Lab: Repositories & Spring Data

1: Changes from Previous Labs

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

Test Resource Files

We have provided the following files which you will utilize in this lab.

`src/test/resources/schema.sql`

`src/test/resources/data.sql`

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_4`

2: 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.

2. 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 we learned about in a previous lesson.

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.

3: Add Spring Data Dependencies

This project was originally created using the `Spring Initializr`, 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'
}
```

2. 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:

     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.

3. 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.

4: 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 {
}
```

2. 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.

3. 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.

4. 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> {
}
```

5. 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) {
}
```

6. Run the tests.

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

5: 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;
   }
   ...
```

2. 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?

3. 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.

```
public interface CashCardRepository {
}
```

Compile the project and note the failure.

`[~/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.

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

6: 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, 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();
     }
 }
```

2. 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.

`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.

3. 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
 }
}
```

4. 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.

7: 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.

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
);
```

2. 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.

3. 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!

4. 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.

5. Rerun the tests.

They pass! Woo hoo!

```
BUILD SUCCESSFUL in 7s
```

8: 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?!