Lab: Testing First

1: 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.

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

   ```
   [~/exercises] $ ./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.

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

   `assertThat(42).isEqualTo(42);`

4. 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.

2: Testing the Data Contract

Now, let's write a test that makes sense for your goal: write the CashCard REST API.

1. Replace the very simple test from the previous exercise with a more comprehensive test, so that the test file looks like this:

```
package example.cashcard;

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;
```

```java
@JsonTest
public class CashCardJsonTest {

    @Autowired
    private JacksonTester<CashCard> json;

    @Test
    public void cashCardSerializationTest() throws IOException {
        CashCard cashCard = new CashCard(99L, 123.45);
        assertThat(json.write(cashCard)).isStrictlyEqualToJson("expected.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);
    }
}
```

The @JsonTest annotation marks the CashCardJsonTest as a test class which uses the Jackson framework (which is included as part of Spring). This provides extensive JSON testing and parsing support. It also establishes all the related behavior to test JSON objects.

- `JacksonTester` is a convenience wrapper to the Jackson JSON parsing library. It handles serialization and deserialization of JSON objects.
- `@Autowired` is an annotation that directs Spring to create an object of the requested type.

2. Run the test.

Run the cashCardSerializationTest() that you just created to test the serialization of the CashCard class.

```
[~/exercises] $ ./gradlew test

> Task :compileTestJava
/home/eduk8s/src/test/java/example/cashcard/CashCardJsonTest.java:16: error: cannot
find symbol
    private JacksonTester<CashCard> json;
                          ^
  symbol:   class CashCard
  location: class CashCardJsonTest
...
> Task :compileTestJava FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':compileTestJava'.
> Compilation failed; see the compiler error output for details.
It’s no surprise that the test failed as a CashCard class does not yet exist. Let’s
create one now.
```

3. Create the `CashCard` class.

To create a CashCard class and the constructor that's used in the cashCardSerializationTest() test, create the file `src/main/java/example/cashcard/CashCard.java` with the following contents (notice that this file is under in the src/main directory, not the src/test directory):

```
package example.cashcard;

public record CashCard(Long id, Double amount) {
}
```

4. Re-run the test.

You should receive a failure because the expected.json file does not exist yet:

```
[~/exercises] $ ./gradlew test
...
CashCardJsonTest > cashCardSerializationTest() FAILED
    java.lang.IllegalStateException: Unable to load JSON from class path resource
[example/cashcard/expected.json]
...
        at
example.cashcard.CashCardJsonTest.cashCardSerializationTest(CashCardJsonTest.java:21)

        Caused by:
        java.io.FileNotFoundException: class path resource
[example/cashcard/expected.json] cannot be opened because it does not exist
...
2 tests completed, 1 failed

> Task :test FAILED
The quickest way to add a new expected.json file is to:
```

5. Create the Cash Card contract file.

Create a file called `expected.json` in `src/test/resources/example/cashcard/expected.json`, with the following content:

`{}`

Note that you'll need to create the directory structure to contain the new file. One way to do this is as follows:

- Right-click on the test portion of the src/test folder in the editor.
- Select New File...
- In the dialog enter resources/example/cashcard/expected.json as the file name.
- Edit the newly-created expected.json file, and enter {} as the only contents.

We are purposely only including an empty JSON document {}, which will cause the test to fail.

6. Run the test.

The test fails again, but this time it's because of a comparison failure. The error messages indicate two fields are missing:

```
CashCardJsonTest > cashCardSerializationTest() FAILED
    java.lang.AssertionError: JSON Comparison failure:
    Unexpected: amount
     ;
    Unexpected: id
        at
example.cashcard.CashCardJsonTest.cashCardSerializationTest(CashCardJsonTest.java:21)
```

7. Complete the Data Contract.

To add data contract fields to the `src/test/resources/example/cashcard/expected.json` JSON file, change the content of the expected.json file to the following:

```
{
  "id": 99,
  "amount": 123.45
}
```

8. Run the test again.

It passes now that the assertions match the contract file.

Congratulations! You've now implemented TDD to create a data contract for your CashCard API.

3: Testing Deserialization

Deserialization is the reverse process of serialization. It transforms data from a file or byte stream back into an object for your application. This makes it possible for an object serialized on one platform to be deserialized on a different platform. For example, your client application can serialize an object on Windows while the backend would deserialize it on Linux.

Serialization and deserialization work together to transform/recreate data objects to/from a portable format. The most popular data format for serializing data is JSON.

Let's write a second test to deserialize data so that it converts from JSON to Java after the first test passes. This test uses a test-first technique where you purposely write a failing test. Specifically: the values for id and amount are not what you expect.

1. In the file `src/test/java/example/cashcard/CashCardJsonTest.java`, add a test:

```
@Test
public void cashCardDeserializationTest() throws IOException {
   String expected = """
           {
               "id":99,
               "amount":123.45
           }
           """;
   assertThat(json.parse(expected))
           .isEqualTo(new CashCard(1000L, 67.89));
   assertThat(json.parseObject(expected).id()).isEqualTo(1000);
   assertThat(json.parseObject(expected).amount()).isEqualTo(67.89);
}
```

2. Run the test.

The test fails:

```
CashCardJsonTest > cashCardDeserializationTest() FAILED
 org.opentest4j.AssertionFailedError:
```

```
 expected: CashCard[id=1000, amount=67.89]
  but was: CashCard[id=99, amount=123.45]
```

3. Correct the erroneous values for id and amount.

You can correct them one at a time. Be sure to re-run the test after each edit.

Here is what the corrected assertions now look like:

```
...
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);
...
```

That's it! You now have a working pair of `serialization/deserialization` tests.

Summary In this lesson you learned about JSON and its importance for modern apps. You also learned how JSON is used in the CashCard application. Finally, you learned the benefits of the test-first approach to software development, then exercised test-first development by testing the JSON data contract for the Cash Card service.