assertThat(response.getBody()).isBlank();
}
```

Notice that we are expecting a semantic HTTP Response Status code of 404 NOT_FOUND. If we request a Cash Card that does not exist then that Cash Card is indeed "not found".

2. Run the test and note the result.

```
expected: 404 NOT_FOUND
but was: 200 OK
```

3. Add `@PathVariable`.

Now let's make the test pass by making the Controller return the specific Cash Card only if we submit the correct identifier.

To do this, first make the Controller aware of the path variable we're submitting by adding the @PathVariable annotation to the handler method argument.

```
@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {
    CashCard cashCard = new CashCard(99L, 123.45);
    return ResponseEntity.ok(cashCard);
}
```

`@PathVariable` makes Spring Web aware of the requestedId supplied in the HTTP request. Now it's available for us to use in our handler method.

4. Utilize `@PathVariable`.

Update the handler method to return an empty response with status NOT_FOUND unless the requestedId is 99.

```
@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {
    if (requestedId.equals(99L)) {
        CashCard cashCard = new CashCard(99L, 123.45);
        return ResponseEntity.ok(cashCard);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

5. Rerun the tests.

Awesome! They pass!

```
BUILD SUCCESSFUL in 7s
```

6: Summary

Congrats! In this lesson you learned how to use test driven development to create your first Family Cash Card REST endpoint: a `GET` that returns a `CashCard` of a certain ID.