Spring Boot Cucumber Tests Example:

Important Words:

* Automated Acceptance Testing
* Gherkin
* Cucumber
* Behavior Driven Development
* StepDefinationClass

Cucumber tries to address the area of acceptance testing. Cucumber allows collaboration between business stakeholder and development team to express the business outcomes. Cucumber has its own global language and adheres to syntax rules known as Gherkin.

It is a Business Readable, Domain Specific Language that lets you describe software’s behavior without detailing how that behavior is implemented. These are the following constructs of the Gherkin language.

* Given : This indicates the prior state of the system. For example, a user must be logged in to perform activities within the site.
* When : This is the specific activity carried out or the functionality tested.
* Then: This is our assert/verification scenario which is the result we expect out of the testing.

## **Spring Boot Application**

We will implement a calculator web service and automate testing with Cucumber. Our focus is on Automated Acceptance testing and not unit testing the application.

**Pom.xml dependencies(Important dependencies):**

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.2.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.2.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

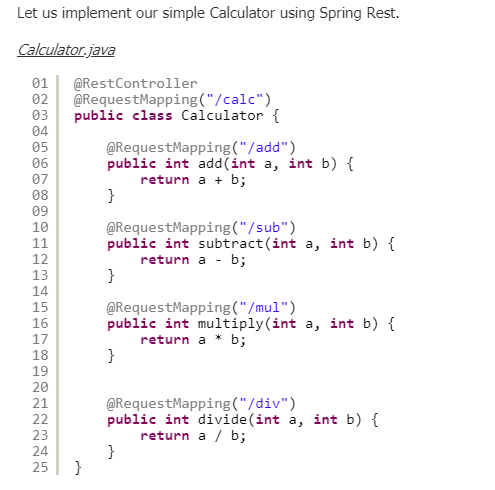
<artifactId>cucumber-spring</artifactId>

<version>1.2.4</version>

<scope>test</scope>

</dependency>

Below is the controller code, we are going to write BDD tests for this.

****

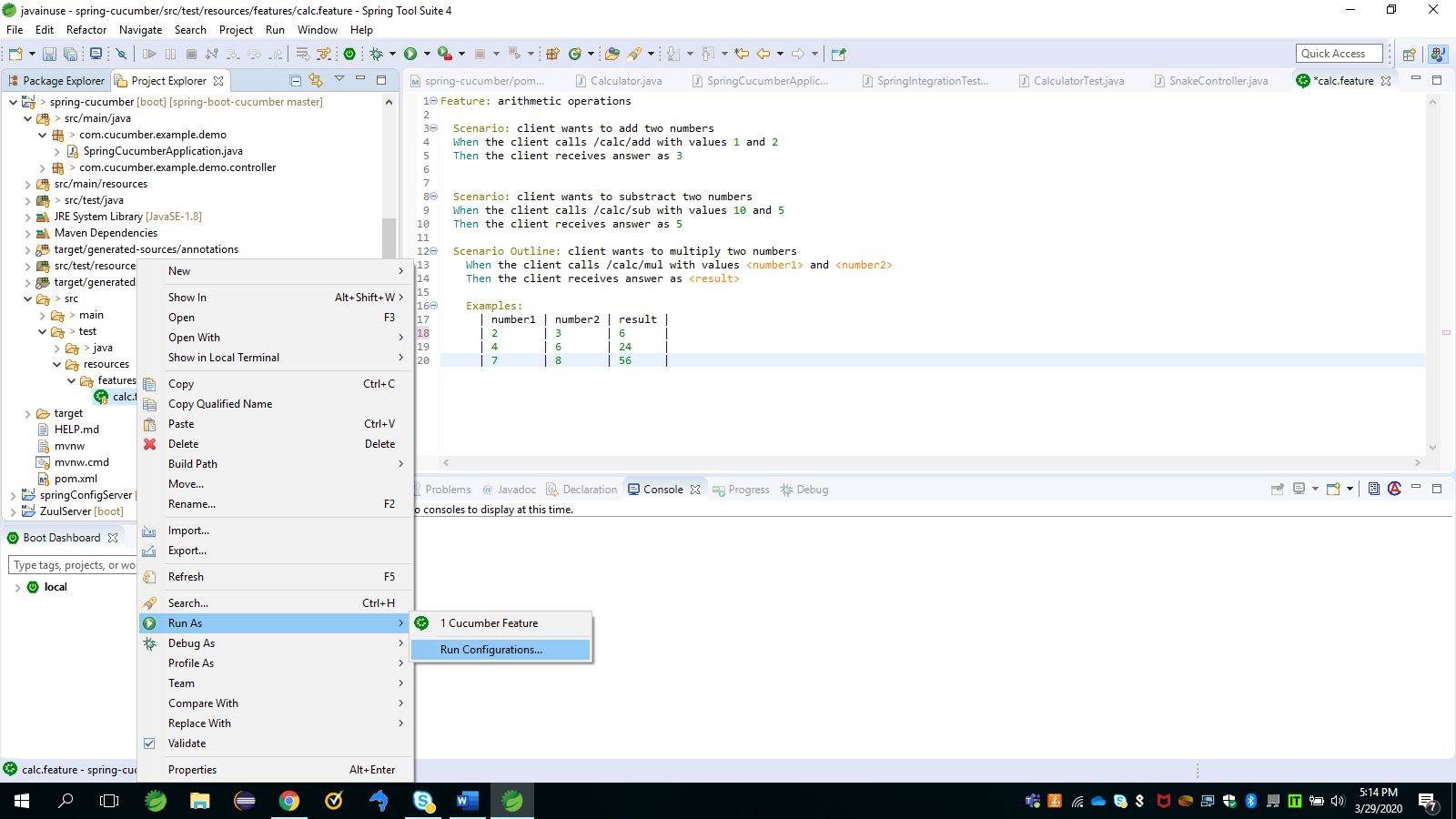
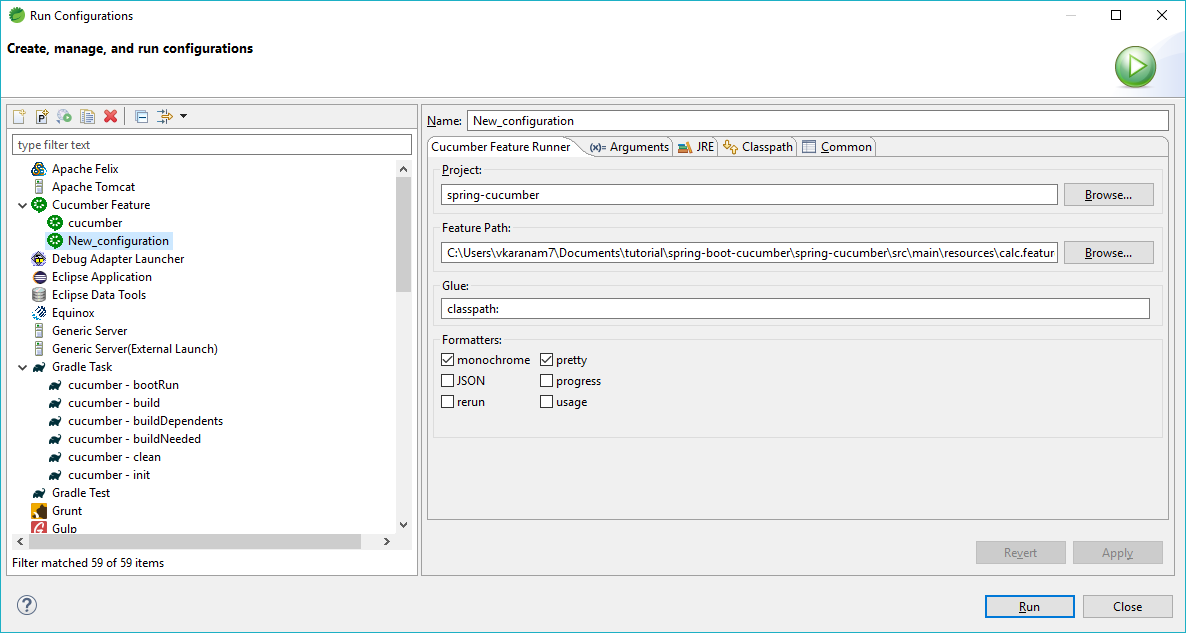
## **Cucumber Acceptance Testing**

The first step is to create a feature file capturing all the testing scenarios.

*calc.feature*

|  |  |
| --- | --- |
|  |  |

For the above calc.feature file, we need to generateStepDefinition Class.

**Click on RUN, this will generate code to test acceptance testing for the controller as shown below.**

Feature: arithmetic operations

Scenario: client wants to add two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:3

When the client calls /calc/add with values 1 and 2

Then the client receives answer as 3

Scenario: client wants to substract two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:8

When the client calls /calc/sub with values 10 and 5

Then the client receives answer as 5

Scenario Outline: client wants to multiply two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:12

When the client calls /calc/mul with values <number1> and <number2>

Then the client receives answer as <result>

Examples:

Scenario Outline: client wants to multiply two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:18

When the client calls /calc/mul with values 2 and 3

Then the client receives answer as 6

Scenario Outline: client wants to multiply two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:19

When the client calls /calc/mul with values 4 and 6

Then the client receives answer as 24

Scenario Outline: client wants to multiply two numbers # C:/Users/vkaranam7/Documents/tutorial/spring-boot-cucumber/spring-cucumber/src/test/resources/features/calc.feature:20

When the client calls /calc/mul with values 7 and 8

Then the client receives answer as 56

5 Scenarios (5 undefined)

10 Steps (10 undefined)

0m0.000s

You can implement missing steps with the snippets below:

@When("^the client calls /calc/add with values (\\d+) and (\\d+)$")

public void the\_client\_calls\_calc\_add\_with\_values\_and(int arg1, int arg2) throws Throwable {

// Write code here that turns the phrase above into concrete actions

throw new PendingException();

}

@Then("^the client receives answer as (\\d+)$")

public void the\_client\_receives\_answer\_as(int arg1) throws Throwable {

// Write code here that turns the phrase above into concrete actions

throw new PendingException();

}

@When("^the client calls /calc/sub with values (\\d+) and (\\d+)$")

public void the\_client\_calls\_calc\_sub\_with\_values\_and(int arg1, int arg2) throws Throwable {

// Write code here that turns the phrase above into concrete actions

throw new PendingException();

}

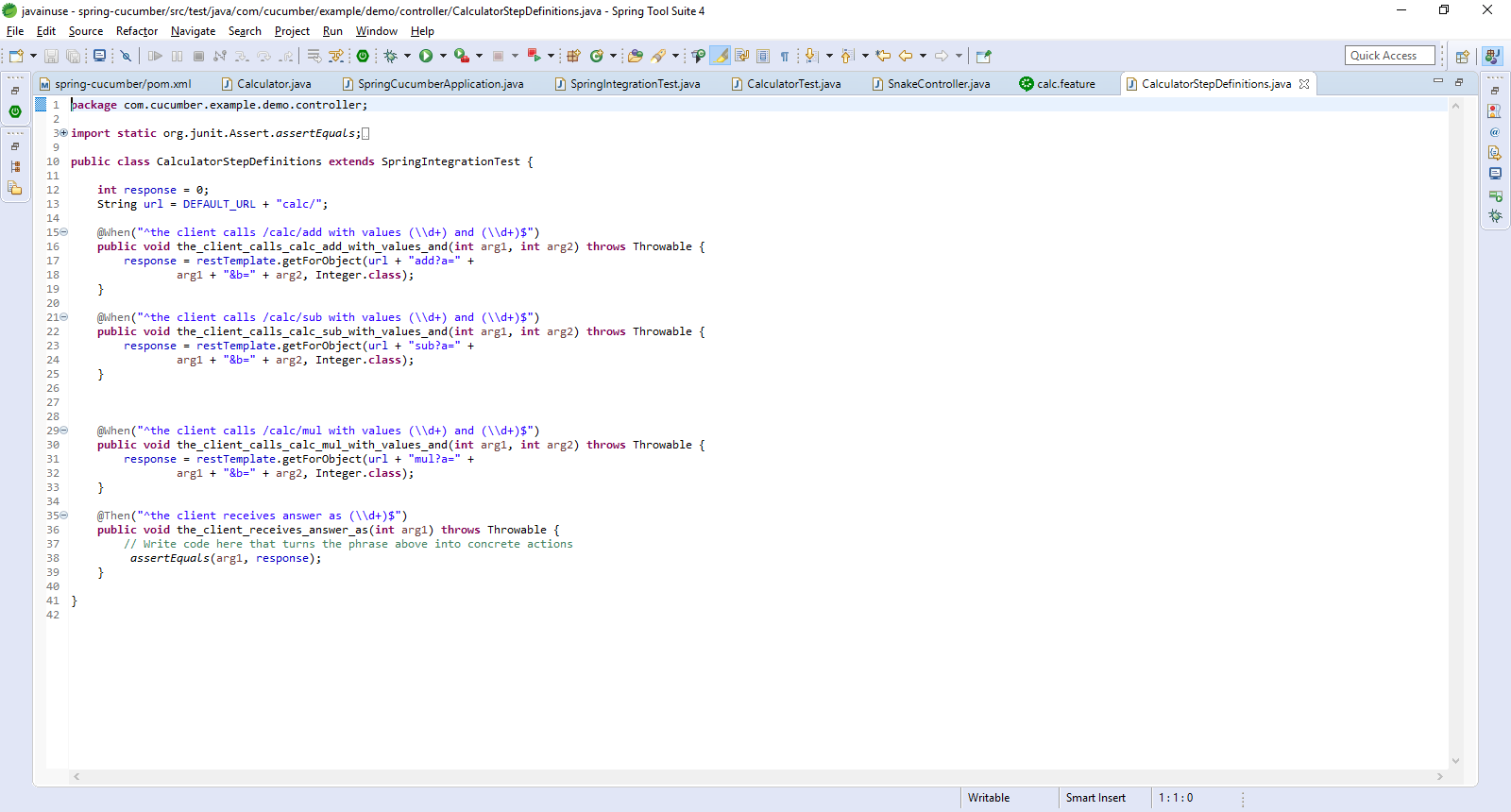
@When("^the client calls /calc/mul with values (\\d+) and (\\d+)$")

public void the\_client\_calls\_calc\_mul\_with\_values\_and(int arg1, int arg2) throws Throwable {

// Write code here that turns the phrase above into concrete actions

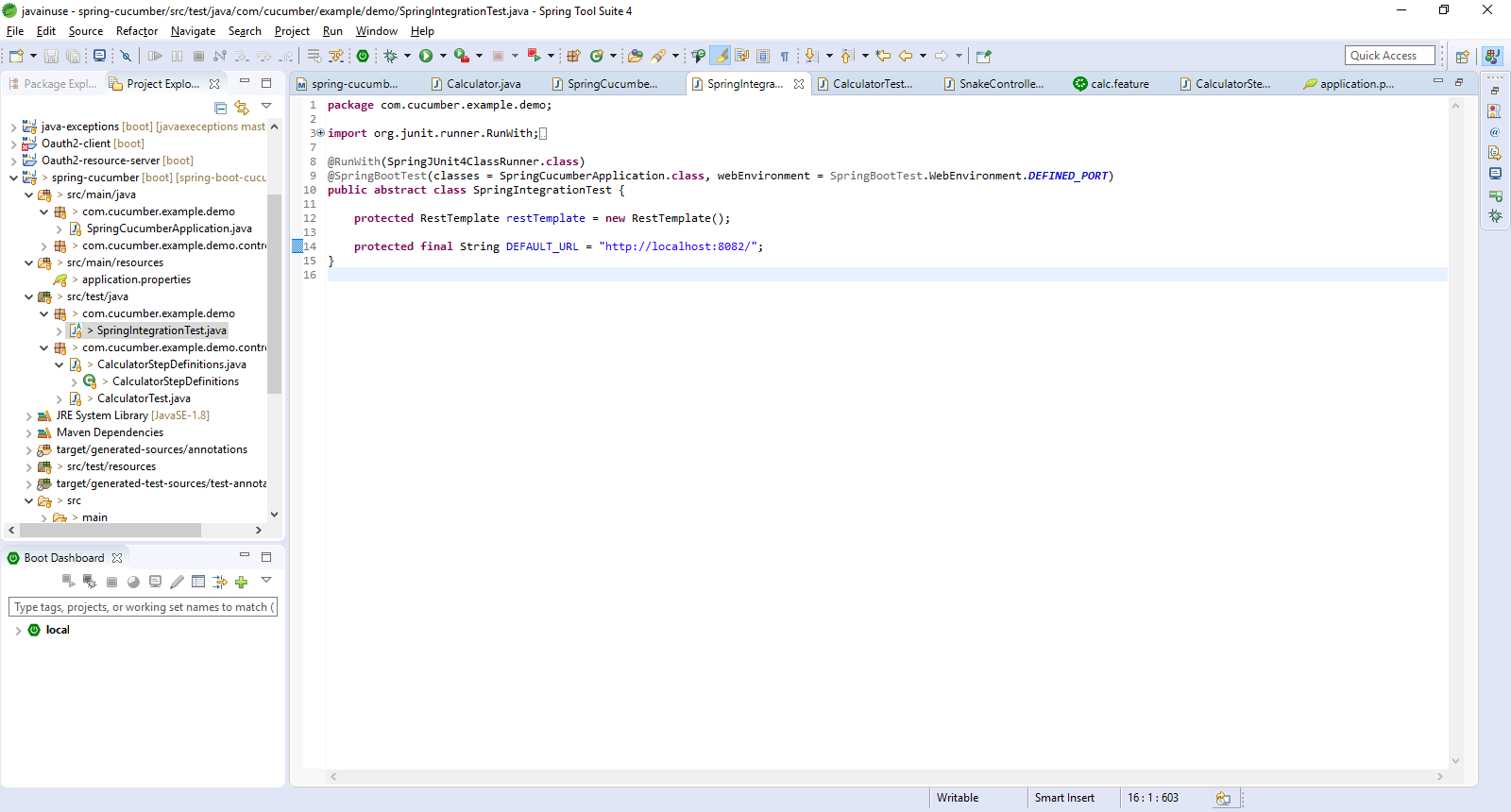
throw new PendingException();

}



* This file has the step definition snippets copied from the console.
* It also contains the multiplication step, where we have passed multiple values in feature file using **scenario outline** feature.
* It uses restTemplate from the base class to connect to the URL (add/sub/mul/div) based on the scenario and stores the response in its instance variable.
* The assertion is a straight forward comparison of the value with its stored instance variable.

We will look at the base class below as that provides the Spring configuration for the Step definition class to be executed.



* This class will be run with SpringJunitRunner which is used to run the step definition class.
* This class is configured as SpringBootTestclass and we specify our application to run in classes options. This is our source SpringBoot Application class specified in the previous section.
* We want our application to run in specified port and hence the use of DEFINED\_PORT. As discussed in the previous section, We can specify the port to run in application.properties under the resources folder.
* We create an instance of RestTemplate to be used in all our subclasses.
* Since we specify the port, We can be sure of the URL and hence the base URL for our application is constructed and stored.

## **Cucumber Scenario Outline**

if we wanted to test the same scenario with multiple data points, Cucumber provides the option of a Scenario Outline. We have to modify our feature file to create a Scenario Outline.

Scenario Outline: client wants to multiply two numbers

When the client calls /calc/mul with values <number1> and <number2>

Then the client receives answer as <result>

Examples:

| number1 | number2 | result |

| 2 | 3 | 6 |

| 4 | 6 | 24 |

| 7 | 8 | 56 |

o kick off the test execution, a Spring Boot Execution class has to be created and configured under the test directory.

*CalculatorTest.java*

|  |  |
| --- | --- |
| 1  2  3  4 | @RunWith(Cucumber.class)  @CucumberOptions(features = "src/test/resources")  public class CalculatorTest {  } |

This specifies that the test suite has to be run with Cucumber runner and Cucumber has to identify the feature file under the parent directory specified with features options. Cucumber looks for a features folder under the directory for the presence of .feature files. Running the above class produces the following output.

**Reference:** <https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-cucumber-tests-example/>