Example of creating Cucumber based BDD tests using JUnit5 and Spring Dependency Injection

2021-10-12 23:00:00 +0530

# Introduction

I am very passionate about testing. Have been so for very many years now. I am a firm believer of the idea that developers themselves should take responsibility of writing high quality tests. And the tests should not be limited to *Unit Tests*, but should also have *Integration Tests*, touching all layers of the Application. Gone are the days when testing was frowned upon by the developer community, and was the solely, exclusively only owned by the so called *Testing Team*. As we work in more Agile environment, where frequent releases is the norm, *Testing*, especially, *Integration Testing* has become an important tool to make this happen with adequate confidence. The norm is to now run our Integration Tests in our CI/CD Pipeline, in an automated way, so that we receive instant feedback.

BDD is one of the best ways to describe the behavior of an Application that is crisp and easy to understand. I use BDD in most of my Application these days along with Cucumber. In this post, I will not harp about the merits of either, but just assume people are already familiar with these terms.

What we will be discussing today is how to write elegant Integration Tests using Cucumber, JUnit5 and Spring for Dependency Injection.

## Cucumber and JUnit5

I have been using Cucumber for close to 5 years. And I have seen it evolve. The one issue I have always felt recently is the apparent lack of support for JUnit5. Of course this past year things have improved a lot, but I remember, I had to struggle to run my Cucumber tests on JUnit5 exclusively. The reason is scant documentation.

## Cucumber and Dependency Injection

Are you serious? What Dependency Injection? This is just some throwaway testing code. Why do we need a DI Framework here?

Well, people who have used Cucumber in their Java Projects, I am pretty sure, that at some point, they have faced this issue about how to share State or exchange information between Step Definitions? I have seen people go crazy and use lot of static methods and non final static variables to achieve that. Of course, that comes with a huge maintenance cost, and ultimately makes the code brittle and unreadable. And this is where a good DI Framework comes into rescue. Cucumber supports a host of [DI Frameworks](https://cucumber.io/docs/cucumber/state/#dependency-injection). Today, however, we will limit ourselves to using Spring.

# What are we testing?

Well, lets pretend that we have a library service, where we have created just 2 broad REST endpoints related to Authors and Books. The Service is written in Spring Boot, with an in-memory H2 Database. The REST end points that are exposed, allow us to list these items and then search them with a partial name. These are GET calls. We also have a single PUT call that allows us the change the Author of a Book.

We would be writing integration tests that will make REST calls to these endpoints and make sure that they are returning expected results.

# Lets build

## pom.xml

From Cucumber 7.x release, they have introduced a BOM. This has made our lives much easier. First, we will define these BOMs under the *dependencyManagement* section as shown below:

<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>io.cucumber</groupId>  
 <artifactId>cucumber-bom</artifactId>  
 <version>7.0.0</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit</groupId>  
 <artifactId>junit-bom</artifactId>  
 <version>5.8.1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

Lets then move on to add the actual dependencies:

<dependency>  
 <groupId>io.cucumber</groupId>  
 <artifactId>cucumber-java</artifactId>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>io.cucumber</groupId>  
 <artifactId>cucumber-spring</artifactId>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>io.cucumber</groupId>  
 <artifactId>cucumber-junit-platform-engine</artifactId>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>org.junit.platform</groupId>  
 <artifactId>junit-platform-suite</artifactId>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <scope>test</scope>  
</dependency>  
<dependency>  
 <groupId>org.junit.platform</groupId>  
 <artifactId>junit-platform-console</artifactId>  
 <scope>test</scope>  
</dependency>

## The TestRunner

This is the key class that helps JUnit5 detect and run Cucumber feature files. There is a subtle difference between 6.x and 7.x. In 6.x, we used to have the *@Cucumber* annotation. In 7.x, they have deprecated this and recommend using the *JUnit Platform Suite* as shown below.

import org.junit.platform.suite.api.ConfigurationParameter;  
import org.junit.platform.suite.api.SelectClasspathResource;  
import org.junit.platform.suite.api.Suite;  
  
import io.cucumber.core.options.Constants;  
  
@Suite  
@SelectClasspathResource("cukes")  
@ConfigurationParameter(key = Constants.PLUGIN\_PUBLISH\_QUIET\_PROPERTY\_NAME, value = "true")  
@ConfigurationParameter(key = Constants.FILTER\_TAGS\_PROPERTY\_NAME, value = "@book or @author")  
public class RunCucumberTest {  
  
}

By convention, the feature files are expected to be under the same package as the TestRunner. In case it is different, for example, located under the *src/test/resources/cukes*, we can specify that by using the *@SelectClasspathResource(“cukes”)* annotation.

Various configuration parameters can be passed into it through the *@ConfigurationParameter* annotation.

## Spring Integration

### Defining a TestConfiguration

Let us start by defining a TestConfiguration that will do my ComponentScan and also define any Beans that I would need.

@ComponentScan  
@Configuration  
public class SpringTestConfig {  
  
 @Bean  
 public RestTemplate restTemplate() {  
 return new RestTemplate();  
 }  
  
}

The *@ComponentScan* ensures that any Spring annotated classes defined under this package, will be picked up by Spring DI.

### Integration with Cucumber

In order to integrate Spring DI with Cucumber glue code or Step Definitions, we would need to do the below:

import org.springframework.test.context.ContextConfiguration;  
import io.cucumber.spring.CucumberContextConfiguration;  
  
@CucumberContextConfiguration  
@ContextConfiguration(classes = { SpringTestConfig.class })  
public class CucumberSpringContextConfig {  
  
}

Notice how we use the *@ContextConfiguration* to tie the TestConfiguration that we have defined above into Cucumber. This will ensure that all Spring Beans are available for use in our Cucumber Step Definitions.

### Sharing State between Step Definitions

This is how my *author.feature* file looks like:

@author  
Feature: Test all author rest endpoints  
  
Background:  
 Given The application is available at "http://localhost:8080"  
 And Health check is fine at "/actuator/health"  
  
Scenario: List all available authors  
 When I fetch authors at "/v1/author"  
 Then I should find 2 authors

Look closely and you will see that the idea is to define the base URL which is the host name *http://localhost:8080* only once in the *Given* of the *Background*. All subsequent steps just define the rest of the path like */actuator/health* and */v1/author*. This is done to make this more readable and promoting reuse.

But then, we would need the ability to pass on the base URL from the *Given* to all the subsequent steps that needs it.

Another scenario where we would need to share state is, between the *When* and *Then*. In the above example, *When* is expected to fetch the list of authors and *Then* is supposed to verify the number of authors found.

We write the below SharedDTO class that we can use for this.

@Component  
@ScenarioScope  
@Data  
public class SharedDTO {  
  
 private String baseUrl;  
 private List<Author> authors;  
 private List<Book> books;  
 private int booksAffected;  
  
}

Then, in the *AuthorSteps* step definition, just *@Autowire* this, as you would any other Spring Bean. The @ScenarioScope ensures that this Bean is destroyed and recreated for every scenario.

public class AuthorSteps {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @Autowired  
 private SharedDTO sharedDto;  
  
 @Given("The application is available at {string}")  
 public void applicationBaseUrl(String baseUrl) {  
 sharedDto.setBaseUrl(baseUrl);  
 }  
  
 @When("I fetch authors at {string}")  
 public void fetchAuthors(String uri) {  
 List<Author> authors = restTemplate.exchange(sharedDto.getBaseUrl() + uri, HttpMethod.GET, null,  
 new ParameterizedTypeReference<List<Author>>() {}).getBody();  
 sharedDto.setAuthors(authors);  
 }  
  
 @Then("I should find {int} authors")  
 public void fetchAuthors(int authorCount) {  
 assertEquals(authorCount, sharedDto.getAuthors().size());  
 }   
  
}

Note how we are setting the Base URL on the *Given* and the using that in *When*. Also, we are setting the *List<Author>* in *When* and using it in *Then* to verify the count.

### Reference

Spring Cucumber Reference: <https://github.com/paawak/cucumber-jvm/tree/master/spring>

JUnit 5 Example: <https://github.com/paawak/cucumber-jvm/tree/master/examples/calculator-java-junit5>

# Addendum: How to define Scenario Outline

Though a bit off topic, could not resist the temptation of adding this nugget here.

## When to use?

I have a REST endpoint to search for Authors with partial First Name or Last Name. I would like to try different names and assert the count of Authors returned. Instead of writing various scenarios, each for a name, we could use a *Scenario Outline* as below:

Scenario Outline: Get authors by name  
 When I search for author by name at "/v1/author/<name>"  
 Then I should find <count> authors  
  
 Examples:  
 | name | count |  
 | BALAI | 1 |  
 | sara | 1 |  
 | a | 2 |  
 | A | 2 |

What we are doing is putting parameters between *<>*. We have 2 parameters above: *<name>* and *<count>*. Then, in the *Examples* section, we are providing the values for these parameters. The Steps defined under the *Scenario Outline* would be called as many times as the number of rows in the *Examples* with the specified values.

# Sources

The source code can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/cucumber/cucumber-with-junit5-spring>.