Lab: Implementing DELETE

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_9`

1: Overview

In this lab, we'll implement hard delete in our Cash Card API, using the API specifications.

Credentials for test user Kumar

If you have taken previous labs in this course you'll notice the following changes to the codebase, which we have made on your behalf to make this lab easier to understand and complete.

We have added credentials for the user kumar2 to the testOnlyUsers bean in `src/main/java/example/cashcard/SecurityConfig.java`.

Let's implement the Delete endpoint!

2: Test the Happy Path

Let's start with the simplest happy path: successfully deleting a CashCard which exists.

We need the Delete endpoint to return the 204 NO CONTENT status code.

Write the test.

Add the following method to src/test/java/example/cashcard/CashCardApplicationTests.java:

@Test @DirtiesContext void shouldDeleteAnExistingCashCard() { ResponseEntity response = restTemplate .withBasicAuth("sarah1", "abc123") .exchange("/cashcards/99", HttpMethod.DELETE, null, Void.class); assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT); } Notice that we've added the @DirtiesContext annotation. We'll add this annotation to all tests which change the data. If we don't, then these tests could affect the result of other tests in the file.

Why not use RestTemplate.delete()? Notice that we're using RestTemplate.exchange() even though RestTemplate supplies a method that looks like we could use: RestTemplate.delete(). However, let's look at the signature:

public class RestTemplate ... { public void delete(String url, Object... uriVariables) The other methods we've been using (such as getForEntity() and exchange()) return a ResponseEntity, but delete() does not. Instead, it's a void method. Why is this?

The Spring Web framework supplies the delete() method as a convenience, but it comes with some assumptions:

A response to a DELETE request will have no body. The client shouldn't care what the response code is unless it's an error, in which case, it'll throw an exception. Given those assumptions, no return value is needed from delete().

But the second assumption makes delete() unsuitable for us: We need the ResponseEntity in order to assert on the status code! So we won't use the convenience method, but rather let's use the more general method: exchange().

Run the tests.

As always, we use ./gradlew test to run the tests.

[~/exercises] $ ./gradlew test ... CashCardApplicationTests > shouldDeleteAnExistingCashcard() FAILED org.opentest4j.AssertionFailedError: expected: 204 NO_CONTENT but was: 403 FORBIDDEN The test failed because the DELETE /cashcards/99 request returned a 403 FORBIDDEN.

At this point you probably expected this result: Spring Security returns a 403 response for any endpoint which is not mapped.

We need to implement the Controller method! So let's do it.

Implement the Delete endpoint in the Controller.

Add the following method to the CashCardController class:

@DeleteMapping("/{id}") private ResponseEntity deleteCashCard(@PathVariable Long id) { return ResponseEntity.noContent().build(); } Run the tests. They pass!

So are we done? Not yet!

We haven't written the code to actually delete the item. Let's do that next.

We write the test first, of course.

Test that we're actually deleting the CashCard.

Add the following assertions to the shouldDeleteAnExistingCashcard() method:

void shouldDeleteAnExistingCashCard() { ResponseEntity response = restTemplate .withBasicAuth("sarah1", "abc123") .exchange("/cashcards/99", HttpMethod.DELETE, null, Void.class); assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);

```
// Add the following code:
ResponseEntity<String> getResponse = restTemplate
        .withBasicAuth("sarah1", "abc123")
        .getForEntity("/cashcards/99", String.class);
assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
```

} Understand the test code.

We want to test that the deleted Cash Card is actually deleted, so we try to GET it, and assert that the result code is 404 NOT FOUND.

Run the test. Does it pass? Of course not!

CashCardApplicationTests > shouldDeleteAnExistingCashcard() FAILED org.opentest4j.AssertionFailedError: expected: 404 NOT_FOUND but was: 200 OK What do we need to do to make the test pass? Write some code to delete the record!

Let's go.

3: Implement the DELETE Endpoint

Now we need to write a Controller method which will be called when we send a DELETE request with the proper URI.

Add code to the Controller to delete the record.

Change the CashCardController.deleteCashCard() method:

@DeleteMapping("/{id}") private ResponseEntity deleteCashCard(@PathVariable Long id) { cashCardRepository.deleteById(id); // Add this line return ResponseEntity.noContent().build(); } The change is straightforward:

We use the @DeleteMapping with the "{id}" parameter, which Spring Web matches to the id method parameter. CashCardRepository already has the method we need: deleteById() (it's inherited from CrudRepository). Run the tests, and watch them pass!

Great, what to do next?

4: Test case: The Cash Card doesn't exist

Our contract states that we should return 404 NOT FOUND if we try to delete a card that doesn't exist.

Write the test.

Add the following test method to CashCardApplicationTests:

@Test void shouldNotDeleteACashCardThatDoesNotExist() { ResponseEntity deleteResponse = restTemplate .withBasicAuth("sarah1", "abc123") .exchange("/cashcards/99999", HttpMethod.DELETE, null, Void.class); assertThat(deleteResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND); } Run the tests.

CashCardApplicationTests > shouldNotDeleteACashCardThatDoesNotExist() FAILED org.opentest4j.AssertionFailedError: expected: 404 NOT_FOUND but was: 204 NO_CONTENT No surprise here—we need to enforce the security of our app by checking that the user trying to delete a card is the owner. Spring Security does a lot of things for us, but this is not one of them.

5: Enforce Ownership

We need to check whether the record exists. If not, we should not delete the Cash Card, and return 404 NOT FOUND.

Make the following changes to CashCardController.deleteCashCard:

private ResponseEntity deleteCashCard(@PathVariable Long id, Principal principal // Add Principal to the parameter list) { // Add the following 3 lines: if (!cashCardRepository.existsByIdAndOwner(id, principal.getName())) { return ResponseEntity.notFound().build(); } ... } Let's be sure to the add the Principal parameter!

We're using the Principal in order to check whether the Cash Card exists, and at the same time, enforce ownership.

Add the existsByIdAndOwner() method to the Repository.

Also, let's add the new method existsByIdAndOwner() to CashCardRepository:

public interface CashCardRepository extends CrudRepository<CashCard, Long>, PagingAndSortingRepository<CashCard, Long> { ... boolean existsByIdAndOwner(Long id, String owner); ... } Understand the Repository code.

We added logic to the Controller method to check whether the Cash Card ID in the request actually exists in the database. The method we'll use is CashCardRepository.existsByIdAndOwner(id, username).

This is another case where Spring Data will generate the implementation of this method as long as we add it to the Repository.

So why not just use the findByIdAndOwner() method and check whether it returns null? We could absolutely do that! But, such a call would return extra information (the content of the Cash Card retrieved), so we'd like to avoid it as to not introduce extra complexity.

If you'd rather not use the existsByIdAndOwner() method, that's ok! You may choose to use findByIdAndOwner(). The test result will be the same!

Watch the test pass.

Let's run the test and, no big surprise, the test passes!

```
[~/exercises] $ ./gradlew test
...
BUILD SUCCESSFUL in 8s
```

6: Refactor

At this point, we have an opportunity to practice the Red, Green, Refactor process. We've already done Red (the failing test), and Green (the passing test). Now we can ask ourselves, should we Refactor anything?

Here's the body of our CashCardController.deleteCashCard method:

```
if (!cashCardRepository.existsByIdAndOwner(id, principal.getName())) {
    return ResponseEntity.notFound().build();
}
cashCardRepository.deleteById(id);
return ResponseEntity.noContent().build();
```

You might find the following version, which is logically equivalent but slightly simpler, to be easier to read:

```
if (cashCardRepository.existsByIdAndOwner(id, principal.getName())) {
    cashCardRepository.deleteById(id);
    return ResponseEntity.noContent().build();
}
return ResponseEntity.notFound().build();
```

The differences are slight, but removing a not-operator (!) from an if statement often makes for easier-to-read code, and readability is important!

If you do find the second version easier to read and understand, then replace the existing code with the new version.

Run the tests again. They still pass!

```
[~/exercises] $ ./gradlew test
...
BUILD SUCCESSFUL in 8s
```

7: Hide Unauthorized Records

At this point, you may ask yourself, "Are we done?" You are the best person to answer that question! If you want, take a couple minutes to refresh yourself with the accompanying lesson in order to see if we've tested and implemented every aspect of the API contract for DELETE.

OK, that was time well-spent, wasn't it? That's right - There is one more case that we have yet to test: What if the user attempts to delete a Cash Card owned by someone else? Response should be 404 NOT FOUND in this case. Thats enough information for us to write a test for that case:

In CashCardApplicationTests.java, add the following test method at the end of the class:

```
```
@Test
void shouldNotAllowDeletionOfCashCardsTheyDoNotOwn() {
    ResponseEntity<Void> deleteResponse = restTemplate
            .withBasicAuth("sarah1", "abc123")
            .exchange("/cashcards/102", HttpMethod.DELETE, null, Void.class);
    assertThat(deleteResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```
```

Run the test

Do you think the test will pass? Take a moment to predict the outcome, then run the test.

It passed! That's great news.

We have written a test for a specific case in our API. The test passed without any code changes! Now let's consider a question which may have occurred to you: Why do I need that test, since it passes without having to make any code changes? Isn't the purpose of TDD to use tests to guide the implementation of the application? If that's the case, why did I bother to write that test?

Answers:

Yes, that is one of many benefits that TDD provides: A process to guide the creation of code in order to arrive at a desired outcome. Tests themselves, though, have another purpose, separate from TDD: Tests are a powerful safety net to enforce correctness. Since the test you just wrote tests a different case than those already written, it provides value. If someone were to make a code change which caused this new test to fail, then you will have caught the error before it became an issue! Yay for Tests. One More Test

But wait, you say. Shouldn't I test that the record that I tried to delete still exists in the database - that it didn't get deleted? Yes! That is a valid test. Thanks for mentioning it! Add the following code to the test method, to verify that the record you tried unsuccessfully to delete is still there:

```
    void shouldNotAllowDeletionOfCashCardsTheyDoNotOwn() {
...
        // Add this code at the end of the test method:
        ResponseEntity<String> getResponse = restTemplate
                .withBasicAuth("kumar2", "xyz789")
                .getForEntity("/cashcards/102", String.class);
        assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
    }
```

Do you think the test will pass? Of course it will! (right?)

Run the test

Just to make sure ... Please run all the tests and ensure that they pass.

8: Summary

In this lesson, you implemented a RESTful Delete endpoint which does not "leak" security information to would-be attackers. You also had a chance to do a small, but useful refactoring to practice the Red, Green, Refactor process.

This is the last of the CRUD operations to implement in the API, which brings us to a successful conclusion of our technical work! Congrats!