Lab: Implementing PUT

1: Overview

In this lab, we'll allow users of the Family Cash Card REST API to update a CashCard using an HTTP PUT.

Let's implement this functionality now!

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_8`

2: Write the Test First

As we have in almost every lab, let's begin with a test.

What is the functionality we want our application to have? How do we want our application to behave?

Let's define this now, and then work our way towards satisfying our aspirations.

Write the Update test.

We will use PUT to update CashCards.

Note that we'll expect a 204 NO_CONTENT response instead of a 200 OK. The 204 indicates that the action was successfully performed and no further action is needed by the caller.

Edit `src/test/java/example/cashcard/CashCardApplicationTests.java` and add the following test, which updates Cash Card 99 and sets its amount to 19.99.

Don't forget to add the two new imports.

```
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
...
@Test
@DirtiesContext
void shouldUpdateAnExistingCashCard() {
    CashCard cashCardUpdate = new CashCard(null, 19.99, null);
    HttpEntity<CashCard> request = new HttpEntity<>(cashCardUpdate);
    ResponseEntity<Void> response = restTemplate
            .withBasicAuth("sarah1", "abc123")
            .exchange("/cashcards/99", HttpMethod.PUT, request, Void.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
}
```

Learning Moment: what's up with restTemplate.exchange()? Did you notice that we're not using RestTemplate in the same way we have in our previous tests?

All other tests use RestTemplate.xyzForEntity() methods such as getForEntity() and postForEntity().

So, why are we not following the same pattern utilizing putForEntity()?

Answer: putForEntity() does not exist! Read more about it here in the GitHub issue about the topic.

Luckily RestTemplate supports multiple ways of interacting with REST APIs, such as RestTemplate.exchange().

Let's learn about RestTemplate.exchange() now.

Understand RestTemplate.exchange().

Let's understand more about what's happening.

The exchange() method is a more general version of the xyzForEntity() methods we've used in other tests: exchange() requires the verb and the request entity (the body of the request) to be supplied as parameters.

Using getForEntity() as an example, you can imagine that the following two lines of code accomplish the same goal:

`.exchange("/cashcards/99", HttpMethod.GET, new HttpEntity(null), String.class);`

The above line is functionally equivalent to the following line:

`.getForEntity("/cashcards/99", String.class);`

Now let's explain the test code.

First we create the HttpEntity that the exchange() method needs:

`HttpEntity<CashCard> request = new HttpEntity<>(existingCashCard);`

Then we call exchange(), which sends a PUT request for the target ID of 99 and updated Cash Card data:

.exchange("/cashcards/99", HttpMethod.PUT, request, Void.class); Run the tests.

It's time to watch our tests fail for the right reasons.

For what reason will our new test fail?

Note that we'll always use ./gradlew test to run our tests.

```
[~/exercises] $ ./gradlew test
...
CashCardApplicationTests > shouldUpdateAnExistingCashCard() FAILED
 org.opentest4j.AssertionFailedError:
 expected: 204 NO_CONTENT
  but was: 403 FORBIDDEN
...
BUILD FAILED in 6s
```

Answer: we haven't implemented a PUT request method handler yet!

As you can see, the test failed with a 403 FORBIDDEN response code.

With no Controller endpoint, this PUT call is forbidden! Spring Security automatically handled this scenario for us.

Next, let's implement the Controller endpoint.

3: Implement @PutMapping in the Controller

Following the pattern we've used this far in our Controller, let's implement the PUT endpoint in our CashCardController.

Add a minimal `@PutMapping`.

Edit `src/main/java/example/cashcard/CashCardController.java` and add the PUT endpoint.

```
@PutMapping("/{requestedId}")
private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
```

```
CashCard cashCardUpdate) {
    // just return 204 NO CONTENT for now.
    return ResponseEntity.noContent().build();
}
```

This Controller endpoint is fairly self-explanatory:

The @PutMapping supports the PUT verb and supplies the target requestedId. The @RequestBody contains the updated CashCard data. Return an HTTP 204 NO_CONTENT response code for now just to get started. Run the tests.

What do you think will happen?

Let's run them now.

```
...
CashCardApplicationTests > shouldUpdateAnExistingCashCard() PASSED
...
BUILD SUCCESSFUL in 6s
```

They pass!

But, that isn't very satisfying, is it?

We haven't updated the CashCard!

Let's do that next.

Enhance the test to verify a successful update.

Similar to other verifications we've performed in our test suite, let's assert that the update was successful.

```
void shouldUpdateAnExistingCashCard() {
  ...
  assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);

  ResponseEntity<String> getResponse = restTemplate
          .withBasicAuth("sarah1", "abc123")
          .getForEntity("/cashcards/99", String.class);
  assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
  DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
  Number id = documentContext.read("$.id");
  Double amount = documentContext.read("$.amount");
  assertThat(id).isEqualTo(99);
  assertThat(amount).isEqualTo(19.99);
}
```

Understand the test updates.

```
ResponseEntity<String> getResponse = restTemplate
    .withBasicAuth("sarah1", "abc123")
    .getForEntity("/cashcards/99", String.class);
```

Here we fetch CashCard 99 again so we can verify that it was updated.

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

```
assertThat(amount).isEqualTo(19.99);
```

Next, we assert that we've retrieved the correct CashCard -- 99 -- and that its amount was successfully updated to 19.99.

Run the tests.

Our expectations are legit, but we haven't updated the CashCardController to match.

The test should fail with valuable error messages, right?

Let's run the tests now.

```
CashCardApplicationTests > shouldUpdateAnExistingCashCard() FAILED
 org.opentest4j.AssertionFailedError:
 expected: 19.99
  but was: 123.45
```

Excellent! We expect an amount of 19.99, but without any changes we are still returning 123.45.

Let's update the Controller.

Update CashCardController to perform the data update.

As with our other handler methods, let's ensure that we guarantee only the CashCard owner can perform the updates -- the logged-in Principal must be the same as the Cash Card owner:

```
@PutMapping("/{requestedId}")
private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
CashCard cashCardUpdate, Principal principal) {
    CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
principal.getName());
    CashCard updatedCashCard = new CashCard(cashCard.id(), cashCardUpdate.amount(),
principal.getName());
    cashCardRepository.save(updatedCashCard);
    return ResponseEntity.noContent().build();
}
```

Understand the CashCardController updates.

We're following a similar pattern as in the @PostMapping createCashCard() endpoint.

@PutMapping("/{requestedId}") private ResponseEntity putCashCard(@PathVariable Long requestedId, @RequestBody CashCard cashCardUpdate, Principal principal) { We've added the Principal as a method argument, provided automatically by Spring Security.

Thanks once again, Spring Security!

cashCardRepository.findByIdAndOwner(requestedId, principal.getName()); Here we scope our retrieval of the CashCard to the submitted requestedId and Principal (provided by Spring Security) to ensure only the authenticated, authorized owner may update this CashCard.

CashCard updatedCashCard = new CashCard(cashCard.id(), cashCardUpdate.amount(), principal.getName()); cashCardRepository.save(updatedCashCard); Finally, build a CashCard with updated values and save it.

That was a lot! Let's run the tests and assess where we're at.

Run the tests.

They pass!

```
...
CashCardApplicationTests > shouldUpdateAnExistingCashCard() PASSED
...
BUILD SUCCESSFUL in 6s
```

We've successfully implemented updating a CashCard.

But what happens if we attempt to update a CashCard that does not exist?

Let's test that scenario next.

4: Additional Testing and Spring Security's Influence

What would happen if we tried to update a Cash Card that doesn't exist?

Let's add a test and find out.

Try to update a Cash Card that doesn't exist.

Add the following test to CashCardApplicationTests.

```
@Test
void shouldNotUpdateACashCardThatDoesNotExist() {
    CashCard unknownCard = new CashCard(null, 19.99, null);
    HttpEntity<CashCard> request = new HttpEntity<>(unknownCard);
    ResponseEntity<Void> response = restTemplate
            .withBasicAuth("sarah1", "abc123")
            .exchange("/cashcards/99999", HttpMethod.PUT, request, Void.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

Here we'll attempt to update a Cash Card with ID 99999, which does not exist.

We should expect a generic 404 NOT_FOUND error.

Run the tests.

Let's make a hypothesis about what will happen when we run the tests.

Run the tests and observe the results.

... CashCardApplicationTests > shouldNotUpdateACashCardThatDoesNotExist() FAILED
org.opentest4j.AssertionFailedError: expected: 404 NOT_FOUND but was: 403 FORBIDDEN Well that's...interesting. Why did we get a 403 FORBIDDEN?

Before we run them again, let's edit build.gradle to enable additional test output.

```
test {
 testLogging {
     ...
     // Change to `true` for more verbose test output
     showStandardStreams = true
 }
}
```

After rerunning the tests, search the output for the following:

```
CashCardApplicationTests > shouldNotUpdateACashCardThatDoesNotExist() STANDARD_OUT
...
java.lang.NullPointerException: Cannot invoke "example.cashcard.CashCard.id()" because
"cashCard" is null
```

A NullPointerException! Why a NullPointerException?

Looking at CashCardController.putCashCard we can see that if we don't find the cashCard then method calls to cashCard will result in a NullPointerException. That makes sense.

But why is a NullPointerException thrown in our Controller resulting in a 403 FORBIDDEN instead of a 500 INTERNAL_SERVER_ERROR, given the server "crashed?"

Learning Moment: Spring Security and Error Handling Our Controller is returning 403 FORBIDDEN instead of an 500 INTERNAL_SERVER_ERROR because Spring Security is automatically implementing a best practice regarding how errors are handled by Spring Web.

It's important to understand that any information returned from our application might be useful to a bad actor attempting to violate our application's security. For example: knowledge about actions that causes our application to crash -- a 500 INTERNAL_SERVER_ERROR.

In order to avoid "leaking" information about our application, Spring Security has configured Spring Web to return a generic 403 FORBIDDEN in most error conditions. If almost everything results in a 403 FORBIDDEN response then an attacker doesn't really know what's going on.

Now that we understand what is happening, let's fix it.

Don't crash.

Though we're thankful to Spring Security, our application should not crash - we shouldn't allow our code to throw a NullPointerException. Instead, we should handle the condition when cashCard == null, and return a generic 404 NOT_FOUND HTTP response.

Update CashCardController.putCashCard to return 404 NOT_FOUND if no existing CashCard is found.

```
@PutMapping("/{requestedId}")
private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
CashCard cashCardUpdate, Principal principal) {
    CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
principal.getName());
    if (cashCard != null) {
        CashCard updatedCashCard = new CashCard(cashCard.id(),
cashCardUpdate.amount(), principal.getName());
        cashCardRepository.save(updatedCashCard);
        return ResponseEntity.noContent().build();
    }
    return ResponseEntity.notFound().build();
}
```

Run the tests.

Our tests pass now that we're returning a 404 when no CashCard is found for this user.

```
BUILD SUCCESSFUL in 6s
```

5: Refactor the Controller Code

Let's reinforce your usage of the Red, Green, Refactor development loop.

We have just completed several tests focused on updating an existing Cash Card.

Do we have any opportunities to simplify, reduce duplication, or otherwise refactor our code without changing behavior?

Continue on to address several refactoring opportunities.

Simplify the Code Remove the Optional.

This might be controversial: since we are not taking advantage of features of Optional in CashCardController.findById, and no other Controller methods use an Optional, let's simplify our code by removing it.

Edit `CashCardController.findById` to remove the usage of the Optional:

```
// remove the unused Optional import if present
// import java.util.Optional;

@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById(@PathVariable Long requestedId, Principal
principal) {
    CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
principal.getName());
    if (cashCard != null) {
        return ResponseEntity.ok(cashCard);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Run the tests.

Because we have not changed any functionality, the tests will continue to pass.

```
./gradlew test
...
BUILD SUCCESSFUL in 6s
```

Reduce Code Duplication

Note that both CashCardController.findById and CashCardController.putCashCard have nearly identical code that retrieves a target CashCard from the CashCardRepository using information from a CashCard and Principal.

Let's reduce code duplication by extracting a helper method named findCashCard and utilizing it in both .findById and .putCashCard.

This will allow us to update how we retrieve a CashCard in one place as the Controller changes over time.

Create a shared findCashCard method.

Create a new private method in CashCardController named findCashCard, using functionality we've written before:

private CashCard findCashCard(Long requestedId, Principal principal) { return cashCardRepository.findByIdAndOwner(requestedId, principal.getName()); } Update CashCardController.findById and rerun the tests.

Next, utilize the new findCashCard method in CashCardController.findById.

@GetMapping("/{requestedId}") public ResponseEntity findById(@PathVariable Long requestedId, Principal principal) { CashCard cashCard = findCashCard(requestedId, principal); ... No functionality has changed, so no tests should fail when we rerun the tests.

```
./gradlew test
...
BUILD SUCCESSFUL in 6s
```

Update `CashCardController.putCashCard` and rerun the tests.

Similar to the previous step, utilize the new findCashCard method in CashCardController.putCashCard.

```
@PutMapping("/{requestedId}")
private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
CashCard cashCardUpdate, Principal principal) {
    CashCard cashCard = findCashCard(requestedId, principal);
    ...
```

As with the other refactoring we have performed, no functionality has changed, so no tests should fail when we rerun the tests.

```
./gradlew test
...
BUILD SUCCESSFUL in 6s
```

Look at that! We have refactored our code by simplifying it and also reduced code duplication.

6: Summary

In this lesson, you learned how to implement an HTTP PUT endpoint that allows an authenticated, authorized owner to update their Cash Card.

You also learned how Spring Security automatically manages error handling by Spring Web to ensure that security-sensitive information isn't accidentally revealed when a Controller encounters an error.

In addition, you reinforced your understanding and utilization of the Red, Green, Refactor development cycle by refactoring our Controller code without changing its functionality.