|  |  |
| --- | --- |
|  | **2020** |
|  | **Royal Bank Of Scotland**  **Chittaranjan Swain** |

|  |
| --- |
| **[CUCUMBER – bdd framework]** |
| Cucumber is a testing tool that supports Behavior Driven Development (BDD) framework. You will learn cucumber concepts along with a Project here. You Will also learn about Git and Jenkins in this. |

Welcome to this journey to learn Cucumber (Cucumber Tutorial). Cucumber is a buzz word these days. Everybody is talking about how fun it is to use Cucumber. So let’s understand more on Cucumber and Software development model it follows.

Cucumber is a testing framework which supports Behavior Driven Development (BDD). It lets us define application behavior in plain meaningful English text using a simple grammar defined by a language called **Gherkin**. Cucumber itself is written in Ruby, but it can be used to “test” code written in Ruby or other languages including but not limited to Java, C# and Python.

## TDD vs BDD:-

## What’s the Difference? - BDD and TDD may seem very similar since they are both testing strategies for a software application. In both cases, the developer writes the test before writing the code to make the test pass. And in both cases, the tests can be used as part of an automated testing framework to prevent bugs.

### What you’re testing? - BDD is designed to test an application’s behavior from the end user’s standpoint, whereas TDD is focused on testing smaller pieces of functionality in isolation. In the prior example, the TDD test asserts the result of a specific method, while the BDD test is only concerned about the result of the higher level scenario.

### How you’re testing? - BDD involves product managers, developers, and test engineers who collaborate to come up with concrete examples of desirable functionality. There is a high level of communication before any implementation. By comparison, TDD can be done by a solo developer without any external input from product managers or stakeholders.

## How They Work Together? - It’s important to note that BDD and TDD aren’t mutually exclusive — many agile teams use TDD without using BDD. However, BDD ensures that most use-cases of the application work on a higher level and provides a greater level of confidence.

## For example, a development team may use BDD to come up with higher level tests that confirm an application's behavior. When implementing the specifics, developers may create separate unit tests to ensure the robustness of the components, especially since these components may be reused elsewhere across the application.

Rspec is a great example of TDD and BDD principles combined into a single framework for Ruby applications.

## Implementation Differences - Test-driven development has been widely adopted by Agile development firms and there are many different tools to help teams get on the same page. Unfortunately, there are fewer tools for behavior-driven development since it involves communication between business and technical teams.

## [HipTest](https://hiptest.com/) is a BDD test management suite that’s designed to bridge the gap and make continuous testing a lot easier. Easy-to-use scenario editors let business teams easily define and update features, while integrations with automation frameworks and CI/CD pipelines simplify writing the tests.

## The plain-text language used in the features and the aggregated data from the integrations helps create a living documentation that can be referenced by technical or business teams anytime. For example, business users can ensure that a certain feature has been developed and still functions as of the latest CI run.

## The good news with BDD is that it doesn’t require reworking any existing TDD practices—it only requires an investment in a little more communication that can pay off in the form of less ambiguity and greater confidence in the product.

## The Bottom Line - Test-driven development has become the default approach for agile software development over the past several years. The approach minimizes bugs reaching production and ensures that software can be continuously released without issue.

## Behavior-driven development represents an evolution beyond TDD, where business goals can be better communicated to developers. By bridging the gap between business and technical teams, BDD helps reduce any confusion about acceptance criteria, identify potential problems with user stories early, and ensure that the application functions as-expected for end users.

## Introduction: -

In order to get better advantage of the software testing, organizations are nowadays taking a step forward. They implement important acceptance test scenarios while development is in-progress. This approach is commonly known as **Behavior Driven Development** (BDD).

Behavior Driven Development gives us an opportunity to create test scripts from both the developer’s and the customer’s perspective as well. So in the beginning, developers, project managers, QAs, user acceptance testers and the product owner (stockholder), all get together and brainstorm about which test scenarios should be passed in order to call this software/application successful. This way they come up with a set of test scenarios. All these test scripts are in simple English language, so it serves the purpose of documentation also.

## Example

If we are developing a user authentication feature, then the following can be few key test scenarios, which needs to get passed in order to call it a success.

* The user should be able to login with correct username and correct password.
* The user should not be able to login with incorrect username and correct password.
* The user should not be able to login with correct username and incorrect password.

## How it Works

By the time the code is ready, test scripts are ready too. The code has to pass the test scripts defined in BDD. If it does not happen, code refactoring will be needed. Code gets freezed only after successful execution of defined test scripts.



It is a very simple notion, but what we need in order to get this concept implemented. The answer is, Behavior Driven Development (BDD) Framework. **Cucumber** is one such open source tool, which supports behavior driven development. To be more precise, Cucumber can be defined as a testing framework, driven by plain English text. It serves as documentation, automated tests, and a development aid – all in one.

So what does Cucumber do? It can be described in the following steps −

Cucumber reads the code written in plain English text (Language Gherkin – to be introduced later in this tutorial) in the feature file (to be introduced later).

It finds the exact match of each step in the step definition (a code file - details provided later in the tutorial).

The piece of code to be executed can be different software frameworks like **Selenium, Ruby on Rails**, etc. Not every BDD framework tool supports every tool.

This has become the reason for Cucumber's popularity over other frameworks, like **JBehave, JDave, Easyb,** etc.

Cucumber supports over a dozen different software platforms like −

* Ruby on Rails
* Selenium
* PicoContainer
* Spring Framework
* Watir

## Advantages of Cucumber Over Other Tools

* Cucumber supports different languages like Java.net and Ruby. Cucumber supports 14 languages right now, including Python on the JVM also called Jython.
* It acts as a bridge between the business and technical language. We can accomplish this by creating a test case in plain English text.
* It allows the test script to be written without knowledge of any code; it allows the involvement of non-programmers as well.
* It serves the purpose of end-to-end test framework unlike other tools.
* Due to simple test script architecture, Cucumber provides code reusability.

## Gherkin: -

So far, we have got an understanding of Cucumber and what it does. It executes the test scripts, which have been defined in the feature file (to be covered in subsequent chapters). The language, in which this executable feature files is written, is known as **Gherkin**. Gherkin is a plain English text language, which helps the tool - Cucumber to interpret and execute the test scripts.

Gherkin provides the common set of keywords in English text, which can be used by people amongst the different community and yet get the same output in the form of test scripts. Example: -

**Feature** − Login functionality for a social networking site.

**Given** I am a social networking site user.

**When** I enter username as username1.

**And** I enter password as password1.

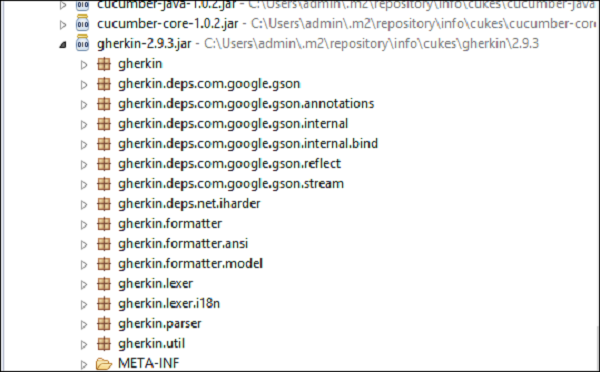
**Then** I should be redirected to the home page of the site.

The above-mentioned scenario is of a feature called user login. All the words highlighted in bold are **Gherkin keywords.**

Example of few other keywords −

* Background
* But
* \*
* Scenario Outline
* Examples

Gherkin will parse each step written in step definition file. So the steps mentioned in the feature file and the step definition file should match. You can locate the Gherkin jars in the Maven Dependency folder in the Package Explorer. It gets downloaded along with the other Cucumber jars. It will look like the following screenshot −



Another interesting fact about Gherkin is, it supports not only English but many other native languages such as French, Finnish, Indonesian, Hungarian, Hindi, Urdu, Gujarati, etc.

## Feature:-

A **Feature** can be defined as a standalone unit or functionality of a project. Let’s take a very common example of a social networking site. How does the feature of this product/project look like? Few basic features can be determined as −

* Create and remove the user from the social networking site.
* User login functionality for the social networking site.
* Sharing photos or videos on the social networking site.
* Sending a friend request.
* Logout.

By now, it is clear that, **Each independent functionality of the product under test can be termed as a feature when we talk about Cucumber**. It is a best practice later when you start testing, that before deriving the test scripts, we should determine the features to be tested.

A feature usually contains a list of scenarios to be tested for that feature. A file in which we store features, description about the features and scenarios to be tested is known as **Feature File**. We will see more about feature files in the following chapter.

The keyword to represent a feature under test in Gherkins is “Feature”. The suggested best practice is, to write a small description of the feature beneath the feature title in the feature file. This will fulfill the need of a good documentation as well.

### Example

**Feature** − Login functionality for a social networking site.

The user should be able to login into the social networking site if the username and the password are correct.

The user should be shown the error message if the username and the password are incorrect.

The user should be navigated to home page, if the username and password are correct.

## Feature Files

The file, in which Cucumber tests are written, is known as **feature files**. It is advisable that there should be a separate feature file, for each feature under test. The extension of the feature file needs to be **“.feature”.**

One can create as many feature files as needed. To have an organized structure, each feature should have one feature file.

For Example −

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Feature** | **Feature File name** |
| 1 | User Login | userLogin.feature |
| 2 | Share the Post | sharePost.feature |
| 3 | Create Account | createAccount.feature |
| 4 | Delete Account | deleteAccount.feature |

The naming convention to be used for feature name, feature file name depends on the individual’s choice. There is no ground rule in Cucumber about names.

A simple feature file consists of the following keywords/parts −

* **Feature** − Name of the feature under test.
* **Description** (optional) − Describe about feature under test.
* **Scenario** − What is the test scenario.
* **Given** − Prerequisite before the test steps get executed.
* **When** − Specific condition which should match in order to execute the next step.
* **Then** − What should happen if the condition mentioned in WHEN is satisfied.

### Example

**Feature** − User login on social networking site.

The user should be able to login into the social networking site when the username and the password are correct.

The user should be shown an error message when the username and the password are incorrect.

The user should be navigated to the home page if the username and the password are correct.

**Outline** − Login functionality for a social networking site.

The given user navigates to Facebook. When I enter Username as "<username>" and Password as "<password>". Then, login should be unsuccessful.

| username | password |

| username1 | password1 |

**\* AND** keyword is used to show conjunction between two conditions. **AND** can be used with any other keywords like **GIVEN, WHEN** and **THEN**.

There are no logic details written in the feature file.

## Steps Definitions: -

The next target is to test or run the feature file and in order to test the feature file, we need to write the implementation or step definition for each step in the feature file in java. When Cucumber executes a Step in a Scenario it will look for a matching Step Definition to execute.

## What is Step Definition?

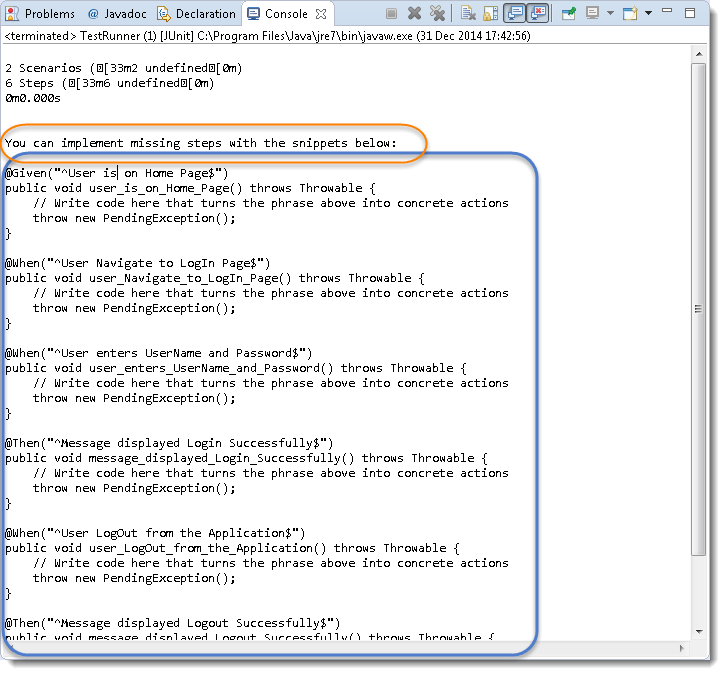
A Step Definition is a small piece of code with a pattern attached to it or in other words a Step Definition is a java method in a class with an annotation above it. An annotation followed by the pattern is used to link the Step Definition to all the matching Steps, and the code is what Cucumber will execute when it sees a Gherkin Step. Cucumber finds the Step Definition file with the help of the Glue code in ***Cucumber Options***. We will cover different Cucumber Options in the next chapter.

## Add a Step Definition file

1) Create a new ***Class*** file in the ‘***stepDefinition***‘ package and name it as ‘***Test\_Steps***‘, by right click on the Package and select New > Class. Do not check the option for ‘***public static void main***‘ and click on ***Finish*** button.

2) Take a look at the message in the console window. This message was displayed, when we ran the ***Test\_Runner*** class.

2) Notice, the eclipse console window says ‘***You can implement missing steps with the snippets below:***‘. It is very easy to implement all the steps, all you need to do is to copy the complete text marked in a blue box and paste it into the above created ***Test\_Steps*** class.



3) As of now, the test will show many errors on ‘***@***‘ **annotations**. Mouse hover at the annotations and import the ‘***cucumber.api.java.en***‘ for all the annotations.

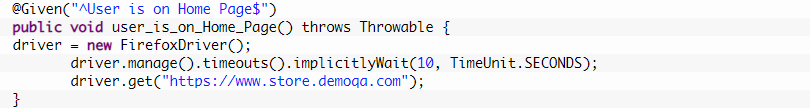
## Step_Definition_2

## Add Selenium Java code in the Step Definition methods

1) Now take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the first method ‘**@Given(“^User is on Home Page$”)**‘.

* Launch the Browser
* Navigate to Home Page

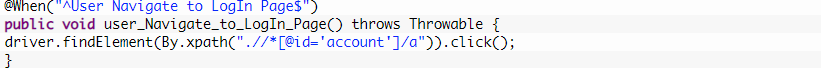
Method will look like this now:



2) Take out the code take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the second method ‘***@When(“^User Navigate to LogIn Page$”)***‘.

* Click on the LogIn link

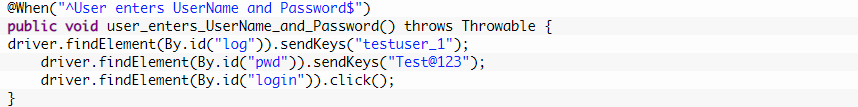
Method will look like this now:



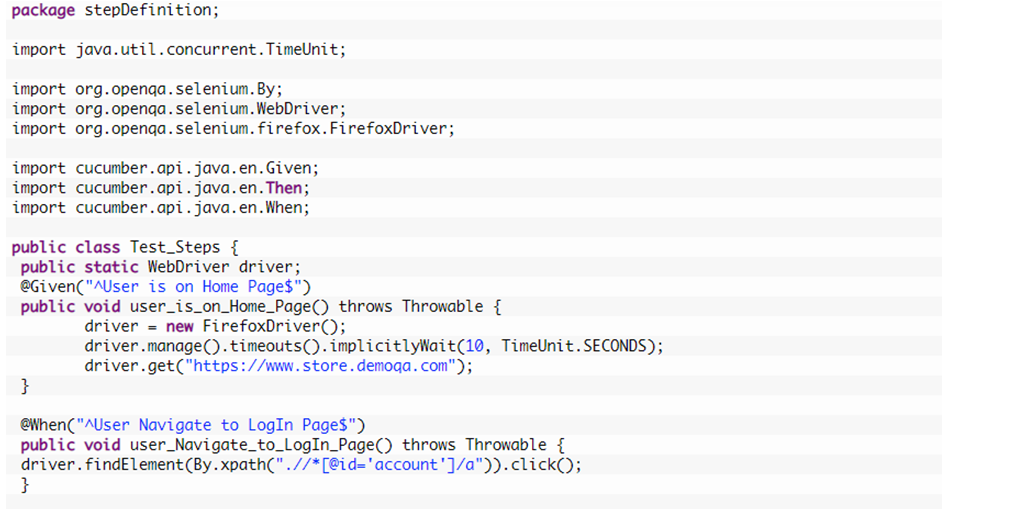
3) Take out the code take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the second method ‘***@When(“^User enters UserName and Password$”)***‘.

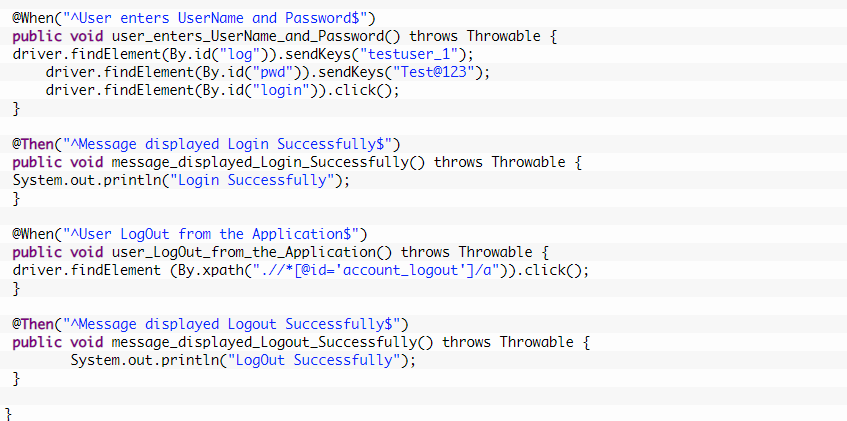
* Enter UserName and Password
* Click on Submit button

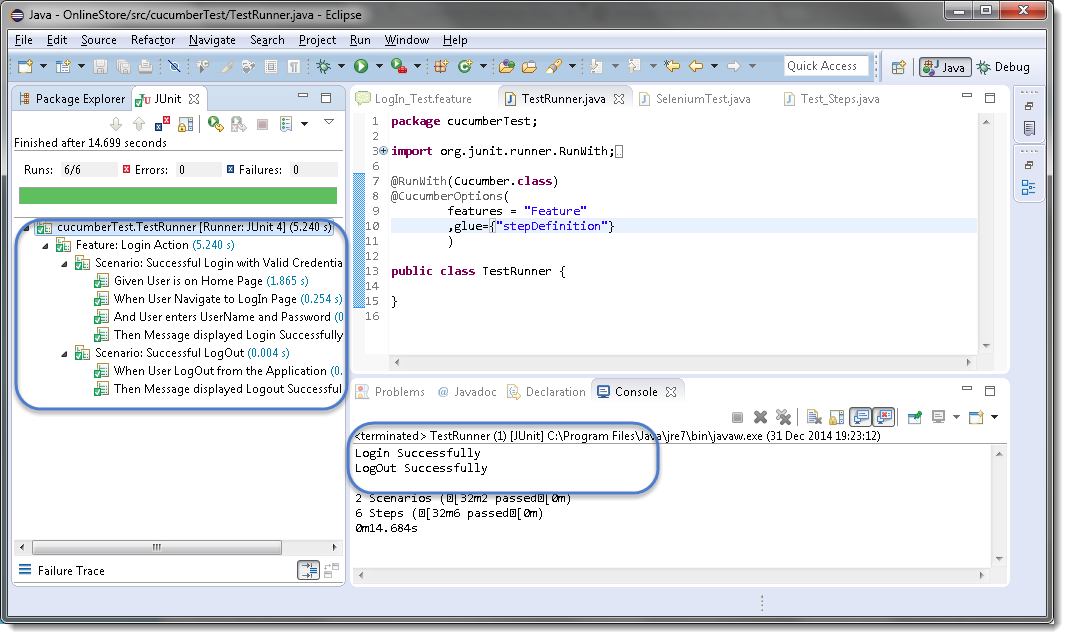
Method will look like this now:



4) Do the same steps for the rest of the methods as well and complete Test\_Steps class will look like this:



**Run the Cucumber Test -** Now we are all set to run the first Cucumber test. Right Click on ***TestRunner*** class and Click ***Run As***  > ***JUnit Test.***Cucumber will run the script the same way it runs in Selenium WebDriver and the result will be shown in the left hand side project explorer window in JUnit tab.



Cucumber starts it’s execution by reading the feature file steps. As soon as Cucumber reaches the first step for e.g. Given the statement of Scenario, it looks for the same statement in the Step Definition file, the moment it finds the statement, it executes the piece of code written inside the function.

## Scenario: -

**Scenario** is one of the core Gherkin structures. Every scenario starts with the keyword “Scenario:” (or localized one) and is followed by an optional scenario title. Each feature can have one or more scenarios and every scenario consists of one or more steps. A very simple example of scenario can be −

**Scenario** − Verify Help Functionality.

**Given** user navigates to Facebook.

**When** the user clicks on Help,

**Then** the Help page opens.

Consider a case, where we need to execute a test scenario more than once. Suppose, we need to make sure that the login functionality is working for all types of subscription holders. That requires execution of login functionality scenario multiple times. Copy paste the same steps in order to just re-execute the code, does not seem to be a smart idea. For this, Gherkin provides one more structure, which is scenario outline.

**Scenario outline** is similar to scenario structure; the only difference is the provision of multiple inputs. As you can see in the following example, the test case remains the same and non-repeatable. At the bottom we have provided multiple input values for the variables “Username” and “Password”. While running the actual test, Cucumber will replace the variable with input values provided and it will execute the test. Once pass-1 has been executed, the test will rerun for second iteration with another input value. Such variable or placeholders can be represented with ”<>” while mentioning with gherkin statements.

## Example

**Scenario Outline** − Login functionality for a social networking site. The given user navigates to Facebook.

When the user logs in using the Username as "<username>" and the Password as "<password>", then login should be successful.

| username | password |

| user1 | password1 |

| user2 | password2 |

There are a few tips and tricks to smartly define the Cucumber scenarios.

* Each step should be clearly defined, so that it does not create any confusion for the reader.
* Do not repeat the test scenario, if needed use scenario outline to implement repetition.
* Develop a test step in a way that, it can be used within multiple scenarios and scenario outlines.
* As far as possible, keep each step completely independent. For example: “Given the user is logged in”. This can be divided into two steps
  + Given the user enters the user name.
  + Clicks on login.

## Annotation: -

**Annotation** is a predefined text, which holds a specific meaning. It lets the compiler/interpreter know, what should be done upon execution. Cucumber has got the following few annotations −

* **Given** −
  + It describes the pre-requisite for the test to be executed.
  + Example − GIVEN I am a Facebook user
* **When** −
  + It defines the trigger point for any test scenario execution.
  + Example − WHEN I enter "<username>"
* **Then** −
  + Then holds the expected result for the test to be executed.
  + Example − THEN login should be successful.
* **And** −
  + It provides the logical AND condition between any two statements. AND can be used in conjunction with GIVEN, WHEN and THEN statement.
  + Example − WHEN I enter my "<username>" AND I enter my "<password>"
* **But** −
  + It signifies logical OR condition between any two statements. OR can be used in conjunction with GIVEN, WHEN and THEN statement.
  + Example − THEN login should be successful. BUT home page should not be missing.
* **Scenario** −
  + Details about the scenario under the test needs to be captured after the keyword “Scenario:”
  + Example −

Scenario:

GIVEN I am a Facebook user

WHEN I enter my

AND I enter my

THEN login should be successful.

BUT home page should not be missing.

* **Scenario Outline** − (To be covered later)
* **Examples** − (To be covered later)
* **Background** −
  + Background generally has the instruction on what to setup before each scenario runs. However, it gets executed after “Before” hook (to be covered later). So this is ideal to be used for code when we want to set up the web-browser or we want to establish the database connectivity.
    - Example −

Background:

Go to Facebook home page.

## Example Scenario

Let’s automate a scenario in order to understand annotations better.

**Step 1**

Create a Maven Test Project named as **AnnotationTest**.

* Go to File → New → Others → Maven → Maven Project → Next.
* Provide group Id (group Id will identify your project uniquely across all projects).
* Provide artifact Id (artifact Id is the name of the jar without version. You can choose any name which is in lowercase).
* Click on Finish.
* Open **pom.xml** −
  + Go to package explorer on the left hand side of Eclipse.
  + Expand the project CucumberTest.
  + Locate pom.xml file.
  + Right-click and select the option, Open with “Text Editor”.
* Add dependency for Selenium − This will indicate Maven, which Selenium jar files are to be downloaded from the central repository to the local repository.
  + Open pom.xml is in edit mode, create dependencies tag (<dependencies></dependencies>), inside the project tag.
  + Inside the dependencies tag, create dependency tag (<dependency></dependency>).
  + Provide the following information within the dependency tag.

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>2.47.1</version>

</dependency>

* Add dependency for Cucumber-Java: This will indicate Maven, which Cucumber files are to be downloaded from the central repository to the local repository.
  + Create one more dependency tag.
  + Provide following information within the dependency tag.

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

* Add dependency for Cucumber-JUnit: This will indicate Maven, which Cucumber JUnit files are to be downloaded from the central repository to the local repository.
  + Create one more dependency tag.
  + Provide the following information within the dependency tag.

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

* Add dependency for JUnit - This will indicate Maven, which JUnit files are to be downloaded from the central repository to the local repository.
  + Create one more dependency tag.
  + Provide the following information within the dependency tag.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

* Add dependency for JUnit - This will indicate Maven, which JUnit files are to be downloaded from the central repository to the local repository.
  + Create one more dependency tag.
  + Provide the following information within the dependency tag.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

* Verify binaries.
  + Once pom.xml is edited successfully, save it.
  + Go to Project → Clean − It will take a few minutes.

**Step 2**

Create a package named Annotation under **src/test/java**

* Select the newly created project.
* Right-click and select ‘New’.
* Select option as ‘Package’.
* Name it as ‘Annotation’.
* Save it.

**Step 3**

Create a feature file named **annotation.feature**.

* Select and right-click on the package outline.
* Click on ‘New’ file.
* Give the file a name such as **outline.feature**.
* Write the following text within the file and save it.

Feature: annotation

#This is how background can be used to eliminate duplicate steps

Background:

User navigates to Facebook Given

I am on Facebook login page

#Scenario with AND

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

#Scenario with BUT

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

But Relogin option should be available

**Step 4**

Create a step definition file.

* Select and right-click on the package outline.
* Click on ‘New’ file.
* Give the file name as **annotation.java**
* Write the following text within the file and save it.

package Annotation;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class annotation {

WebDriver driver = null;

@Given("^I am on Facebook login page$")

public void goToFacebook() {

driver = new FirefoxDriver();

driver.navigate().to("https://www.facebook.com/");

}

@When("^I enter username as \"(.\*)\"$")

public void enterUsername(String arg1) {

driver.findElement(By.id("email")).sendKeys(arg1);

}

@When ("^I enter password as \"(.\*)\"$")

public void enterPassword(String arg1) {

driver.findElement(By.id("pass")).sendKeys(arg1);

driver.findElement(By.id("u\_0\_v")).click();

}

@Then("^Login should fail$")

public void checkFail() {

if(driver.getCurrentUrl().equalsIgnoreCase(

"https://www.facebook.com/login.php?login\_attempt=1&lwv=110")){

System.out.println("Test1 Pass");

} else {

System.out.println("Test1 Failed");

}

driver.close();

}

@Then("^Relogin option should be available$")

public void checkRelogin() {

if(driver.getCurrentUrl().equalsIgnoreCase(

"https://www.facebook.com/login.php?login\_attempt=1&lwv=110")){

System.out.println("Test2 Pass");

} else {

System.out.println("Test2 Failed");

}

driver.close();

}

}

**Step 5**

Create a runner class file.

* Select and right-click on the package outline.
* Click on ‘New’ file.
* Give the file a name, such as **runTest.java**
* Write the following text within the file and save it.

package Annotation;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

@Cucumber.Options(format = {"pretty", "html:target/cucumber"})

public class runTest { }

**Step 6**

Run the test using option −

* Select runTest.java file from the package explorer.
* Right-click and select the option **‘Run as’**
* Select JUnit test.

You will observe the following things when you run this class file −

* Facebook opens in a new Firefox web-browser instance.
* TOM will be passed as an input to the username field.
* JERRY will be passed as an input to the password field.
* Login will be clicked.
* Message will be displayed on the browser regarding unsuccessful login.
* In the console, you will see “Test Pass” printed
* Step result 1. to 5. Will be re-executed for username as "" and password as "".

## Scenario outline: -

**Scenario outline** basically replaces variable/keywords with the value from the table. Each row in the table is considered to be a scenario. Let’s continue with the same example of Facebook login feature. So far we have been executing one scenario: Upon providing the correct user name, login is successful. Now, suppose we want to check if login is successful for all three possible types of inputs, which are username, email address, or phone number. To achieve this, we will need to write three different scenarios, where each scenario will vary with the type of input, login is successful. In this case, scenarios will look like the following.

**Scenario:**

Given user navigates to Facebook

When I enter correct username and password

Then login should be successful

**Scenario:**

Given user navigates to Facebook

When I enter correct email address and password

Then login should be successful

**Scenario:**

Given user navigates to Facebook

When I enter correct phone number and password

Then login should be successful

Here, if we see closely, for above three scenarios: statements are the same, only the input parameter (username/email address/phone number) is changing. That’s where the importance of scenario outline comes into picture.

When we define any scenario with scenario outline, we can specify one test scenario and at the bottom of it we can provide a number of inputs. The scenario will get executed as many times as the number of inputs provided.

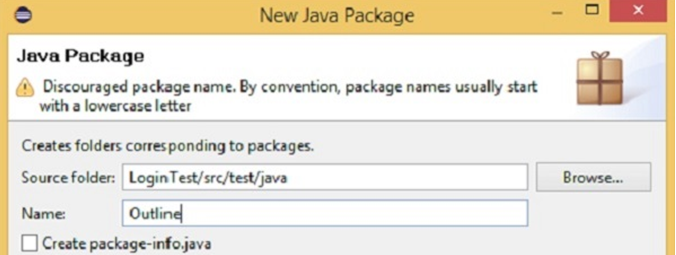
## Example

Let’s create an example of a scenario outline −

**Step 1** − Create a Maven Test Project named as **ScenarioOutlineTest**

* Go to File → New → Others → Maven → Maven Project → Next. Provide group Id (group Id will identify your project uniquely across all projects). Provide artifact Id (artifact Id is the name of the jar without version. You can choose any name which is in lowercase). Click on Finish.
* Open **pom.xml** – Add all the dependencies.
* Verify binaries.
  + Once pom.xml is edited successfully, save it.
  + Go to Project → Clean − It will take a few minutes.

**Step 2** − Create a package named “outline” under **src/test/java**



**Step 3** − Create a feature file named **“outline.feature”**

* Select and right-click on the **package outline.**
* Click on ‘New’ file.
* Give the file name such as **“outline.feature”**
  + Write the following text within the file and save it.

**Feature** − Scenario Outline

**Scenario Outline** − Login functionality for a social networking site.

**Given** user navigates to Facebook

**When** I enter Username as "<username>" and Password as "<password>"

**Then** login should be unsuccessful

**Example** −

| username | password |

| username1 | password1 |

| username2 | password2 |

**Note** − Here, example annotation describes the range of input to be provided upon scenario execution. Test scenario will be executed for each of the input provided. So, in the given example, test scenario will be executed three times.

**Step 4** − Create a step definition file. Select and right-click on the package outline. Click on ‘New’ file. Give the file name as **stepdefinition.java**

* Write the following text within the file and save it.

package Outline;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class stepdefinition {

WebDriver driver = null;

@Given("^user navigates to facebook$")

public void goToFacebook() {

driver = new FirefoxDriver();

driver.navigate().to("https://www.facebook.com/");

}

@When("^I enter Username as \"([^\"]\*)\" and Password as \"([^\"]\*)\"$")

public void I\_enter\_Username\_as\_and\_Password\_as(String arg1, String arg2) {

driver.findElement(By.id("email")).sendKeys(arg1);

driver.findElement(By.id("pass")).sendKeys(arg2);

driver.findElement(By.id("u\_0\_v")).click();

}

@Then("^login should be unsuccessful$")

public void validateRelogin() {

if(driver.getCurrentUrl().equalsIgnoreCase(

"https://www.facebook.com/login.php?login\_attempt=1&lwv=110")){

System.out.println("Test Pass");

} else {

System.out.println("Test Failed");

}

driver.close();

}

}

**Note** − In the above code, we have to define a function having two input argument: one username and other will be for password. So for each set of input provided in the example tag, Set of GIVEN, WHEN and THEN will be executed.

**Step 5** − Create a runner class file. Select and right-click on the package outline. Click on ‘New’ file. Give the file name such as **runTest.java**

* Write the following text within the file and save it.

package Outline;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

@Cucumber.Options(format = {"pretty", "html:target/cucumber"})

public class runTest { }

* Run the test using option −
  + Select **runTest.java** file from package explorer.
  + Right-click and select the option **‘Run as’**.
  + Select JUnit test.

You will observe the following things when you run this class file

* Facebook opens in a new Firefox web-browser instance.
* Username1 and password1 will be passed as an input to username and password field.
* Login will be clicked.
* Message will be displayed on the browser regarding unsuccessful login.
* In the console, you will see “Test Pass” printed.
* **Step result 1 to 5 will be re-executed for username2 and password2.**

In nutshell, when scenario does not change but only the data value gets changed, it is advisable to use scenario outline data tables.

## Cucumber Options: -

So far in the series of Cucumber tutorial we have covered Feature files, Gherkins, Step Definitions, Annotations, Test Runner Class and many other things. There is no doubt that you cannot set up the BDD framework until you know all the concepts but there are still few more areas that are very important to know in the life of Cucumber Automation such as Cucumber Options, Regular Expressions, Page Object factory and few others. Let’s start with Cucumber Options.

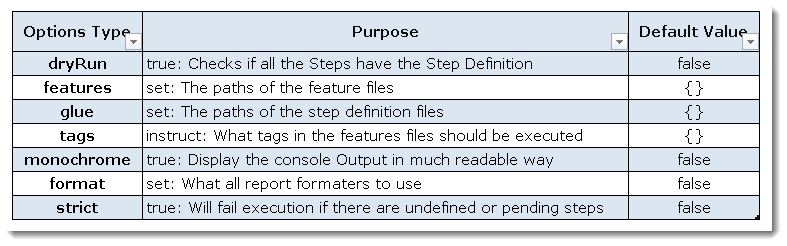
## What is Cucumber Options?

In layman language, ***@CucumberOptions*** are like property files or settings for your test. Basically @CucumberOptions enables us to do all the things that we could have done if we have used cucumber command line. This is very helpful and of utmost importance, if we are using IDE such eclipse only to execute our project. You must have noticed that we set a few options in the ‘***TestRunner’*** class in the previous chapter.

***TestRunner Class***



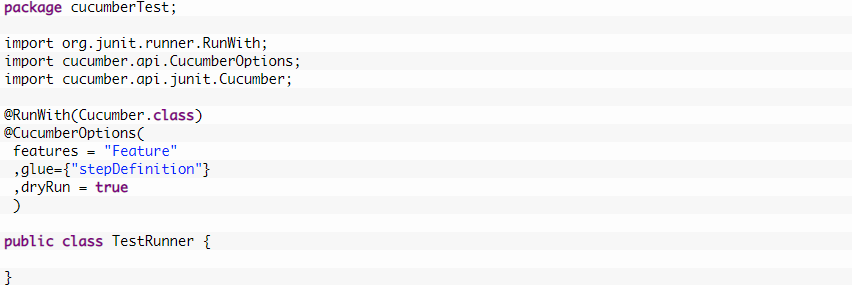
So in the above example, we have just set two different Cucumber Options. One is for Feature File and the other is for Step Definition file. We will talk about it in detail now but with this, we can say that @CucumberOptions are used to set some specific properties for the Cucumber test. Following Main Options are available in Cucumber:



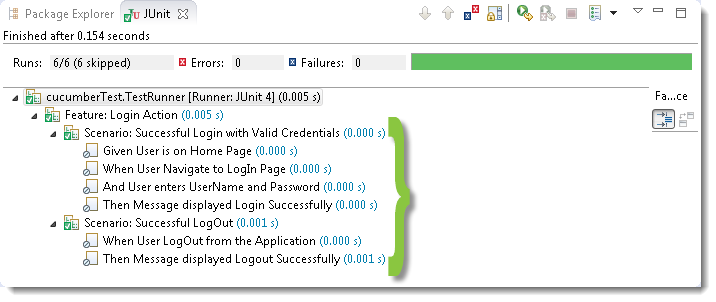
### *****Dry Run*****

***dryRun*** option can either set as ***true*** or ***false***. If it is set as true, it means that Cucumber will only check that every Step mentioned in the Feature File has corresponding code written in Step Definition file or not. So in case any of the functions are missed in the Step Definition for any Step in Feature File, it will give us the message. For practice just add the code ‘***dryRun = true***‘ in ***TestRunner*** class:

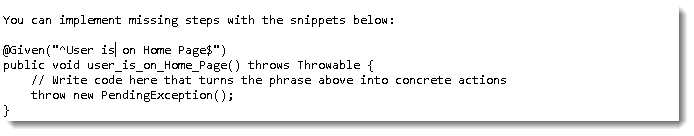
***TestRunner Class***



Now give it a run by Right Click on ***TestRunner*** class and Click ***Run As***  > ***JUnit Test.***Cucumber will run the script and the result will be shown in the left-hand side project explorer window in JUnit tab.



Take a look at the time duration at the end of the every Steps, it is (***0.000s***). It means none of the Step is executed but still, Cucumber has made sure that every Step has the corresponding method available in the Step Definition file. Give it a try, remove the ‘***@Given(“^User is on Home Page$”)***‘ statement from the ***Test\_Steps*** class and run the ***TestRunner*** class again. You would get the following message:



### *****Monochrome*****

This option can either set as ***true*** or ***false***. If it is set as true, it means that the console output for the Cucumber test are much more readable. And if it is set as false, then the console output is not as readable as it should be. For practice just add the code ‘***monochrome = true***‘ in ***TestRunner*** class:

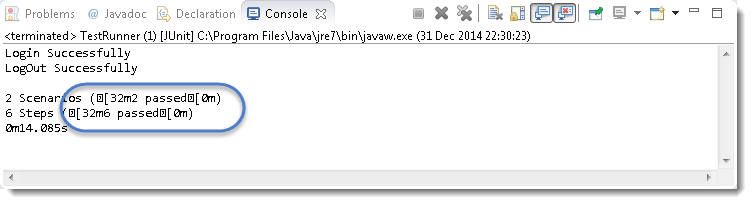
***TestRunner Class***



Now give it a run by Right Click on ***TestRunner*** class and Click ***Run As***  > ***JUnit Test.***Cucumber will run the script and Console Output will display like this:



This time change the value from true to false and run the ***TestRunner*** class again. This time the Console Output will look like this:

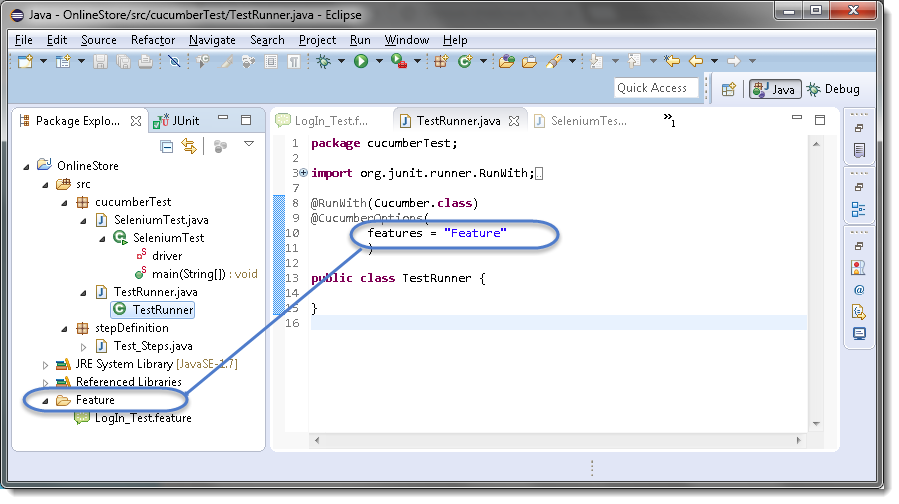


***Features Options***helps Cucumber to locate the Feature file in the project folder structure. You must have notices that we have been specifying the Feature Option in the ***TestRunner*** class since the first chapter. All we need to do is to specify the folder path and Cucumber will automatically find all the ‘***.features***‘ extension files in the folder. It can be specified like:

***features = “Feature“***

Or if the Feature file is in the deep folder structure

***features = “src/test/features“***



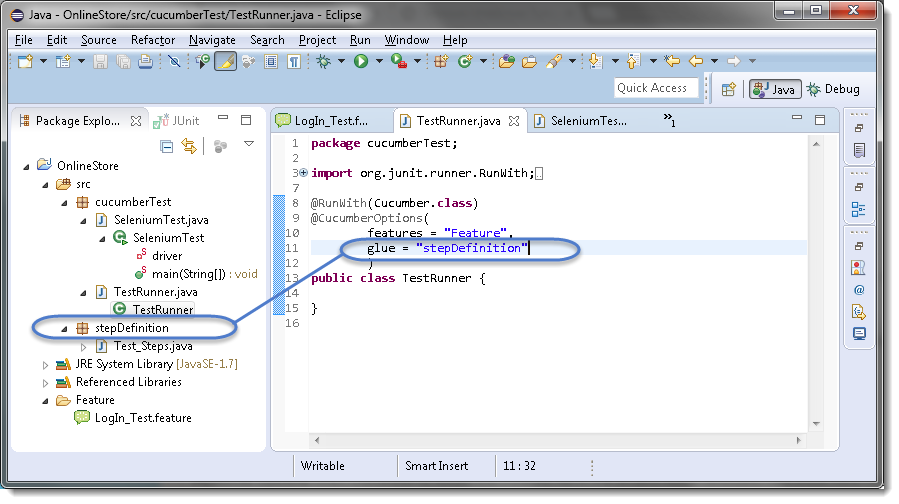
### *****Glue*****

It is almost the same think as Features Option but the only difference is that it helps Cucumber to locate the ***Step Definition file.*** Whenever Cucumber encounters a Step, it looks for a Step Definition inside all the files present in the folder mentioned in **Glue Option**. It can be specified like:

***glue = “stepDefinition“***

Or if the Step Definition file is in the deep folder structure

***glue = “src/test/stepDeinition“***



**Format Option** is used to specify different formatting options for the output reports. Various options that can be used as for-matters are:

***Pretty:***Prints the Gherkin source with additional colors and stack traces for errors. Use below code:

***format = {“pretty“}***

***HTML:***This will generate a HTML report at the location mentioned in the for-matter itself. Use below code:

***format = {“html:Folder\_Name“}***

***JSON:***This report contains all the information from the gherkin source in JSON Format. This report is meant to be post-processed into another visual format by 3rd party tools such as Cucumber Jenkins. Use the below code:

***format = {“json:Folder\_Name/cucumber.json“}***

***JUnit:*** This report generates XML files just like Apache Ant’s JUnit report task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports. use the below code:

***format = { “junit:Folder\_Name/cucumber.xml“}***

## Tags: -

Let’s say you have got many different feature files that cover all the different functionality of the application. Now there can be a certain situation in the project where you like to execute just a **SmokeTests** or **End2EndTests** or may be **RegressionTests**.

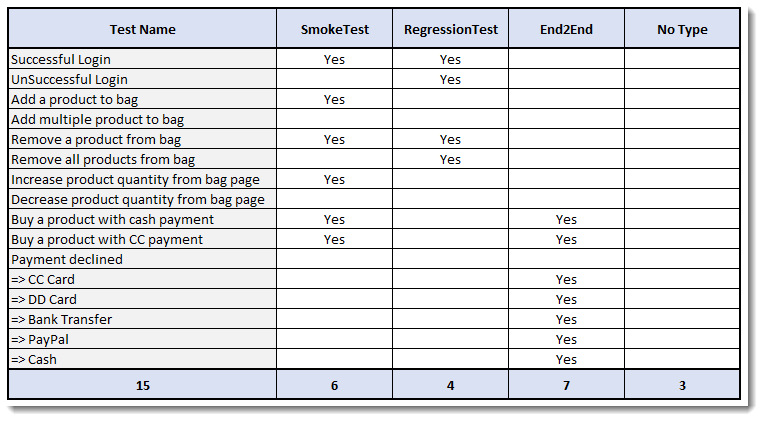
One approach is that you start creating new feature files with the name of the type like **SmokeTests.features** or**End2EndTests.feature** and copy-paste your existing tests in the same. But this would make the project filthy and would require more maintenance in future. So how to manage execution in such cases?

*For this, Cucumber has already provided a way to organize your scenario execution by using***tags***in feature file. We can define each scenario with a useful tag. Later, in the runner file, we can decide which specific tag (and so as the scenario(s)) we want Cucumber to execute. Tag starts with “***@***”. After “@” you can have any relevant text to define your tag like***@SmokeTests***just above the scenarios you like to mark. Then to target these tagged scenarios just specify the tags names in the***CucumberOptions***as***tags = {“@SmokeTests”}.**

*Tagging not just specifically works with Scenarios, it also works with***Features***. Means you can also tag your features files.***Any tag that exists on a Feature will be inherited by Scenario, Scenario Outline or Examples.**

## How to run Cucumber Tests in Groups using Cucumber Tags?

Let’s understand this with an example. Below is an excel sheet containing a list of scenarios of a single feature.



***Things to Note:***

* *Few scenarios are part of the Smoke Test, Regression Test, and End2End Test.*
* *Few scenarios are part of two or more Test Types. For example, the first test is considered as Smoke as well as Regression.*
* *Few scenarios are not at all tagged*
* *The last scenario of Payment Declined, it is a single scenario but has five different test data. So this will be considered as five different scenarios.*

***Feature file will look like this***

@FunctionalTest

Feature: ECommerce Application

@SmokeTest @RegressionTest

Scenario: Successful Login

Given This is a blank test

@RegressionTest

Scenario: UnSuccessful Login

Given This is a blank test

@SmokeTest

Scenario: Add a product to bag

Given This is a blank test

Scenario: Add multiple product to bag

Given This is a blank test

@SmokeTest @RegressionTest

Scenario: Remove a product from bag

Given This is a blank test

@RegressionTest

Scenario: Remove all products from bag

Given This is a blank test

@SmokeTest

Scenario: Increase product quantity from bag page

Given This is a blank test

Scenario: Decrease product quantity from bag page

Given This is a blank test

@SmokeTest @End2End

Scenario: Buy a product with cash payment

Given This is a blank test

@SmokeTest @End2End

Scenario: Buy a product with CC payment

Given This is a blank test

@End2End

Scenario Outline: Payment declined

Given This is a blank test

Examples:

|PaymentMethod|

|CC Card|

|DD Card|

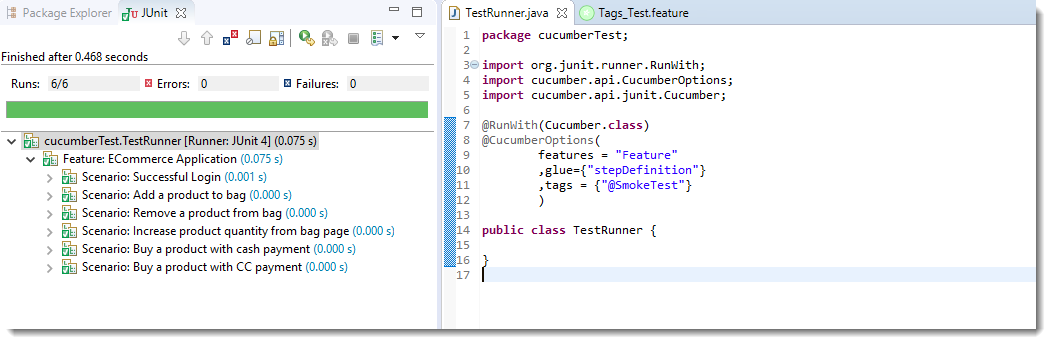
|Bank Transfer|

|PayPal|

|Cash|

### *****Running single Cucumber Feature file or single Cucumber Tag*****

***Execute all tests tagged as @SmokeTests***

  
***Note***: In the excel sheet and in the feature file paste above if you count the scenarios which are tagged as @SmokeTests, you will find the count is 6 and the same count is also displayed under Junit tab.

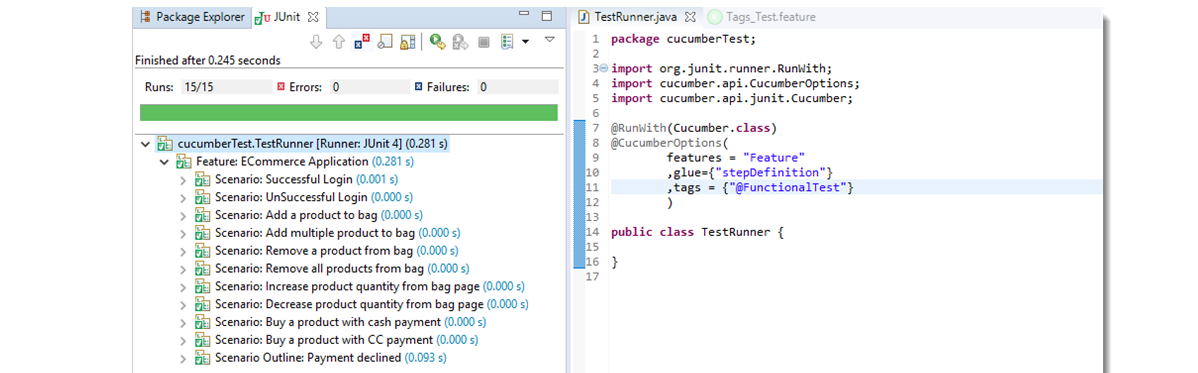
***Execute all tests tagged as @End2End***

### Cucumber Group Tags 9

***Note***: A special thing to note here is that, the last scenario ***Payment declined*** has five different data examples. So every example is considered as a separate test. Due to which the total test number is 7.

***Execute all tests of a Feature tagged as @FunctionalTest : Feature Tagging***

Not only tags work with Scenario, tags work with Feature Files as well. Feature files pasted above is also tagged as ***@FunctionTests***. Let’s just see how to executes all the tests in this feature.



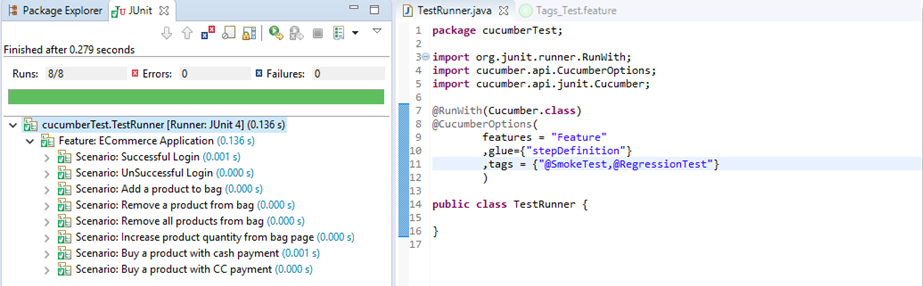
***Note***: All the test exists in the feature file are executed.

## Logically ANDing and ORing Tags

Requirements are complicated, it will not always simple like executing a single tag. It can be complicated like executing scenarios that are tagged either as @SmokeTest or @RegressionTest. It can also be like executing scenarios that are tagged both as @SmokeTest and @RegressionTest. Cucumber tagging gives us the capability to choose what we want with the help of ANDing and ORing.

***Execute all tests tagged as @SmokeTest OR @RegressionTest***

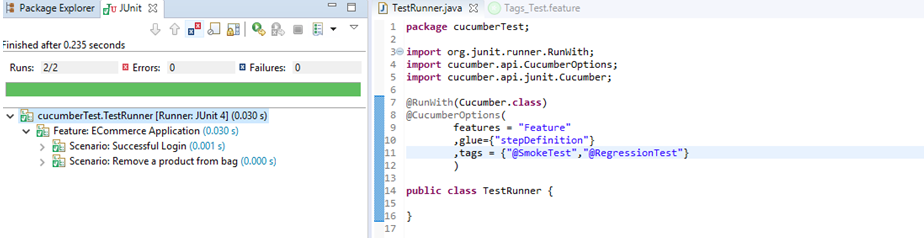
Tags that are ***comma-separated*** are ORed.



***Note***: OR means scenarios that are tagged either as @SmokeTest OR @RegressionTest.

***Execute all tests tagged as @SmokeTest AND @RegressionTest***

Tags which are passed in separate ***quotes*** are ANDed

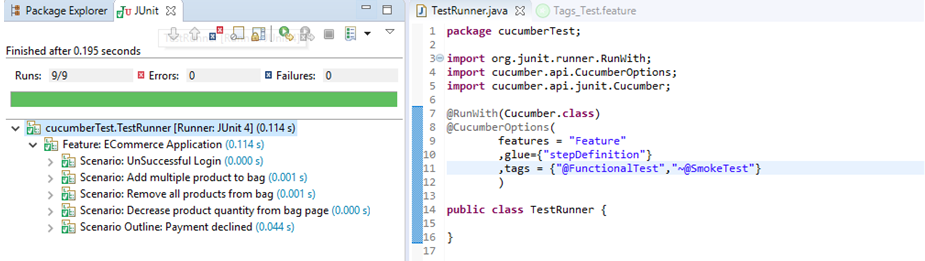


***Note***: There are only two scenarios in our feature file which have both tags together.

## How to Ignore Cucumber Tests

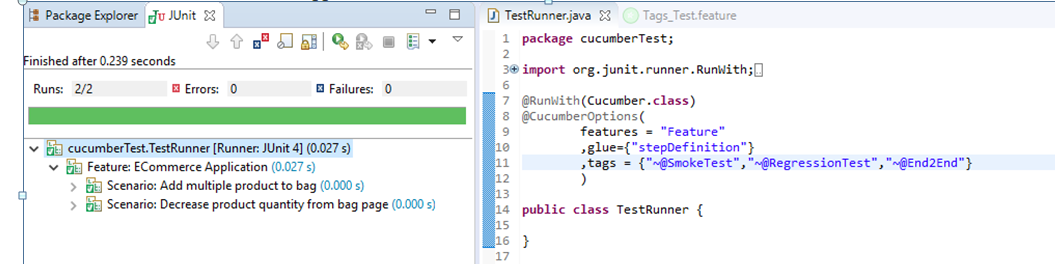
This is again a good feature of Cucumber Tags that you can even skip tests in the group execution. Special Character ***~***is used to skip the tags. This also works both for Scenarios and Features. And this can also works in conjunction with AND or OR.

***Execute all tests of the feature tagged as @FunctionalTests but skip scenarios tagged as @SmokeTest***

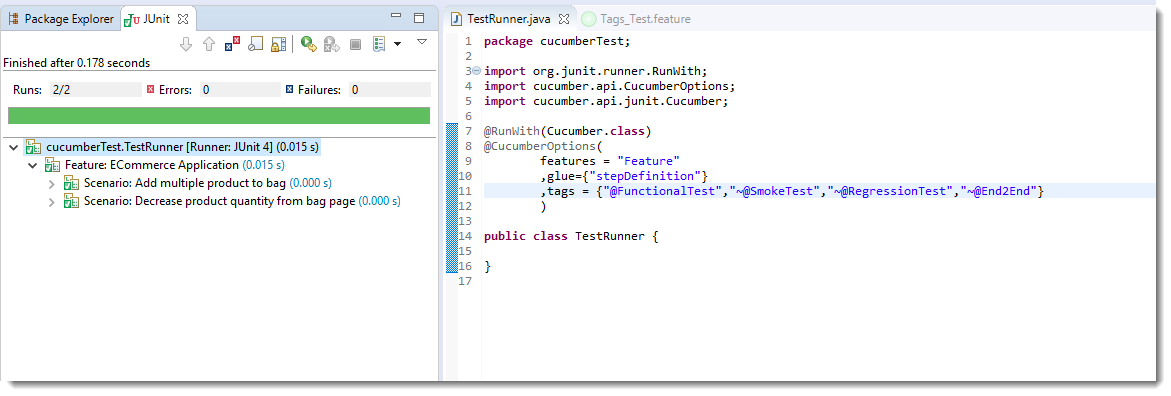


Note: This is AND condition, which means all the scenarios tagged as @FunctionalTest but not @SmokeTest. So total tests are 15 and smoke tests are 6, so it ran just 9 tests.

***Execute all tests which are not at all tagged in all the Features***



***Execute all tests which are not at all tagged in Single Feature***



It is fun to play with tags, especially when you have many features files with multiple scenarios.

**Data Driven Testing in Cucumber: -**

Most commercial automated software tools on the market support some sort of **Data Driven Testing**, which allows to automatically run a test case multiple times with different input and validation values. As *Selenium WebDriver* is more an automated testing framework than a ready-to-use tool. It takes extra efforts to support *data driven testing* in automated tests.

This is very often required in any automated test to pass data or to use the same test again with different data set. And the good part is that the **Cucumber** inherently supports **Data Driven Testing using Scenario Outline**. There are different ways to use the data insertion within the *Cucumber*and outside the *Cucumber*with external files.

**Data-Driven Testing in Cucumber**

* Parameterization without Example Keyword

**Data-Driven Testing in Cucumber using Scenario Outline**

* Parameterization with Example Keyword
* Parameterization using Tables

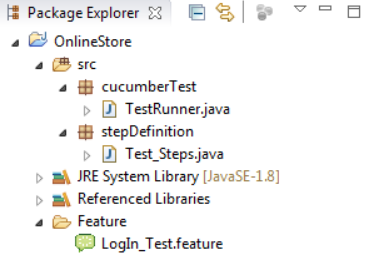
**Data-Driven Testing in Cucumber using External Files**

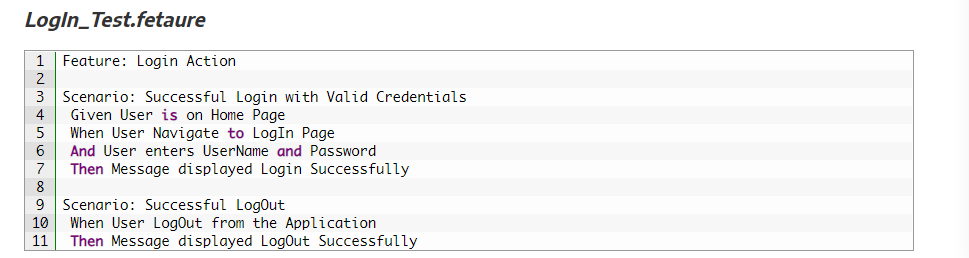
* Parameterization using Excel Files
* Parameterization using Json
* Parameterization using XML

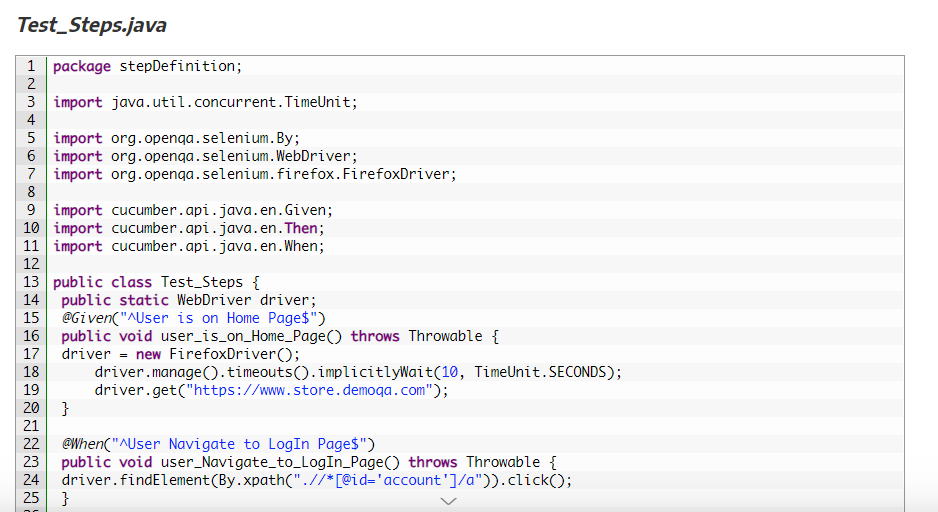
**Scenario Outline** – This is used to run the same scenario for 2 or more different sets of test data. **E.g**. In our scenario, if you want to register another user you can data drive the same scenario twice.

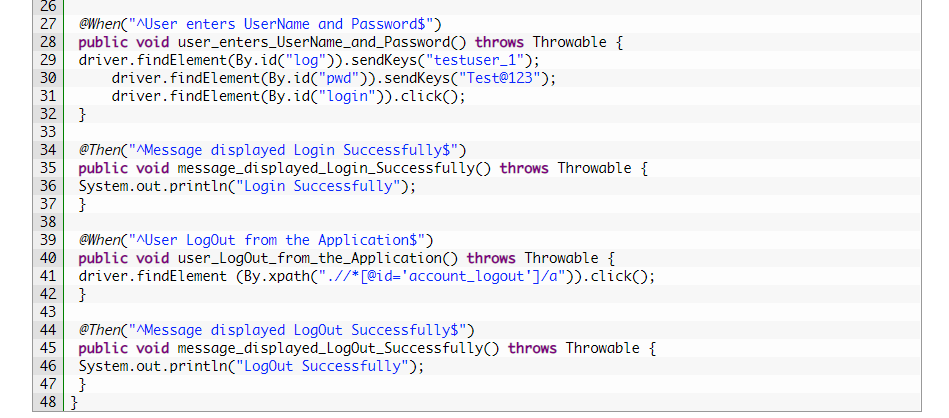
**Examples** – All scenario outlines have to be followed with the Examples section. This contains the data that has to be passed on to the scenario.

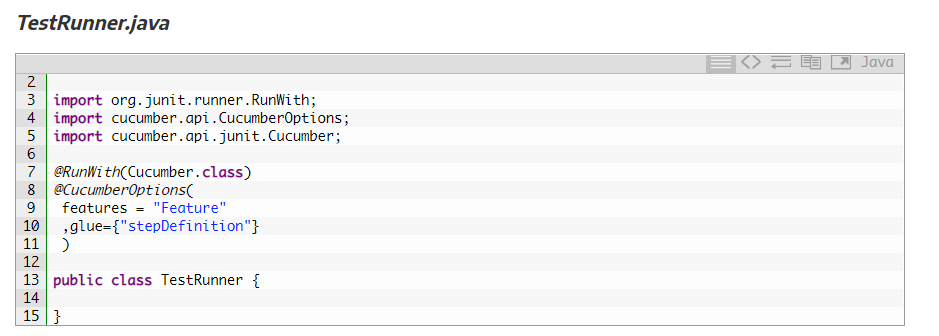
The project folder structure and code should be in the below state.









****

### Parameterizing without Example Keyword

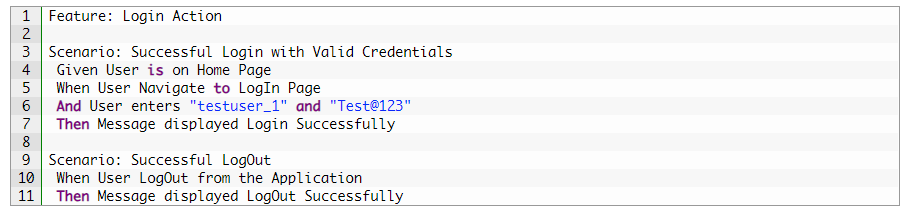
Now the task is to ***Parameterizing the UserName and Password***. Which is quite logical, why would anybody want to hardcode the UserName & Password of the application. As there is a high probability of changing both.

1) Go to the ***Feature File*** and change the statement where passing Username & Password as per below:

***And User enters “testuser\_1” and “Test@123“***

In the above statement, we have passed Username & Password from the Feature File which will feed in to Step Definition of the above statement automatically. Cucumber will do the trick for us. After the above changes, the code will look like this:

***LogIn\_Test.feature***

****

2) Changes in the Step Definition file is also required to make it understand the Parameterization of the feature file. So, it is required to update the Test Step in the Step Definition file which is linked with the above-changed Feature file statement. Use the below code:

***@When(“^User enters \”(.\*)\” and \”(.\*)\”$”)***

The same can be achieved by using the below code as well:

***@When(“^User enters \”([^\”]\*)\” and \”([^\”]\*)\”$”)***

With the help of the above statements, Cucumber will understand that the associated Test\_Step is expecting some parameters.

3) Same parameters should also go into the associated Test\_Step. As the Test step is nothing but a simple Java method, syntax to accept the parameter in the Java method is like this:

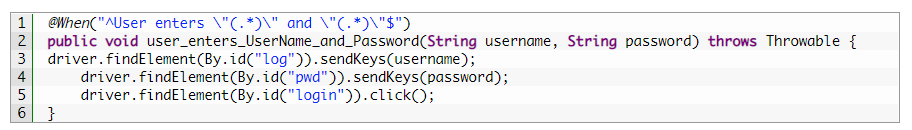
***public void user\_enters\_UserName\_and\_Password(String username, String password) throws Throwable {***

***}***

4) Now the last step is to feed the parameters in the actual core statements of Selenium WebDriver. Use the below code:

***driver.findElement(By.id(“log“)).sendKeys(username);***  
***driver.findElement(By.id(“pwd“)).sendKeys(password);***  
***driver.findElement(By.id(“login“)).click();***

After making the above changes, the method will look like this:

****

5) Run the test by Right Click on ***TestRunner class*** and Click ***Run As  > JUnit Test*** Application. You would notice that the Cucumber will open the Website in the browser and enter username & password which is passed from the Feature File.

**Data Driven Testing Using Examples Keyword: -**

In the last chapter of [**Parameterization in Cucumber**](https://toolsqa.com/cucumber/data-driven-testing-in-cucumber/), we learned how to parameterize data. But with that trick, only limited functionality can be achieved of Data-Driven. As the test can be run multiple times. But by now that you know the anatomy of a Data-Driven test, here’s a trick that simplifies the process of **Data-Driven testing using Cucumber**. Cucumber inherently supports Data-Driven testing by the use of the **Scenario Outline** and **Examples** section. It is with these keywords that Cucumber allows for easy Data-Driven testing to be completed where no changes need to be made to the Java file. In this tutorial we learn, How to **Implement a Scenario Outline in Data-Driven testing using Examples Keyword?**

Example keyword can only be used with the Scenario Outline Keyword.

* **Scenario Outline** – This is used to run the same scenario for 2 or more different sets of test data. E.g. In our scenario, if you want to register another user you can data drive the same scenario twice.
* **Examples** – All scenario outlines have to be followed with the Examples section. This contains the data that has to be passed on to the scenario.

## Data-Driven Testing Using Examples Keyword

If you understood the concept of [***Parameterization in Cucumber***](https://toolsqa.com/cucumber/data-driven-testing-in-cucumber/), you would find this one very easy. In this tutorial as well I am taking the same LogIn test scenario.

1) Enter the***Example Data*** just below the LogIn Scenario of the Feature File.

***Examples:***  
***| username  | password  |***  
***| testuser\_1 | Test@153 |***  
***| testuser\_2 | Test@153 |***

***Note***: The table must have a header row corresponding to the variables in the Scenario Outline steps.

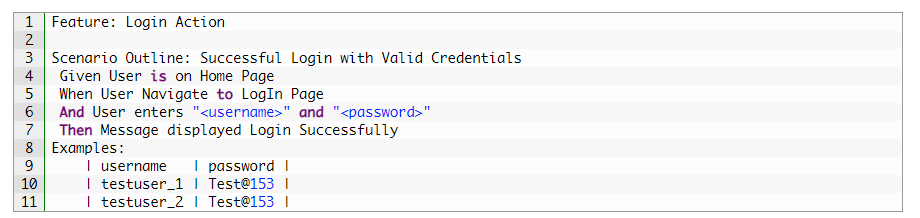
The Examples section is a table where each argument variable represents a column in the table, separated by “***|***”. Each line below the header represents an individual run of the test case with the respective data. As a result, if there are 3 lines below the header in the Examples table, the script will run 3 times with its respective data.

2) Need to update the Statement in the feature file, which tells Cucumber to enter username & Password.

***And User enters <username> and <password>***

Cucumber understands the above statement syntax and looks for the ***Examples*** Keyword in the test to read the Test Data.

***The complete code will look like this***: And There are no changes in **TestRunner**class.



There are no changes in **TestRunner**class.

There are no changes in **Test\_Steps** file from the previous chapter.

This takes the *parameterization* one step further: now our scenario has “**variables**” and they get filled in by the values in each row. To be clear: by defining this, the scenario will run two times, passing in one row at a time. This makes it very easy to define a lot of examples, edge cases, and special outcomes.  Instead of hardcoding the test data, variables are defined in the Examples section and used in the *Scenario Outline* section.

**Data Tables:-**

***Data Tables in Cucumber*** are quite interesting and can be used in many ways. DataTables are also used to handle large amounts of data. They are quite powerful but not the most intuitive as you either need to deal with a ***list of maps*** or a ***map of lists***. Most of the people get confused with Data tables & Scenario outline, but these two works completely differently.

## Difference between Scenario Outline & Data Table

***Scenario Outline:***

* This uses Example keyword to define the test data for the Scenario
* This works for the whole test
* Cucumber automatically run the complete test the number of times equal to the number of data in the Test Set

***Test Data:***

* No keyword is used to define the test data
* This works only for the single step, below which it is defined
* A separate code needs to understand the test data and then it can be run single or multiple times but again just for the single step, not for the complete test

While working on automation, we may face variety of scenarios. Each scenario carries a different meaning and needs.

Since the beginning, we have been taking an example of login functionality for a social networking site, where we just had two input parameters to be passed. Let’s think of some more possibility. How about “New user registration” functionality? Typically, what can be the input parameters while registering a new user for a social networking site? Something like the following −

* User Name
* Email Address
* Password
* Re-enter password
* Birth date
* Gender
* Phone number

**Feature** − New user registration.

Verify that the new user registration is unsuccessful after passing the incorrect inputs.

**Given** I am on a new user registration page.

**When** I enter the user name and an e-mail address as email address and password as, and re-enter password as and Birthdate as and Gender as and phone number as then the user registration should be unsuccessful.

It looks a bit messy at first glance. So, is there any better way to manage such chunk of inputs? Answer can be “Data Table”. Data table is a set of input to be provided for a single tag. This tag can be GIVEN, WHEN, or THEN.

Let’s write the above scenario with the help of data table and it will look like the following −

**Given** I am on a new user registration page

**When** I enter valid data on the page

| Fields | Values |

| First Name | Tom |

| Last Name | Kenny |

| Email Address | someone@someone.com |

| Re-enter Email Address | someone@someone.com |

| Password | Password1 |

| Birthdate | 01 |

**Then** the user registration should be successful.

The complete scenario is same as what we have done earlier. But the only difference is in this, we are not passing parameters in the step line and even we are not using Examples test data. We declared the data under the step only. So we are using Tables as arguments to Steps.

If you run the above scenario without implementing the step, you would get the following error in the Eclipse console window.

## DataTable_Cucumber_1

## Example

Let’s automate an example of a data table.

**Step 1** − Create a Maven Test Project named “DataTableTest”.

* Edit pom.xml −

**Step 2** − Create a package named **dataTable** under **src/test/java**

**Step 3** − Create a Feature file.

* Create a feature file, named as **dataTable**.feature inside the package dataTable (see section scenario outline for more detailed steps).
* Write the following text.

**Feature** − Data table

Verify that the new user registration is unsuccessful after passing incorrect inputs.

**Scenario:**

**Given** I am on the new user registration page

**When** I enter invalid data on the page

| Fields | Values |

| First Name | Tom |

| Last Name | Kenny |

| Email Address | someone@someone.com |

| Re-enter Email Address | someone@someone.com |

| Password | Password1 |

| Birthdate | 01 |

**Then** the user registration should be unsuccessful

* Save the file.

**Step 4** − Create step definition file.

* Create the step definition file named as ‘dataTable.java’ inside the package dataTable (see section scenario outline for more detailed steps).
* Write the following code.

package dataTable;

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.openqa.selenium.support.ui.Select;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

import cucumber.table.DataTable;

public class stepdefinition {

WebDriver driver = null;

@Given("^I am on new user registration page$")

public void goToFacebook() {

//Intiate web browser instance. driver = new FirefoxDriver();

driver.navigate().to("https://www.facebook.com/");

}

@When("^I enter invalid data on the page$")

public void enterData(DataTable table){

//Initialize data table

List<list> data = table.raw();

System.out.println(data.get(1).get(1));

//Enter data

driver.findElement(By.name("firstname")).sendKeys(data.get(1).get(1));

driver.findElement(By.name("lastname")).sendKeys(data.get(2).get(1));

driver.findElement(By.name("reg\_email\_\_")).sendKeys(data.get(3).get(1));

driver.findElement(By.name("reg\_email\_confirmation\_\_")).

sendKeys(data.get(4).get(1));

driver.findElement(By.name("reg\_passwd\_\_")).sendKeys(data.get(5).get(1));

Select dropdownB = new Select(driver.findElement(By.name("birthday\_day")));

dropdownB.selectByValue("15");

Select dropdownM = new Select(driver.findElement(By.name("birthday\_month")));

dropdownM.selectByValue("6");

Select dropdownY = new Select(driver.findElement(By.name("birthday\_year")));

dropdownY.selectByValue("1990");

driver.findElement(By.className("\_58mt")).click();

// Click submit button driver.findElement(By.name("websubmit")).click();

}

@Then("^User registration should be unsuccessful$")

public void User\_registration\_should\_be\_unsuccessful() {

if(driver.getCurrentUrl().equalsIgnoreCase("https://www.facebook.com/")){

System.out.println("Test Pass");

} else {

System.out.println("Test Failed");

}

driver.close();

}

}

* Save the file.

**Step 5** − Create a runner class file.

* Create runner class named as runTest.java inside the package.
* Write the following code.

package dataTable;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

@Cucumber.Options(format = {"pretty", "html:target/cucumber"})

public class runTest { }

* Save the file. Run JUnit test.

You may observe the following things upon successful execution.

* Facebook website gets loaded.
* Data will be entered on the registration page.
* Submit button will be clicked.
* We will see that home page will not displayed and “Test Pass” will be written on the console.

**Comment: -**

Comment is basically a piece of code meant for documentation purpose and not for execution. Be it a step definition file or a feature file, to make it more readable and understandable. So, it is important to use/put comments at appropriate places in the file. This also helps while debugging the code. Cucumber feature files can have comments at any place. To put comments, we just need to start the statement with “#” sign.

Different programming languages have got different norms for defining the comments. Let’s see how Cucumber deals with it.

* Step definition file − If you are using Java as a platform then mark your comments with “//”.
* Feature File − In case of feature file, we just need to put # before beginning your comment.

## Example

The highlighted text in the program refer to the comments in the code.

Feature: annotation

#This is how background can be used to eliminate duplicate steps

Background:

User navigates to Facebook

Given I am on Facebook login page

#Scenario with AND

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

#Scenario with BUT

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

But Relogin option should be available

**Hooks:-**

**What are Hooks in Cucumber?** - Cucumber supports ***hooks***, which are blocks of code that run ***before*** or ***after*** each scenario. You can define them anywhere in your project or step definition layers, using the methods ***@Before*** and ***@After***. ***Cucumber Hooks*** allows us to better manage the code workflow and helps us to reduce the code redundancy. We can say that it is an unseen step, which allows us to perform our scenarios or tests.

**Why Cucumber Hooks? -** In the world of testing, you must have encountered the situations where you need to perform the prerequisite steps before testing any test scenario. This prerequisite can be anything from:

* Starting a webdriver
* Setting up DB connections
* Setting up test data
* Setting up browser cookies
* Navigating to certain page
* or anything before the test

In the same way, there are always after steps as well of the tests like:

* Killing the WebDriver
* Closing DB connections
* Clearing the test data
* Clearing browser cookies
* Logging out from the application
* Printing reports or logs
* Taking screenshots on error
* or anything after the test

To handle these kinds of situations, cucumber hooks are the best choice to use. Unlike [***TestNG Annotaions***](https://toolsqa.com/selenium-webdriver/testng-annotations-groups-depends/), cucumber supports only two hooks (Before & After) which works at the start and the end of the test scenario. As the name suggests, @before hook gets executed well before any other test scenario, and @after hook gets executed after executing the scenario.

## How to implement Hooks in Cucumber Test

Let’s do some easy and small examples of Cucumber Hooks just to understand the concept. I will bring the intelligent usage of Hooks in my later tutorial series of **Designing Framework with Cucumber**.

### ****Test Hooks with Single Scenario****

**Feature File**

**Feature**: Test Hooks

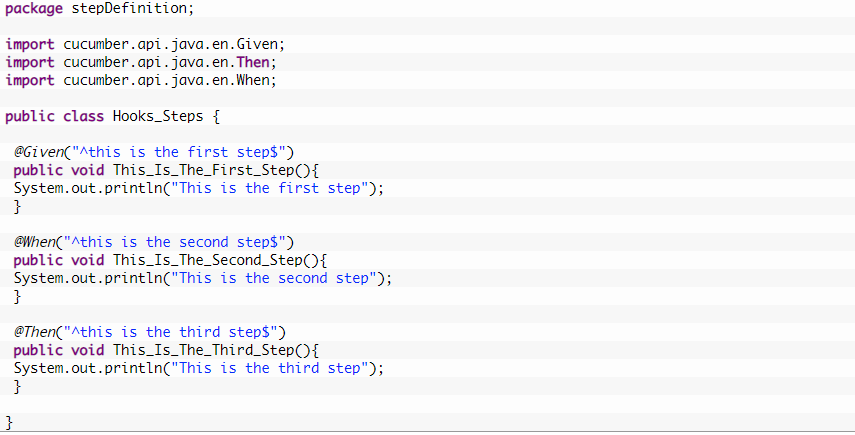
**Scenario**: This scenario is to test hooks functionality

**Given** this is the first step

**When** this is the second step

**Then** this is the third step

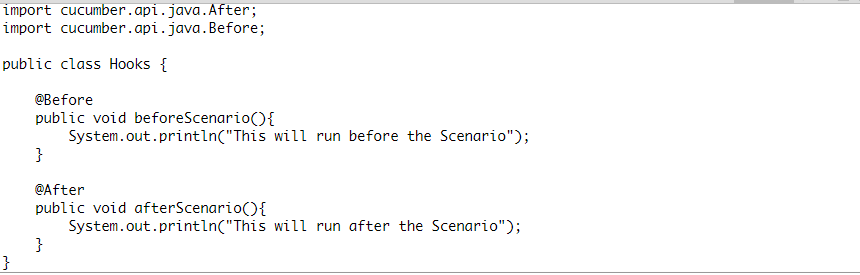
***Step Definitions***



***Note***: There is no logic used in the step definitions. Just printing the step summary log.

**Hooks**

package utilities;

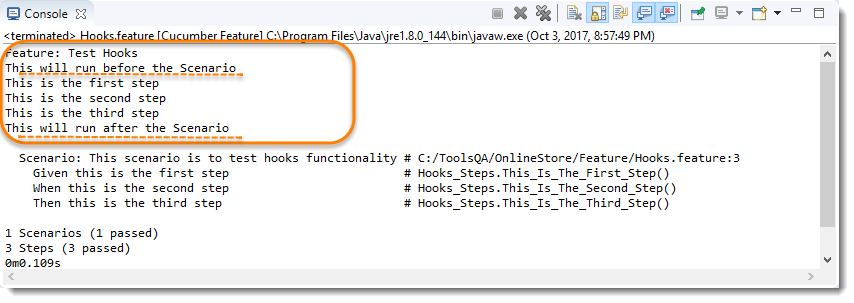


***Things to note***

* *An important thing to note about the after hook is that even in case of test fail, after hook will execute for sure.*
* *Method name can be anything, need not to be beforeScenario() or afterScenario(). can also be named as setUp() and tearDown().*
* *Make sure that the package import statement should be****import cucumber.api.java.After; & import cucumber.api.java.Before;***

Often people mistaken and import Junit Annotations, so be careful with this.

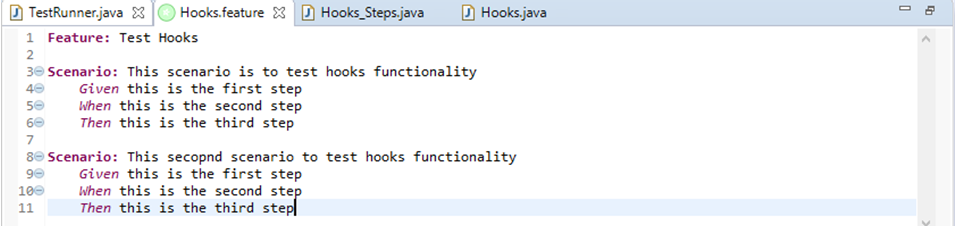
***Output***



No need for explanation, it is self-explanatory 🙂

### ****Test Hooks with Multiple Scenarios****

I just wanted to show you the reaction of Hooks with the multiple scenarios. Let’s just add one more Test Scenario in the feature file and run the feature again.

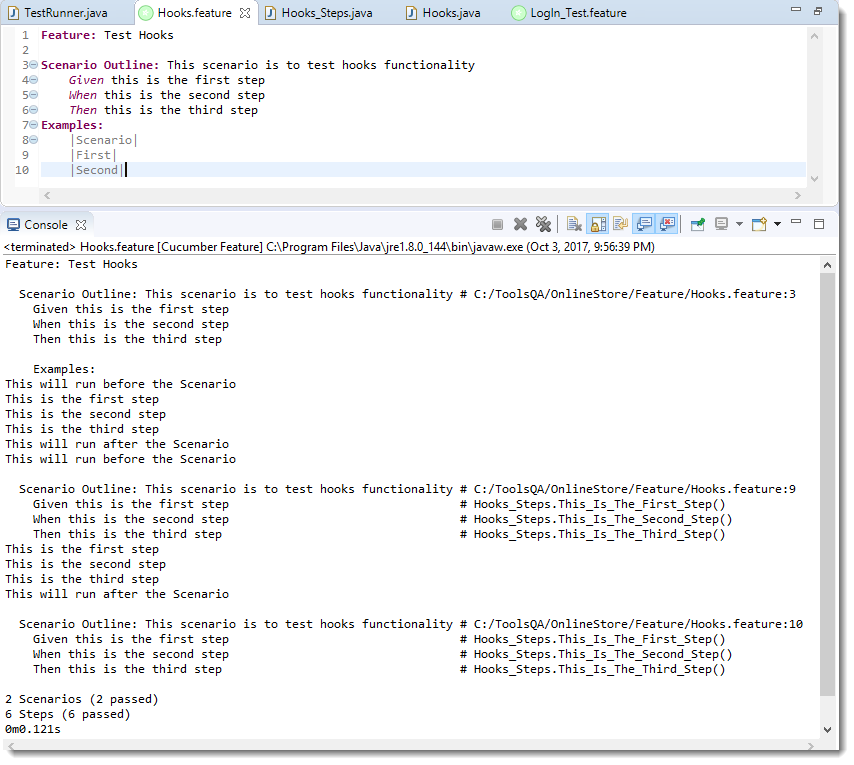




***Note***: Scenario Hooks execute before and after every scenario. In the above example, executed two times for two scenarios.

### ****Test Hooks with Example Scenarios****

Lets take a look when we have Scenario Outline with Examples.



***Note***: Again, in cucumber, every example is considered as a separate scenario. So the output is the same as the second example above.

**Tagged Hooks -** Now we know that if we need to do anything before or after the test, we can use @Before & @After hooks. But this scenario works till the time our prerequisites are the same for all the scenarios. For example till the time prerequisite for any test is to start the browser, hooks can solve our purpose. But what if we have different perquisites for different scenarios. And we need to have different hooks for different scenarios.

Again, Cucumbers has given a feature of Tagged Hooks to solve the above situation where we need to perform different tasks before and after scenarios.

We can also indicate if we want before and after hooks to be executed with a specific tag only. **Example** − @Before(‘@Web’). The same concept of tag logical and/or can be applied with hooks as well. **Example** − @Before(@dev,@wip), @Before(@dev,~@wip)

# Execution Order of Hooks: -

In this chapter, we will learn about ***Execution Order of Hooks***. If you ever have worked with [***TestNG***](https://toolsqa.com/selenium-webdriver/testng-prioritizing-sequencing/), you must know that it performs the execution in a certain order. The same way [***Cucumber***](https://toolsqa.com/cucumber/cucumber-tutorial/) also executes the hooks in a certain order. But there are ways to change the order of the executing according to the need of the test or the framework.

Order hooks to run in a particular sequence is easy to do. As we already know the way to specify hooks in cucumber-like putting an annotation just above the scenario. Ordering also works the same way but the only difference is that it required an extra parameter. This extra parameter decides the order of execution of the certain hook.

***For example*** ***@Before***, and if you want to specify the order it will become ***@Before(value = 1)***.

The same goes with any [***Tags***](https://toolsqa.com/cucumber/cucumber-tags/) or [***Hooks***](https://toolsqa.com/cucumber/cucumber-hooks/) available in Cucumber including [***Tagged Hooks***](https://toolsqa.com/cucumber/tagged-hooks-in-cucumber/) as well.

### ****Exercise on Order Hooks****

Let’s take a different approach this time and do an exercise with the multiple hooks without any ordering value. Later we will bring order value and see the difference in output.

**Feature File**

**Feature**: Test Order Hooks

**Scenario**: First scenario to test Order Hooks functionality

Given this is the first step

When this is the second step

Then this is the third step

**Scenario**: Second scenario to test Order Hooks functionality

Given this is the first step

When this is the second step

Then this is the third step

This is the same plain feature file that we used in previous chapters on Tags, Hooks, and Tagged Hooks.

**Step Definitions**

package stepDefinition;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class Hooks\_Steps {

@Given("^this is the first step$")

public void This\_Is\_The\_First\_Step(){

System.out.println("This is the first step");

}

@When("^this is the second step$")

public void This\_Is\_The\_Second\_Step(){

System.out.println("This is the second step");

}

@Then("^this is the third step$")

public void This\_Is\_The\_Third\_Step(){

System.out.println("This is the third step");

}

}

Again, steps definitions are also same as previous chapters.

**Hooks**

package utilities;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class Hooks {

@Before

public void beforeScenario(){

System.out.println("This will run before the every Scenario");

}

@Before

public void beforeScenarioStart(){

System.out.println("-----------------Start of Scenario-----------------");

}

@After

public void afterScenarioFinish(){

System.out.println("-----------------End of Scenario-----------------");

}

@After

public void afterScenario(){

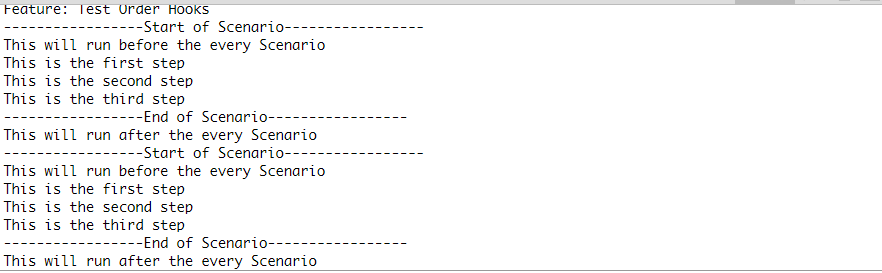
System.out.println("This will run after the every Scenario");

}

}

Above we mentioned two before and two after hooks. Execute the feature file as a whole and see the output below.

***Output***



I would say that I want —–End of Scenario—— to be printed after the This will run after the every Scenario.

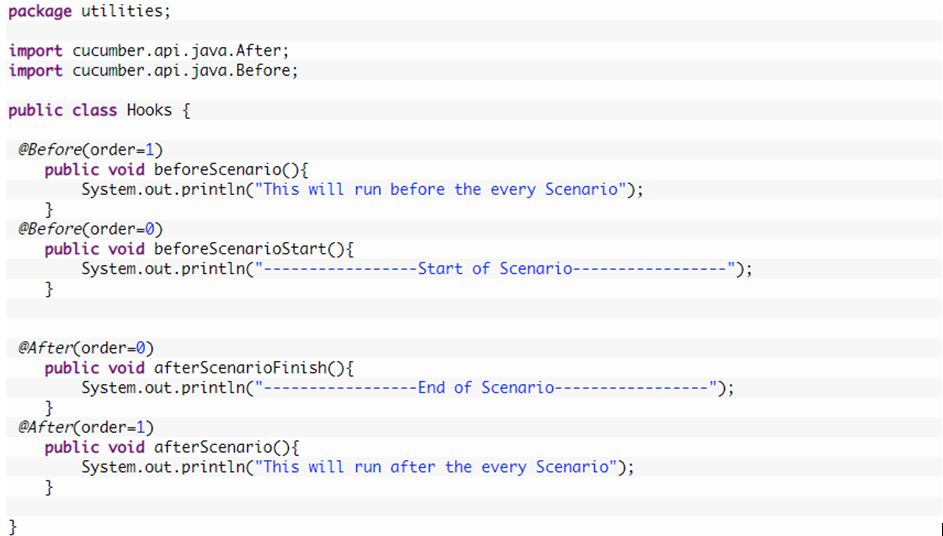
### ****How to set the Order or Priority of Cucumber Hooks?****

***The very important thing to note here is:***

* ***@Before(order = int) :*** This runs in increment order, means value 0 would run first and 1 would be after 0.
* ***@After(order = int) :*** This runs in decrements order, means apposite of @Before. Value 1 would run first and 0 would be after 1.

So, as per the logic above the Hooks file will look like below.

***Hooks***



OutPut



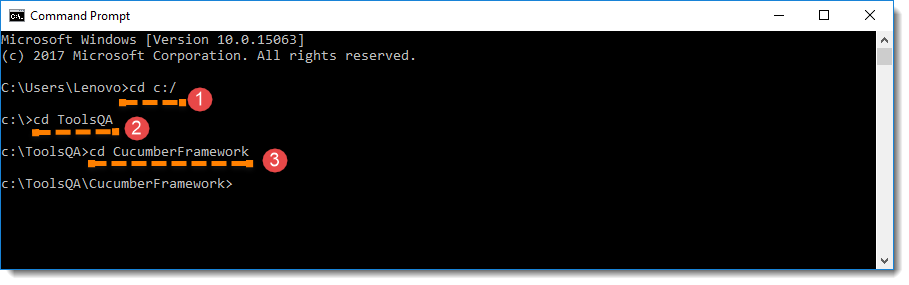
Now just play around with the Hooks + Order, also try to figure out how it behaves when you use the Ordering with Tagged Hooks.

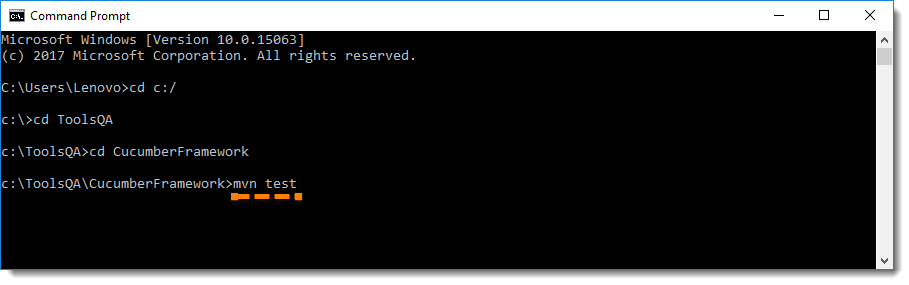
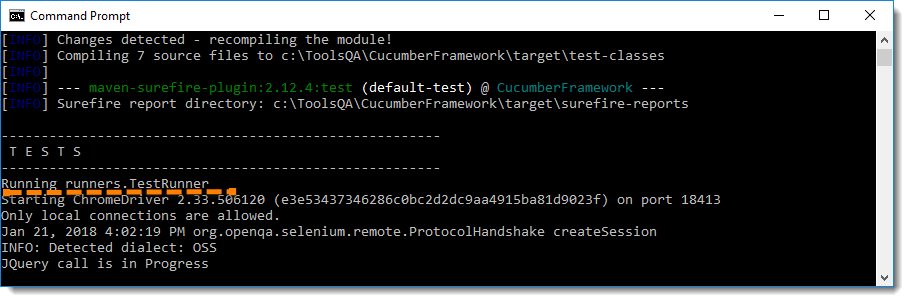
## Run Cucumber Test from Command Line / Terminal: -

There are different ways to run Cucumber Test from command line. Tests can be run by using JUnit and Maven as well. But maven is the most suggested way and has extra benefits to it. This is why we started this Project as Maven project. And remember, Maven has a lot of advantages over other build tools, such as dependency management, lots of plugins and the convenience of running integration tests. Maven will allow our test cases to be run in different flavors, such as from the ***Terminal***, integrating with ***Jenkins***, and ***parallel*** ***execution***.

### *****Run Test from Command Line*****

1. Open the***command prompt*** and ***cd*** until the project root directory.



2. First, let’s run all the Cucumber Scenarios from the command prompt. Since it’s a Maven project and we have added Cucumber in ***test scope*** dependency and all features are also added in ***src/test*** packages, run the following command in the command prompt: ***mvn test***  
  
You would notice below that it actually triggered the ***TestRunner*** file.  


***Build Success Output***

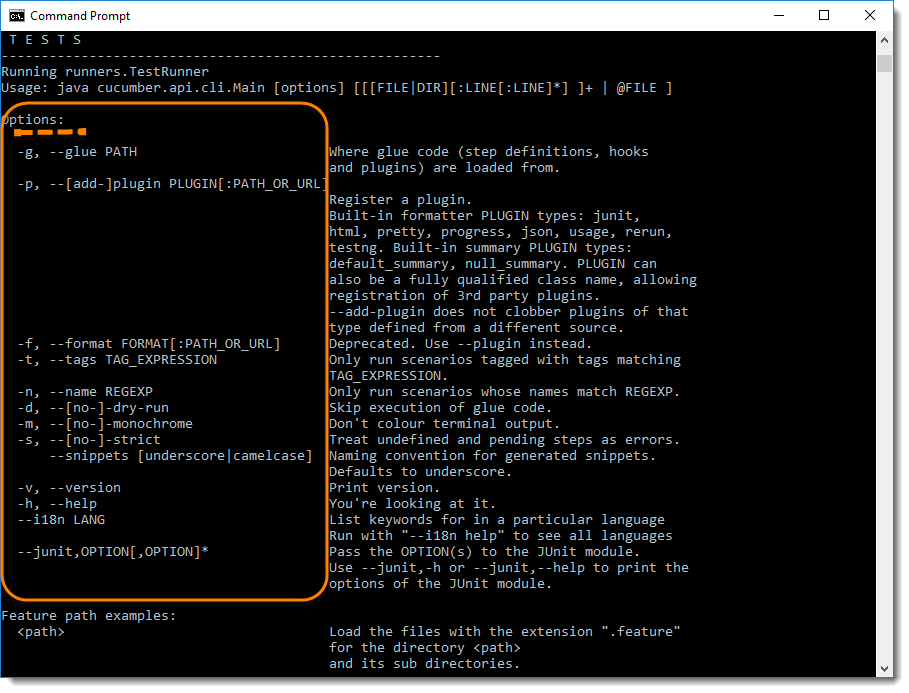


***Note***: mvn test runs Cucumber Features using Cucumber’s JUnit Runner. The @RunWith (Cucumber.class) annotation on the TestRunner class tells JUnit to kick off Cucumber. Cucumber run time parses the command-line options to know what Feature to run, where the Glue Code lives, what plugins to use, and so on.

On the other hand, if you run test from eclipse when you use the JUnit Runner, these options are generated from the @CucumberOptions annotation on your test.

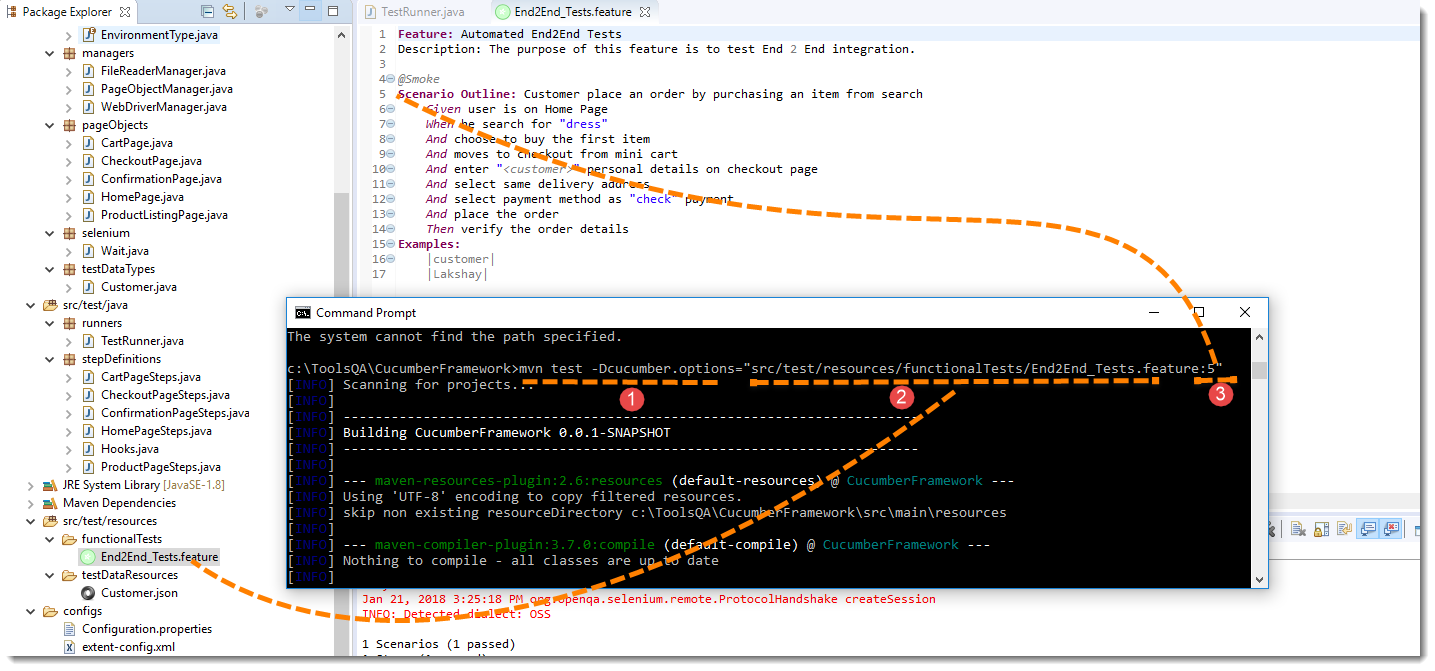
## Overriding Cucumber Options

At times, you would need to override the configurations specified in @CucumberOptions. If we want to override the configurations mentioned in the ***Runner***, then we need to use following command: **mvn test –DCucumber.options=”Your Options”**

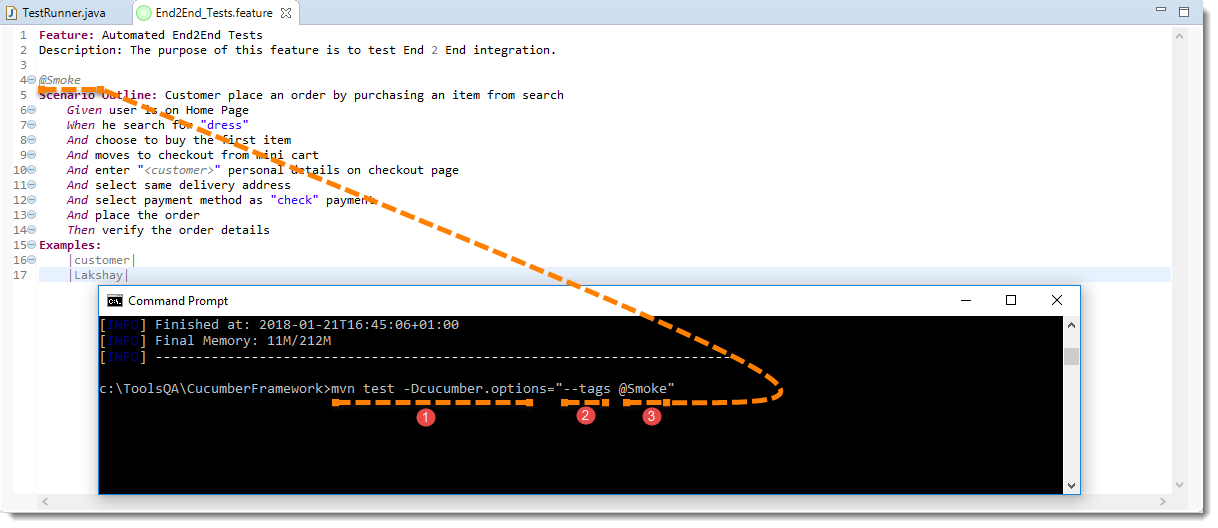
If you need help on these Cucumber options, then enter the following command in the command prompt and look at the output: ***mvn test -Dcucumber.options=”–help”***  


***Note :***This gives you the list of parameters that can be passed through command line using CucumberOptions.

### ****Running a Scenario from Command Line****

If we want to run single ***Scenario*** from the ***cmd,***this is how we specify***:***mvn test -Dcucumber.options=”feature file path” + “line number of the scenario”  
[](https://toolsqa.com/wp-content/gallery/cucumber/Running-Cucumber-Test-from-Command-Line-5.png)  
***Note***: In the preceding code, ***“5”*** is the Feature file line number where a Scenario starts.

### Running Scenarios using Tags from Command Line

If you want to run the test cases associated to ***Tags***, this is how we specify : mvn test -Dcucumber.options=”–tags @tag Name”  


### ****Overriding Report Plugin and Path****

If you want to generate a different report or if you wish to change the report path. This how you do it: mvn test -Dcucumber.options=”–plugin junit:target/cucumber-reports/report.xml”.

Please look at our [***Cucumber Report***](https://toolsqa.com/selenium-cucumber-framework/cucumber-reports/) to see what all reports you can generate using @CucumberOptions.

### ****Running a Feature file only from Command Line****

If you want cucumber to run just a single feature file or multiple feature file, you can pass parameter for the same from command line. This how you do it:

mvn test -Dcucumber.options=”src/test/resources/functionalTests/End2End\_Tests.feature”

### ****Passing multiple Parameter at once****

Its also possible to pass multiple options at once. This is how it can be done mvn test -Dcucumber.options=”SomeThing” -Dcucumber.options=”SomeThing” -Dcucumber.options=”SomeThing”

**mvn test -Dcucumber.options=”src/test/resources/functionalTests/End2End\_Tests.feature” -Dcucumber.options=”–tags @Smoke”**

**BackGround: -**

***Background in Cucumber***is used to define a step or series of steps that are common to all the tests in the feature file. It allows you to add some context to the scenarios for a feature where it is defined. A Background is much like a scenario containing a number of steps. But it runs before each and every scenario were for a feature in which it is defined.

For example, to purchase a product on any E-Commerce website, you need to do the following steps:

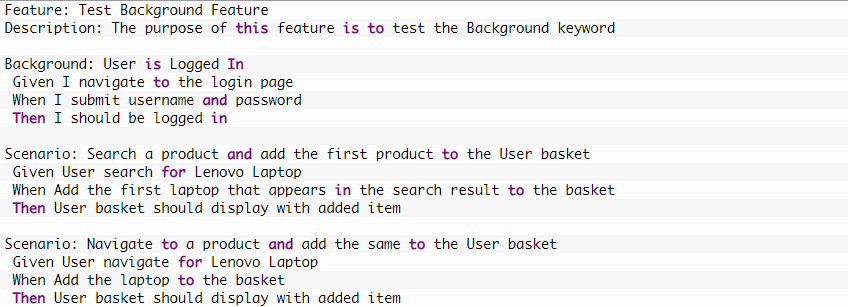
* Navigate to Login Page
* Submit UserName and Password

After these steps only you will be able to add a product to your cart/basket and able to perform the payment. Now as we are in a feature file where we will be testing only the Add to Cart or Add to Bag functionality, these tests become common for all tests. So instead of writing them again and again for all tests, we can move it under the Background keyword.

**Background in Cucumber**

Let’s start with a simple exercise to build the understanding of Background usage in Cucumber test. If we create a feature file of the scenario we explained above, this is how it will look like:

***Feature File***

****

In the above example, we have two different scenarios where a user is adding a product from search and directly from the product page. But the common step is to logIn to website for both the scenario. This is why we create another Scenario for LogIn but named it as Background rather than a Scenario. So that it executes for both the Scenarios

**Step Definitions – As usual**

***Note***: Hope you noticed the impact. The background ran two times in the feature before each scenario.

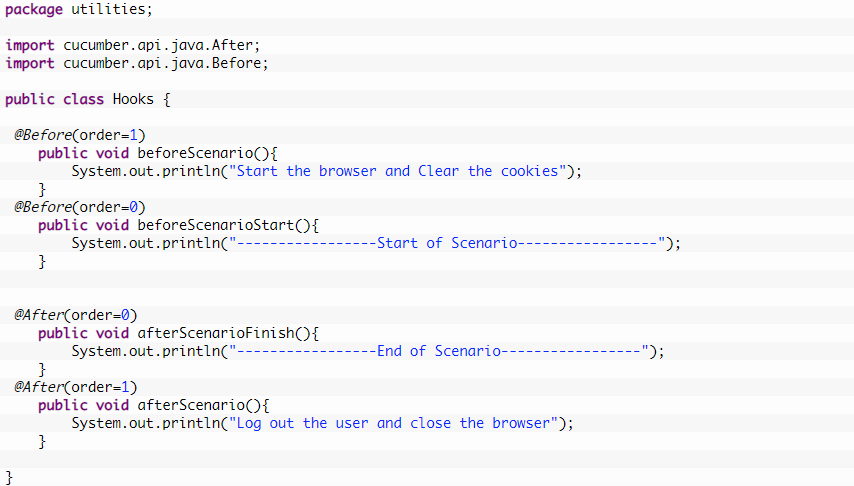
### *****Background with Hooks*****

This is so interesting to see the working of Background with Hooks. The background is run before each of your scenarios but after any of your [***@Before hook***](https://toolsqa.com/cucumber/cucumber-hooks/).

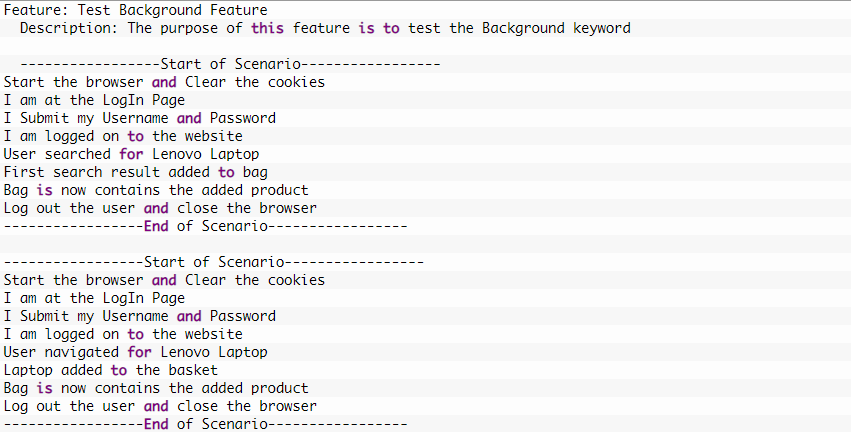
To get it straight, let’s assign a task to the Before & After Hook in the same test.

* @Before: Print the starting logs
* @Before: Start browser and Clear the cookies
* @After:  Close the browser
* @After: Print the closing logs

**Hook File**



**OutPut**



### ****Good practices for using Background****

It is really necessary to understand the right usage of Background. As hooks as well give similar kind of functionality and moreover almost all the tasks can be done by hooks as well. This is why it is critical to use the background in the right place in the test.

**Feature Dependency**

Any feature level dependency should be tie with the background and any scenario level dependency should be tie with hooks.

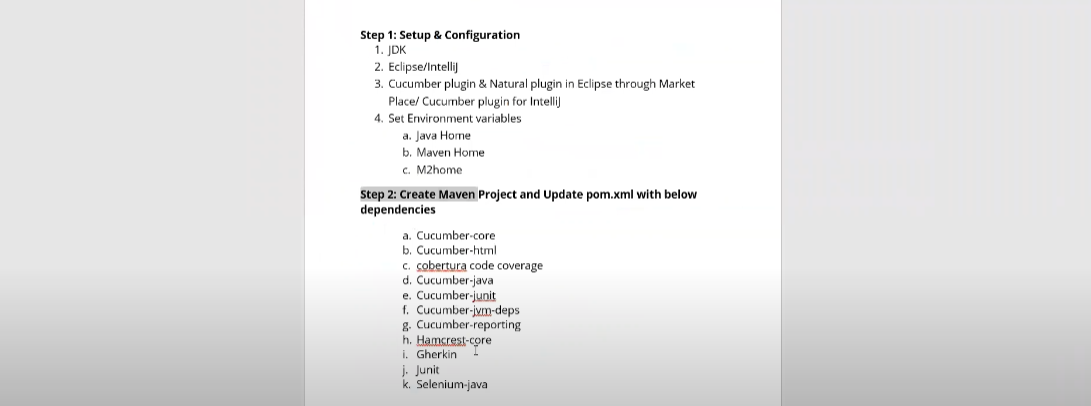
**Keep Background short.**

You’re expecting the user to actually remember this stuff when reading your scenarios. If the background is more than 4 lines long, can you move some of the irrelevant details into high-level steps? See Calling Steps from Step Definitions.

**Make your Background section vivid.**

You should use colorful names and try to tell a story, because the human brain can keep track of stories much better than it can keep track of names like “User A”, “User B”, “Site 1”, and so on.

**PROJECT**



**Add all the dependencies in the POM.xml file.**

**Install Cucumber plugin from eclipse marketplace.**

A Sample POM.xml file below

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>Project4\_nopCommerce\_BDD\_Cucumber</groupId>

<artifactId>Project4\_nopCommerce\_BDD\_Cucumber</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>Project4\_nopCommerce\_BDD\_Cucumber</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.sun</groupId>

<artifactId>tools</artifactId>

<version>1.6</version>

<scope>system</scope>

<systemPath>C:\Program Files\Java\jdk1.8.0\_231\lib\tools.jar</systemPath>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>6.1.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>3.141.59</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>6.1.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>6.1.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-core</artifactId>

<version>6.1.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-html</artifactId>

<version>0.2.7</version>

</dependency>

<dependency>

<groupId>net.sourceforge.cobertura</groupId>

<artifactId>cobertura</artifactId>

<version>2.1.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-core</artifactId>

<version>2.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>gherkin</artifactId>

<version>13.0.0</version>

</dependency>

<dependency>

<groupId>net.masterthought</groupId>

<artifactId>cucumber-reporting</artifactId>

<version>5.3.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-jvm-deps</artifactId>

<version>1.0.6</version>

<scope>provided</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.18.1</version>

<configuration>

<testFailureIgnore> false </testFailureIgnore>

<suiteXmlFiles>

<suiteXmlFile>src//test//resources//runner//testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.18.1</version>

<configuration>

<testFailureIgnore>false</testFailureIgnore>

<suiteXmlFiles>

<suiteXmlFile>src/test/resources/runner/testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

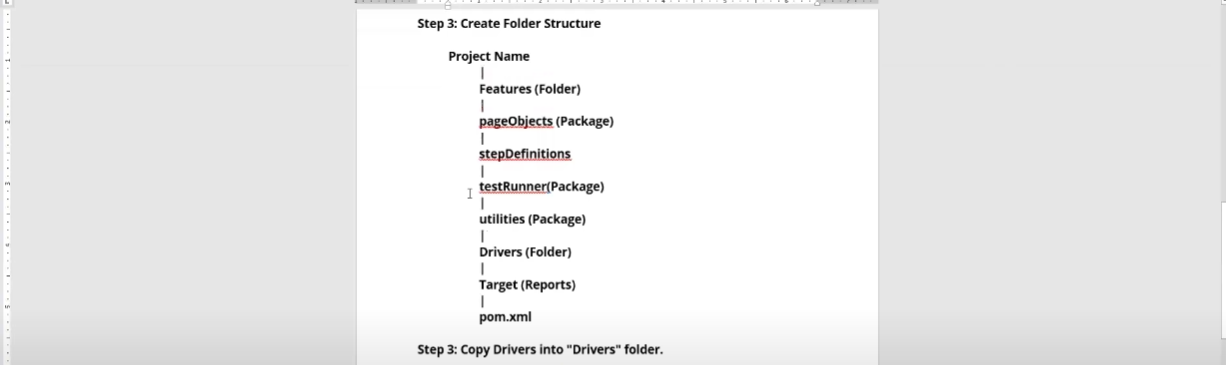
</plugin>

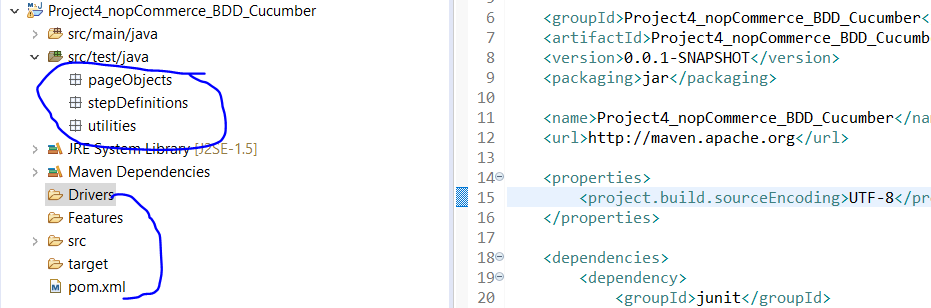
</plugins>

</build>

</project>

Create folder structure in eclipse

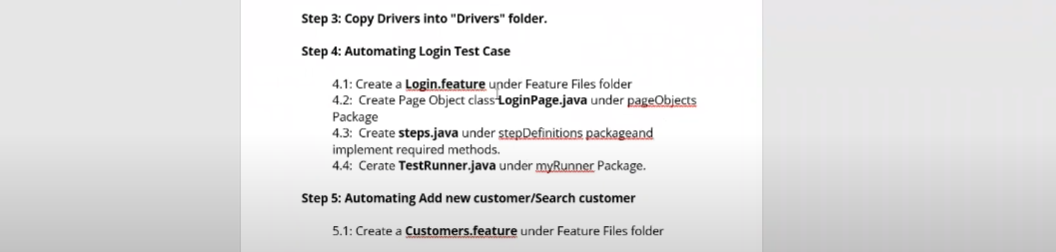




**Webpage to be automated -** [**https://admin-demo.nopcommerce.com/**](https://admin-demo.nopcommerce.com/)

Email id - [admin@yourstore.com](mailto:admin@yourstore.com)

Password – admin



Source of Information –

<https://www.tutorialspoint.com/cucumber/cucumber_overview.htm>

<https://www.toolsqa.com/cucumber-tutorial/>