TEST WITH REST-ASSURED

AGENDA

- Concept Study
- REST Assured
- Unit Testing
- Integration Testing
- Requests and Tests for GET, POST, PUT, PATCH, DELETE
- Parameterization and Data-Driven Testing
- Data-Driven Testing Example with Ms Excel
- Conclusion

Concept Study

The Spring team advocates test-driven development (TDD). Testing is an integral part of enterprise software development.

REST-assured was designed to simplify the testing and validation of REST APIs and is highly influenced by testing techniques used in dynamic languages such as Ruby and Groovy.

The library has solid support for HTTP, starting of course with the verbs and standard HTTP operations, but also going well beyond these basics. In this guide, we are going to explore REST-assured and we're going to use it for unit testing as well as integration testing

REST Assured

Testing and validating REST services in Java is harder than in dynamic languages such as Ruby and Groovy. REST Assured brings the simplicity of using these languages into the Java domain.

REST Assured is a Java DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests. Rest-Assured library also provides the ability to validate the HTTP Responses received from the server. For e.g. we can verify the Status code, Status message, Headers and even the Body of the response. This makes Rest-Assured a very flexible library that can be used for testing.

REST Assured

❑ Maven Dependency

For Using REST-Assured with Spring boot, we need those dependencies

- **spring-boot-starter-web**
- **io.rest-assured for REST Assured. It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**
- **org.hamcrest uses existing predicates – called matcher classes – for making assertions. It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**

Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing ensures that API components will function properly.

Here are some characteristics of Unit Testing

- **Designed to test specific sections of code**
- **Percentage of lines of code tested is code coverage**
- **Should be ‘unity’ and execute very fast**
- **Should have no external dependencies i.e. no database, no Spring context, etc.**

Unit Testing

Both Testng and Junit are Testing framework used for Unit Testing. TestNG is similar to JUnit. Few more functionalities are added to it that makes TestNG more powerful than Junit. TestNG is a testing framework inspired by JUnit and NUnit.

Here is the table that shows the features supported by JUnit and TestNG.

	JUnit4	TestNG
Annotation Support	✔	✔
Suite Test	✔	✔
Ignore Test	✔	✔
Exception Test	✔	✔
Timeout	✔	✔
Parameterized Test	✔	✔
Dependency Test	✘	✔

Unit Testing

Both JUnit and TestNG uses annotations and almost all the annotations looks similar.

Regarding to these comparisons above, we have decided to work with TestNG instead of Junit. So we will add another dependency for TestNG

- **org.testing , It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**

❑ Unit Testing Example

We want to implement a Unit Test using the GET Method with a public API. We will proceed as follow

- ✓ **Create a Test class(Test01_GET) to implement 2 tests methods**
- ✓ **Test by launching all test using IntelliJ**

Unit Testing

- Unit Testing Example

```
================================================
Default Suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
================================================


Process finished with exit code 0
```

Integration Testing

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved. Designed to test behaviors between objects and parts of the overall system

Ideally, we should keep the integration tests separated from the unit tests and should not run along with the unit tests. Here are some characteristics of Integration Testing

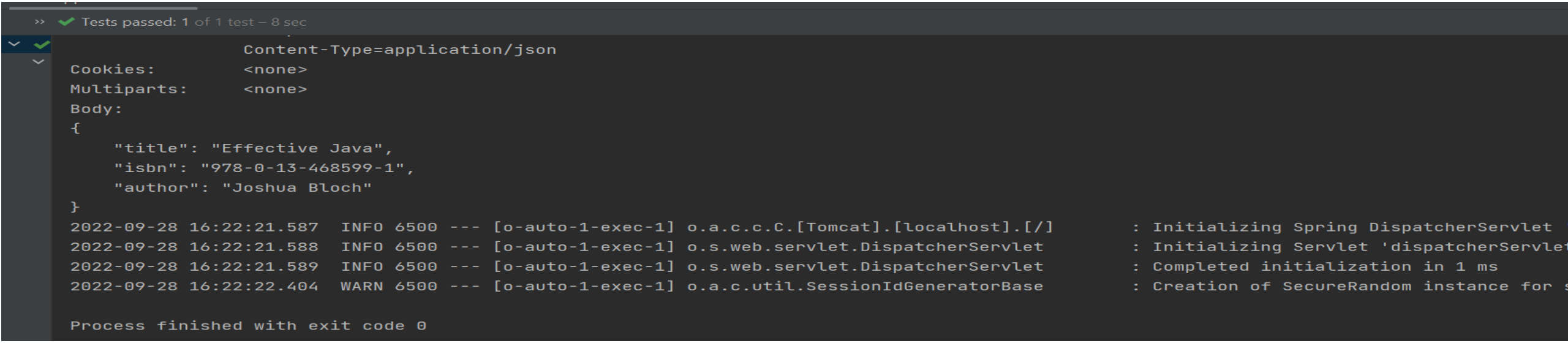
- **Much larger scope**
- **Can include the Spring Context, database, and message brokers**
- **Will run much slower than unit tests**

Integration Testing

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved.

Ideally, we should keep the integration tests separated from the unit tests and should not run along with the unit tests.

For quickly implement the integration testing, we create a little book management application and we do integration testing on ShouldCreateBook (). Here is the result



Requests and Tests for GET, POST, PUT, PATCH, DELETE

❑ GET

For this Request/Test, we proceed as follow

- ✓ Create a Test class(Tests_GET) to implement 3 tests methods **(each method is a different manner)**
- ✓ Test by launching all test using IntelliJ

```
===============================================
Default Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Requests and Tests for GET, POST, PUT, PATCH, DELETE

❑ POST

For our POST Requests and Tests, we need to add another dependency because we want to work with JSON. So we have decided to work with Json Simple dependency

- **com.googlecode.json-simple for Json Simple. It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**

We will then proceed as follow

✓ Create a Test class(Tests_POST) to implement 1 tests method

✓ Test by launching the test using IntelliJ

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Requests and Tests for GET, POST, PUT, PATCH, DELETE

❑ PUT

For our PUT Requests and Tests, we need to add another dependency because we want to work with JSON. So we have decided to work with Json Simple dependency

- **com.googlecode.json-simple for Json Simple. It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**

We will then proceed as follow

✓ Create a Test class(Tests_PUT) to implement 1 tests method

✓ Test by launching the test using IntelliJ

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Requests and Tests for GET, POST, PUT, PATCH, DELETE

❑ DELETE

For our DELETE Requests and Tests, We will proceed as follow

- ✓ Create a Test class(Tests_DELETE) to implement 1 tests method
- ✓ Test by launching the test using IntelliJ

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Requests and Tests for GET, POST, PUT, PATCH, DELETE

❑ PATCH

For our PATCH Requests and Tests, we need to add another dependency because we want to work with JSON. So we have decided to work with Json Simple dependency

- **com.googlecode.json-simple for Json Simple. It is not available on https://start.spring.io/, we need to add it manually from the maven repository.**

We will then proceed as follow

✓ Create a Test class(Tests_PATCH) to implement 1 tests method

✓ Test by launching the test using IntelliJ

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Parameterization and Data-Driven Testing

❑ Data-Driven Testing

For Data-Driven Testing, TestNG provide us a DataProvider. Here are some characteristics of this DataProvider.

- **Helps to write data driven tests**
- **Same test can be run multiple times with diff sets of data**
- **Annotation @DataProvider**
- **The annotated method is used to return object containing test data**
- **This test data can be used in actual tests**

To quickly implement it, we will install a full fake REST API with zero coding in less than 30 seconds by follow this link https://github.com/typicode/json-server

We can apply Data-Driven Testing for every Http methods such as POST, GET, PUT, PATCH, and DELETE. In our case, we will just use the POST method. We will proceed as follow

- ✓ **Create 2 class: DataDrivenExample class for our Test and DataForTest for our data**
- ✓ **Test by launching the test using IntelliJ**

Parameterization and Data-Driven Testing

❑ Data-Driven Testing

```
{
    "firstName": "Henry",
    "lastName": "Ford",
    "subjectId": 2
}

================================================
Default Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
================================================


Process finished with exit code 0
```

Parameterization and Data-Driven Testing

- Parameterization

Parameterization is a little bit similar to DataProvider, it is useful when you have a small amount of Data. But in the age of Big Data, we don’t need it anymore.

Data-Driven Testing Example with Ms Excel

Ms Excel is very useful for every company, because It has been used for a while. The integration of Excel with Spring boot is very useful for us when we want to build software for company. Because all of them have already have some data in form of Excel files, and we want to migrate these data into their information system.

In this part, we will cover how to test our Spring boot API when working with Excel. For this we will proceed as follow

- **Add maven dependency for excel reading and writing (poi-ooxml)**
- **Create a folder and add excel file and create data in the file**
- **Create a class and create functions to get row count**
- **Create a function to get data from excel**
- **Create a constructor to get excelPath and sheetName**
- **Create a new class and call the excel functions**

Data-Driven Testing Example with Ms Excel

Ms Excel is very useful for every company, because It has been used for a while. The integration of Excel with Spring boot is very useful for us when we want to build software for company. Because all of them have already have some data in form of Excel files, and we want to migrate these data into their information system.

In this part, we will cover how to test our Spring boot API when working with Excel. For this we will proceed as follow

- **Add maven dependency for excel reading and writing (poi-ooxml)**
- **Create a folder and add excel file and create data in the file**
- **Create a class and create functions to get row count**
- **Create a function to get data from excel**
- **Create a constructor to get excelPath and sheetName**
- **Create a new class and call the excel functions**

Data-Driven Testing Example with Ms Excel

Now, we can do our test and get this result

```
No of rows : 3
Fomatted value : Bell

Process finished with exit code 0
```

Conclusion

We have learned how to do testing of our API with REST-Assured. The links of our resources and GitHub repository are attached below

https://www.youtube.com/watch?v=JJ7Tp7_fX4c&list=PLhW3qG5bs-L8dDZPP1tpQldU4tzZdPfRv

https://www.youtube.com/watch?v=zEkI8xi3Mjs&list=PLFjB4VDnlT_3qpIsrd-hwBtMSJRhAuayu&index=3

https://www.baeldung.com/spring-boot-testing

https://github.com/typicode/json-server

https://github.com/Dilane-Kamga/TEST-WITH-REST-ASSURED.git

MERCI

Pour votre attention