Lab: Returning a list with GET

1: Changes from the Previous Lab

We've made the following changes from the previous lab.

Added a couple more Cash Card data fixtures to data.sql Refactored `CashCardJsonTest.java` to incorporate the new data fixtures. Renamed expected.json to single.json, and added another data contract JSON file: list.json. Added some imports to the Test classes, so you don't have to! Added the @DirtiesContext annotation to CashCardApplicationTests.json. This list is just the summary. We'll expand on each point throughout the lab instructions.

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_6`

2: Testing the New Data Contract

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

Since we are introducing a new data contract, we'll start by testing it!

Look at the new data fixtures.

Look at the list.json file. It contains the following JSON array:

```
[
  { "id": 99, "amount": 123.45 },
  { "id": 100, "amount": 1.0 },
  { "id": 101, "amount": 150.0 }
]
```

This is our new data contract containing a list of Cash Cards, matching the data in the new data.sql file; go ahead, look at the data.sql file to verify that the JSON file's values match.

Now open the `CashCardsJsonTest.java` file. Note class-level variable cashCards is configured to contain the following Java array:

```
cashCards = Arrays.array(
     new CashCard(99L, 123.45),
     new CashCard(100L, 100.00),
     new CashCard(101L, 150.00));
```

If you look closely you'll see that the one of the CashCard objects in our test does not match the test data in data.sql. This is to set us up to write a failing test!

Add a serialization test for the Cash Card list.

Add a new test to `CashCardJsonTest.java`:

```
@Test
void cashCardListSerializationTest() throws IOException {
   assertThat(jsonList.write(cashCards)).isStrictlyEqualToJson("list.json");
}
```

The test code is self-explanatory: It serializes the cashCards variable into JSON, then asserts that list.json should contain the same data as the serialized cashCards variable.

Run the tests.

Can you predict whether the test will fail, and if it does, what the cause of the failure will be? Go ahead, make the call! What do you think will happen?

Verify your prediction by running the tests.

Note that we will always run ./gradlew test to run the tests.

```
[~/exercises] $ ./gradlew test
...
> Task :test FAILED
...
java.lang.AssertionError: JSON Comparison failure: [1].amount
Expected: 1.0
     got: 100.0
```

Your prediction was correct (hopefully)! The test failed. Happily, the error message points out the exact spot where the failure occurs: the amount field of the second CashCard in the array (index [1]) isn't what was expected.

Fix and rerun the tests.

Change `cashCards[1].amount` to the correct value (in list.json), and watch the test pass!

```
...
new CashCard(100L, 1.00),
...
```

When you rerun the tests you will see that they pass.

`BUILD SUCCESSFUL in 7s`

Add a deserialization test.

Now let's test deserialization. Add the following test:

```
@Test
void cashCardListDeserializationTest() throws IOException {
   String expected="""
         [
            { "id": 99, "amount": 123.45 },
            { "id": 100, "amount": 100.00 },
            { "id": 101, "amount": 150.00 }
         ]
         """;
   assertThat(jsonList.parse(expected)).isEqualTo(cashCards);
}
```

Again, we have intentionally asserted an incorrect value to make it obvious what the test is testing.

Run the tests.

When you run the tests you will see the incorrect value was caught.

```
[~/exercises] $ ./gradlew test
...
> Task :test FAILED
...
 expected:
   [CashCard[id=99, amount=123.45],
       CashCard[id=100, amount=1.0],
       CashCard[id=101, amount=150.0]]
  but was:
   [CashCard[id=99, amount=123.45],
       CashCard[id=100, amount=100.0],
       CashCard[id=101, amount=150.0]]
```

This time, the test failed because we deserialized the expected JSON String, and compared it to the cashCards variable. Again, that pesky $100.00 Cash Card does not match the expectation.

Change the expectation, rerun, and watch the test pass:

```
String expected="""
    [
        { "id": 99, "amount": 123.45 },
        { "id": 100, "amount": 1.00 },
        { "id": 101, "amount": 150.00 }
    ]
    """;
```

```
[~/exercises] $ ./gradlew test
...
CashCardJsonTest > cashCardListDeserializationTest() PASSED
```

Now that we've tested the data contract, let's move on to the Controller endpoint.

3: Test for an Additional GET Endpoint

Write a failing test for a new GET endpoint.

Let's add a new test method which expects a GET endpoint which returns multiple CashCard objects.

In `CashCardApplicationTests.java`, add a new test:

```
 @Test
 void shouldReturnAllCashCardsWhenListIsRequested() {
     ResponseEntity<String> response = restTemplate.getForEntity("/cashcards", String.class);
     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
 }
```

Here we're making a request to the /cashcards endpoint. Since we're getting the entire list of cards, we don't need to specify any additional information in the request.

Run the tests and observe the failure.

The test fails because we haven't implemented a Controller endpoint to handle this GET request.

How do you think it will fail? Perhaps a 404 NOT FOUND?

In a previous lesson we wrote a test that failed because no endpoint yet existed to match the route being requested. The result was a 404 NOT FOUND error. We might expect the same thing to happen when we run the new test, since we haven't added any code to the Controller.

Let's see what happens. Run the test and search for the following failure:

```
expected: 200 OK
 but was: 405 METHOD_NOT_ALLOWED
```

The error messages don't make it clear why we're receiving a 405 METHOD_NOT_ALLOWED error. The reason is a bit hard to discover, so we'll quickly summarize it: we've already implemented a /cashcards endpoint, but not for a GET verb.

This is Spring's process:

Spring receives a request to the /cashcards endpoint. There is no mapping for the HTTP GET verb at that endpoint. There is, however, a mapping to that endpoint for the HTTP POST verb. It's the endpoint for the Create operation that we implemented in a previous lesson! Therefore, Spring reports a 405 METHOD_NOT_ALLOWED error instead of 404 NOT FOUND -- the route was indeed found, but it doesn't support the GET verb. Implement the GET endpoint in the Controller.

To get past the 405 error, we need to implement the /cashcards endpoint in the Controller using a @GetMapping annotation:

```
@GetMapping()
public ResponseEntity<Iterable<CashCard>> findAll() {
   return ResponseEntity.ok(cashCardRepository.findAll());
}
```

Understand the handler method.

Once again we are using one of Spring Data's built-in implementations: CrudRepository.findAll(). Our implementing Repository, CashCardRepository, will automatically return all CashCard records from the database when findAll() is invoked.

Rerun the tests.

When we run the tests again we see they all pass, including the test for the GET endpoint for a CashCard list.

```
[~/exercises] $ ./gradlew test
...
BUILD SUCCESSFUL in 7s
```

4: Enhance the List Test

As we've done in previous lessons, we've tested that our Cash Card API Controller is "listening" for our HTTP calls and does not crash when invoked, this time for a GET with no further parameters.

Let's enhance our tests and make sure the correct data is returned from our HTTP request.

Enhance the test.

First, let's fill out the test to assert on the expected data values:

```
@Test
void shouldReturnAllCashCardsWhenListIsRequested() {
    ResponseEntity<String> response = restTemplate.getForEntity("/cashcards", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

    DocumentContext documentContext = JsonPath.parse(response.getBody());
    int cashCardCount = documentContext.read("$.length()");
    assertThat(cashCardCount).isEqualTo(3);

    JSONArray ids = documentContext.read("$..id");
    assertThat(ids).containsExactlyInAnyOrder(99, 100, 101);

    JSONArray amounts = documentContext.read("$..amount");
    assertThat(amounts).containsExactlyInAnyOrder(123.45, 100.0, 150.00);
}
```

Understand the test.

```
documentContext.read("$.length()");
...
documentContext.read("$..id");
...
documentContext.read("$..amount");
```

Check out these new JsonPath expressions!

```
documentContext.read("$.length()") calculates the length of the array.

.read("$..id") retrieves the list of all id values returned, while .read("$..amount") collects
all amounts returned.
```

To learn more about JsonPath, a good place to start is here in the JsonPath documentation.

assertThat(...).containsExactlyInAnyOrder(...) We have not guaranteed the order of the CashCard list -- they come out in whatever order the database chooses to return them. Since we don't specify the order, containsExactlyInAnyOrder(...) asserts that while the list must contain everything we assert, the order does not matter.

Run the tests.

What do you think the test result will be?

```
Expecting actual:
  [123.45, 1.0, 150.0]
to contain exactly in any order:
  [123.45, 100.0, 150.0]
elements not found:
  [100.0]
and elements not expected:
  [1.0]
```

The failure message points out exactly the cause of the failure. We've sneakily written a failing test which expects the second Cash Card to have an amount of $100.00, whereas in list.json the actual value is $1.00.

Correct the tests and rerun.

Change the expectation for the $1 Cash Card:

```
assertThat(amounts).containsExactlyInAnyOrder(123.45, 1.00, 150.00);
```

And watch the test pass!

```
[~/exercises] $ ./gradlew test
...
CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() PASSED
...
BUILD SUCCESSFUL in 6s
```

5: Test Interaction and @DirtiesContext

Let's take a moment now to talk about the @DirtiesContext annotation. You'll see three uses of this annotation in the CashCardApplicationTests class: one on the class definition, and two (commented out, for now) on method definitions. Let's explain.

First, comment out the class-level annotation:

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) //@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD) class CashCardApplicationTests { Run all the tests:

[~/exercises] $ ./gradlew test ... org.opentest4j.AssertionFailedError: expected: 3 but was: 4 ... at app//example.cashcard.CashCardApplicationTests.shouldReturnAllCashCardsWhenListIsRequested(CashCardApplicationTests.java:70) Our new shouldReturnAllCashCardsWhenListIsRequested test didn't pass this time! Why?

The reason is that one of the other tests is interfering with our new test by creating a new Cash Card. @DirtiesContext fixes this problem by causing Spring to start with a clean slate, as if those other tests hadn't been run. Removing it (commenting it out) from the class caused our new test to fail.

Learning Moment Although you can use @DirtiesContext to work around inter-test interaction, you shouldn't use it indiscriminately; you should have a good reason. Our reason here is to clean up after creating a new Cash Card.

Leave DirtiesContext commented out at the class level, and uncomment it on the method which creates a new Cash Card:

```
//@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
class CashCardApplicationTests {
   ...

    @Test
    @DirtiesContext
    void shouldCreateANewCashCard() {
   ...
```

Run the tests, and they pass!

6: Pagination

Let's now implement paging, starting with a test!

We have 3 CashCards in our database. Let's set up a test to fetch them one at a time (page size of 1), then have their amounts sorted from highest to lowest (descending).

Write the pagination test.

Add the following test to CashCardApplicationTest, and note that we are adding parameters to the HTTP request of ?page=0&size=1. We will handle these in our Controller later.

```
@Test
void shouldReturnAPageOfCashCards() {
    ResponseEntity<String> response = restTemplate.getForEntity("/cashcards?page=0&size=1",
String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

    DocumentContext documentContext = JsonPath.parse(response.getBody());
    JSONArray page = documentContext.read("$[*]");
    assertThat(page.size()).isEqualTo(1);
}
```

Run the tests.

When we run the tests we should not be surprised that all CashCards are returned.

expected: 1 but was: 3 Implement pagination in the CashCardController.

So, let's add our new endpoint to the Controller! Add the following method to the CashCardController (don't delete the existing findAll() method):

```
@GetMapping
public ResponseEntity<List<CashCard>> findAll(Pageable pageable) {
    Page<CashCard> page = cashCardRepository.findAll(
            PageRequest.of(
                    pageable.getPageNumber(),
                    pageable.getPageSize()
    ));
    return ResponseEntity.ok(page.getContent());
}
```

Understand the pagination code.

findAll(Pageable pageable) Pageable is yet another object that Spring Web provides for us. Since we specified the URI parameters of page=0&size=1, pageable will contain the values we need.

PageRequest.of(pageable.getPageNumber(), pageable.getPageSize())); PageRequest is a basic Java Bean implementation of Pageable. Things that want paging and sorting implementation often support this, such as some types of Spring Data Repositories.

Does our CashCardRepository support Paging and Sorting yet? Let's find out.

Try to compile.

When we run the tests we discover that our code doesn't even compile!

```
[~/exercises] $ ./gradlew test
...
> Task :compileJava FAILED
exercises/src/main/java/example/cashcard/CashCardController.java:50: error: method findAll in
interface CrudRepository<T,ID> cannot be applied to given types;
        Page<CashCard> page = cashCardRepository.findAll(
                                                 ^
  required: no arguments
  found:    PageRequest
```

But of course! We haven't changed the Repository to extend the additional interface. So let's do that. In CashCardRepository.java, also extend PagingAndSortingRepository:

Extend PagingAndSortingRepository and rerun tests.

Update CashCardRepository to also extend PagingAndSortingRepository.

Don't forget to add the new import!

import org.springframework.data.repository.PagingAndSortingRepository; ...

public interface CashCardRepository extends CrudRepository<CashCard, Long>, PagingAndSortingRepository<CashCard, Long> { ... } Now our repository does support Paging and Sorting.

But our tests still fail! Search for the following failure:

[~/exercises] $ ./gradlew test ... Failed to load ApplicationContext java.lang.IllegalStateException: Failed to load ApplicationContext ... Caused by: java.lang.IllegalStateException: Ambiguous mapping. Cannot map 'cashCardController' method example.cashcard.CashCardController#findAll(Pageable) to {GET [/cashcards]}: There is already 'cashCardController' bean method example.cashcard.CashCardController#findAll() mapped. (The actual output is immensely long. We've included the most helpful error message in the output above.)

Understand and resolve the failure.

So what happened? We didn't remove the existing findAll() Controller method.

Why is this a problem (even though we have unique method names and everything compiles!)?

The problem is that we have two methods mapped to the same endpoint. Spring detects this error at runtime, during the Spring startup process.

So let's remove the offending old findAll() method:

```
// Delete this one:
@GetMapping()
public ResponseEntity<Iterable<CashCard>> findAll() {
    return ResponseEntity.ok(cashCardRepository.findAll());
}
```

Run the tests and ensure that they pass.

```
BUILD SUCCESSFUL in 7s
```

Next, let's implement Sorting.

7: Sorting

We'd like the Cash Cards to come back in an order that makes sense to humans. So let's order them by amount in a descending order with the highest amounts first.

Write a test (which we expect to fail).

Add the following test to CashCardApplicationTests:

```
@Test
void shouldReturnASortedPageOfCashCards() {
    ResponseEntity<String> response = restTemplate.getForEntity("/cashcards?
page=0&size=1&sort=amount,desc", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

    DocumentContext documentContext = JsonPath.parse(response.getBody());
    JSONArray read = documentContext.read("$[*]");
    assertThat(read.size()).isEqualTo(1);

    double amount = documentContext.read("$[0].amount");
```

```
    assertThat(amount).isEqualTo(150.00);
}
```

Understand the test.

The URI we are requesting contains both pagination and sorting information: /cashcards?page=0&size=1&sort=amount,desc

page=0: Get the first page. Page indexes start at 0. size=1: Each page has size 1. sort=amount,desc The extraction of data (using more JSONPath!) and accompanying assertions expect that the returned Cash Card is the $150.00 one.

Do you think the test will pass? Before running it, try to figure out whether it will or not. If you think it won't pass, where do you think the failure will be?

Run the test.

```
[~/exercises] $ ./gradlew test
...
org.opentest4j.AssertionFailedError:
 expected: 150.0
  but was: 123.45
```

The test expected to get the $150.00 Cash Card, but it got the $123.45 one. Why?

The reason is that since we didn't specify a sort order, the cards are returned in the order they are returned from the database. And this happens to be the same as the order in which they were inserted.

An important observation: Not all databases will act the same way. It should now make even more sense why we specify a sort order (instead of replying on the database's default order).

Implement sorting in the Controller.

Adding sorting to the Controller code is a super simple single line addition. In the CashCardController class, add an additional parameter to the PageRequest.of() call:

```
PageRequest.of(
     pageable.getPageNumber(),
     pageable.getPageSize(),
     pageable.getSort()
));
```

The getSort() method extracts the sort query parameter from the request URI.

Run the tests again. They pass!

CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() PASSED Wait, write one more test!

To get a little more confidence in the test, let's do an experiment. In the test, change the sort order from descending to ascending:

```
ResponseEntity<String> response = restTemplate.getForEntity("/cashcards?
page=0&size=1&sort=amount,asc", String.class);
```

This should cause the test to fail because the first Cash Card in ascending order should be the $1.00 card. Run the tests and observe the failure:

```
CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED
org.opentest4j.AssertionFailedError:
 expected: 150.0
  but was: 1.0
```

Correct! This result reinforces our confidence in the test. Instead of writing a whole new test, we used an existing one to run a little experiment.

Now let's change the test back to request descending sort order so that it passes again.

8: Paging and Sorting defaults

We now have an endpoint which requires the client to send four pieces of information: The page index and size, and the sort order and direction. This is a lot to ask, so let's make it easier on them.

Write a new test which doesn't send any pagination or sorting parameters.

We'll write a test that expects reasonable defaults for the parameters. The defaults will be:

Sort by amount ascending. A page size of something larger than 3, so that all of our fixtures will be returned.

```
@Test
void shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() {
    ResponseEntity<String> response = restTemplate.getForEntity("/cashcards", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

    DocumentContext documentContext = JsonPath.parse(response.getBody());
    JSONArray page = documentContext.read("$[*]");
    assertThat(page.size()).isEqualTo(3);

    JSONArray amounts = documentContext.read("$..amount");
    assertThat(amounts).containsExactly(1.00, 123.45, 150.00);
}
```

Run the tests. The test failure shows:

All the Cash Cards are being returned, since the (page.size()).isEqualTo(3) assertion succeeded. BUT: They are not sorted since the (amounts).containsExactly(1.00, 123.45, 150.00) assertion fails: Actual and expected have the same elements but not in the same order, at index 0 actual element was: 123.45 whereas expected element was: 1.0 Make the test pass.

Change the implementation by adding a single line to the Controller method:

... PageRequest.of(pageable.getPageNumber(), pageable.getPageSize(), pageable.getSortOr(Sort.by(Sort.Direction.ASC, "amount")))); ... Run the tests, and watch them pass!

Understand the implementation.

So, what just happened?

The answer is that the getSortOr() method provides default values for the page, size, and sort parameters. The default values come from two different sources:

Spring provides the default page and size values (they are 0 and 20, respectively). A default of 20 for page size explains why all three of our Cash Cards were returned. Again: we didn't need to explicitly define these defaults. Spring provides them "out of the box".

We defined the default sort parameter in our own code, by passing a Sort object to getSortOr():

Sort.by(Sort.Direction.ASC, "amount") The net result is that if any of the three required parameters are not passed to the application, then reasonable defaults will be provided.

9: Summary

In this lesson, we implemented a "GET many" endpoint and added sorting and pagination. These accomplished two things:

1. Ensured that the data received from the server is in a predictable and undertstandable order.
2. Protected the client and server from being overwhelmed by a large amount of data (the page size puts a cap on the amount of data that can be returned in a single response).