Lab: Implementing POST

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_5`

1: Test the HTTP POST Endpoint

As we've done in previous labs, we'll begin by writing a test of what we expect success to look like.

Add a test for the POST endpoint.

The simplest example of success is a non-failing HTTP POST request to our Family Cash Card API. We'll test for a 200 OK response instead of a 201 CREATED for now. Don't worry, we'll change this soon.

Edit `src/test/java/example/cashcard/CashCardApplicationTests.java` and add the following test method.

```
@Test
void shouldCreateANewCashCard() {
   CashCard newCashCard = new CashCard(null, 250.00);
   ResponseEntity<Void> createResponse = restTemplate.postForEntity("/cashcards",
newCashCard, Void.class);
   assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
}
```

Understand the test.

CashCard newCashCard = new CashCard(null, 250.00); The database will create and manage all unique CashCard.id values for us. We should not provide one.

restTemplate.postForEntity("/cashcards", newCashCard, Void.class); This is very similar to restTemplate.getForEntity, but we must also provide newCashCard data for the new CashCard.

In addition, and unlike restTemplate.getForEntity, we don't expect a CashCard to be returned to us, so we expect a Void response body.

Run the tests.

We'll always use ./gradlew test to run our tests.

```
[~/exercises] $ ./gradlew test
What do you expect will happen?

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

We shouldn't be surprised by the 404 NOT_FOUND error. We have not added the POST endpoint yet!

Let's do that next.

2: Add the POST endpoint

The POST endpoint is similar to the GET endpoint in our CashCardController, but uses the @PostMapping annotation from Spring Web.

The POST endpoint must accept the data we are submitting for our new CashCard, specifically the amount.

But what happens if we don't accept the CashCard?

Add the POST endpoint without accepting CashCard data.

Edit `src/main/java/example/cashcard/CashCardController.java` and add the following method.

Don't forget to add the import for PostMapping.

```
import org.springframework.web.bind.annotation.PostMapping;
...

@PostMapping
private ResponseEntity createCashCard() {
   return null;
}
```

Note that by returning nothing at all, Spring Web will automatically generate an HTTP Response Status code of 200 OK.

Run the tests.

When we rerun the tests, they pass.

`BUILD SUCCESSFUL in 7s`

But, this isn't very satisfying -- our POST endpoint does nothing!

So let's make our tests better.

3: Testing based on semantic correctness

We want our Cash Card API to behave as semantically correctly as possible. Meaning, users of our API should not be surprised by how it behaves.

Let's refer to the official Request for Comments for HTTP Semantics and Content (RFC 7231) for guidance as to how our API should behave.

For our POST endpoint, review this section about HTTP POST; note that we have added emphasis:

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created ...

We'll explain more about this specification as we write our test.

Let's start by updating the POST test.

Update the shouldCreateANewCashCard test.

Here's how we'll encode the HTTP specification as expectations in our test. Be sure to add the additional import.

```
import java.net.URI;
...

@Test
void shouldCreateANewCashCard() {
   CashCard newCashCard = new CashCard(null, 250.00);
   ResponseEntity<Void> createResponse = restTemplate.postForEntity("/cashcards",
newCashCard, Void.class);
   assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);

   URI locationOfNewCashCard = createResponse.getHeaders().getLocation();
   ResponseEntity<String> getResponse =
restTemplate.getForEntity(locationOfNewCashCard, String.class);
   assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
}
```

Understand the test updates.

We have made quite a few changes. Let's review.

assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED); According to the official specification:

the origin server SHOULD send a 201 (Created) response ...

We now expect the HTTP response status code to be 201 CREATED, which is semantically correct if our API creates a new CashCard from our request.

URI locationOfNewCashCard = createResponse.getHeaders().getLocation(); The official spec continue to state the following:

send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created ...

In other words, when a POST request results in the successful creation of a resource, such as a new CashCard, the response should include information for how to retrieve that resource. We'll do this by supplying a URI in a Response Header named "Location".

Note that URI is indeed the correct entity here and not a URL; a URL is a type of URI, while a URI is more generic.

```
ResponseEntity<String> getResponse = restTemplate.getForEntity(locationOfNewCashCard,
String.class);
assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
```

Finally, we'll use the Location header's information to fetch the newly created CashCard.

Run the tests.

Unsurprisingly, they fail on the first changed assertion.

```
expected: 201 CREATED
  but was: 200 OK
```

Let's start fixing stuff!

4: Implement the POST Endpoint

Our POST endpoint in the CashCardController is currently empty. Let's implement the correct logic.

Return a 201 CREATED status.

As we incrementally make our test pass, we can start by returning 201 CREATED.

As we learned earlier, we must provide a Location header with the URI for where to find the newly created CashCard. We're not quite there yet, so we'll use a placeholder URI for now.

Be sure to add the two new import statements.

```
import java.net.URI;
import org.springframework.web.bind.annotation.RequestBody;
...

 @PostMapping
 private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest)
{
     return ResponseEntity.created(URI.create("/what/should/go/here?")).build();
 }
```

Run the tests.

Remarkably, our new test passes until the last line.

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

Here we expect to have retrieved our newly created CashCard, which we haven't created or returned from our CashCardController. Thus, our expectation fails with a result of NOT_FOUND.

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

Save the new CashCard and return its location.

Let's add the rest of the POST implementation, which we will describe in detail.

Be sure to add the new import.

```
import org.springframework.web.util.UriComponentsBuilder;
...

@PostMapping
private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest,
UriComponentsBuilder ucb) {
   CashCard savedCashCard = cashCardRepository.save(newCashCardRequest);
   URI locationOfNewCashCard = ucb
            .path("cashcards/{id}")
            .buildAndExpand(savedCashCard.id())
            .toUri();
   return ResponseEntity.created(locationOfNewCashCard).build();
}
```

Next we'll go over these changes in detail.

5: Understand CrudRepository.save

This line in `CashCardController.createCashCard` is deceptively simple:

```
CashCard savedCashCard = cashCardRepository.save(newCashCardRequest);
```

As learned in previous lessons and labs, Spring Data's CrudRepository provides methods that support creating, reading, updating, and deleting data from a data store. cashCardRepository.save(newCashCardRequest) does just as it says: it saves a new CashCard for us, and returns the saved object with a unique id provided by the database. Amazing!

6: Understand the other changes to CashCardController

Our CashCardController now implements the expected input and results of an HTTP POST.

`createCashCard(@RequestBody CashCard newCashCardRequest, ...)`

Unlike the GET we added earlier, the POST expects a request "body". This contains the data submitted to the API. Spring Web will deserialize the data into a CashCard for us.

```
URI locationOfNewCashCard = ucb
   .path("cashcards/{id}")
   .buildAndExpand(savedCashCard.id())
   .toUri();
```

This is constructing a URI to the newly created CashCard. This is the URI that the caller can then use to GET the newly-created CashCard.

Note that savedCashCard.id is used as the identifier, which matches the GET endpoint's specification of cashcards/<CashCard.id>.

Where did UriComponentsBuilder come from?

We were able to add UriComponentsBuilder ucb as a method argument to this POST handler method and it was automatically passed in. How so? It was injected from our now-familiar friend, Spring's IoC Container. Thanks, Spring Web!

return ResponseEntity.created(locationOfNewCashCard).build(); Finally, we return 201 CREATED with the correct Location header.

7: Final Testing and Learning Moment

Run the tests.

They pass!

`BUILD SUCCESSFUL in 7s`

The new CashCard was created, and we used the URI supplied in the Location response header to retrieve the newly created resource.

Add more test assertions.

If you'd like, add more test assertions for the new id and amount to solidify your learning.

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

// Add assertions such as these
DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
Number id = documentContext.read("$.id");
Double amount = documentContext.read("$.amount");

assertThat(id).isNotNull();
assertThat(amount).isEqualTo(250.00);
```

The additions verify that the new CashCard.id is not null, and the newly created CashCard.amount is 250.00, just as we specified at creation time.

Learning Moment Earlier we stated that the database (via the Repository) would manage creating all database id values for us.

What would happen if we provided an id for our new, unsaved CashCard?

Let's find out.

Update the test to submit a CashCard.id

Change the id submitted from null to one that does not exist, such as 44L.

```
@Test
void shouldCreateANewCashCard() {
   CashCard cashCard = new CashCard(44L, 250.00);
   ...
```

In addition, edit build.gradle to enable more verbose test output, which will help us identify the upcoming test failure.

```
test {
  testLogging {
    ...
    // Set to `true` for more detailed logging.
    showStandardStreams = true
  }
}
```

Run the tests.

When we run the test we see that the the API crashes with a 500 status code.

```
expected: 201 CREATED
 but was: 500 INTERNAL_SERVER_ERROR
```

Let's find out why the test is failing.

Find and understand the database failure.

Search the test output for the following message:

Failed to update entity [CashCard[id=44, amount=250.0]]. Id [44] not found in database. The Repository is trying to find CashCard with id of 44 and throwing an error when it cannot find it. Interesting! Can you guess why?

Supplying an id to cashCardRepository.save is supported when an update is performed on an existing resource.

We'll cover this scenario in a later lab focused on updating an existing CashCard.

In this Learning Moment you learned that the API requires that you not supply a CashCard.id when creating a new CashCard.

Should we validate that requirement in the API? You betcha! Again, stay tuned for how to do that in a future lesson.

8: Summary

In this lab you learned how simple it is to add another endpoint to our API -- the POST endpoint. You also learned how to to use that endpoint to create and save a new `CashCard` to our database using Spring Data. Not only that, but the endpoint accurately implements the HTTP POST specification, which we verified using test driven development. The API is starting to be useful!