Acceptance testing with Cucumber

1. Configuration

```
<dependency>
	<groupId>io.cucumber</groupId>
	<artifactId>cucumber-java</artifactId>
	<version>LATEST_VERSION</version>
	<scope>test</scope>
</dependency>
<dependency>
	<groupId>io.cucumber</groupId>
	<artifactId>cucumber-spring</artifactId>
	<version>LATEST_VERSION</version>
	<scope>test</scope>
</dependency>
```

FOR UNIT TESTING:

```
<dependency>
	<groupId>io.cucumber</groupId>
	<artifactId>cucumber-junit</artifactId>
	<version>LATEST_VERSION</version>
	<scope>test</scope>
</dependency>
```

2. Introduction

In a BDD approach, you should start writing your acceptance test first and let the tests guide your implementation code. Your workflow should look like this:

1. Write the acceptance test
2. Launch it and see it fail to see what the next step is
3. Write a unit test for that next step
4. See it fail so you know what the implementation for that test is
5. Repeat 2, 3 and 4 until the acceptance test passes correctly

Cucumber tests consist of two parts: features, and steps (also called scenarios). Features should define a user story, while steps should define all the possible scenarios that we could encounter within that user story.

Basic Cucumber implementation should consist of *.feature* files, containing the features and scenarios of a particular user story, written in a specific pseudo language. The feature should look like the following:

- **Feature** (Title, mandatory)
  - **As a** (Description, optional)
  - **I want to** (Description, optional)
  - **So that** (Description, optional)

Every feature should have one or more scenarios, represented in the following way, in Gherkin language:

- **Scenario**: A unitary part of the feature. Represented in natural language and must be a self-contained unit.
    - **Given** All the preconditions that must be satisfied so that the scenario can occur.
    - **When** Some event is triggered, some action happens.
    - **Then** We expect some result.
    - **[And]** Optional.

Next, we need to create a so-called "glue code". It is a method that links a single Gherkin scenario with Java code. For this, we have 2 options: either use regular expressions, or Cucumber expressions.

Cucumber Expressions offer similar functionality to Regular Expressions, with a syntax that is more human to read and write. Cucumber Expressions are also extensible with parameter types.

By default, Cucumber will assume you are using Cucumber Expressions. To use Regular Expressions, add anchors (starting with ^ and ending with $) or forward slashes (/).

For example, if we have the following in our Gherkin file:
Given *I have 42 cucumbers in my belly*
Can be matched by:
Given *I have {int} cucumbers in my belly*

Allowed parameters:

- {int} Matches integers, for example 71 or -19.
- {float} Matches floats, for example 3.6, .8 or -9.2.
- {word} Matches words without whitespace, for example banana (but not banana split)
- {string} Matches single-quoted or double-quoted strings, for example "banana split" or 'banana split' (but not banana split). Only the text between the quotes will be extracted. The quotes themselves are discarded. Empty pairs of quotes are valid and will be matched and passed to step code as empty strings.
- {} anonymous Matches anything (/.*/).

3. Cucumber implementation example

We have the following controller:

```
@RestController
public class VersionController {
   @GetMapping("/version")
   public String getVersion() {
      return "1.0";
   }
}
```

We have this Gherkin file:

```
Feature: the version can be retrieved
  Scenario: client makes call to GET /version
    When the client calls /version
    Then the client receives status code of 200
    And the client receives server version 1.0
```

All we need to run our Cucumber tests with JUnit is to create a single empty class with an annotation @RunWith(Cucumber.class):

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources")
public class CucumberIntegrationTest {
}
```

Now all the Cucumber definitions can go into a separate Java class which extends SpringIntegrationTest:

```
public class StepDefs extends SpringIntegrationTest {
    @When("the client calls /version")
     public void the_client_issues_GET_version() throws Throwable{
        executeGet("http://localhost:8080/version");
    }

    @Then("the client receives status code of {int}")
    public void the_client_receives_status_code_of(int statusCode) throws Throwable {
        HttpStatus currentStatusCode = latestResponse.getTheResponse().getStatusCode();
        assertThat("status code is incorrect : "+
        latestResponse.getBody(), currentStatusCode.value(), is(statusCode));
    }

    @And("the client receives server version {}")
    public void the_client_receives_server_version_body(String version) throws Throwable {
        assertThat(latestResponse.getBody(), is(version));
    }
}
```

4. Additional information

- **LISTS:** In Cucumber you can pass a list as a parameter in a step definition. The simplest way to do it is following this syntax:

```
Given the following animals:
  | cow   |
  | horse |
  | sheep |

@Given("the following animals:")
public void the_following_animals(List<String> animals) {
}
```

- **HOOKS**: Cucumber allows the use of Hooks. Hooks are blocks of code that can run at various points in the Cucumber execution cycle. **@Before** hooks run before the first step of each scenario. **@After** hooks run after the last step of each scenario. **@BeforeStep** and **@AfterStep** are hooks invoked before and after a step

- **TAGS:** Cucumber allows the use of **Tags** for categorizing and separating feature or scenario groups:
  @billing
  Feature: Verify billing
  @important
  Scenario: Missing product description
  Given hello
  Scenario: Several products
  Given hello

References:

- **Cucumber tutorial:** https://www.baeldung.com/cucumber-spring-integration
- **Documentation:** https://cucumber.io/