**Initializing and Developing a Family Cash Card Microservice with Spring Boot**

**Introduction**

Welcome to our first lab activity! In this hands-on session, you'll embark on the journey of creating your own microservice application, "Family Cash Card", using Spring Boot and Docker. This simple yet powerful application is designed to assist parents in managing allowances for their children, making it an effective solution for a common problem.

Using a modern microservice architecture, you'll leverage Spring Boot's capability to build stand-alone, production-grade applications quickly and Docker's containerization to ensure our service runs consistently across platforms. Your task will be to develop a Spring Boot application, starting from Spring Initializr, and progressively integrating key features, such as implementing GET and POST methods, configuring a database, managing data, and working with repositories via Spring Data.

By the end of this lab, you will not only have a functional Family Cash Card application but also a deeper understanding of Spring Boot and Docker within a microservices context. This hands-on project will equip you with practical knowledge and skills that are crucial in the modern development ecosystem.

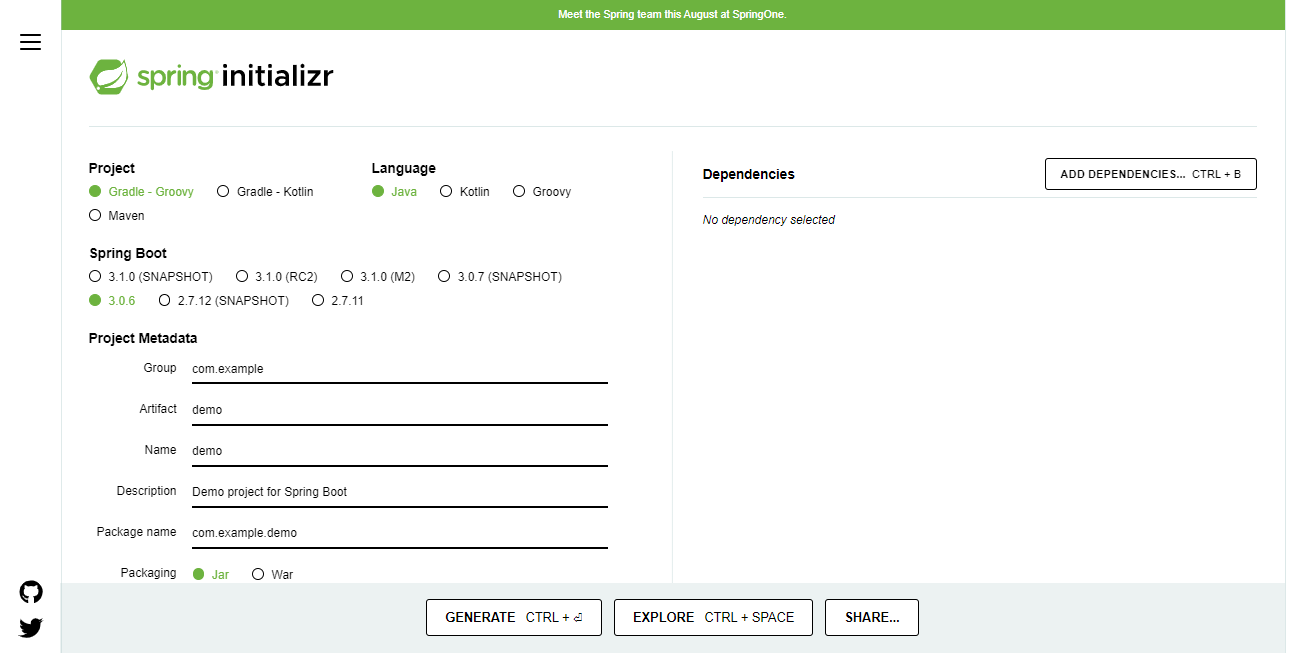
Should you find yourself stuck at any point during the lab, please refer to the solution provided in this GitHub repository: <https://github.com/fenago/cashcard-solution>. Use this as a guide to help you overcome any challenges you might encounter during your development process. However, I encourage you to try to solve the problems on your own first, as this will help enhance your understanding and problem-solving skills.

# Lab: Spring Initializr (You can skip to step 6 to load into gitpod)

1: Spring Initializr

Complete the following steps to use Spring Initializr to set up the Family Cash Card REST API application.

1. Open <https://start.spring.io> : *Spring Initializr*



1. Select the following options:
   * Project: **Gradle - Groovy**
   * Language: **Java**
   * SpringBoot: Choose the latest **3.0X** version (3.0.6)
2. Enter the following values next the corresponding Project Metadata fields:
   * Group: example
   * Artifact: cashcard
   * Name: CashCard
   * Description: CashCard service for Family Cash Cards
   * Packaging: **Jar**
   * Java: **17**

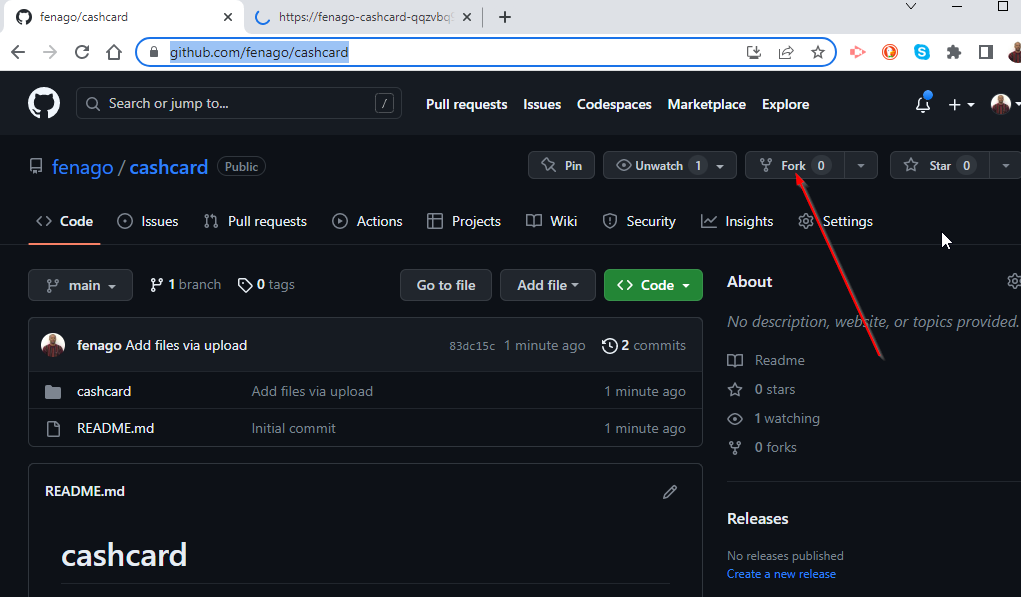
**Note:** You do not have to enter the "Package name" field -- Spring Initializr will fill this in for you!

1. Select the **ADD DEPENDENCIES...** button from the **Dependencies** panel.
2. Select the following option, since we know that we will be creating a web application:
   * Web options: **Spring Web**

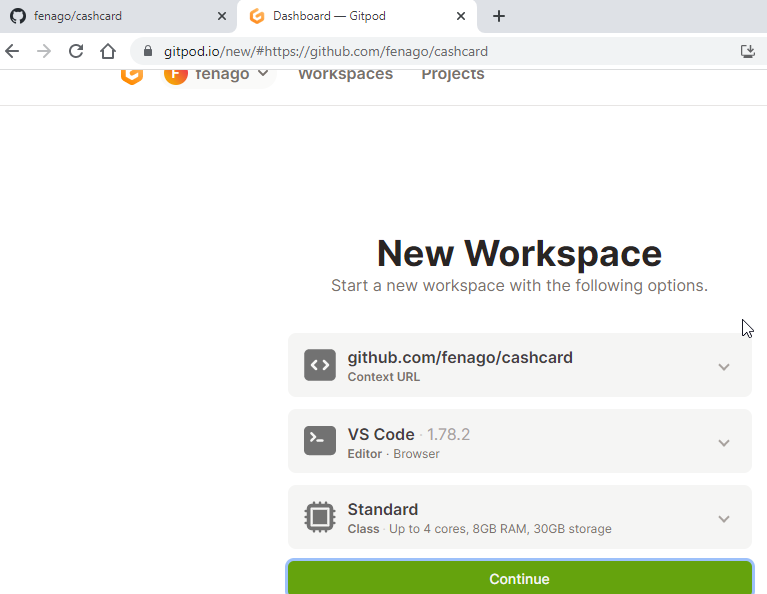
***Later on in the course, you will be adding additional dependencies without using Spring Initializr.***

1. Click the **CREATE** button. Spring Initializr generates a zip file of code and unzips it in your home directory. You can also use this link with the defaults preset: <https://start.spring.io/#!type=gradle-project&language=java&platformVersion=3.0.6&packaging=jar&jvmVersion=17&groupId=example&artifactId=cashcard&name=CashCard&description=CashCard%20service%20for%20Family%20Cash%20Cards&packageName=example.cashcard&dependencies=web>

Shortcut: go here and fork this repo: <https://github.com/fenago/cashcard>



Once forked, go to this URL to start your environment: <https://gitpod.io/#https://github.com/fenago/cashcard>



1. From the command line in the *Terminal* tab, enter the following commands to use the gradle wrapper to build and test the generated application.

Go to the cashcard directory in the *Terminal* dashboard tab.

[~] $ cd cashcard

[~/cashcard] $

You may get a permission error, if this is the case then execute:

[~/cashcard] $ chmod 755 gradlew

[~/cashcard] $ sdk list java

[~/cashcard] $ sdk install java 17.0.7-tem

Next, run the ./gradlew build command:

[~/cashcard] $ ./gradlew build

The output shows that the application passed the tests and was successfully built.

Downloading https://services.gradle.org/distributions/...`

...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%

> Task :test

2022-10-26 20:41:09.313 INFO 329 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown initiated...

2022-10-26 20:41:09.317 INFO 329 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown completed.

BUILD SUCCESSFUL in 45s

7 actionable tasks: 7 executed



Summary

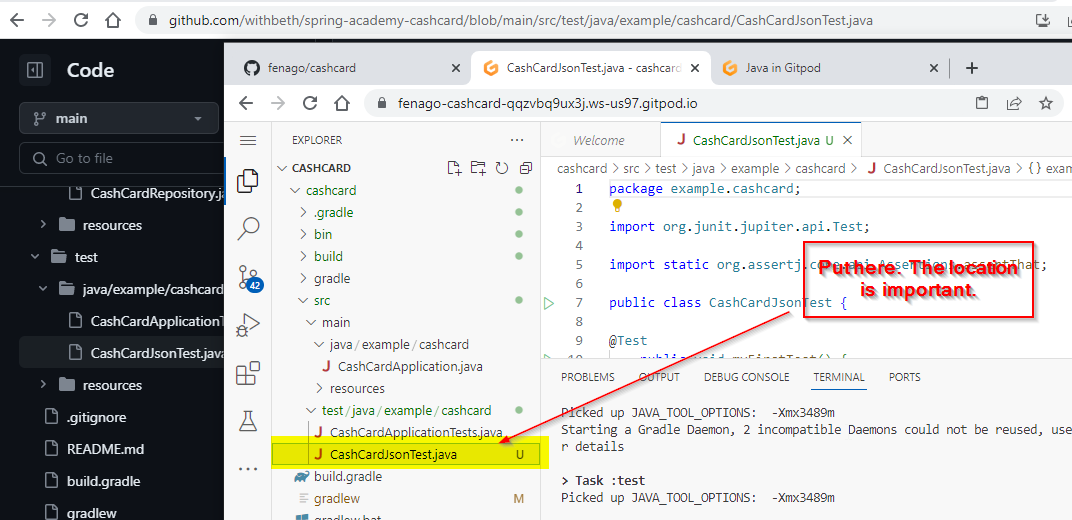
Congrats! You’ve just learned how to quickly and easily bootstrap a Spring Boot codebase using Spring Initializr.

# Lab: Testing First

# Writing a Failing Test

Here we’ll cover a brief introduction to the JUnit test library and the Gradle build tool. We’ll also use the Test-First approach to building software.

Test classes in a standard Java project typically are in the src/test/java/**<package>** directory, where **<package>** is a multi-directory hierarchy if the package is multi-level. For example, in our case the package is example.cashcard, so our test files are in the src/test/java/**example/cashcard** directory.



1. Use the editor to create the file CashCardJsonTest.java in the src/test/java/example/cashcard directory. Start with the simplest thing you can imagine: a single test method with a single statement:

**package** example.cashcard;

**import** org.junit.jupiter.api.Test;

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

**public** **class** **CashCardJsonTest** {

@Test

**public** **void** **myFirstTest**() {

assertThat(1).isEqualTo(42);

}

}

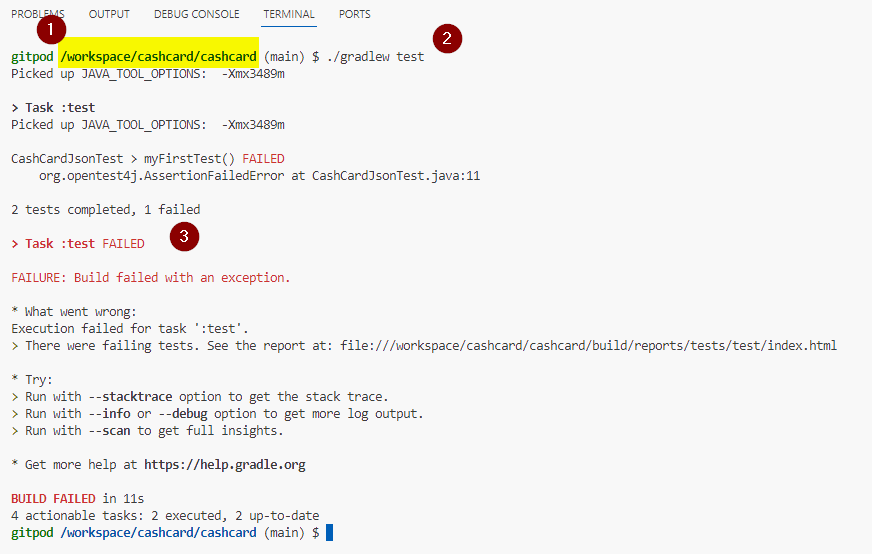
The @Test annotation is part of the JUnit library, and the assertThat method is part of the AssertJ library. Both of these libraries are imported after the package statement.

A common convention (but not a requirement) is to always use the Test suffix for test classes. We’ve done that here. The full class name CashCardJsonTest gives you a clue about the nature of the test we're about to write.

In true Test-First fashion, we've written a failing test first. It's important to have a failing test first so you can have high confidence that whatever you did to fix the test actually worked.

Don't worry that the test (asserting that 1 is equal to 42), as well as the test method name, seem strange. We're about to change them.

1. Run the test from the command line in your terminal (make sure you are in the cashcard directory first):



[~/cashcard] $ ./gradlew test

You should receive output like this (we've omitted some of the less important output):

Task :test

CashCardApplicationTests > contextLoads() PASSED

CashCardJsonTest > myFirstTest() FAILED

org.opentest4j.AssertionFailedError:

expected: 42

but was: 1

...

at app//example.cashcard.CashCardJsonTest.myFirstTest(CashCardJsonTest.java:11)

2 tests completed, 1 failed

This is the expected output from the Gradle build tool when you have a failing test. In this case, your new test failed, whereas the existing CashCardsApplicationTest from the previous lesson succeeded.

The pertinent failure information is towards the top of the output:

expected: 42

but was: 1

You might have expected this, as the number 1 is not equal to the number 42.

In order to "fix" the test, you can assert a statement that you know is true:

assertThat(42).isEqualTo(42);

Now run the test again. It passes!

[~/exercises] $ ./gradlew test

> Task :test

CashCardJsonTest > myFirstTest() PASSED

CashCardApplicationTests > contextLoads() PASSED

BUILD SUCCESSFUL **in** 4s

Congratulations! You’ve successfully completed an iteration of test-first development: Write a failing test, then correct the code so that the test passes. You’re now ready to proceed with using Test-First methodology to write the Cash Card REST API.

# Lab: Implementing GET

# 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);

}

}

1. 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.

1. 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.

[~/cashcard] $ ./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?

1. Understand the test failure.

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

Why is it failing due to an unexpected 404 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.

# Lab: Repositories & Spring Data

# Add files to your environment

### **Test Resource Files**

Add the following files which you will utilize in this lab.

* src/test/resources/schema.sql

/\*

CREATE TABLE cash\_card

(

    ID     BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

    AMOUNT NUMBER NOT NULL DEFAULT 0

);

\*/

* src/test/resources/data.sql

/\*

INSERT INTO CASH\_CARD(ID, AMOUNT) VALUES (99, 123.45);

\*/

* src/main/java/example/cashcard/CashCard.java

package example.cashcard;

public record CashCard(Long id, Double amount) {

}

* 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;

@RestController

@RequestMapping("/cashcards")

public class CashCardController {

    @GetMapping("/{requestedId}")

    public ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {

        if (requestedId.equals(99L)) {

            CashCard cashCard = new CashCard(99L, 123.45);

            return ResponseEntity.ok(cashCard);

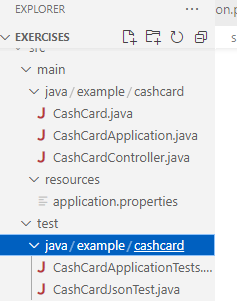
        } else {

            return ResponseEntity.notFound().build();

        }

    }

}



# Review the Current Data Management Pattern

Our Family Cash Card REST API currently relies upon CashCard data hard-coded directly into our CashCardController. Our tests in CashCardApplicationTests assert that this data is present.

We know that a web Controller should not manage data. This is a violation of Separation of Concerns. Web traffic is web traffic, data is data, and healthy software has architectures dedicated to each area.

1. Review CashCardController.

Note lines such as the following:

...

**if** (requestedId == 99L) {

CashCard cashCard = **new** **CashCard**(99L, 123.45);

**return** ResponseEntity.ok(cashCard);

...

This is data management. Our Controller shouldn't be concerned with checking IDs or creating data.

1. Review CashCardApplicationTests.

Interestingly, while our tests make assertions about the data, they don't rely upon or specify how that data is created or managed. This decoupling is important and will help us make the changes we need.

### **Prepare to Refactor to use a Repository and Database**

Refactoring is the act of altering the implementation of a software system without altering its inputs, outputs, or behavior.

Our tests will allow us to change the implementation of our Cash Card API's data management from hard-coded data inside our Controller to utilizing a Repository and database.

This lab is a continuous example of the Red, Green, Refactor development loop.

As we refactor, our tests will periodically fail when we run them. We'll know we've successfully removed all hard-coded data from our Controller and "migrated" that data (and data management) to a database-backed Repository when our tests pass again.

# Add Spring Data Dependencies

This project was originally created using the [Spring Initializr](https://start.spring.io/), which allowed us to automatically add dependencies to our project. However, now we must manually add dependencies to our project.

1. Add dependencies for Spring Data and a database.

In build.gradle

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

testImplementation 'org.springframework.boot:spring-boot-starter-test'

// Add the two dependencies below

implementation 'org.springframework.data:spring-data-jdbc'

testImplementation 'com.h2database:h2'

}

1. Understand the dependencies.

The two dependencies we added are related, but different.

* + implementation 'org.springframework.data:spring-data-jdbc'
  + Spring Data has many implementations for a variety of relational and non-relational database technologies. Spring Data also has several abstractions on top of those technologies. These are commonly called an Object-Relational Mapping framework, or ORM.

Here we'll elect to use Spring Data JDBC. From the Spring Data JDBC [documentation](https://spring.io/projects/spring-data-jdbc):

*Spring Data JDBC aims at being conceptually easy...This makes Spring Data JDBC a simple, limited, opinionated ORM.*

* + testImplementation 'com.h2database:h2'
  + Database management frameworks only work if they have a linked database. H2 is a "very fast, open source, JDBC API" SQL database implemented in Java. It works seamlessly with Spring Data JDBC.
  + Note testImplementation

This tells Spring Boot to make the H2 database available only when running tests. Eventually we'll need a database outside of a testing context, but not yet.

1. Run the tests.

This will both install the dependencies and verify that their addition hasn't broken anything.

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

[~/exercises] $ ./gradlew test

The dependencies are now installed! You might notice additional output compared to previous labs, such as Shutting down embedded database. Spring Auto Configuration is now starting and configuring an H2 database for us to use with tests.

# Create the CashCardRepository

1. Create the CashCardRepository.

Create src/main/java/example/cashcard/CashCardRepository.java and have it extend CrudRepository.

**package** example.cashcard;

**import** org.springframework.data.repository.CrudRepository;

**public** **interface** **CashCardRepository** **extends** **CrudRepository** {

}

1. Understand extend CrudRepository.

This is where we tap into the magic of Spring Data and its data repository pattern.

CrudRepository is an interface supplied by Spring Data. When we extend it (or other sub-Interfaces of Spring Data's Repository), Spring Boot and Spring Data work together to automatically generate the CRUD methods that we need to interact with a database.

We'll use one of these CRUD methods, findById, later in the lab.

1. Run the tests.

We can see that everything compiles, however our application crashes badly upon startup. Digging through the failure messages we find this:

java.lang.IllegalArgumentException: Could not resolve domain type of interface example.cashcard.CashCardRepository

This cryptic error means that we haven't indicated which data object the CashCardRepository should manage. For our application, the "domain type" of this repository will be the CashCard.

1. Configure the CashCardRepository.

Edit the CashCardRepository to specify that it manages the CashCard's data, and that the datatype of the Cash Card ID is Long.

**public** **interface** **CashCardRepository** **extends** **CrudRepository**<CashCard, Long> {

}

1. Configure the CashCard.

When we configure the repository as CrudRepository<CashCard, Long> we indicate that the CashCard's ID is Long. However, we still need to tell Spring Data which field is the ID.

Edit the CashCard class to configure the id as the @Id for the CashCardRepository.

Don't forget to add the new import.

**package** example.cashcard;

// Add this import

**import** org.springframework.data.annotation.Id;

**public** **record** **CashCard**(@Id Long id, Double amount) {

}

1. Run the tests.

The tests pass, but we haven't made any meaningful changes to the code...yet!

# Inject the CashCardRepository

Although we've configured our CashCard and CashCardRepository classes, we haven't utilized the new CashCardRepository to manage our CashCard data. Let's do that now.

1. Inject the CashCardRepository into CashCardController.

Edit CashCardController to accept a CashCardRepository.

@RestController

@RequestMapping("/cashcards")

**public** **class** **CashCardController** {

**private** CashCardRepository cashCardRepository;

**public** **CashCardController**(CashCardRepository cashCardRepository) {

this.cashCardRepository = cashCardRepository;

}

...

1. Run the tests.

If you run the tests now, they'll all pass, despite no other changes to the codebase utilizing the new, required constructor CashCardController(CashCardRepository cashCardRepository).

BUILD SUCCESSFUL in 7s

So how is this possible?

1. Behold Auto Configuration and Construction Injection!

Spring's Auto Configuration is utilizing its dependency injection (DI) framework, specifically constructor injection, to supply CashCardController with the correct implementation of CashCardRepository at runtime.

Magical stuff!

### **Learning Moment**

1. Temporarily change the CashCardRepository to remove the implementation of CrudRepository.
2. **public** **interface** **CashCardRepository** {
3. }
4. Compile the project and note the failure.
5. [~/exercises] $ ./gradlew build

Searching through the output, we find this line:

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'example.cashcard.CashCardRepository' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {}

Clues such as NoSuchBeanDefinitionException, No qualifying bean, and expected at least 1 bean which qualifies as autowire candidate tell us that Spring is trying to find a properly configured class to provide during the dependency injection phase of Auto Configuration, but none qualify. We can satisfy this DI requirement by implementing the CrudRepository.

Be sure to undo the temporary changes to CashCardRepository before moving on.

# Use the CashCardRepository for Data Management

You are finally ready to use the CashCardRepository!

1. Find the CashCard using findById

The CrudRepository interface provides [many helpful methods](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findById-ID-), including findById(ID id).

Update the CashCardController to utilize this method on the CashCardRepository and update the logic; be sure to import java.util.Optional;

**import** java.util.Optional;

...

@GetMapping("/{requestedId}")

**public** ResponseEntity<CashCard> **findById**(@PathVariable Long requestedId) {

Optional<CashCard> cashCardOptional = cashCardRepository.findById(requestedId);

**if** (cashCardOptional.isPresent()) {

**return** ResponseEntity.ok(cashCardOptional.get());

} **else** {

**return** ResponseEntity.notFound().build();

}

}

1. Understand the changes.

We've just altered the CashCardController.findById in several important ways.

* + Optional<CashCard> cashCardOptional = cashCardRepository.findById(requestedId);
  + We're calling CrudRepository.findById which returns an Optional. This smart object might or might not contain the CashCard for which we're searching. Learn more about Optional [here](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html).
  + cashCardOptional.isPresent()
  + and
  + cashCardOptional.get()

This is how you determine if findById did or did not find the CashCard with the supplied id.

If cashCardOptional.isPresent() is true then the repository successfully found the CashCard and we can retrieve it with cashCardOptional.get().

If not, the repository has not found the CashCard.

1. Run the tests.

We can see that the tests fail with a 500 INTERNAL\_SERVER\_ERROR.

CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED

org.opentest4j.AssertionFailedError:

expected: 200 OK

but was: 500 INTERNAL\_SERVER\_ERROR

This means the Cash Card API "crashed".

We need a bit more information. Let's temporarily update the test output section of build.gradle with showStandardStreams = true so our test runs will produce a lot more output.

test {

testLogging {

events "passed", "skipped", "failed" //, "standardOut", "standardError"

showExceptions true

exceptionFormat "full"

showCauses true

showStackTraces true

// Change from false to true

showStandardStreams = true

}

}

1. Rerun the tests.

Note that the test output is much more verbose.

Searching through the output we find these failures:

org.h2.jdbc.JdbcSQLSyntaxErrorException: Table "CASH\_CARD" not found (this database is empty); SQL statement:

SELECT "CASH\_CARD"."ID" AS "ID", "CASH\_CARD"."AMOUNT" AS "AMOUNT" FROM "CASH\_CARD" WHERE "CASH\_CARD"."ID" = ? [42104-214]

The cause of our test failures is clear: Table "CASH\_CARD" not found means we don't have a database nor any data.

# Configure the Database

Our tests expect the API to find and return a CashCard with id of 99. But, we just removed the hard-coded CashCard data and replaced it with a call to cashCardRepository.findById.

Now our application is crashing, complaining about a missing database table named CASH\_CARD.

org.h2.jdbc.JdbcSQLSyntaxErrorException: Table "CASH\_CARD" not found (this database is empty);

We need to help Spring Data configure the database and load some sample data, such as our friend, CashCard 99.

Spring Data and H2 can automatically create and populate the in-memory database we need for our test. We've provided these files for you here, but you'll need to amend them.

1. Edit schema.sql.

Spring Data will automatically configure a database by tests if we provide src/test/resources/schema.sql.

And we have! But, it's currently disabled.

Edit src/test/resources/schema.sql and remove the block-comment /\* ... \*/.

**CREATE** **TABLE** cash\_card

(

ID BIGINT GENERATED **BY** **DEFAULT** **AS** **IDENTITY** **PRIMARY** KEY,

AMOUNT NUMBER **NOT** **NULL** **DEFAULT** 0

);

1. Understand schema.sql.

A database schema is a "blueprint" for how data is stored in a database. We won't cover database schemas in depth here.

Our database schema reflects the CashCard object that we understand, which contains an id and an amount.

1. Rerun the tests.

**Note:** If the test output is too verbose, revert the change in build.gradle performed previously.

Our tests no longer crash with a 500 INTERNAL\_SERVER\_ERROR. However, now we get a 404 NOT\_FOUND

CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED

org.opentest4j.AssertionFailedError:

expected: 200 OK

but was: 404 NOT\_FOUND

Translation: our repository can't find CashCard with id of 99. So why not?

Although we've helped Spring Data create a test database by un-commenting schema.sql, it's still an empty database.

Let's go load some data!

1. Load test data from data.sql.

Not only can Spring Data create our test database, but it can also load data into it, which we can use in our tests.

Similar to schema.sql, we've provided src/test/resources/data.sql, but its contents are commented-out.

Let's remove the block comments in src/test/resources/data.sql.

**INSERT** **INTO** CASH\_CARD(ID, AMOUNT) **VALUES** (99, 123.45);

This SQL statement inserts a row into the CASH\_CARD table with an ID=99 and AMOUNT=123.45, which matches the values we expect in our tests.

1. Rerun the tests.

They pass! Woo hoo!

BUILD SUCCESSFUL in 7s

# Summary

You've now successfully refactored the way the Family Cash Card API manages its data. Spring Data is now creating an in-memory H2 database and loading it with test data, which our tests utilize to exercise our API.

Furthermore, we didn't change any of our tests! They actually guided us to a correct implementation. How awesome is that?!

# Lab: Implementing POST

# 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.

1. 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);

}

1. 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.

1. 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.

# 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?

1. 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.

1. 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.

# 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](https://www.rfc-editor.org/rfc/rfc7231)) for guidance as to how our API should behave.

For our POST endpoint, review this section about [HTTP POST](https://www.rfc-editor.org/rfc/rfc7231#section-4.3.3); 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.

1. 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);

}

1. 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](https://www.rfc-editor.org/rfc/rfc7231#section-7) named "Location".

Note that URI is indeed the correct entity here and not a URL; a [URL is a type of URI](https://www.w3.org/TR/uri-clarification/#contemporary), 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.

1. Run the tests.

Unsurprisingly, they fail on the first changed assertion.

expected: 201 CREATED

but was: 200 OK

Let's start fixing stuff!

# Implement the POST Endpoint

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

1. 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();

}

1. 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

1. 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.

# 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!

# 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.

# Final Testing and Learning Moment

1. 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.

1. 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.

1. 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

}

}

1. 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.

1. 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.

# 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!

# Lab: Returning a list with GET

Changes from the Previous Lab

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

1. Added a couple more Cash Card data fixtures to data.sql

INSERT INTO CASH\_CARD(ID, AMOUNT) VALUES (99, 123.45);

INSERT INTO CASH\_CARD(ID, AMOUNT) VALUES (100, 1.00);

INSERT INTO CASH\_CARD(ID, AMOUNT) VALUES (101, 150.00);

1. Refactored CashCardJsonTest.java to incorporate the new data fixtures.

package example.cashcard;

import org.assertj.core.util.Arrays;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.json.JsonTest;

import org.springframework.boot.test.json.JacksonTester;

import java.io.IOException;

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

@JsonTest

public class CashCardJsonTest {

    @Autowired

    private JacksonTester<CashCard> json;

    @Autowired

    private JacksonTester<CashCard[]> jsonList;

    private CashCard[] cashCards;

    @BeforeEach

    void setUp() {

        cashCards = Arrays.array(

                new CashCard(99L, 123.45),

                new CashCard(100L, 100.00),

                new CashCard(101L, 150.00));

    }

    @Test

    public void cashCardSerializationTest() throws IOException {

        CashCard cashCard = cashCards[0];

        assertThat(json.write(cashCard)).isStrictlyEqualToJson("single.json");

        assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.id");

        assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.id")

                .isEqualTo(99);

        assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.amount");

        assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.amount")

                .isEqualTo(123.45);

    }

    @Test

    public void cashCardDeserializationTest() throws IOException {

        String expected = """

                {

                    "id": 99,

                    "amount": 123.45

                }

                """;

        assertThat(json.parse(expected))

                .isEqualTo(new CashCard(99L, 123.45));

        assertThat(json.parseObject(expected).id()).isEqualTo(99);

        assertThat(json.parseObject(expected).amount()).isEqualTo(123.45);

    }

}

1. Add or rename expected.json to single.json (in src/test/resources/example), and added another data contract JSON file: list.json.

list.json

[

  {"id": 99, "amount": 123.45 },

  {"id": 100, "amount": 1.00 },

  {"id": 101, "amount": 150.00 }

]

single.json

{

  "id": 99,

  "amount": 123.45

}

1. Added some imports to the Test classes, so you don't have to!
2. Added the @DirtiesContext annotation to CashCardApplicationTests.json.

CashCardApplicationTests.java

package example.cashcard;

import com.jayway.jsonpath.DocumentContext;

import com.jayway.jsonpath.JsonPath;

import net.minidev.json.JSONArray;

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 org.springframework.test.annotation.DirtiesContext;

import java.net.URI;

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

import static org.springframework.test.annotation.DirtiesContext.\*;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@DirtiesContext(classMode = ClassMode.AFTER\_EACH\_TEST\_METHOD)

class CashCardApplicationTests {

    @Autowired

    TestRestTemplate restTemplate;

    @Test

    //@DirtiesContext

    void shouldReturnACashCardWhenDataIsSaved() {

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

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

        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);

    }

    @Test

    void shouldNotReturnACashCardWithAnUnknownId() {

        ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/1000", String.class);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT\_FOUND);

        assertThat(response.getBody()).isBlank();

    }

    @Test

    //@DirtiesContext

    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);

        DocumentContext documentContext = JsonPath.parse(getResponse.getBody());

        Number id = documentContext.read("$.id");

        Double amount = documentContext.read("$.amount");

        assertThat(id).isNotNull();

        assertThat(amount).isEqualTo(250.00);

    }

}

CashCardJsonTest.java

package example.cashcard;

import org.assertj.core.util.Arrays;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.json.JsonTest;

import org.springframework.boot.test.json.JacksonTester;

import java.io.IOException;

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

@JsonTest

public class CashCardJsonTest {

    @Autowired

    private JacksonTester<CashCard> json;

    @Autowired

    private JacksonTester<CashCard[]> jsonList;

    private CashCard[] cashCards;

    @BeforeEach

    void setUp() {

        cashCards = Arrays.array(

                new CashCard(99L, 123.45),

                new CashCard(100L, 100.00),

                new CashCard(101L, 150.00));

    }

    @Test

    public void cashCardSerializationTest() throws IOException {

        CashCard cashCard = cashCards[0];

        assertThat(json.write(cashCard)).isStrictlyEqualToJson("single.json");

        assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.id");

        assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.id")

                .isEqualTo(99);

        assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.amount");

        assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.amount")

                .isEqualTo(123.45);

    }

    @Test

    public void cashCardDeserializationTest() throws IOException {

        String expected = """

                {

                    "id": 99,

                    "amount": 123.45

                }

                """;

        assertThat(json.parse(expected))

                .isEqualTo(new CashCard(99L, 123.45));

        assertThat(json.parseObject(expected).id()).isEqualTo(99);

        assertThat(json.parseObject(expected).amount()).isEqualTo(123.45);

    }

}

This list is just the summary. We'll expand on each point throughout the lab instructions.

# Testing the New Data Contract

As we've done in previous workshops, 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!

1. 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!

1. 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.

1. 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.

1. 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

1. 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.

1. 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.

Test for an Additional GET Endpoint

1. 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.

1. 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:

* 1. Spring receives a request to the /cashcards endpoint.
  2. There is no mapping for the HTTP GET verb at that endpoint.
  3. 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!
  4. 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.

1. 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());

}

1. 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.

1. 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

# 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.

1. 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);

}

1. 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](https://github.com/json-path/JsonPath).

* + 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.

1. 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.

1. 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

# 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 two methods which create new cards:

//@DirtiesContext(classMode = ClassMode.AFTER\_EACH\_TEST\_METHOD)

**class** **CashCardApplicationTests** {

...

@Test

@DirtiesContext

**void** **shouldCreateANewCashCard**() {

...

@Test

@DirtiesContext

**void** **shouldReturnACashCardWhenDataIsSaved** () {

Run the tests, and they pass!

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).

1. 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);

}

1. Run the tests.

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

expected: 1

but was: 3

1. 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());

}

1. 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.

1. 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:

1. 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.)

1. 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.

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.

1. 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);

}

1. 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?

1. Run the test.
2. [~/exercises] $ ./gradlew test
3. ...
4. org.opentest4j.AssertionFailedError:
5. expected: 150.0
6. 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).

1. 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

1. 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.

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.

1. 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

1. 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!

1. 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.

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).