CUCUMBER

**For Concepts :** Refer to [**https://github.com/automateeverything891/CucumberConcepts**](https://github.com/automateeverything891/CucumberConcepts)

**For Framework Code :** Refer to[**https://github.com/automateeverything891/CucumberFramework**](https://github.com/automateeverything891/CucumberFramework)

# What is BDD

Behaviour Driven Development (BDD) is a software development approach that has evolved from [TDD](https://blog.testlodge.com/what-is-tdd/) (Test Driven Development). It differs by being written in a shared language, which improves communication between tech and non-tech teams and stakeholders. In both development approaches, tests are written ahead of the code, but in BDD, tests are more user-focused and based on the system’s behaviour.

# Cucumber & Behaviour Driven Development

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.

Behavior Driven testing is an extension of TDD. Like in TDD and BDD also we write tests first and then add application code. The major difference that we get to see here are

* Tests are written in plain descriptive English type grammar
* Tests are explained as behavior of application and are more user-focused
* Using examples to clarify requirements

This difference brings in the need to have a language that can define, in an understandable format.

# Gherkin – Business Driven Development

Before diving into Gherkin, it is necessary to understand the importance and need of a common language across different domains of project. By different domains I By different domains I mean Clients, Developers, Testers, Business analysts and the Managerial team. Let’s start by talking about usual problems of a development project first and then we will move to a solution, while doing so we will come across the need for a common language.

# Cucumber BDD for Testing

1. Cucumber helps improve communication

Cucumber assists in improving communication between technical and non-technical members of the same project. Let’s have a look at the requirement below and its automation tests:

As an Admin User,

I would like to login using Valid user credentials

Feature: Login Functionality

Scenario: Valid Login

Given navigate to Login page

When user is in login page

Then user logins using valid credential

And home page should be displayed

With TestNG, the above test scenario can implement as below:

@test

public void validLogin() {

// Open the Browser and navigate to login page

WebDriverManager.chromedriver().setup();

WebDriver driver = new ChromeDriver();

driver.get(<https://ui.freecrm.com/>);

String title = driver.getTitle();

Assert.assertEquals(title, "ui.freecrm.com");

// Enter valid credentials and login

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

// User should be in Home page

String homepage = driver.findElement(By.xpath("//span[text()='Home']")).getText();

System.out.println(homepage);

Assert.assertEquals(homepage, "Home");

}

We can write the same test case using Cucumber:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Feature: Login Functionality  Scenario: Valid Login  Given navigate to Login page "https://ui.freecrm.com/"  When user is in login page  Then user logins using valid credential "josephkuruvilla891@gmail.com" and "Jakay07@"  And home page should be displayed |

Both automation test scripts above execute well to complete the test automatically. But do all testers of your team make out these tests? Is there a possibility of other business analysts and other stakeholders using these tests again at the acceptance testing (AT) stage?

The automation test with TestNG may be tough for most manual testers and BAs to catch up. Moreover, it is not possible to use this test again for AT. As a result, based on these flaws mentioned before, this can not be considered as a suitable method.

In contrast, we develop/ create the automation test using Cucumber in a business domain language or in natural language, which all members of the software project team can easily make out. Communication is vital for any development team, especially in the Agile team. There are usually a lot of continuous chats, discussions, or even arguments happening among developers and testers to figure out what the correct behavior of a feature is. By using Cucumber, the developers can develop the same feature specification now for testing by testers. It is a powerful tool because it can help lower the risk of misunderstanding as well as the communication breakdown.

2. Cucumber is an Automated Acceptance Testing Tool

The acceptance test is generally carried out by BAs/customers to make sure that the development team has built specific features. A common activity in this testing stage is verifying the system against the original requirements with specific, real data from production. Cucumber testing not only follows the requirements as its test scenarios but also helps BAs or Product Manager to adjust test data quickly. Here is a demonstration with a little adjustment:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | As an Admin User,  I would like to login using Valid user credentials  Feature: Login Functionality  Scenario: Valid Login  Given navigate to Login page  When user is in login page  Then user logins using valid credential  And home page should be displayed |

We write the automation test in the Cucumber framework:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Feature: Login Functionality  Scenario: Valid Login  Given navigate to Login page "https://ui.freecrm.com/"  When user is in login page  Then user logins using valid credential “<username>” and “<password>”  And home page should be displayed  Examples:  | username | password |  | [josephkuruvilla891@gmail.com](mailto:josephkuruvilla891@gmail.com) | Jakay07@ | |

3. All testers can take part in automation test with Cucumber BDD

In addition to improving communication within the members of the same testing team, Cucumber also helps leverage tester’s skills efficiently. The expertise gap always exists in every organization. In other words, some testers have great technical expertise in programming utilizing automated testing, while others are performing manual testing with limited programming skills in the same team. Thanks to Cucumber, all testers, irrespective of their skill levels, can participate in the process of performing automation tests.

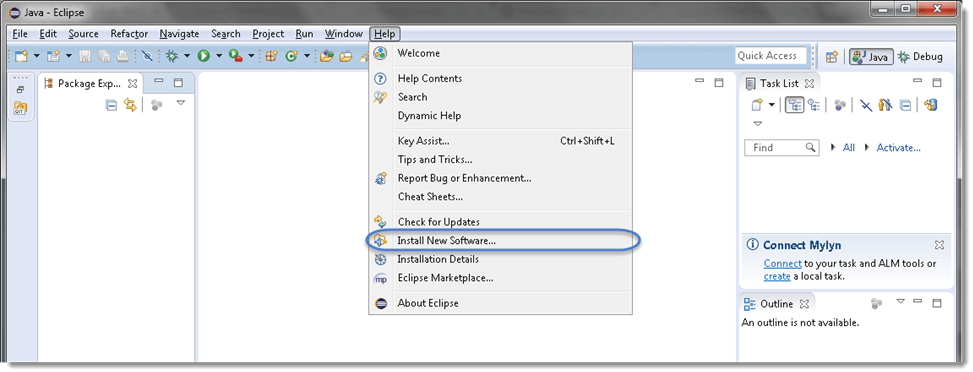
Let’s take a look at the above example:

* Any tester who knows the business logic and workflow can write feature files, add more scenarios, and test datasets.
* Moreover, any tester who has a basic knowledge of programming and know-how to create objects, access properties, call methods, can generate step definitions.
* Any tester with a higher programming skill level can take part in the process of making a framework, define data source connection, and so on.

# Install Cucumber Eclipse Plugin

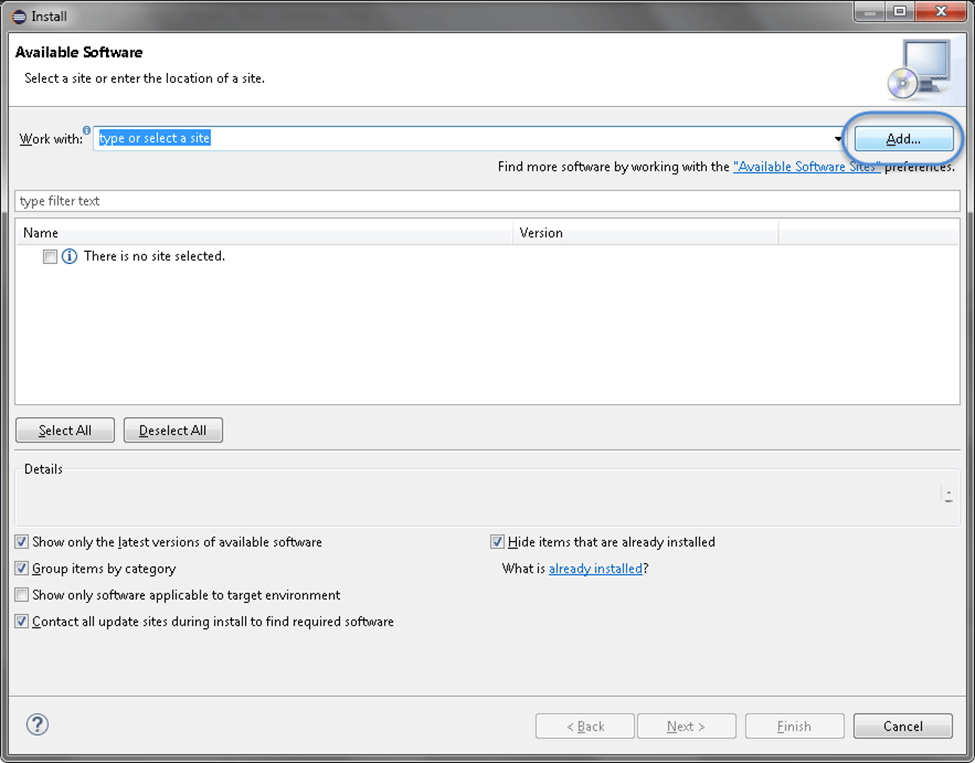
**Step 1**

In the first step, make sure the availability of a good Internet connection. Now, launch the **Eclipse IDE** then go to Help menu, and click "**Install New Software**".



**Step 2**

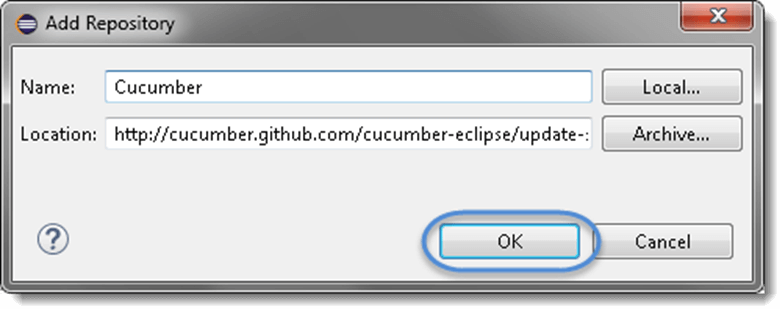
After clicking "**Install New Software**", a window will be prompted, on this window, click the "**Add**" button.



**Step 3**

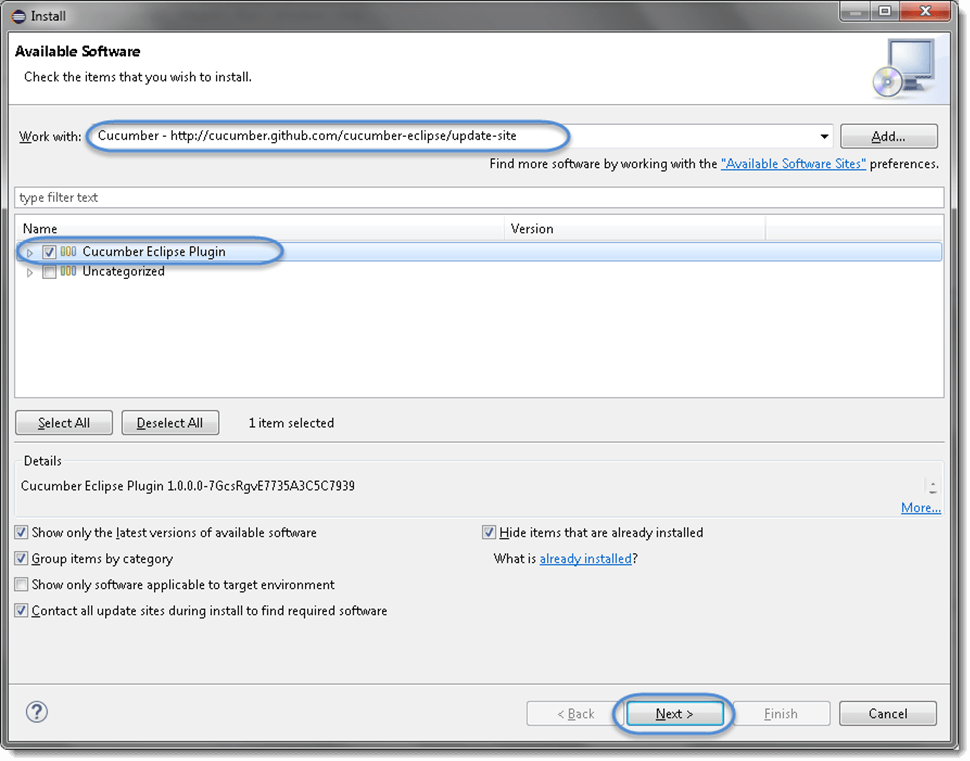
After clicking the "**Add**" button, give the **Name** in the text box as per your choice. We provided "**Cucumber**".

Now, in the Location text box type "[**http://cucumber.github.com/cucumber-eclipse/update-site**](http://cucumber.github.com/cucumber-eclipse/update-site)" as locationand then click **OK**.



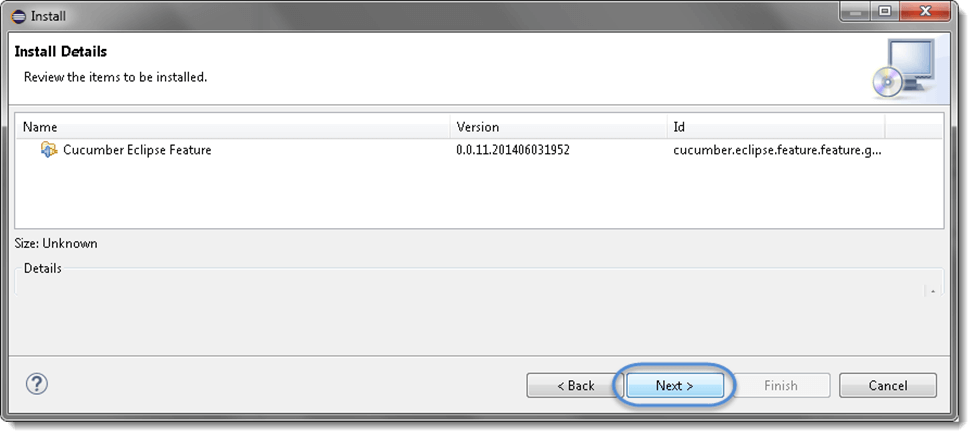
**Step 4**

Now, you will come back to the previous window, but this time you will see "**Cucumber Eclipse Plugin**" in the software list. Just click "**CheckBox**" and then the "**Next**" button.



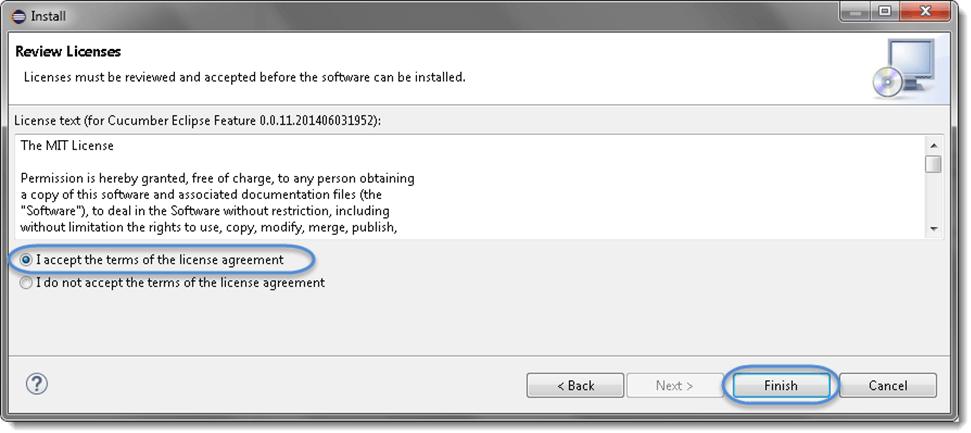
**Step 5**

Now, click on the "**Next**" button.



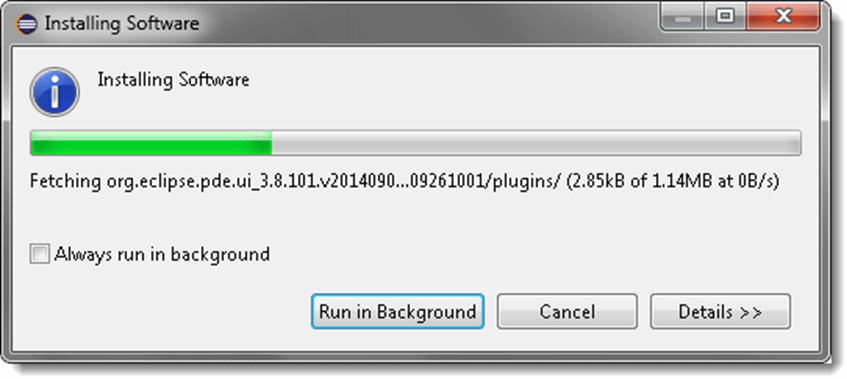
**Step 6**

Click the check box "***I accept the terms of the license agreement***" on the license window then click **Finish**.



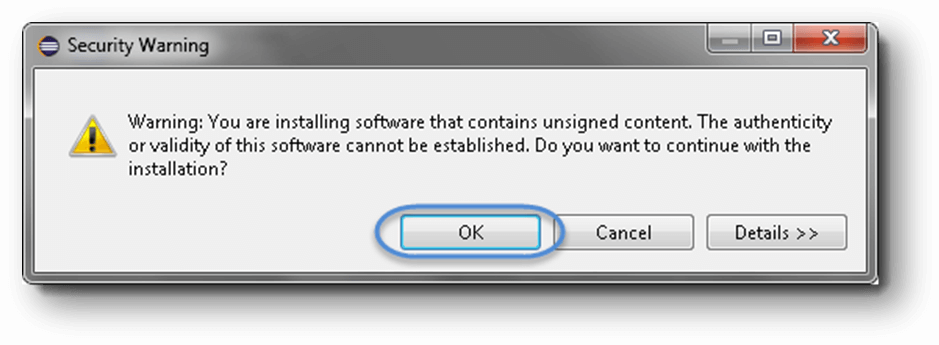
**Step 7**

Now, the installation will be started. It can take some time to be completed.



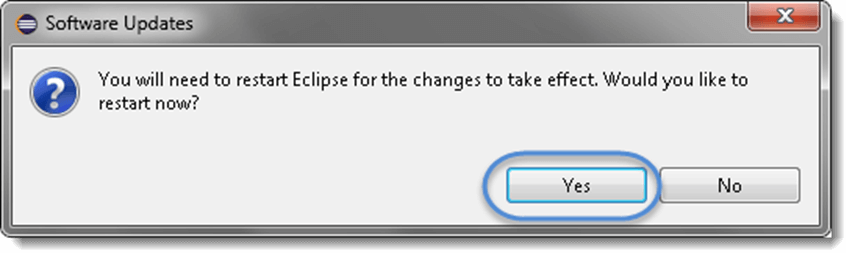
**Step 8**

If you encounter a Security warning, just click **OK**.



**Step 9**

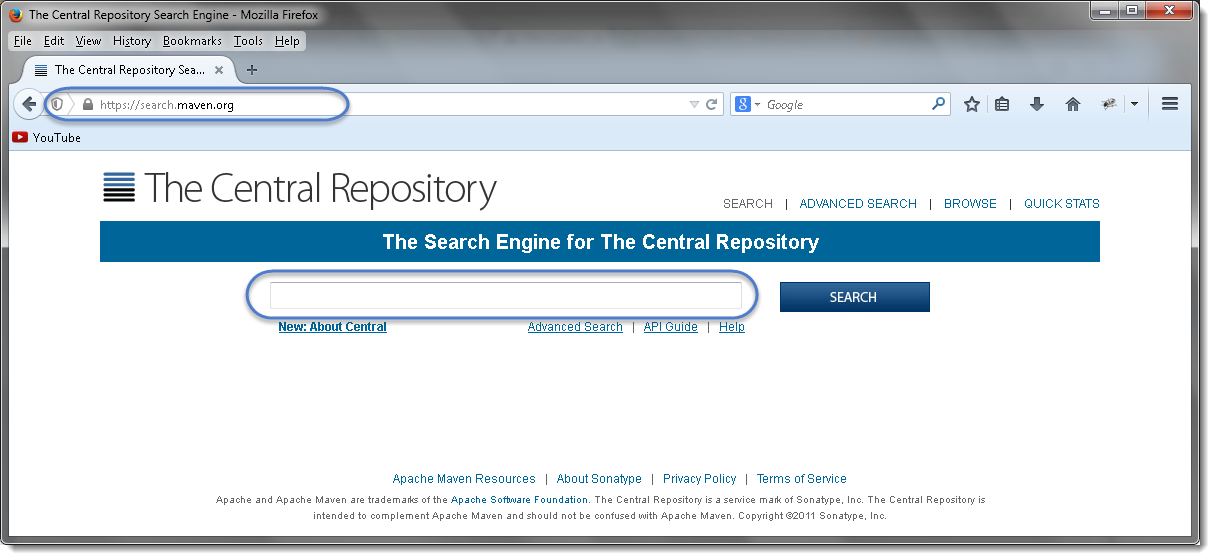
The installation has been completed, now just click the "**Yes**" button.



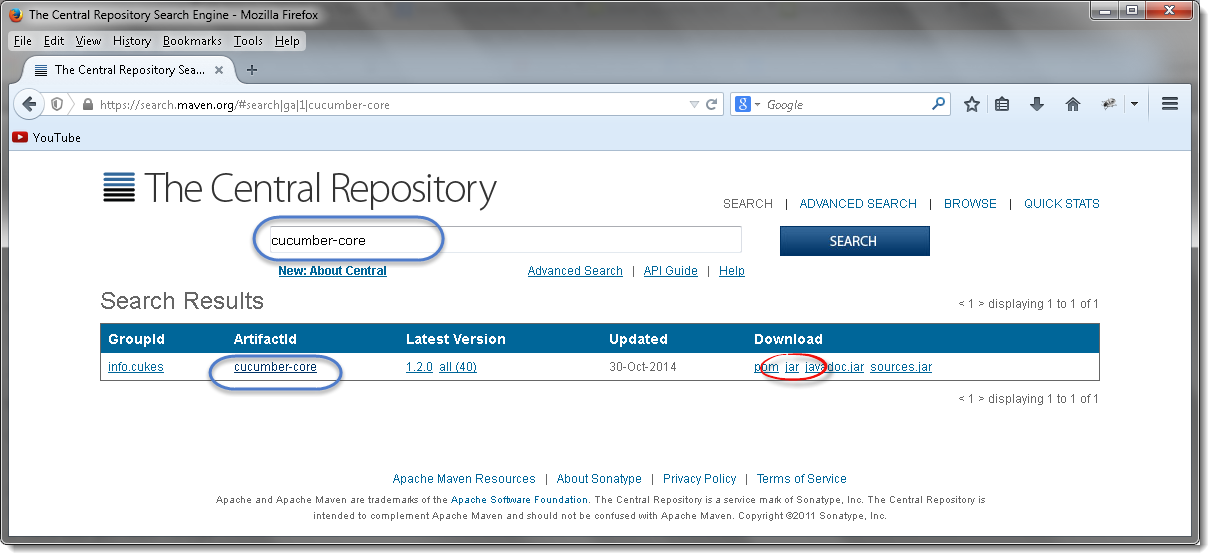
# Download Cucumber for Eclipse

**Option 1 : Download Cucumber Jars from Online Maven Repository**

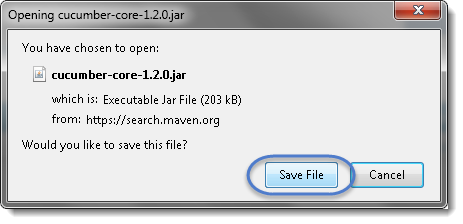
1) Go to [https://search.maven.org](https://search.maven.org/).



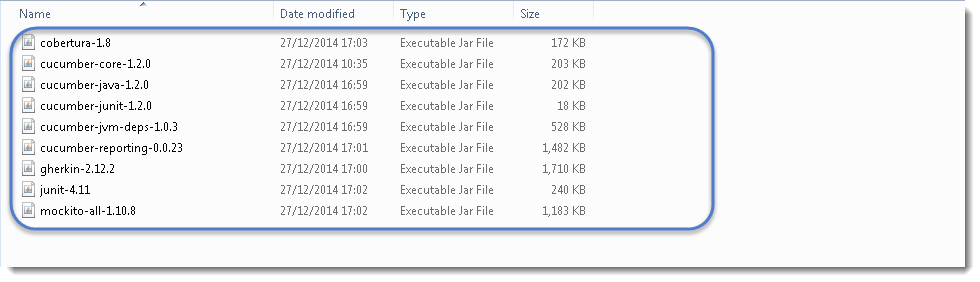
2) Search for cucumber-core in the Central Maven Repository. It will return the Cucumber Core jars. Click on the jar.



3) It will display a pop up and ask you to save the cucumber core jar file.

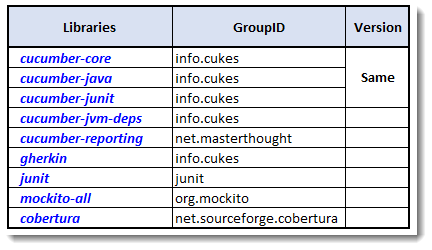


4) Just like this, one by one search for every other jar file mentioned above and downloads these to your drive.



Note that core, java and junit files all need to be the same file version e.g. 1.2.0

Also, one most important thing to know is to download the correct Jars. As when you search for these libraries, you will get many options to download. So below chart will help you to install the right libraries from GroupID.

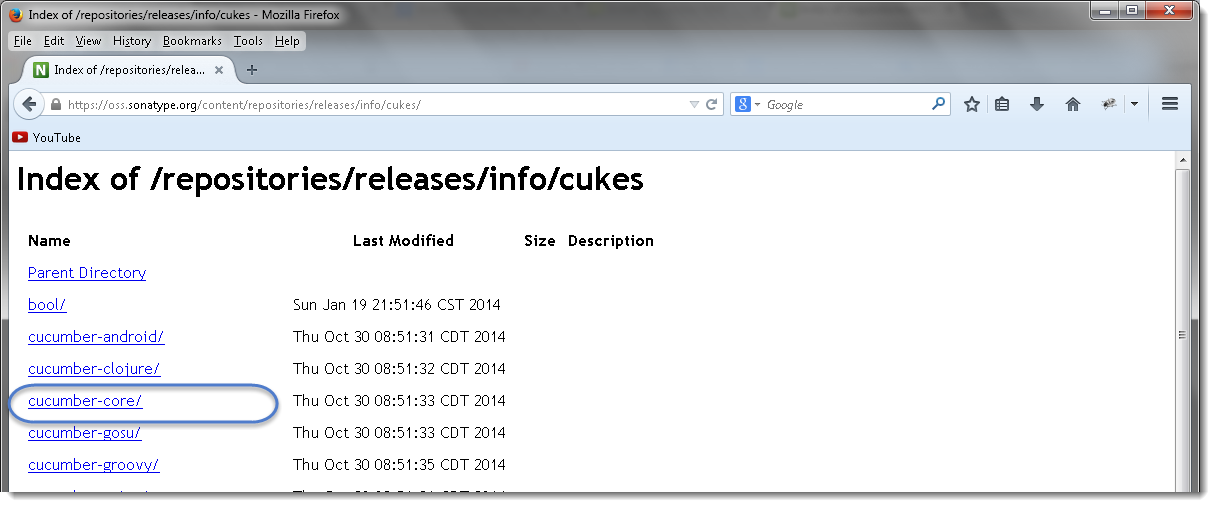


I got below versions on Oct’17 for Cucumber

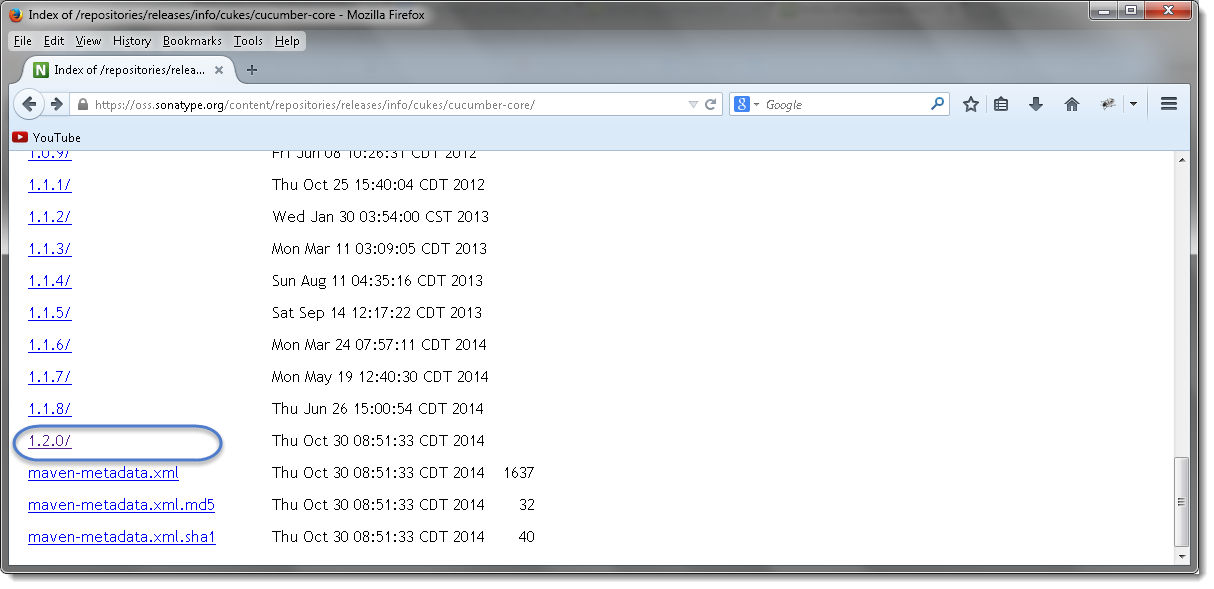
* cobertura-2.1.1
* cucumber-core-1.2.5
* cucumber-java-1.2.5
* cucumber-junit-1.2.5
* cucumber-jvm-deps-1.0.5
* cucumber-reporting-3.10.0
* gherkin-2.12.2
* junit-4.12
* mockito-all-2.0.2-beta

**Option 2 : Download Cucumber Jars from oss.sonatype.org**

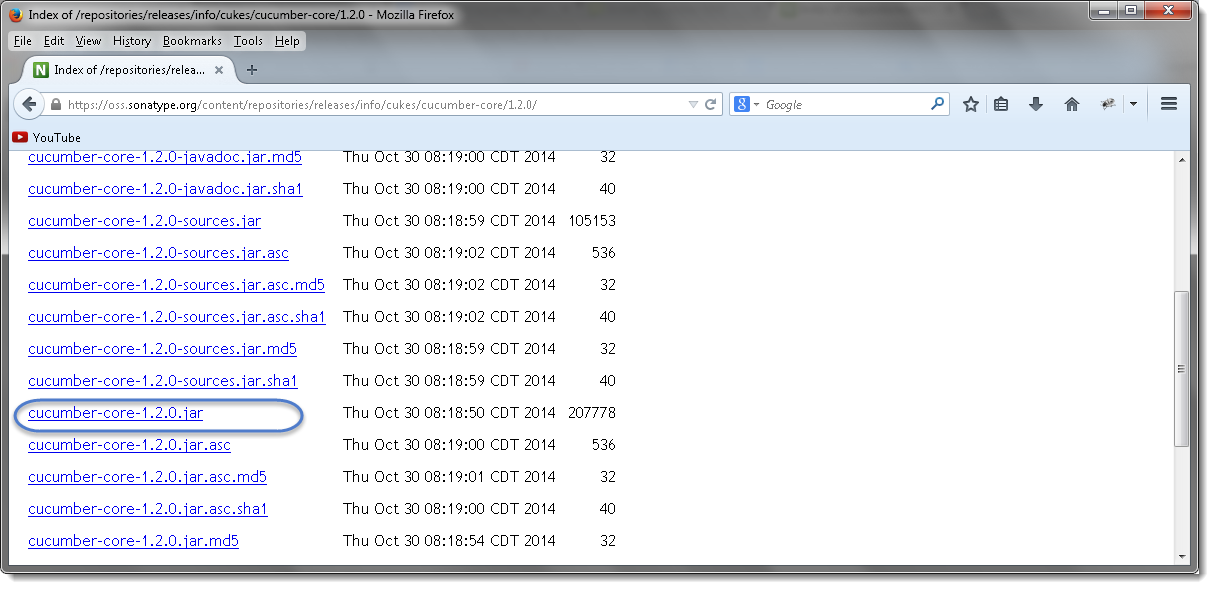
1) Go to <https://oss.sonatype.org/content/repositories/releases/info/cukes/> . Here also all the cucumber jars are available. Start with cucumber-core.



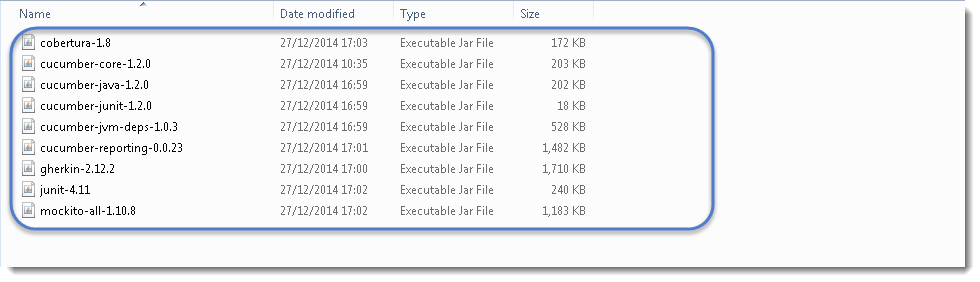
2) Click on the latest version at the bottom of the page, which is version 1.2.0 as of now by Dec’14.



3) Once clicked on the version, it will display all types of jar available to download. Click on cucumber-core-1.2.0.jar .



4) Just like this, download all of these to your drive and every other jar is available on the same page.



Note that core, java and junit files all need to be the same file version e.g. 1.2.0

**Option 3 : Download Cucumber Jars from Maven dependencies**

This is the most common and effective way of setting up cucumber with eclipse. But this is a bit tricky for the people who do not have much experience with Maven and may end up wasting a lot of time. There are few prerequisites for setting up cucumber in eclipse.

1. [Install Maven in Eclipse IDE](https://toolsqa.com/java/maven/how-to-install-maven-eclipse-ide/)
2. [Create a New Maven Project in Eclipse](https://toolsqa.com/java/maven/create-new-maven-project-eclipse/)

Once Maven is installed on eclipse and a Maven project is created, the next step is to add cucumber dependencies on the project. I have written a nice tutorial on Maven and how to add dependencies into Maven projects. Please follow Step 4 : Add Dependencies to the Maven Repository at [this article](https://toolsqa.com/java/maven/configure-selenium-continuous-integration-maven/). The way we have added Selenium dependencies in the article the same way add the below mentioned dependencies into the Maven POM.

Do not forget to add all the dependencies for all the below mentioned jars required for Cucumber set up:

1. Cucumber-java
2. Cucumber-junit
3. Cucumber-jvm-deps
4. Cucumber-reporting
5. Gherkin
6. Junit
7. Selenium-java
8. Webdrivermanager
9. Maven-cucumber-reporting

For example, Open the pom.xml file and Copy the following inside the dependencies tag and save.

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.2.5</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.2.5</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

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

<version>1.0.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>net.masterthought</groupId>

<artifactId>cucumber-reporting</artifactId>

<version>5.0.2</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>gherkin</artifactId>

<version>2.12.2</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13</version>

</dependency>

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>3.141.59</version>

</dependency>

<dependency>

<groupId>io.github.bonigarcia</groupId>

<artifactId>webdrivermanager</artifactId>

<version>3.7.1</version>

</dependency>

<dependency>

<groupId>net.masterthought</groupId>

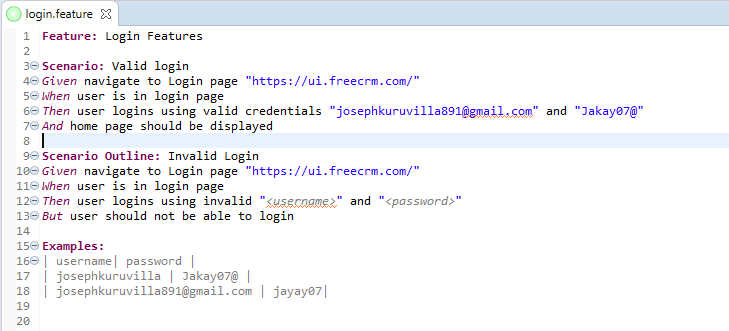
<artifactId>maven-cucumber-reporting</artifactId>

<version>5.3.0</version>

</dependency>

# Feature File

Feature files are the essential part of cucumber which is used to write test automation steps or acceptance tests. This can be used as the live document. The steps are the application specification. All the feature files end with .feature extension.

****

**Feature:**

This gives information about the high-level business functionality (Refer to the previous example) and the purpose of Application under test. Everybody should be able to understand the intent of a feature file by reading the first Feature step. This part is basically kept brief.

**Scenario:**

Basically, a scenario represents a particular functionality which is under test. By seeing the scenario, the user should be able to understand the intent behind the scenario and what the test is all about. Each scenario should follow given, when and then format. This language is called “gherkin”.

* **Given:** As mentioned above, given specifies the pre-conditions. It is basically a known state.
* **When:** This is used when some action is to be performed. As in above example, we have seen when the user tries to log in using username and password, it becomes an action.
* **Then:** The expected outcome or result should be placed here.
* **Background:** Whenever any step is required to perform in each scenario then those steps need to be placed in Background. For Instance: If a user needs to clear the database before each scenario then those steps can be put in the background.
* **And:** And is used to combine two or more same types of action.

**Scenario Outline:**

Scenario outlines are used when the same test has to be performed with different data sets. Let’s take the same example. We have to test login functionality with multiple different sets of username and password.

**Examples:** Lists examples for scenario outlines

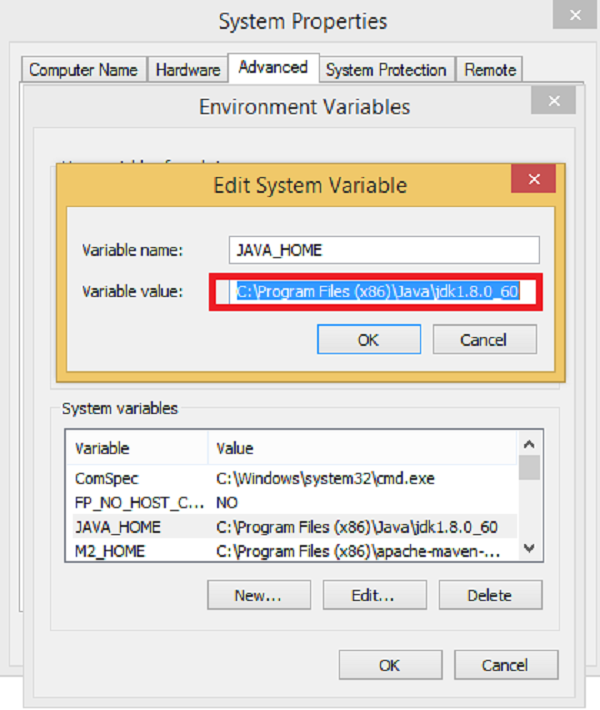
# Configure JUnit

Step 1 − Install Java.

* Download jdk and jre from the link

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

* Accept license agreement.
* Install JDK and JRE.
* Set the environment variable as shown in the following screenshot.

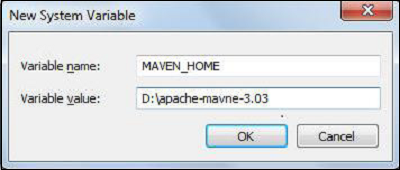


Step 2 − Install Eclipse.

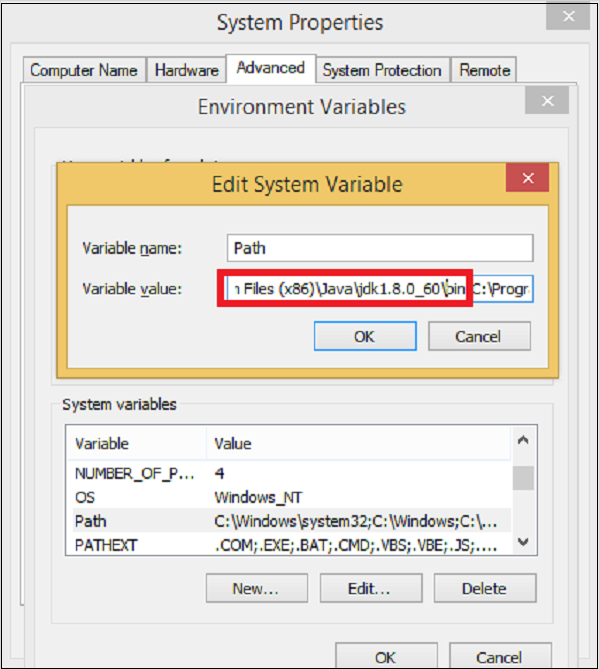
* Why we need − Eclipse is an Integrated Development Environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment.
* How to install −
  + Make sure JAVA is installed on your machine.
  + Download Eclipse from <http://www.eclipse.org/downloads>
  + Unzip and Eclipse are installed.

Step 3 − Install Maven.

* Why we need − Maven is a build automation tool used primarily for Java projects. It provides a common platform to perform activities like generating source code, compiling code, packaging code to a jar, etc. Also later if any of the software versions gets changed, Maven provides an easy way to modify the test project accordingly.
* How to install −
  + Download Maven − <https://maven.apache.org/download.cgi>
  + Unzip the file and remember the location.
  + Create an environment variable MAVEN\_HOME as shown in the following screenshot.



* Edit Path variable and include Maven as shown in the following screenshot.



* Download MAVEN plugin from Eclipse.
  + Open Eclipse.
  + Got to Help → Eclipse Marketplace → Search maven → Maven Integration for Eclipse → INSTALL.

Step 4 − How to configure Cucumber with Maven

* Create a Maven project in Eclipse.
  + 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 a 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 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-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>

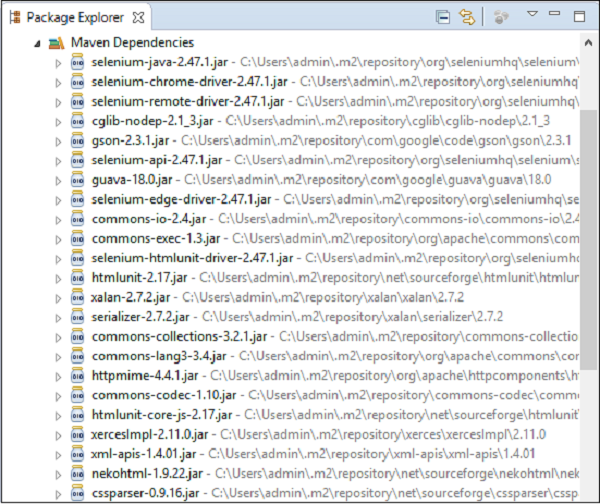
<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

* Verify binaries
  + Once pom.xml is edited successfully, save it.
  + Go to Project → Clean − It will take a few minutes.
  + After that you will be able to see a Maven repository like the following screenshot.



# JUnit Test Runner Class

As Cucumber uses Junit we need to have a Test Runner class. This class will use the Junit annotation @RunWith(), which tells JUnit what is the test runner class. It's more like a starting point for Junit to start executing your tests. In the src folder create a class called TestRunner.

**Test Runner Class**

package MyRunner;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = {"D:\Selenium\CucumberConcepts\src\main\java\Feature"},

// path of the feature file

glue= {"StepDefinition"},

// path of the step definition file

monochrome = true,

// to display the console output in a proper readable format

plugin = {"pretty", "html:test-output", "json:target/cucumber.json", "junit: junit\_xml/cucumber.xml"},

// to generate different type of reports

strict = true,

// it will check if any step is not defined in step definition file

dryRun = false,

//to check the mapping is proper between feature file and step definition file

tags = {"@SmokeTest, @RegressionTest,@End2End"})

public class TestRunner {

}

**Import Statements**

First import statement ‘org.junit.runner.RunWith‘ imports @RunWith annotation from the Junit class. @RunWith annotation tells JUnit that tests should run using the Cucumber class present in the ‘Cucumber.api.junit‘ package.

Second import statement ‘cucumber.api.CucumberOptions‘ imports the @CucumberOptions annotation. This annotation tells Cucumber a lot of things like where to look for feature files, what reporting system to use and some other things also. But as of now in the above test, we have just told it for the Feature file folder.

**Run the Cucumber Test**

Now we are all set to run the first Cucumber test. There are multiple ways and runners to use when it comes to cucumber feature files. We would try to understand how to run it from the IDE first and then from a command line at a later point.

Even from the IDE, there are a couple of ways to run these feature files.

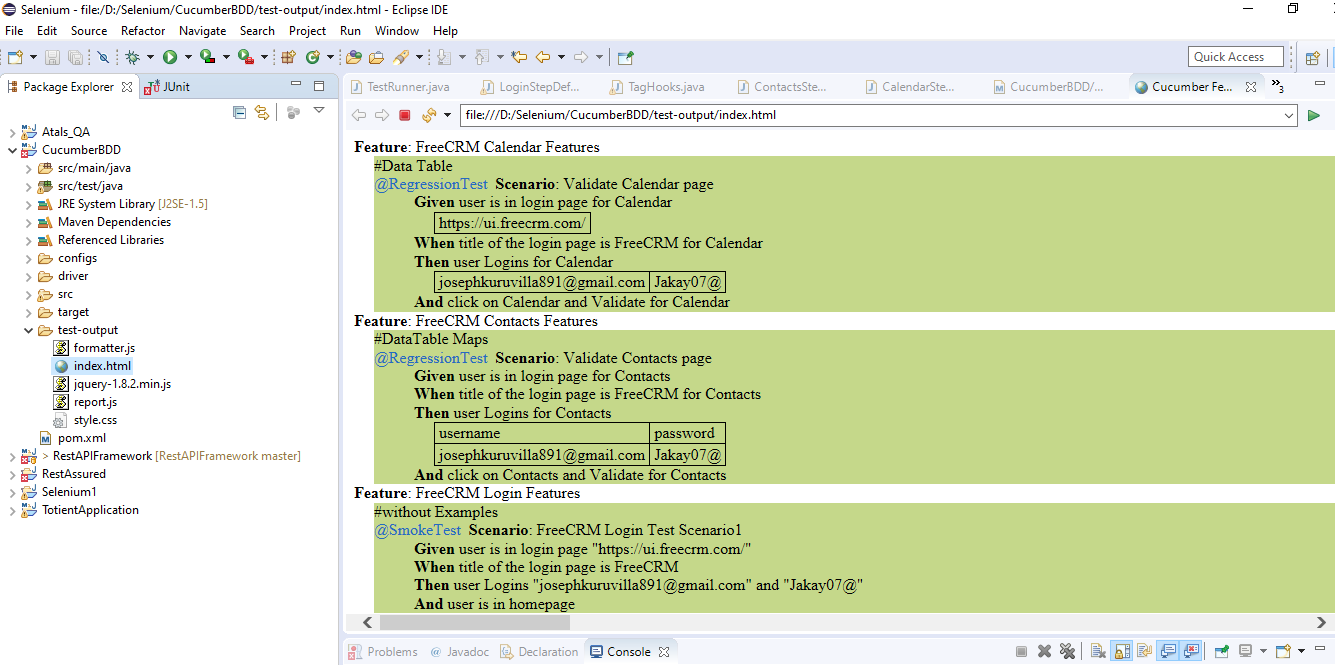
* Click on the Run button on eclipse and you have your test run
* Right Click on TestRunner class and Click Run As > JUnit Test Application

You will think where is the java code that will execute for these tests? Well, don’t worry about that at this moment. Let’s just see what we have on the console window. Here is the text that I got on my console. Look how Cucumber has suggested that you should implement these methods so that the Steps mentioned in the Feature file can be traced to Java methods, which can be executed while executing the feature file.

**Cucumber Reporting:**

Unlike reporting and other third-party tools where we need to do some configuration to view the reporting.Here in Cucumber, we have built-in plugins such as pretty, JSON, HTML, XML which give us the comprehensive report of test execution.

plugin = {"pretty", "html:test-output", "json:target/cucumber.json", "junit: junit\_xml/cucumber.xml"}



# Gherkin Keywords

Gherkin is the format for cucumber specifications. It is a domain specific language which helps you to describe business behavior without the need to go into detail of implementation. This text acts as documentation and skeleton of your automated tests.

**Feature:**

Each Gherkin file begins with a Feature keyword. Feature defines the logical test functionality you will test in this feature file. For e.g, if you are testing a payment gateway your Feature will become Payment Gateway or if you are testing the LogIn functionality then the Feature will become Login. The idea of having a feature file is to put down a summary of what you will be testing. This will serve as the documentation for your tests as well as a good point to start for a new team member. Note that a feature keyword is present at the starting of the feature file.

Feature: Login Features

Or

Feature: Login Features

Description: This feature will test a valid LogIn functionality

Or

Feature: Login Features

This feature will test a valid LogIn functionality

Notice that whatever comes after the Feature: keyword, will be considered as the feature description. Feature description can span across multiple lines as shown above in the second example. Everything after Feature: till the next Keyword is encountered is considered as feature description.

**Background :**

Background keyword is used to define steps that are common to all the tests in the feature file. For example, to purchase a product, you need to do the following steps:

* Navigate to Login Page
* Enter UserName and Password
* Click on Login button

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 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. This is how it will look like:

Feature: Test Background

Background: User is Logged In

Given user navigates to the login page

| https://ui.freecrm.com/ |

When user submit username and password

| josephkuruvilla891@gmail.com | Jakay07@ |

Then user is logged in

Scenario: Create Event

Given navigate to Create New Event page

When user submits Calendar details

| New Event Calendar |

Then calendar should be created

Scenario: Create Contact

Given navigate to Create New Contact page

When user submits contact details

| FirstName | LastName |

| Admin | Super |

#| Admin | Services |

Then contact should be created

**Scenario:**

Each Feature will contain a number of tests to test the feature. Each test is called a Scenario and is described using the Scenario: keyword.

Scenario: Valid login

A scenario is equivalent to a test in our regular development process. Each scenario/test can be basically broken down into three parts:

* Precondition to the test, which represent with (Given) keyword
* Test step execution, which represent with (When) keyword
* Verification of the output with expected result, which represent with (Then)

**When:**

When keyword defines the test action that will be executed. By test action we mean the user input action.

Scenario: Valid Login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

**Then:**

Then keyword defines the Outcome of previous steps. We can understand it best by looking at the test above and adding a Then step there.

Scenario: Valid login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using valid credentials "josephkuruvilla891@gmail.com" and "Jakay07@"

**And:**

“And” keyword is used to add conditions to your steps. Let’s look at it by modifying our example a little

Scenario: Valid login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using valid credentials "josephkuruvilla891@gmail.com" and "Jakay07@"

And home page should be displayed

**But:**

“But” keyword is used to add negative type comments. It is not a hard & fast rule to use but only for negative conditions. It makes sense to use But when you will try to add a condition which is opposite to the premise your test is trying to set. Take a look at the example below:

Scenario Outline: Invalid Login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using invalid "<username>" and "<password>"

But user should not be able to login

Examples:

| username| password |

| josephkuruvilla | Jakay07@ |

**Scenario Outline:** Starts a new scenario with multiple examples

**Examples:** Lists examples for scenario outlines

**Scenario Outline:** Invalid Login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using invalid "<username>" and "<password>"

But user should not be able to login

**Examples:**

| username| password |

| josephkuruvilla | Jakay07@ |

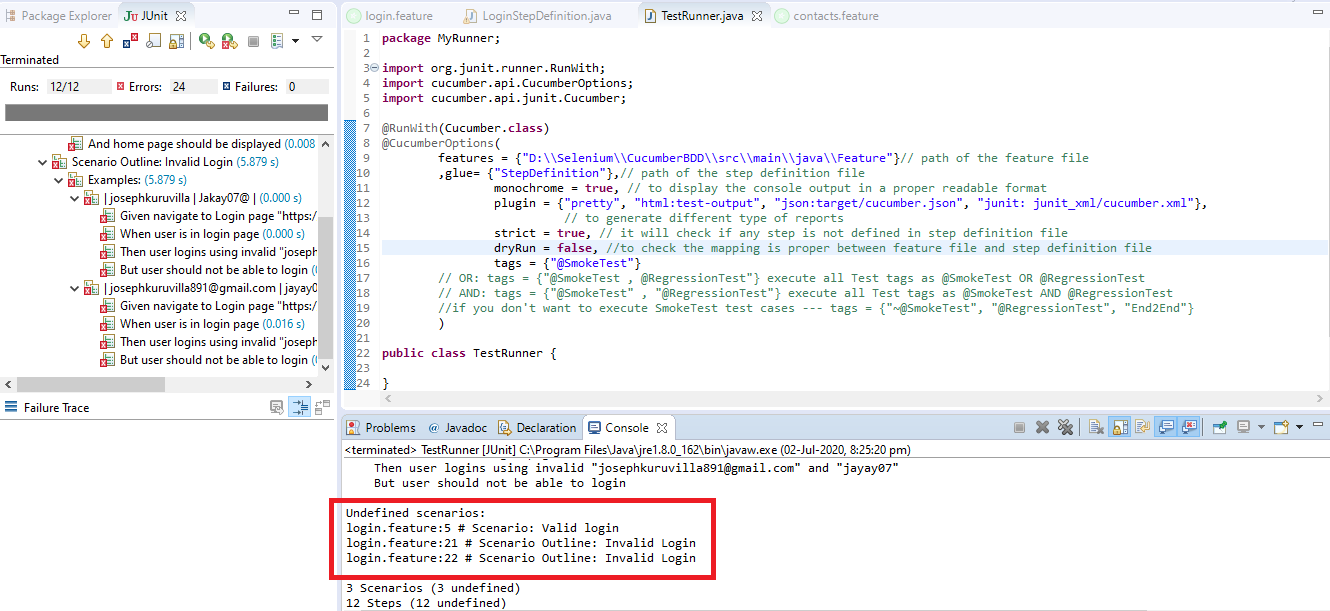
# Step Definition

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**

## **Add a Step Definition file**

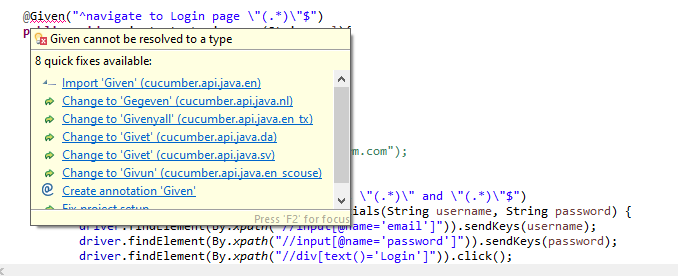
1) Create a new Class file in the ‘stepDefinition‘ package and name it as ‘LoginStepDefinition‘, by right click on the Package and select New > Class. Do not check the option for ‘public static void main‘ and click on the Finish button.

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 “Undefined scenarios”. So, implement all the steps in the scenarios.

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.

****

## Add Selenium Java code in the Step Definition methods

For each step of a Scenario create a method in LoginStepDefinition class

**1)Given** navigate to Login page "https://ui.freecrm.com/"

Method will look like this now:

@Given("^navigate to Login page \"(.\*)\"$")

public void navigate\_to\_Login\_page(String url){

driver.get(url);

}

**2)When** user is in login page

Method will look like this now:

@When("^user is in login page$")

public void user\_is\_in\_login\_page() {

String title = driver.getTitle();

System.out.println(title);

Assert.assertEquals(title, "ui.freecrm.com");

}

**3)Then** user logins using valid credentials "josephkuruvilla891@gmail.com" and "Jakay07@"

Method will look like this now:

@Then("^user logins using valid credentials \"(.\*)\" and \"(.\*)\"$")

public void user\_logins\_using\_valid\_credentials(String username, String password) {

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

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

Step Definition: LoginStepDefinition

package StepDefinition;

import java.util.concurrent.TimeUnit;

import org.junit.Assert;

import org.openqa.selenium.By;

import org.openqa.selenium.chrome.ChromeDriver;

import cucumber.api.java.After;

import cucumber.api.java.Before;

import cucumber.api.java.en.And;

import cucumber.api.java.en.But;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

import io.github.bonigarcia.wdm.WebDriverManager;

public class LoginStepDefinition {

public WebDriver driver;

@Given("^navigate to Login page \"(.\*)\"$")

public void navigate\_to\_Login\_page(String url){

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.manage().window().maximize();

driver.get(url);

}

@When("^user is in login page$")

public void user\_is\_in\_login\_page() {

String title = driver.getTitle();

System.out.println(title);

//Assert.assertEquals(title, "ui.freecrm.com");

}

@Then("^user logins using valid credentials \"(.\*)\" and \"(.\*)\"$")

public void user\_logins\_using\_valid\_credentials(String username, String password) {

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

@Then("^user logins using invalid \"(.\*)\" and \"(.\*)\"$")

public void user\_logins\_using\_invalid(String username, String password) {

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

@And("^home page should be displayed$")

public void home\_page\_should\_be\_displayed() {

String homepage = driver.findElement(By.xpath("//span[text()='Home']")).getText();

System.out.println(homepage);

Assert.assertEquals(homepage, "Home");

driver.close();

}

@But ("^user should not be able to login$")

public void user\_should\_not\_be\_able\_to\_login() {

driver.findElement(By.xpath("//div[text()='Something went wrong...']")).isDisplayed();

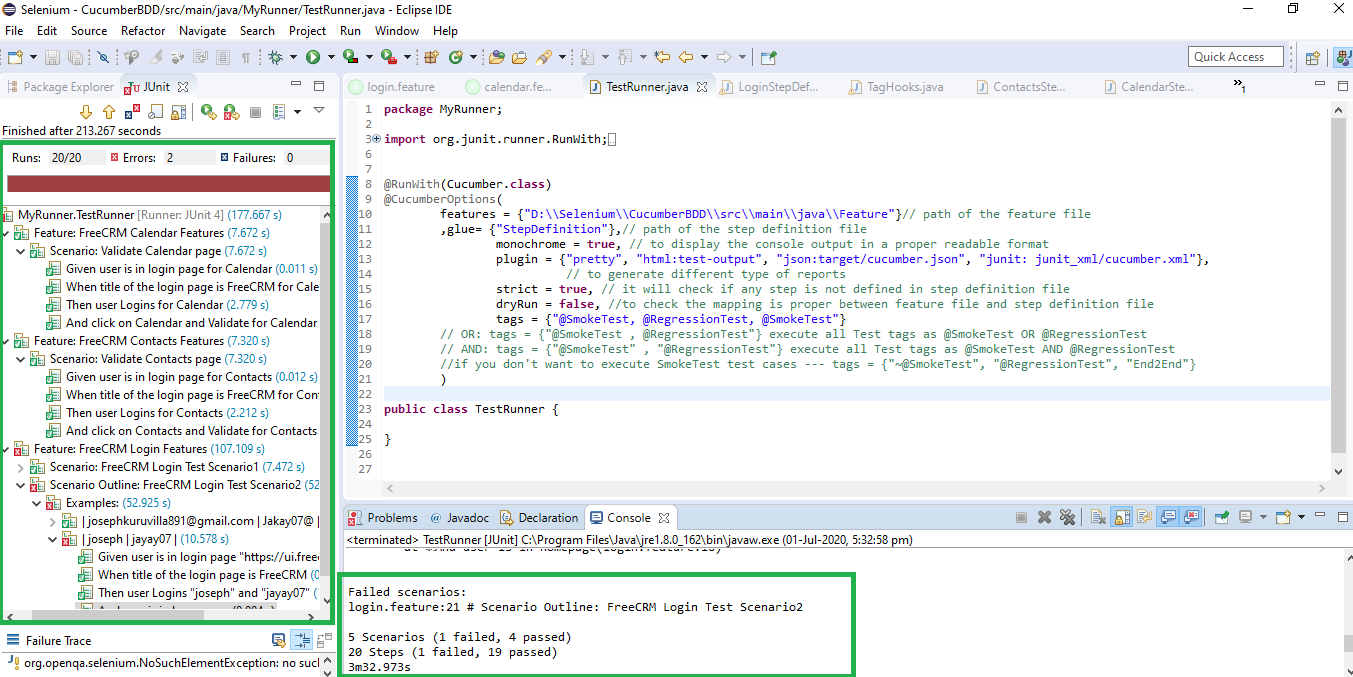
driver.close();

}

}

## **Run the Cucumber Test**

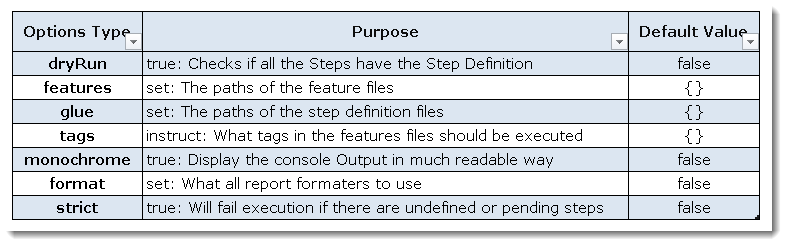
Now we are all set to run the first Cucumber test. Right Click on TestRunner class and Click Run As > JUnit Test. Cucumber will run the script the same way it runs in Selenium WebDriver and the result will be shown in the left hand side project explorer window in JUnit tab.



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.

# Cucumber Options

Following Main Options are available in Cucumber:



**Dry Run**

dryRun option can either be 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**

package MyRunner;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = {"D:\Selenium\CucumberConcepts\src\main\java\Feature"},

// path of the feature file

glue= {"StepDefinition"},

// path of the step definition file

monochrome = true,

// to display the console output in a proper readable format

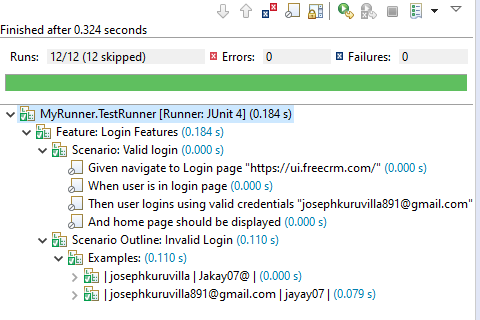
dryRun = true)

//to check the mapping is proper between feature file and step definition file

public class DryRunTestRunner {

}

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 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.

**Monochrome**

This option can either be set as **true** or **false**. If it is set as true, it means that the console output for the Cucumber test is 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**

package MyRunner;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = {"D:\\Selenium\\CucumberConcepts\\src\\main\\java\\Feature"},

// path of the feature file

glue= {"StepDefinition"},

// path of the step definition file

monochrome = true)

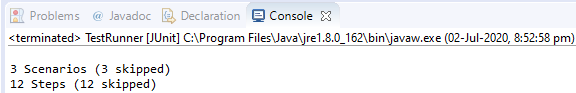
// monochrome = false)

// to display the console output in a proper readable format

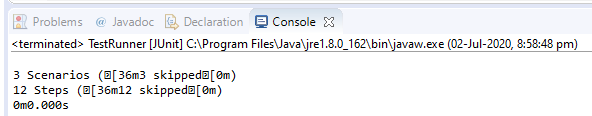
public class MonochromeTestRunner {

}

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:



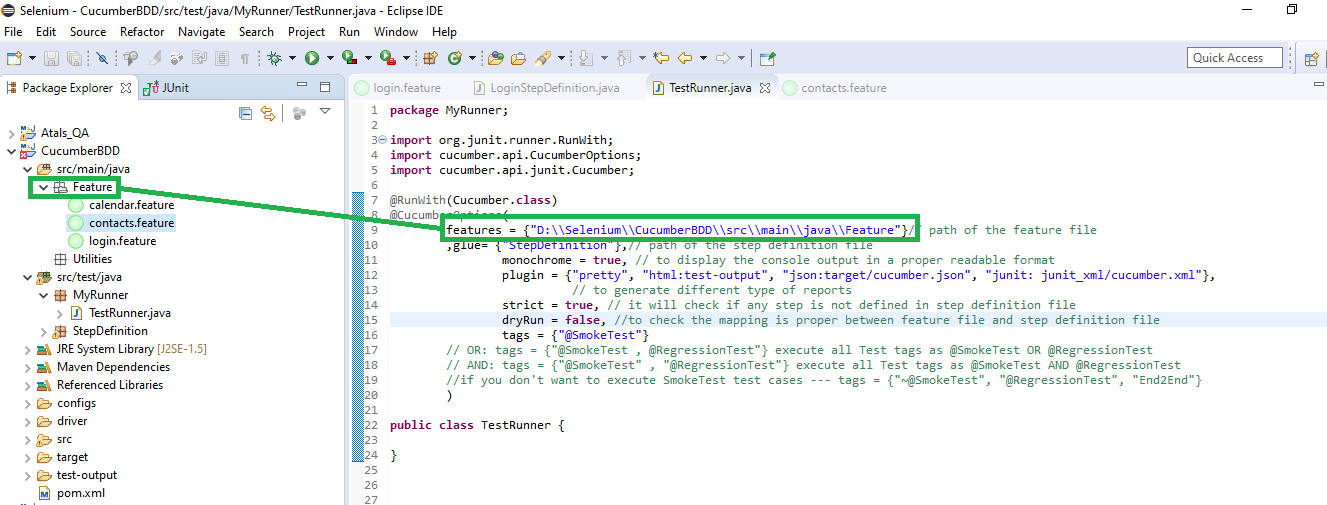
**Features**

Features Options helps Cucumber to locate the Feature file in the project folder structure. You must have noticed 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 = {"D:\\Selenium\\CucumberConcepts\\src\\main\\java\\Feature"}

****

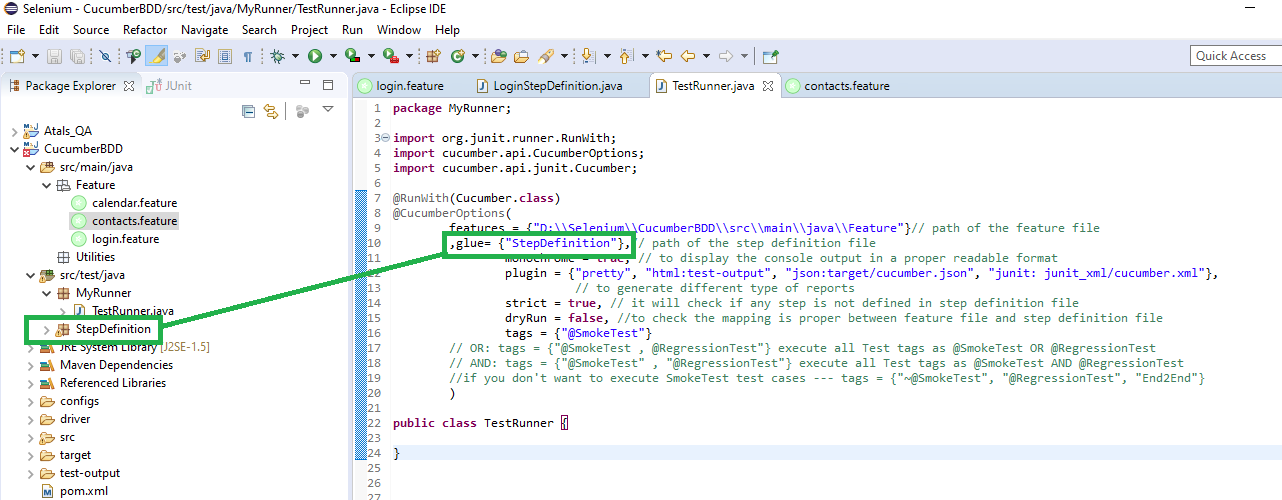
**Glue**

It is almost the same thing 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 = “D:\Selenium\CucumberBDD\src\test\java\StepDefinition“

****

**Format**

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“}

**We can also use these options together.**

@CucumberOptions(plugin = {"pretty" ,"html:test-output" ,

"json:target/cucumber.json" ,

"junit: junit\_xml/cucumber.xml"})

# Data Driven Testing without Using Examples Keyword

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

**Then** user logins using valid credentials "josephkuruvilla891@gmail.com" and "Jakay07@"

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.feature

Scenario: Valid login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using valid credentials "josephkuruvilla891@gmail.com" and "Jakay07@"

And home page should be displayed

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:

@Then("^user logins using valid credentials \"(.\*)\" and \"(.\*)\"$")

3) Same parameters should also go into the associated LoginStepDefinition. 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\_logins\_using\_valid\_credentials(String username, String password) {

}

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.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

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

@Then("^user logins using valid credentials \"(.\*)\" and \"(.\*)\"$")

public void user\_logins\_using\_valid\_credentials(String username, String password) {

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(username);

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(password);

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

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

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

Examples:

| username| password |

| josephkuruvilla | Jakay07@ |

| josephkuruvilla891@gmail.com | jayay07|

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.

Then user logins using invalid "<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:

Scenario Outline: Invalid Login

Given navigate to Login page "https://ui.freecrm.com/"

When user is in login page

Then user logins using invalid "<username>" and "<password>"

But user should not be able to login

Examples:

| username| password |

| josephkuruvilla | Jakay07@ |

| josephkuruvilla891@gmail.com | jayay07|

3) There are no changes in TestRunner class.

4) There are no changes in the StepDefinition file from the previous chapter.

5) Run the test by Right Click on TestRunner class and Click Run As > JUnit Test Application.

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 in Cucumber

**Data Tables in Cucumber** are quite interesting and can be used in many ways. DataTables are also used to handle large amounts of data.

In this example, we will pass the test data using the data table and handle it using **Raw()** method.

Scenario: Validate Calendar page

Given user is in login page2

| https://ui.freecrm.com/ |

When title of the login page is FreeCRM2

Then user Logins2

| josephkuruvilla891@gmail.com | Jakay07@ |

And click on Calendar and Validate page2

The complete scenario is the 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.

**The implementation of the above step will be like this:**

@Given("^user is in login page2$")

public void user\_is\_in\_login\_page2(DataTable URL){

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.manage().window().maximize();

List<List<String>> url = URL.raw();

driver.get(url.get(0).get(0));

}

@When("^title of the login page is FreeCRM2$")

public void title\_of\_the\_login\_page\_is\_FreeCRM2() {

String title = driver.getTitle();

System.out.println(title);

Assert.assertEquals(title, "ui.freecrm.com");

}

@Then("^user Logins2$")

public void user\_Logins2(DataTable credentials) {

List<List<String>> usercredentials = credentials.raw();

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(usercredentials.get(0).get(0)); driver.findElement(By.xpath("//input[@name='password']")).sendKeys(usercredentials.get(0).get(1));

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

@And ("click on Calendar and Validate page2$")

public void click\_on\_Calendar\_and\_Validate\_page2() {

driver.findElement(By.xpath("//span[text()='Calendar']")).click();

String Calendar = driver.findElement(By.xpath("//div[text()='Calendar']")).getText();

System.out.println(Calendar);

Assert.assertEquals(Calendar, "Calendar");

}

# Maps in Data Tables

*Maps in Data Tables* can be used in different ways. ***Headers***can also be defined for the *data tables.* The same step can be executed multiple times with different sets of test data using ***Maps***.

Maps in Data Tables with Header

In the previous chapter of [***Data Tables in Cucumber***](https://toolsqa.com/cucumber/data-tables-in-cucumber/)***,***  we pass *Username & Password* without Header, due to which the test was not much readable. What if there will be many columns. The basic funda of BDD test is to make the Test in Business readable format, so that business users can understand it easily. Setting Header in Test data is not a difficult task in Cucumber. take a look at the below Scenario.

***Feature File Scenario***

Scenario: Validate Contacts page

Given user is in login page3

When title of the login page is FreeCRM3

Then user Logins3

| username | password |

| josephkuruvilla891@gmail.com | Jakay07@ |

And click on Contacts and Validate page3

***The implementation of the above step will be like this:***

@Given("^user is in login page3$")

public void user\_is\_in\_login\_page3(){

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.manage().window().maximize();

driver.get("<https://ui.freecrm.com/>");

}

@When("^title of the login page is FreeCRM3$")

public void title\_of\_the\_login\_page\_is\_FreeCRM3() {

String title = driver.getTitle();

System.out.println(title);

Assert.assertEquals(title, "ui.freecrm.com");

}

@Then("^user Logins3$")

public void user\_Logins3(DataTable credentials) {

for(Map<String, String> usercredentials : credentials.asMaps(String.class, String.class)) { driver.findElement(By.xpath("//input[@name='email']")).sendKeys(usercredentials.get("username"));

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(usercredentials.get("password"));

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

}

@And ("click on Contacts and Validate for page3$")

public void click\_on\_Contacts\_and\_Validate\_page3() {

driver.findElement(By.xpath("//span[text()='Contacts']")).click();

String Contacts = driver.findElement(By.xpath("//div[text()='Contacts']")).getText();

System.out.println(Contacts);

Assert.assertEquals(Contacts, "Contacts");

}

Maps in Data Tables with Multiple Test Data

In this test we will create two contacts. So our test should create one contact first and repeat the same steps again.

***Feature File Scenario***

Scenario: Create New Contact

Given user is in login page3

When title of the login page is FreeCRM3

Then user Logins3

| username | password |

| josephkuruvilla891@gmail.com | Jakay07@ |

And click on Contacts and Validate page3

And create Contact

| FirstName | LastName |

| Admin | Super |

| Admin | Services |

***The implementation of the above step will be like this:***

@Given("^user is in login page3$")

public void user\_is\_in\_login\_page3(){

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.manage().window().maximize();

driver.get("<https://ui.freecrm.com/>");

}

@When("^title of the login page is FreeCRM3$")

public void title\_of\_the\_login\_page\_is\_FreeCRM3() {

String title = driver.getTitle();

System.out.println(title);

Assert.assertEquals(title, "ui.freecrm.com");

}

@Then("^user Logins3$")

public void user\_Logins3(DataTable credentials) {

for(Map<String, String> usercredentials : credentials.asMaps(String.class, String.class)) { driver.findElement(By.xpath("//input[@name='email']")).sendKeys(usercredentials.get("username")); driver.findElement(By.xpath("//input[@name='password']")).sendKeys(usercredentials.get("password"));

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

}

@And ("click on Contacts and Validate for page3$")

public void click\_on\_Contacts\_and\_Validate\_page3() {

driver.findElement(By.xpath("//span[text()='Contacts']")).click();

String Contacts = driver.findElement(By.xpath("//div[text()='Contacts']")).getText();

System.out.println(Contacts);

Assert.assertEquals(Contacts, "Contacts");

}

@And("^create Contact$")

public void create\_Contact(DataTable contactDetails) {

for(Map<String, String> userContactDetails : contactDetails.asMaps(String.class, String.class)) {

driver.findElement(By.xpath("//span[text()='Contacts']")).click();

driver.findElement(By.xpath("//button[text()='New']")).click(); driver.findElement(By.xpath("//input[@name='first\_name']")).sendKeys(userContactDetails.get("FirstName"));

driver.findElement(By.xpath("//input[@name='last\_name']")).sendKeys(userContactDetails.get("LastName"));

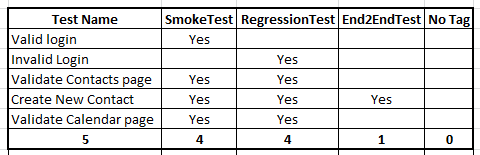
driver.findElement(By.xpath("//button[text()='Save']")).click();

}

# 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 maybe **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.

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



**Feature file will look like this**

@SmokeTest

Scenario: Valid login

@RegressionTest

Scenario Outline: Invalid Login

@SmokeTest @RegressionTest

Scenario: Validate Contacts page

@RegressionTest @SmokeTest @End2EndTest

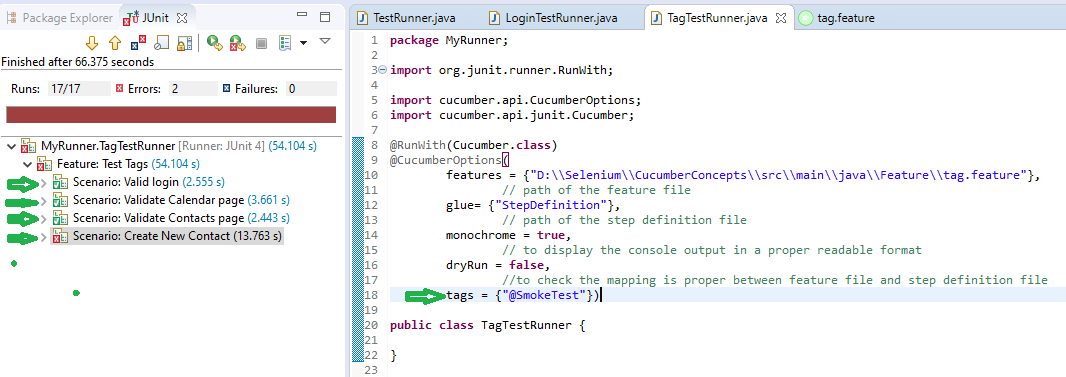
Scenario: Create New Contact

@RegressionTest @SmokeTest

Scenario: Validate Calendar page

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 4.

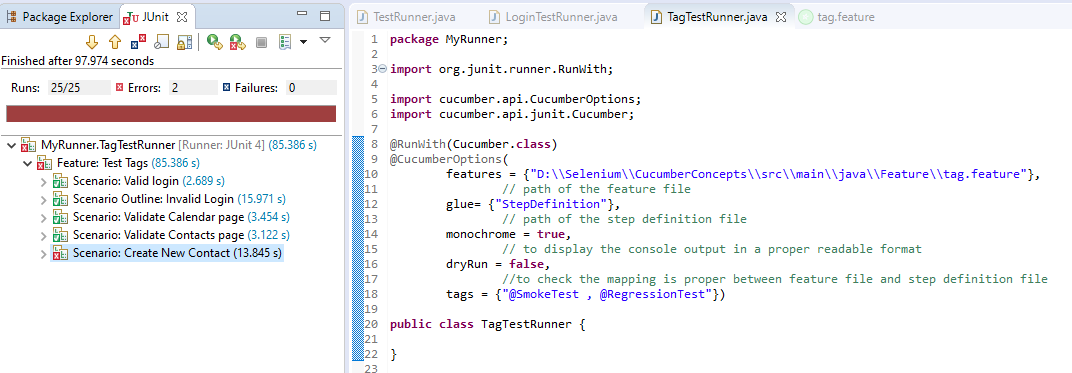
**(It will be Similar to other tags as well)**

**Logical ANDing and ORing Tags**

Requirements are complicated, it will not always be 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**

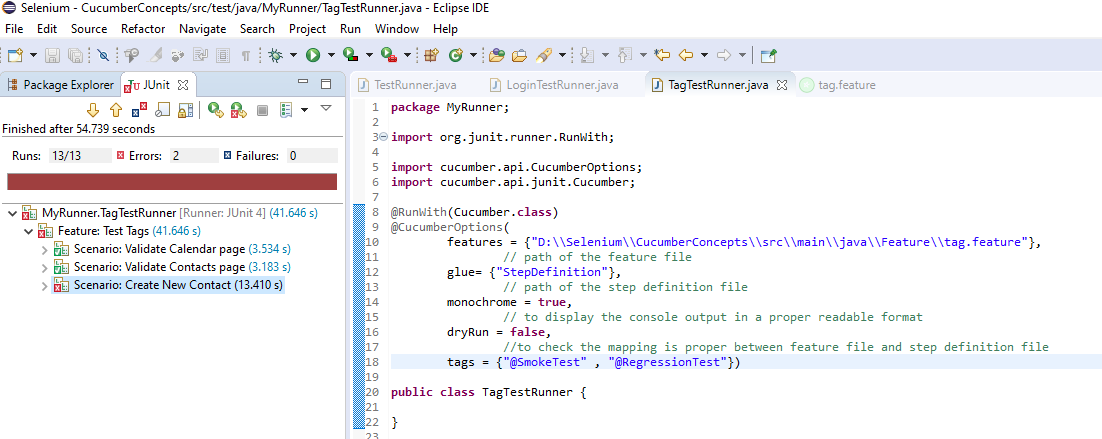
Tags that are **comma-separated** are ORed.



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

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

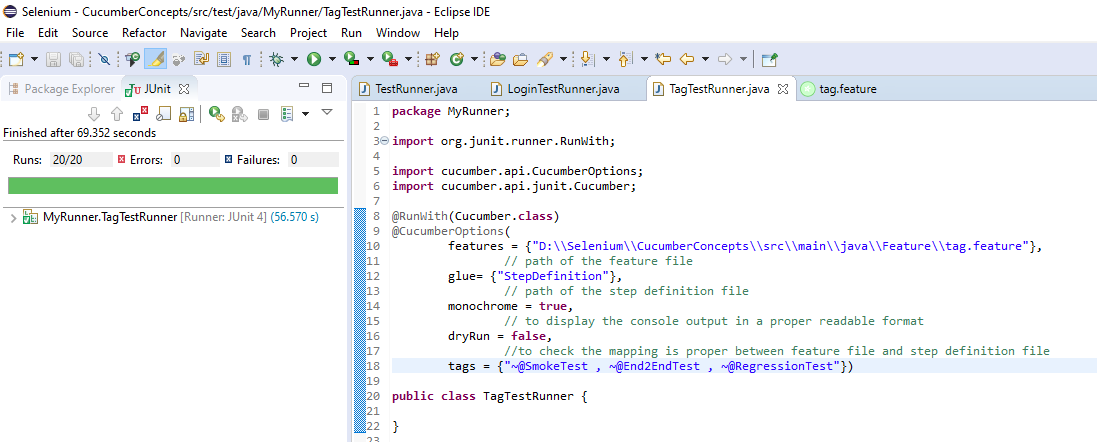
Tags which are passed in separate **quotes** are ANDed



**Note**: There are only 3 scenarios in our feature file which have both tags together.

**How to Ignore Cucumber Tests**

Using “ ~ “ before Tags

****

# Hooks

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 Annotations**](https://toolsqa.com/selenium-webdriver/testng-annotations-groups-depends/), cucumber supports only two hooks (Before & After) which work 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: First Scenario

Given this is First step

When this is Second Step

Then this is Third Step

**Step Definitions**

package StepDefinition;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class HooksStepDefinition {

@Given ("^this is First step$")

public void this\_is\_First\_step(){

System.out.println("First Step");

}

@When ("^this is Second Step$")

public void this\_is\_Second\_Step(){

System.out.println("Second Step");

}

@Then ("^this is Third Step$")

public void this\_is\_Third\_Step(){

System.out.println("Third Step");

}

}

**Hooks**

package StepDefinition;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class Hooks {

@Before

public void beforeScenario(){

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

}

@After

public void afterScenario(){

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

}

}

**Things to note**

* An important thing to note about the after hook is that even in case of a test fail, after hook will execute for sure.
* Method names 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 mistake and import Junit Annotations, so be careful with this.

**Output**

This will run before each Scenario

First Step

Second Step

Third Step

This will run after each Scenario

**Note**: Scenario Hooks execute before and after every scenario.Test Hooks with Example Scenarios

Let's take a look when we have Scenario Outline with Examples.

# 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.

Tagged Hooks in Cucumber

Lets again start with doing a simple exercise to get the concept straight. Just keep three different scenarios in the feature file with the same Given, When & Then steps.

1)-First step is to annotate required scenarios using **@ + AnyName** at the top of the Scenario. For this example, I just annotate each scenario with the sequence order of it, like **@First, @Second & @Third**.

**Feature File**

Feature: Test Tagged Hooks

@First

Scenario: First Scenario

Given this is First step

When this is Second Step

Then this is Third Step

@Second

Scenario: Second Scenario

Given this is First step

When this is Second Step

Then this is Third Step

@Third

Scenario: Third Scenario

Given this is First step

When this is Second Step

Then this is Third Step

2) Create a Step definition file and just print the execution order of the steps in the console.

**Step Definitions**

package StepDefinition;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class HooksStepDefinition {

@Given ("^this is First step$")

public void this\_is\_First\_step(){

System.out.println("First Step");

}

@When ("^this is Second Step$")

public void this\_is\_Second\_Step(){

System.out.println("Second Step");

}

@Then ("^this is Third Step$")

public void this\_is\_Third\_Step(){

System.out.println("Third Step");

}

}

3) Define tagged hooks in Hooks class file. Hooks can be used like **@Before(“@TagName”)**. Create before and after hooks for every scenario. I have also added normal before and after hooks, in case you are not aware, please go to the previous chapter of [**Hooks in Cucumber**](https://toolsqa.com/cucumber/cucumber-hooks/).

**Tagged Hooks**

package StepDefinition;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class TaggedHooks {

@Before ("@First")

public void beforeFirstScenario(){

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

}

@After ("@First")

public void afterFirstScenario(){

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

}

@Before ("@Second")

public void beforeSecondScenario(){

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

}

@After ("@Second")

public void afterSecondScenario(){

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

}

@Before ("@Third")

public void beforeThirdScenario(){

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

}

@After ("@Third")

public void afterThirdScenario(){

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

}

}

**Note**: We learned that @Before & @After hooks run before & after every Scenario. But **@Before(“@First”)** will run only before the first scenario and likewise other tagged hooks. Again, these tag names can be anything and no need to be first, second and third.

4) Run the feature file and observe the output.

This will run before each Scenario

This will run before First Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after First Scenario

This will run before each Scenario

This will run before Second Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after Second Scenario

This will run before each Scenario

This will run before Third Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after Third Scenario

Common Tagged Hooks for Multiple Scenarios

We can have common tagged hooks for multiple scenarios as well. In the below example, I just combined the **@Before(“First”)** and **@Before(“Third”)** by **@Before(“@First, @Third”).** So in this way we do not need to have two different hooks logic.

**Hooks**

package StepDefinition;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class MultipleTaggedHooks {

//Single Tagged Hooks

@Before ("@First2")

public void beforeFirstScenario(){

System.out.println("This will run before First2 Scenario");

}

@After ("@First2")

public void afterFirstScenario(){

System.out.println("This will run after First2 Scenario");

}

@Before ("@Second2")

public void beforeSecondScenario(){

System.out.println("This will run before Second2 Scenario");

}

@After ("@Second2")

public void afterSecondScenario(){

System.out.println("This will run after Second2 Scenario");

}

@Before ("@Third2")

public void beforeThirdScenario(){

System.out.println("This will run before Third2 Scenario");

}

@After ("@Third2")

public void afterThirdScenario(){

System.out.println("This will run after Third2 Scenario");

}

//multiple Tagged Hooks

@Before ("@First2,@Second2,@Third2")

public void beforeFirstSecondThirdScenario(){

System.out.println("This will run before FirstSecondThird2 Scenario");

}

@After ("@First2,@Second2,@Third2")

public void afterFirstSecondThirdThirdScenario(){

System.out.println("This will run after FirstSecondThird2 Scenario");

}

}

**Output**

This will run before each Scenario

This will run before First2 Scenario

This will run before FirstSecondThird2 Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after FirstSecondThird2 Scenario

This will run after First2 Scenario

This will run before each Scenario

This will run before Second2 Scenario

This will run before FirstSecondThird2 Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after FirstSecondThird2 Scenario

This will run after Second2 Scenario

This will run before each Scenario

This will run before Third2 Scenario

This will run before FirstSecondThird2 Scenario

First Step

Second Step

Third Step

This will run after each Scenario

This will run after FirstSecondThird2 Scenario

This will run after Third2 Scenario

# Execution Order of Hooks

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**

**Feature File**

Feature: Test Hooks Execution Order

// Methods in HooksStepDefinition.java

Scenario: First Scenario

Given this is First step

When this is Second Step

Then this is Third Step

Scenario: Second Scenario

Given this is First step

When this is Second Step

Then this is Third Step

Again, steps definitions are also the same as previous chapters.

**Hooks**

package StepDefinition;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class HooksExecutionOrder {

@Before (order=0)

public void beforeFirstScenario(){

System.out.println("Before order=0");

}

@After (order=0)

public void afterFirstScenario(){

System.out.println("After order=0");

}

@Before (order=1)

public void beforeSecondScenario(){

System.out.println("Before order=1");

}

@After (order=1)

public void afterSecondScenario(){

System.out.println("After order=1");

}

}

package StepDefinition;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class Hooks {

@Before

public void beforeScenario(){

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

}

@After

public void afterScenario(){

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

}

}

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

**Output**

Before order=0

Before order=1

This will run before each Scenario

First Step

Second Step

Third Step

This will run after each Scenario

After order=1

After order=0

Before order=0

Before order=1

This will run before each Scenario

First Step

Second Step

Third Step

This will run after each Scenario

After order=1

After order=0

# Background Keyword

**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 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 the Cucumber test. If we create a feature file of the scenario we explained above, this is how it will look like:

**Feature File**

Feature: Test Background

Background: User is Logged In

Given user navigates to the login page

| https://ui.freecrm.com/ |

When user submit username and password

| josephkuruvilla891@gmail.com | Jakay07@ |

Then user is logged in

Scenario: Create Event

Given navigate to Create New Event page

When user submits Calendar details

| New Event Calendar |

Then calendar should be created

Scenario: Create Contact

Given navigate to Create New Contact page

When user submits contact details

| FirstName | LastName |

| Admin | Super |

Then contact should be created

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 the website for both scenarios. 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**

package StepDefinition;

import java.util.List;

import java.util.Map;

import java.util.concurrent.TimeUnit;

import org.junit.Assert;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import cucumber.api.DataTable;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

import io.github.bonigarcia.wdm.WebDriverManager;

public class BackgroundSTepDefinition {

public WebDriver driver;

@Given ("^user navigates to the login page$")

public void user\_navigates\_to\_the\_login\_page(DataTable URL) {

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.manage().window().maximize();

List<List<String>> url = URL.raw();

WebDriverManager.chromedriver().setup();

driver.get(url.get(0).get(0));

String title = driver.getTitle();

System.out.println(title);

Assert.assertEquals(title, "ui.freecrm.com");

}

@When("^user submit username and password$")

public void user\_submit\_username\_and\_password(DataTable credentials) {

List<List<String>> usercredentials = credentials.raw();

driver.findElement(By.xpath("//input[@name='email']")).sendKeys(usercredentials.get(0).get(0));

driver.findElement(By.xpath("//input[@name='password']")).sendKeys(usercredentials.get(0).get(1));

driver.findElement(By.xpath("//div[text()='Login']")).click();

}

@Then("^user is logged in$")

public void user\_is\_logged\_in() {

String homepage = driver.findElement(By.xpath("//span[text()='Home']")).getText();

System.out.println(homepage);

Assert.assertEquals(homepage, "Home");

}

@Given ("^navigate to Create New Event page$")

public void navigate\_to\_Create\_New\_Event\_page() {

driver.findElement(By.xpath("//span[text()='Calendar']")).click();

String Calendar = driver.findElement(By.xpath("//div[text()='Calendar']")).getText();

System.out.println(Calendar);

Assert.assertEquals(Calendar, "Calendar");

driver.findElement(By.xpath("//button[text()='New']")).click();

}

@When("^user submits Calendar details$")

public void user\_submits\_Calendar\_details(DataTable details) {

List<List<String>> calendarDetails = details.raw();

driver.findElement(By.xpath("//input[@name='title']")).sendKeys(calendarDetails.get(0).get(0));

driver.findElement(By.xpath("//div[@name='calendar']")).click();

driver.findElement(By.xpath("//div[text()='Jakay M <josephkuruvilla891@gmail.com>']")).click();

driver.findElement(By.xpath("//button[text()='Save']")).click();

}

@Then("^calendar should be created$")

public void calendar\_should\_be\_created() {

driver.findElement(By.xpath("//button[text()='Save']")).click();

driver.close();

}

@Given ("^navigate to Create New Contact page$")

public void navigate\_to\_Create\_New\_Contact\_page() {

driver.findElement(By.xpath("//span[text()='Contacts']")).click();

String Contacts = driver.findElement(By.xpath("//div[text()='Contacts']")).getText();

System.out.println(Contacts);

Assert.assertEquals(Contacts, "Contacts");

driver.findElement(By.xpath("//button[text()='New']")).click();

}

@When("^user submits contact details$")

public void user\_submits\_contact\_details(DataTable details) {

for(Map<String, String> contactDetails : details.asMaps(String.class, String.class)) {

driver.findElement(By.xpath("//span[text()='Contacts']")).click();

driver.findElement(By.xpath("//button[text()='New']")).click();

driver.findElement(By.xpath("//input[@name='first\_name']")).sendKeys(contactDetails.get("FirstName"));

driver.findElement(By.xpath("//input[@name='last\_name']")).sendKeys(contactDetails.get("LastName"));

driver.findElement(By.xpath("//button[text()='Save']")).click();

}

}

@Then("^contact should be created$")

public void contact\_should\_be\_created() {

driver.findElement(By.xpath("//button[text()='Save']")).click();

driver.close();

}

}

**Note: The background runs two times in the feature before each scenario.**

# Cucumber Automation Framework

**Prerequisite:**

* Install JAVA on your system and set up the environmental variable.
* Install eclipse on your system.
* Install maven plugin in eclipse IDE.

**Step 1:**

Open eclipse IDE.

**Step 2:**

Create the new maven project.

**Step 3:**

POM.xml file needs to add the following dependencies

1. Cucumber-java
2. Cucumber-junit
3. Cucumber-jvm-deps
4. Cucumber-reporting
5. Gherkin
6. Junit
7. Selenium-java
8. Webdrivermanager
9. Maven-cucumber-reporting

**Step 4:**

Under the src/main/java Source Folder create the following packages

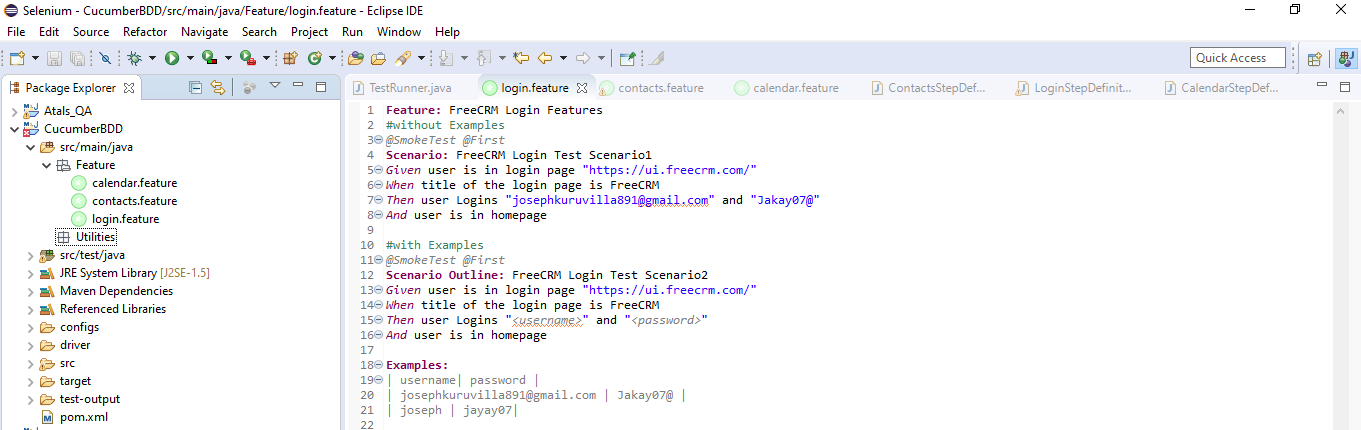
1. Feature
2. Utilities

Under the src/test/java Source Folder create the following packages

1. MyRunner
2. StepDefinition

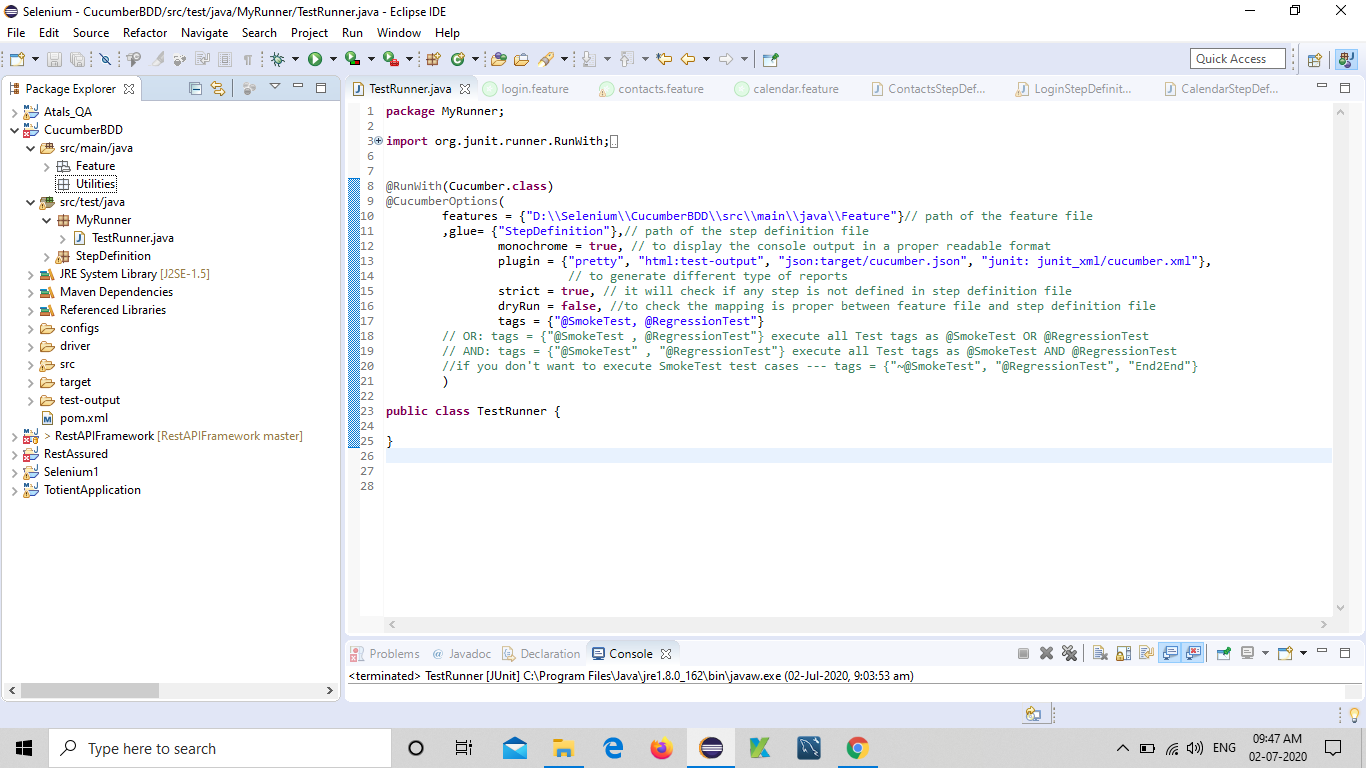
**Step 5:**

Under Feature package create .feature files

****

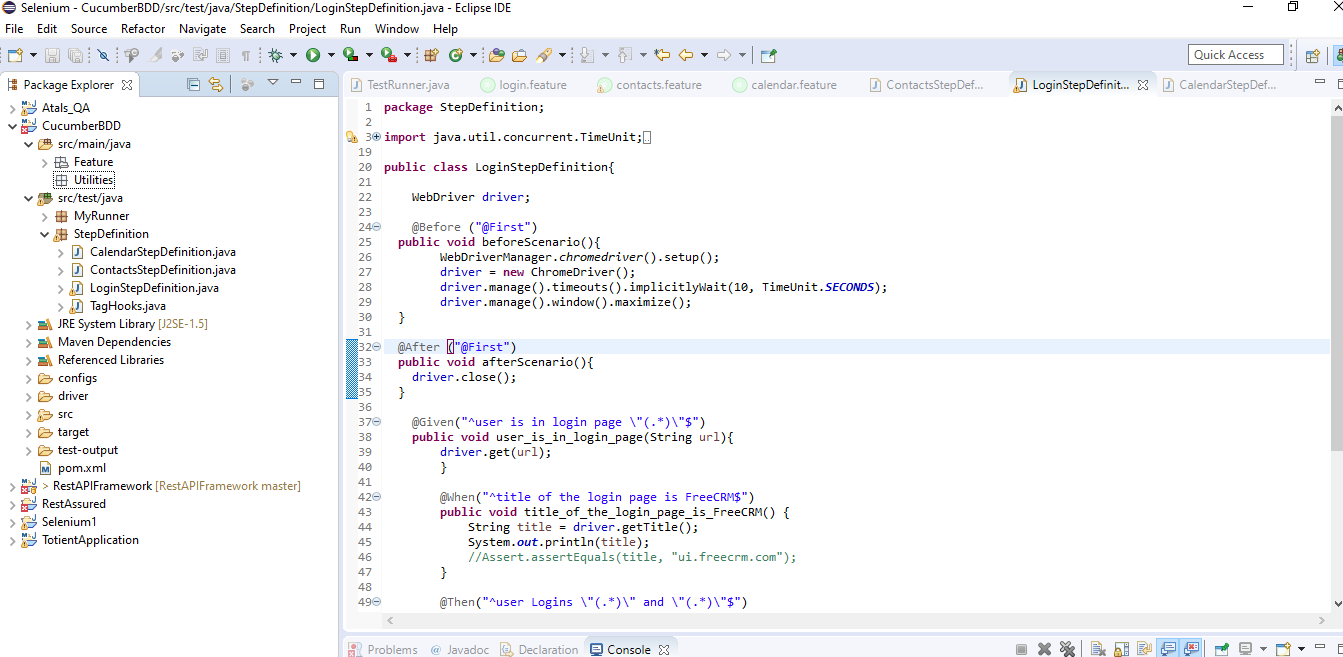
**Step 6:**

Under MyRunner Package create TestRunner.java class



**Step 7:**

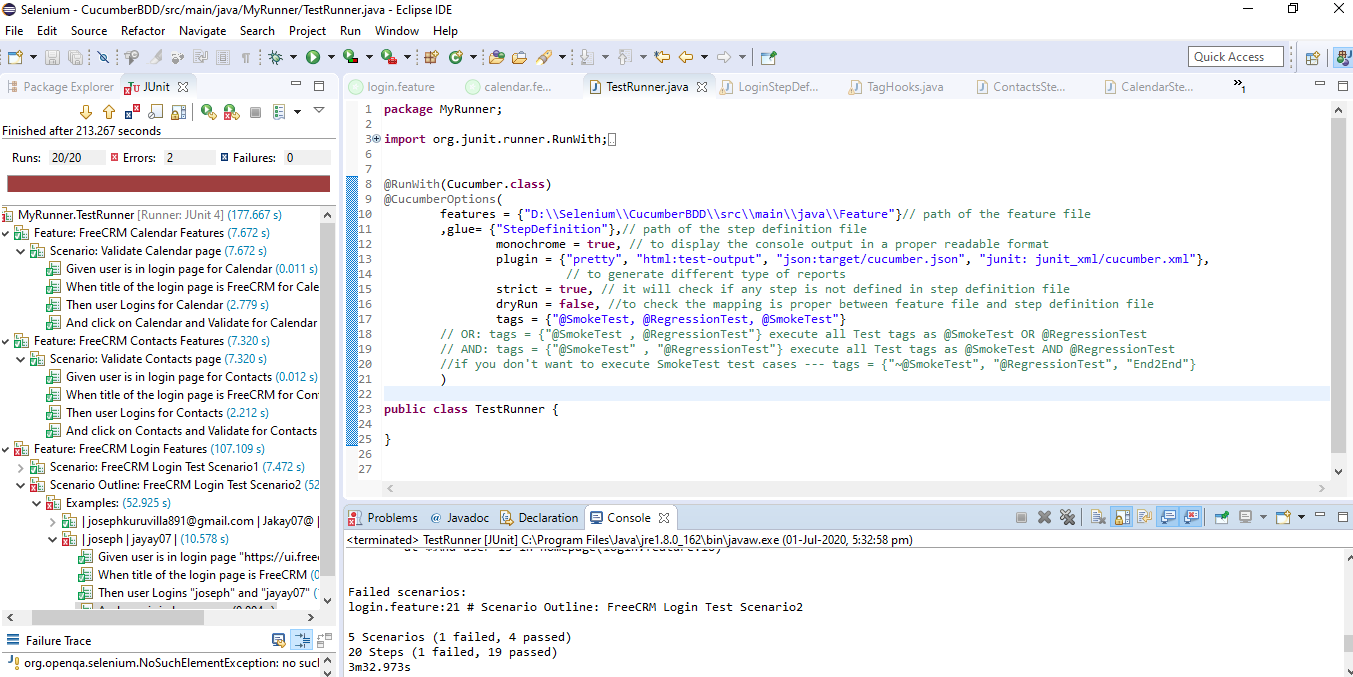
Under StepDefinition Package create (Functionality\_Name)StepDefinition.java classes



**Step 8:**

Run TestRunner.java

JUnit Execution Report will be generated



And HTML Report will be generated in Test-output folder

