Cucumber

Cucumber Testing

Cucumber is a widely used tool for Behaviour Driven Development because it provides an easily understandable testing script for system acceptance and automation testing.

Our Cucumber testing tutorial provides basic and advanced concepts of Cucumber testing. This Cucumber testing tutorial is designed for beginners and professionals.

What is Cucumber Testing

Cucumber is a Behavior Driven Development tool used to develop test cases for the behavior of software's functionality. It plays a supporting role in automated testing.

"Cucumber is a software tool used by the testers to develop test cases for the testing of behavior of the software."

Cucumber tool plays a vital role in the development of acceptance test cases for automation testing. It is mainly used to write acceptance tests for web applications as per the behavior of their functionalities.

It follows a **BDD** (Behavior Driven Development) framework to observe the behavior of the software's functionalities.

It allows business analysts, developers, testers, etc. to automate functional verification and validation in an easily readable and understandable format (e.g., plain English).

We can use Cucumber along with Watir, Selenium, and Capybara, etc. It supports many other languages like **PHP**, **Net**, **Python**, **Perl**, etc.

What is BDD?

BDD (Behavioral Driven Development) is a software development approach that was developed from **Test Driven Development (TDD)**.

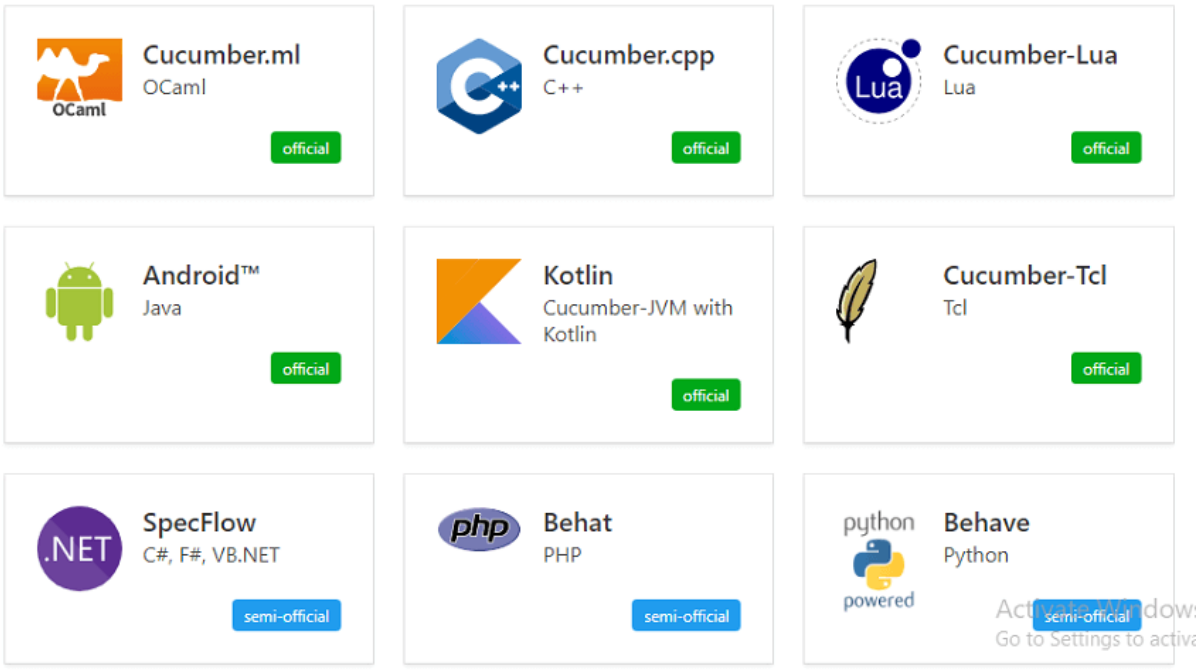
BDD includes test case development in the form of simple English statements inside a feature file, which is human-generated. Test case statements are based on the system's behavior and more user-focused.

BDD is written in simple English language statements rather than a typical programming language, which improves the communication between technical and non-technical teams and stakeholders.

Which language is used in cucumber?

Cucumber tool was originally written in the "**Ruby**" programming language. It was exclusively used only for testing of Ruby as a complement to the **RSpec** BDD framework.

But now, Cucumber supports a variety of different programming languages including Java, JavaScript, PHP, Net, Python, Perl, etc. with various implementations. In Java, it supports **native JUnit**.



Basic Terms of Cucumber

- Feature File
- Features
- Tags
- Scenario
- Gherkin Language
- Step Definition

How does Cucumber Testing Works?

Cucumber test cases are written parallel with the code development of software. These test cases are called step in a Gherkin Language.

- Firstly, Cucumber tool reads the step written in a Gherkin or plain English text inside the feature file.
- Now, it searches for the exact match of each step in the step definition file. When it finds its match, then executes the test case and provides the result as pass or fail.
- The code of developed software must correspond with the BDD defined test scripts. If it does not, then code refactoring will be required. The code gets freeze only after successful execution of defined test scripts.

Software tools supported by Cucumber

The piece of code to be executed for testing may belong to different software tools like **Selenium**, **Ruby on Rails**, etc. But cucumber supports almost all popular software platforms, and this is the reason behind Cucumber's popularity over other frameworks such as **JDave**, **Easyb**, **JBehave**, etc. Some Cucumber supported tools are given below:

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

Advantages of Cucumber Tool

- The main focus of the Cucumber Testing is on the end-user experience, as the success of the software ultimately depends on the end-user experience.
- The test case writing is very easy and understandable.
- It provides an end-to-end testing framework, unlike other tools.
- It supports almost all popular different languages like Java.net, JavaScript Ruby, PHP, etc.
- It works as a bridge between business and technical language, and this bridge is sustainable because of test cases written in a plain English text.
- The testing environment set up and execution both are very quick and easy.

- It is a well efficient tool for testing.

Difference between Cucumber and Selenium

Cucumber	Selenium
It is a **Behavior Driven Development** tool used to develop test cases for the behaviour of software's functionality.	It is an automated testing tool.
Cucumber is a free or open-source BDD (Behavior Driven Development) tool.	Selenium is also a free or open-source testing tool.
Cucumber is a BDD supported tool.	Selenium is a both Functional and Performance (Selenium Grid) testing tool.
Cucumber framework supports many languages, such as Java, Scala, Groovy, etc. beyond Ruby.	Selenium also supports many languages, such as Java, .Net, etc.
Cucumber includes both testers and developers to write automation steps.	Like Cucumber, Selenium also includes both testers and developers to write automation steps.
Cucumber is used to test only web applications.	Like Cucumber, Selenium also used to test only web applications.
Cucumber testing is less reliable as compared to Selenium and QTP.	The process of Selenium makes testing more reliable and dependable.
Cucumber works very fast in Plugin.	Selenium works slower in Plugin than Cucumber.

Behavior Driven Development

BDD (Behavioral Driven Development) is a **software development** approach that was developed from **Test Driven Development (TDD)**.

BDD includes test case development on the basis of the behavior of software functionalities. All test cases are written in the form of simple English statements inside

a feature file, which is human-generated. Acceptance test case statements are entirely focused on user actions.

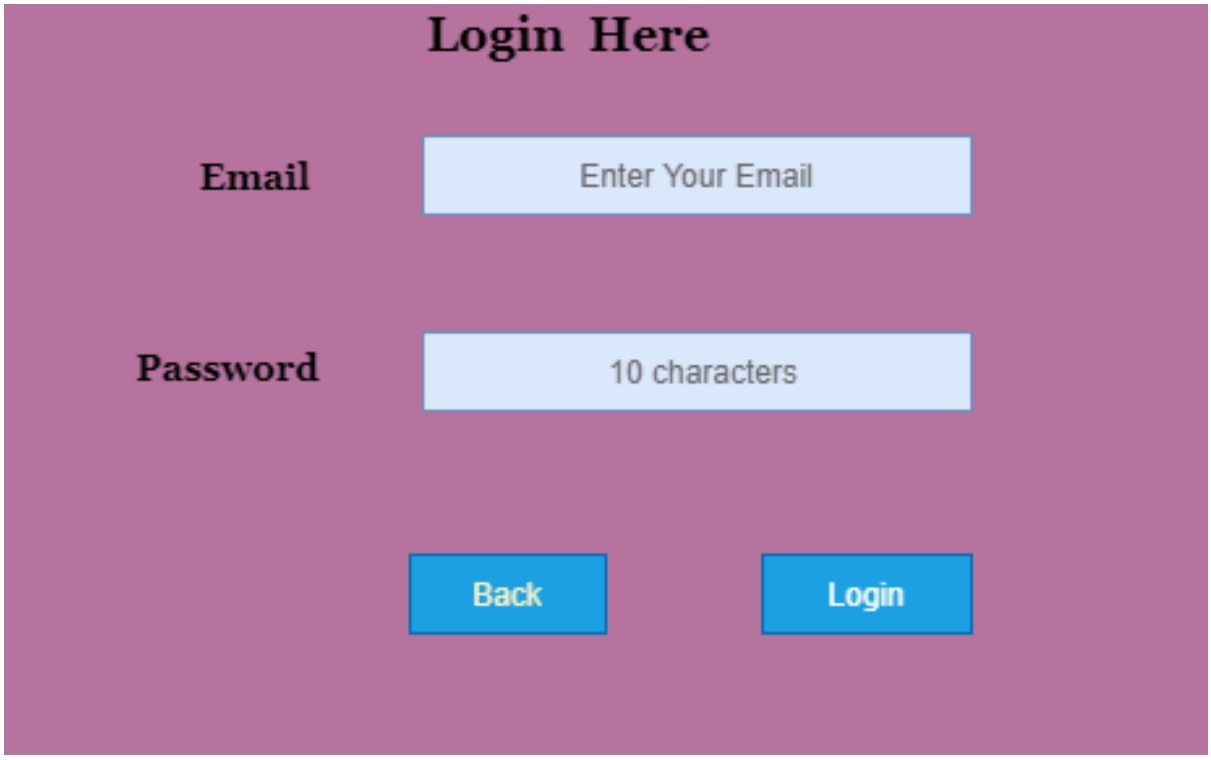
BDD is written in simple English language statements, not in a typical programming language. BDD improves communication between technical and non-technical teams and stakeholders.

Let's understand through an example, how we can develop test cases on the basis of the behavior of a particular function.

In the following example, we are going to take the **login function** of a web application.

Example:

In order to ensure the working of Login Functionality, we are developing acceptance test cases on the basis of BDD.



Feature: Login Function
To enter in the System
User must be able to
Access software when login is successful
Scenario: Login
Given User has its Email
And Password
When User enters the correct Email and Password
Then It should be logged in
Scenario: Unsuccessful Login
When User enters either wrong Email or Password
Then It should be reverse back on the login page with an error message

Need to Choose BDD

TDD works satisfactorily unless the business owners are familiar with the use of the unit testing. Also, their technical skills should strong enough, which is not always possible.

In these circumstances, BDD is advantageous because test cases are written in a common English language, which is easily understandable by all stakeholders.

The familiar, easily understandable language is the most significant advantage of using BDD because it plays a vital role in cooperation between technical and non-technical teams to perform a task with better efficiency.

Characteristics of BDD

Strong collaboration

BDD provides a strong collaboration between involved parties. It is just because of easy test cases which are written in the English language. In cucumber testing, stockholders play a vital role in constructive discussions as only they know the expectations from the software.

High Visibility

Everyone gets strong visibility in the progress of the project due to the easy English language.

The software design follows the business value

BDD gives great importance to business value and needs. By setting priorities with the client, depending on the value provided by them, developers are able to give a better result because they have a strong understanding of how the customer thinks.

The Ubiquitous Language

As mentioned earlier, test cases are written in the ubiquitous language, which is understandable by all the members of the team, whether they are from a technical field or not. This helps to reduc

e misconceptions and misunderstanding between the members related to concepts. Ubiquitous language makes easy joining of new members into the working.

Software development meets the user need.

BDD focuses on the business's needs so that users can be satisfied, and of course, satisfied users imply a growing business. With BDD, tester focuses on the behavior which has more impact than the implementation.

More confidence from the developers' side

The teams using BDD are generally more confident because they do not break the code, and when it comes to their work, a better forecast is done.

Lower Costs

By improving the quality of the code, BDD basically reduces the cost of maintenance and minimizes the risks of the project.

Feature File in Cucumber Testing

The feature file is the essential segment of cucumber tool, which is used to write acceptance steps for automation testing. Acceptance steps generally follow the application specification.

A feature file is usually a common file which stores feature, scenarios, and feature description to be tested.

The feature file is an entry point, to write the cucumber tests and used as a live document at the time of testing.

The extension of the feature file is "**.feature**". Each functionality of the software must have a separate feature file.

Example:

In order to ensure the working of Login Functionality, we are implementing the cucumber test by creating a feature file. It will verify whether the Login Functionality is working properly or not.

```
Feature: Login
Scenario: Login Functionality
Given user navigates to the website javatpoint.com
And there user logs in through Login Window by using Username as "USER" and Password as "PASSWORD"
Then login must be successful.
```

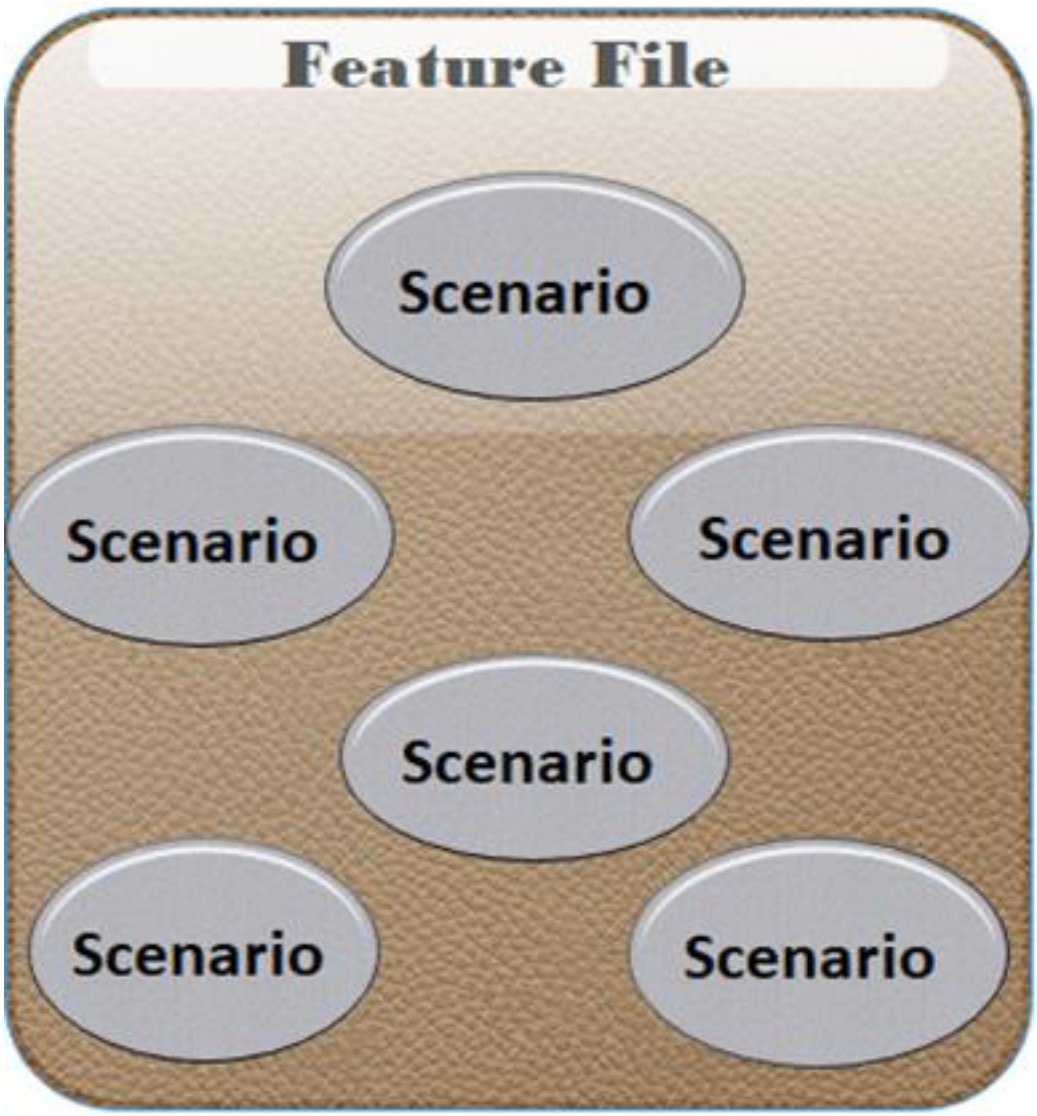
After performing the automation testing, a table is created as a result of automation testing. This table is used in tags.

The resulting table looks like the following table:

Test Name	Smoke Test	Regression Test	End2End	No Type
Sussessful Login	Yes	Yes		
UnSuccessful Login		Yes		
Add a Product to bag	Yes			
Add multiple Product to bag				
Remove a Product from bag	Yes	Yes		
Remove all Product from bag		Yes		
Increase Product quantity from bag page	Yes			
Decrease Product quantity from bag page				
Buy a Product with cash payment	Yes		Yes	
Buy a Product with CC Payment	Yes		Yes	
Payment declined				
→ CC Card			Yes	
→ DD Card			Yes	
→ Bank Transfer			Yes	
→ Paypal			Yes	
→ Cash			Yes	
15	6	4	7	3

Feature file with Multiple Scenario

Feature file can contain multiple scenarios or scenario outlines. We can write all possible Scenarios of a particular feature in a feature file.



By using the keyword "Scenario" or "Scenario Outline", One Scenario can be separated from another.

However, a single feature file can contain any number of scenarios but focuses only on one feature such as registration, login etc at a time. Therefore, it is better to keep the scenarios related to a particular feature in a single feature file.

Scenarios can be executed parallel, or you can execute them together in a group. Let's take an example for more clarity:

Example:

Feature File 1:

Feature: `Registration`

Background:

Given user on the homepage

And user follows "**Sign in**"

@regression

Scenario: Create a New User

When user fills "registration email textbox" with "chitrali.sharma27@gmail.com"

And user clicks "create an account button"

And user enters the following details

| First Name | Chitrali|

| Last Name | Sharma|

| Password | Inquiry@1234 |

| Date | 17| | Month | 02| | Year | 1992 |

And user clicks "register button"

Scenario: User does not follow form validations

When user enters wrong characters

Then error message displayed with invalid password

And user returns back on registration page

Feature File 2:

Feature: Login

Background:

Given user on the login page

And user follows "**Log in**"

@regression @smoke

Scenario: Verification of Login Function

Given user on the Login Page

And user enters "email address" with "**chitrali.sharma27@gmail.com**"

And user enters "password" with "**Inquiry@1234**"

And user click "**log in**" button

Then user should see "**My Account**"

Scenario: Unsuccessful login

Given user on the Login Page

And user enters "email address" with "**chitrali.sharma27@gmail.com**"

And user enters "password" with "**qsder@1234**"

And user clicks "**login**" button

Then error message displayed with wrong password

And user returns back on login page

Comments in Feature File

If we do not need to execute a particular scenario at a time, then we can comment that scenario.

In Eclipse, to comment a multi-line or use block comment first select all the line to be commented and then press **Ctrl + /**. Similarly, to remove comments, we need to press **Ctrl + **. Other IDEs may contain other shortcuts to do this.

While commenting any scenario, do not forget to comment the complete scenario. Otherwise, remaining lines of scenario which are not commented will be considered as a part of the previous scenario.

How to create Feature File

There are several approaches to create a feature file in different IDEs, here we are creating it in Eclipse IDE.

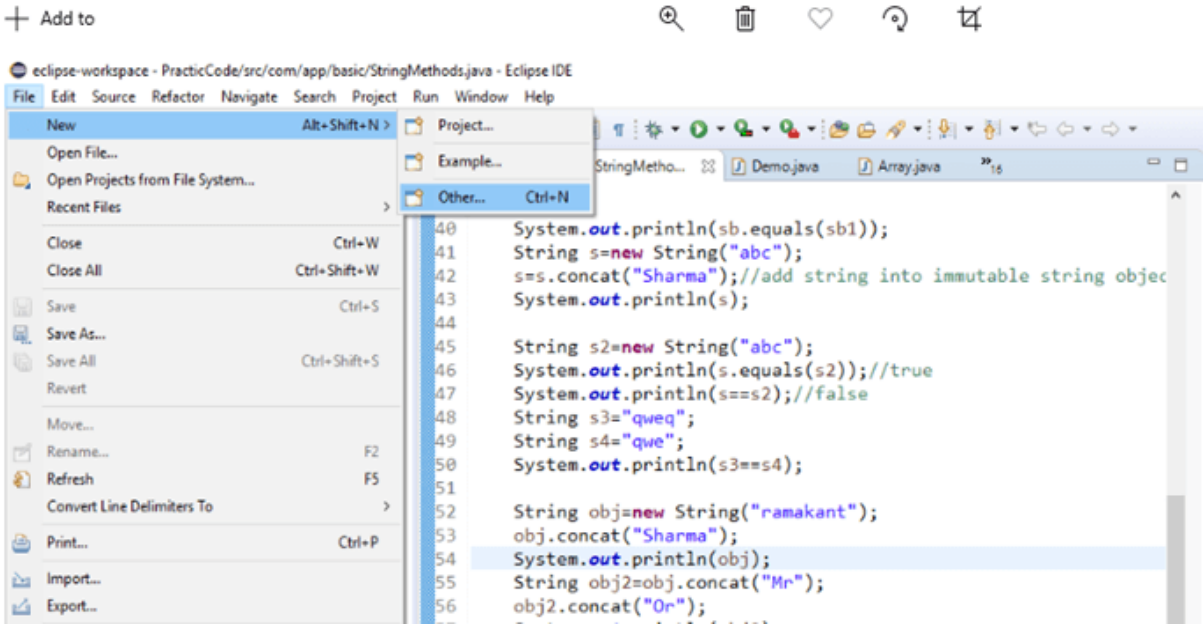
We can create a feature file with the "***.feature***" extension.

Following are the steps to create a feature file by using **eclipse IDE**:

- **Project**
- **Example**
- **Other**

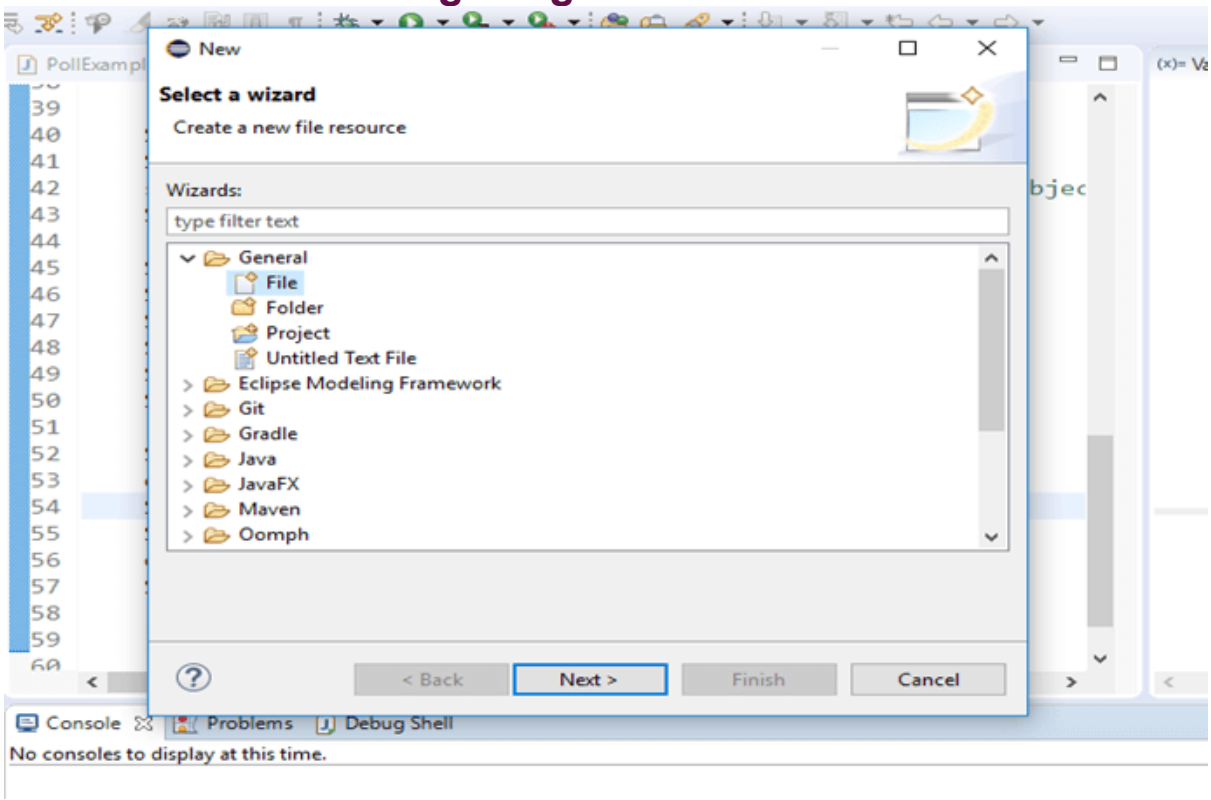
Select **Other** option from these three options.

Consider the following image:



2. After selecting **Other** option, you will get several options, from these options, select **General < File** option, and then click **Next**.

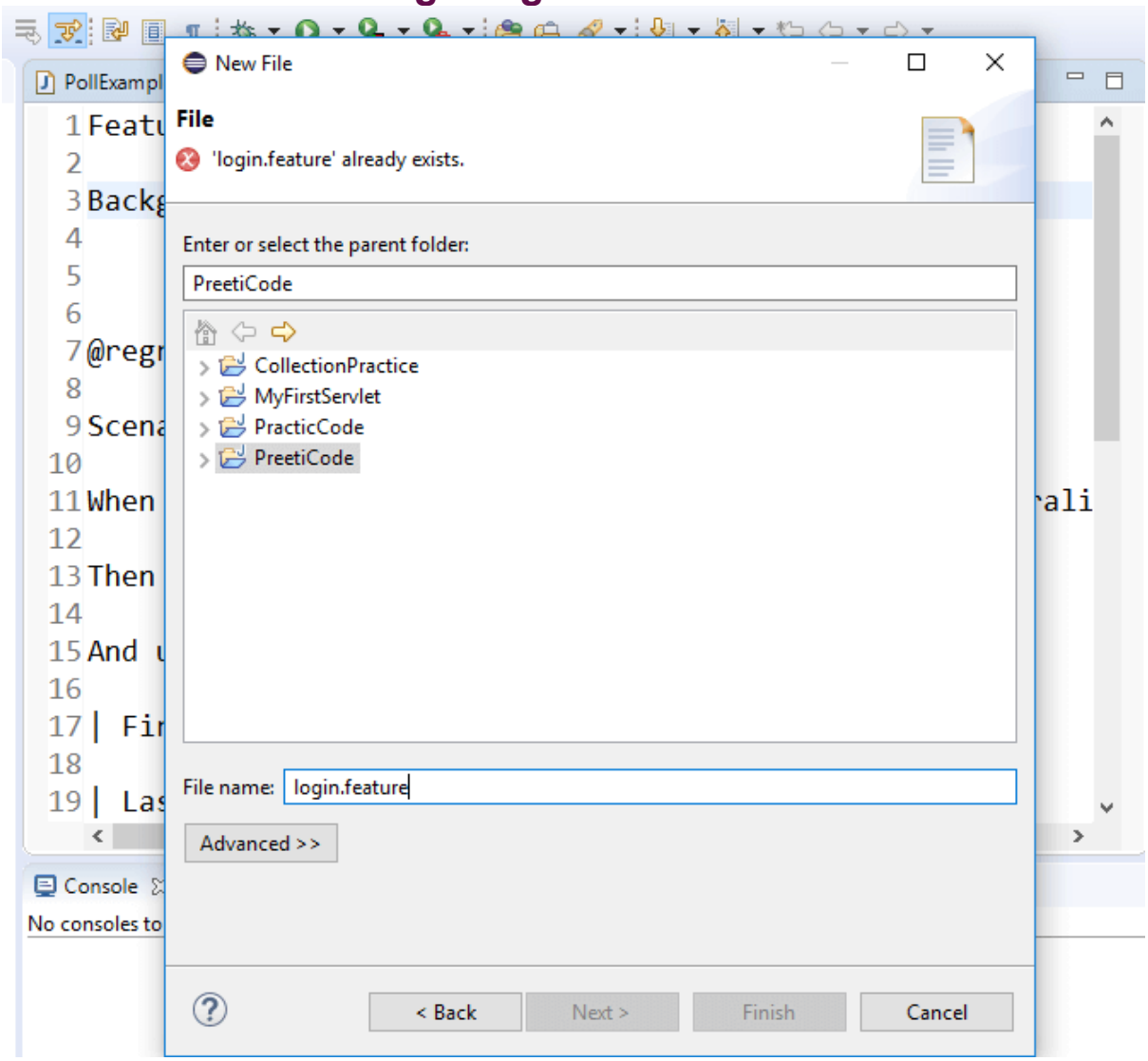
Consider the following image:



3. After clicking the **Next**, select the project inside which you want to create a feature file.

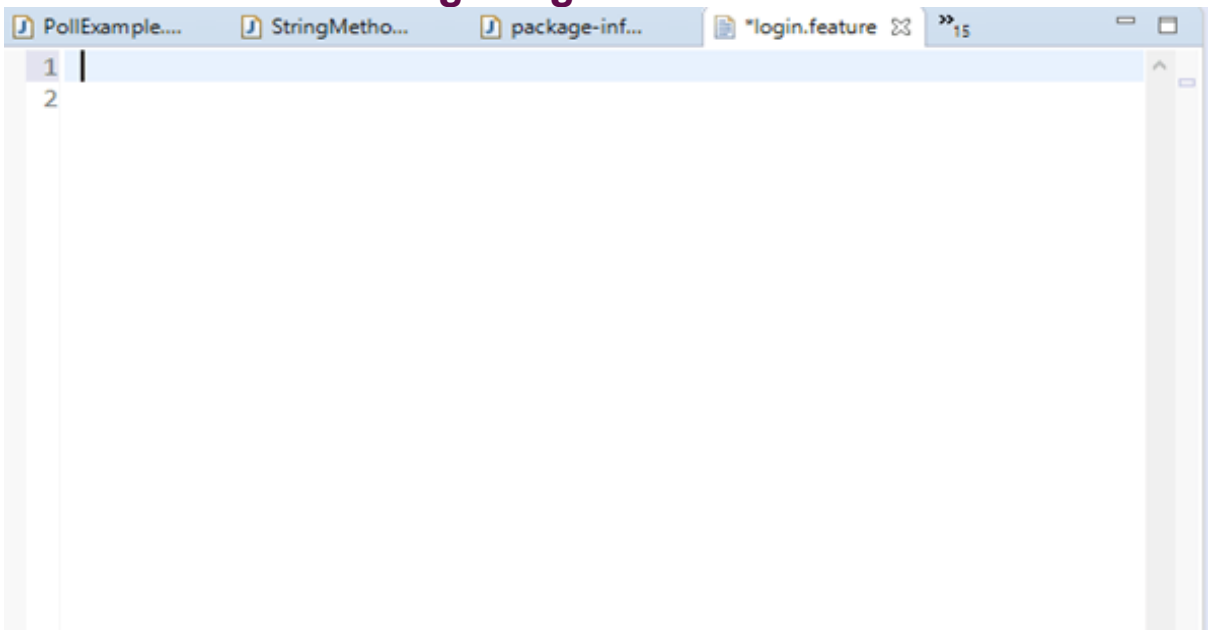
After selecting the project, you can create a feature file by giving a name and "**.feature**" extension. You can provide any name to the feature file on your own choice. After providing a name, click on the **Finish** button.

Consider the following image:



4. Now, the created feature file will appear inside the project.

Consider the following image:



The generated feature file will be similar to the above image. In it, we can write features, scenarios, and feature description of the web application to be tested.

What is tag in Cucumber testing?

In Cucumber, tags are used to associate a test like smoke, regression etc. with a particular scenario.

Tag fulfils the following purposes:

- If we have many scenarios in the feature file, to keep them in one group, we use tags in Cucumber, through which we will be able to prepare reports for specific scenarios under the same tag.
- By default, Cucumber executes all the scenarios inside the **feature file**, but if we need to execute or skip any specific scenario under a specific test, so we can declare scenarios within a tag.

We can declare a tag in a feature file by the following syntax:

```
@TestName
Scenario: Mention the Scenario
```

Where,

@: It is a symbol used to declare a tag.

TestName: It is the name of a specific test.

Scenario: It is a scenario.

Now, if we need to execute a scenario under multiple tests, in this case, we can create a set of multiple tests by using a tag.

Syntax:

@TestName@TestName
Scenario: `Mention the scenario`

Let's understand tag through an example:

Example:

Suppose, a feature file of an application contains 100 test scenarios, and when we test this application through Cucumber testing each time 100 test scenarios will get executed unnecessarily. And due to that, system performance is getting low.

To overcome this problem, we can use a tag.

Let's take an instance of a feature file with few scenarios.

@SmokeTest
Scenario: `Search contact`
Given: `Desired contact will be displayed`
@RegressionTest
Scenario: `Search a deal`
Given: `Desired deal will be displayed`
@SmokeTest
Scenario: `Search an email`
Given: `Desired email will be displayed`

There are two benefits by using the tag in the above feature file:

- First, the @SmokeTest or @RegressionTest tag contains only those scenarios that are applicable to the smoke or regression testing.
- Second, scenarios can be included or excluded as per the requirement at the time of execution.

Now suppose, we need to test only those scenarios which are declared under the smoke test, then we can mention **@SmokeTest** tag inside the testing code in the following way:

```
tags={"@SmokeTest"}
```

After mentioning the tag inside the testing code, only the scenarios which are declared under the smoke test will be tested and remaining will be skipped.

What's the need for tags in cucumber testing?

It looks easy when we just have a few numbers of scenarios in a feature file. However, in real-time projects, it does not happen.

In real-time projects, there may be a large number of feature files, which may have a **different purpose** such as Smoke test/Regression test, **different status** such as Ready for execution/Work in progress, **different prospective** such as Developer/QA/BA, etc.

In order to manage the execution of such large feature files, we use the tag with scenarios inside the feature file.

The benefit of the tag is that we can only test the specific scenario of a feature file, which we need to execute, and not those scenarios which are unnecessary.

How to create a set of multiple tags in cucumber testing?

We can also use multiple tags when we need to perform more than one testing of a single scenario.

Example:

@SmokeTest tag @RegressionTest tag

Scenario: `Search contact`

Given: `Desired contact will be displayed`

@RegressionTest tag

Scenario: `Search a deal`

Given: `Desired deal will be displayed`

@SmokeTest tag

Scenario: `Search an email`

Given: `Desired email will be displayed`

As per the above example, we can test the first scenario for both smoke testing and regression testing.

The testing through multiple tags can be done by using two operators:

- **OR operator**
- **AND operator**

OR operator

The OR operator can be used in the case, when we need to test an application like this, if the application has failed in the first test, then the next test should be checked. If the next test is also failed, then another next test should be checked and so on.

In other words, each test case among from the set of multiple tags must be tested whether it is failed or pass.

To use the OR operator in the test executing program, use the following syntax:

```
Tags= {"@FirstTest, @SecondTest"}
```

Example:

```
tags= {"@SmokeTest, @RegressionTest"}
```

In the above example, OR operator executes all the tagged tests i.e., **@SmokeTest**, **@RegressionTest**.

AND Operator

The AND operator can be used in the case, when we need to test an application like this, if the application has passed in the first test, then jump to check the next test, but if it gets failed, then testing should be terminated.

To use the AND operator in the test executing program, use the following syntax:

```
tags= {"@FirstTest", "@SecondTest"}
```

Example:

```
tags= {"@SmokeTest", "@RegressionTest"}
```

In the above example, AND operator executes the first test, if the result of this test is passed then it will jump to check the next test. But if the result is failed then testing will be terminated.

How to ignore tags in Cucumber testing?

In the case, when we need to skip a test, then we can use Special symbol "**~**" within the tag. This Special Character also works to skip both Scenarios and Features. And this can also work in conjunction with OR or AND.

Example:

Suppose there is a group of **@SmokeTest** and **@RegressionTest** tests. Now, we need to skip the regression test, to do that, consider the following code:

```
tags={"@SmokeTest", "~@RegressionTest"}
```

Installation-
https://www.javatpoint.com/install-cucumber-eclipse-plugin

What is the Data Table in Cucumber?

Data tables are used when we need to test numerous input parameters of a web application. For example, the registration form of the new user involves several parameters to test, so for this, we can use the data table.

Let's understand the data table with an instance of a registration form of javatpoint.com, which is a web application. For this registration form, we are going to create a typical feature file, and later we will create a **feature file with data table** to recognize the difference between them.

The registration form contains the following parameters:

- **User Name**
- **Email**
- **Password**
- **Confirm Password**
- **Birth-date**
- **Gender**
- **Phone Number**

How to create Selenium Maven Project in Eclipse?

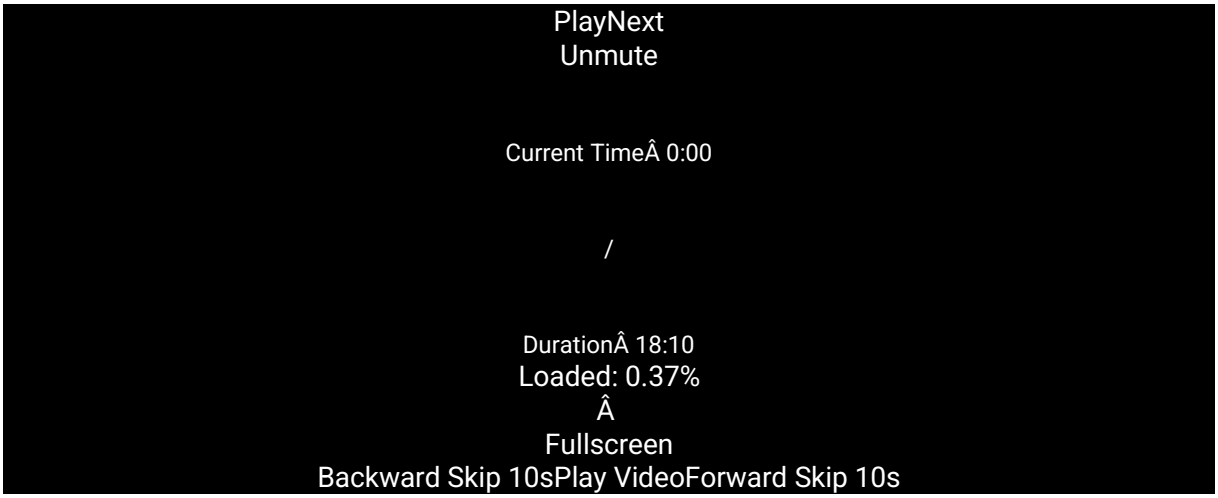
Before creating Selenium Maven testing Project for cucumber testing in Eclipse, first we must have the following dependencies in our computer system:

- **Java**
- **Eclipse**
- **Cucumber Eclipse Plugin**
- **Web driver Java Client**
- **Selenium**

Here we are creating a Maven project to test the registration feature through the data table:

Step 1

In order to create a Maven project first, Go to **File** → **New** → **Others** → **Maven** → **Maven Project** → **Next**.



After clicking the **Next**, a window will prompt. In this window, we need to provide **group Id** (group Id identifies the project uniquely across all projects). We have provided group Id as **"com.javatpoint."** You can give any name on your own choice.

Also, provide **artifact Id** (artifact Id represents the name of the project). We have provided artifact Id as **"CucumberTesting."** You can give any name on your own choice.

Click on **Finish**.

Step 2

Open pom.xml by using the following steps:

- Go to package explorer on the left side of the Eclipse window.
- Expand the project which is created for data table testing in cucumber then select its pom.xml file.
- Now open pom.xml and add the following dependencies.

Add dependency inside pom.xml for Selenium: This will indicate to Maven, which Selenium jar files will be downloaded from the central repository to the local repository.

- In pom.xml file, create a dependencies tag **(<dependencies></dependencies>)**, inside the project tag.
- Now, inside the dependencies tag, create a dependency tag **(<dependency></dependency>)**, and provide the following information within it.

```
1. <dependencies>
2. <dependency>
3.    <groupId> org.seleniumhq.selenium </groupId>
4.    <artifactId> selenium-java </artifactId>
5.    <version> 2.47.1 </version>
6. </dependency>
7. </dependencies>
```

Step 3

Add dependency inside pom.xml for Cucumber-Java: It will indicate to Maven; which Cucumber files will be downloaded from the central repository to the local repository.

- Now, inside the dependencies tag **(<dependencies></dependencies>)**, create a dependency tag **(<dependency></dependency>)**, and provide the following information within it.

```
1. <dependencies>
2. <dependency>
3.  <groupId> info.cukes </groupId>
```

```
4.  <artifactId> cucumber-java </artifactId>
5.  <version> 1.0.2 </version>
6.  <scope> test </scope>
7. </dependency>
8. </dependencies>
```

Step 4

Add dependency for Cucumber-Junit: It will indicate to Maven, which Cucumber JUnit files will be downloaded from the central repository to the local repository.

- Now, inside the dependencies tag **(<dependencies></dependencies>)**, create a dependency tag **(<dependency></dependency>)**, and provide the following information within it.

```
1. <dependencies>
2. <dependency>
3. <groupId> info.cukes </groupId>
4. <artifactId> cucumber-junit </artifactId>
5.  <version> 1.0.2 </version>
6.  <scope> test </scope>
7. </dependency>
8. </dependencies>
```

Step 5

Add dependency for Junit: It will indicate to Maven, which JUnit files will be downloaded from the central repository to the local repository.

- Now, inside the dependencies tag **(<dependencies></dependencies>)**, create a dependency tag **(<dependency></dependency>)**, and provide the following information within it.

```
1. <dependencies>
2. <dependency>
3.  <groupId> junit </groupId>
4.  <artifactId> junit </artifactId>
5. <version> 4.10 </version>
6. <scope> test </scope> </dependency>
7. </dependencies>
```

After completing all dependencies, verify binaries.

- Once pom.xml is completed successfully, then save it.
- Go to your Project → Clean - It can take a few minutes.

Now, create a package named **dataTable** under **src/test/java** folder of your project.

Step 6

Create a Feature file:

- Inside the package dataTable, create a feature file, named **dataTable.**feature.
- Inside the feature file, write the following text.

```
Feature - Data table
Verify that the new user registration is successful after passing correct inputs.
Scenario:
Given the user on the user registration page.
When user enter invalid data on the page
| Fields                 | Values             |
| First Name             | Preeti             |
| Last Name              | Sharma             |
| Email Address          | someone@gmail.com  |
| Re-enter Email Address | someone@gmail.com  |
| Password               |PASSWORD            |
| Birthdate              | 02                 |
Then the user registration should be successful.
```

- Save this file.

Step 7

Creation of the step definition file:

- Create the step definition file inside the package dataTable with extension ".java" and named as 'dataTable.java.'
- Inside the step definition file, write the following code.

```
1. package dataTable;
2. import java.util.List;
3. import org.openqa.selenium.By;
4. import org.openqa.selenium.WebDriver;
```

```
5.  import org.openqa.selenium.WebElement;
6.  import org.openqa.selenium.firefox.FirefoxDriver;
7.  import org.openqa.selenium.support.ui.Select;
8.  import cucumber.annotation.en.Given;
9.  import cucumber.annotation.en.Then;
10. import cucumber.annotation.en.When;
11. import cucumber.table.DataTable;
12. public class StepDefinition {
13.   WebDriver driver = null;
14.   @Given("^I am on user registration page$")
15.   public void goToFacebook() {
16.     //Intiate web browser instance. driver = new FirefoxDriver();
17.     driver.navigate().to("https://www.javaTpoint.com/");
18.   }
19.
20.   @When("^I enter valid data on the page$")
21.   public void enterData(DataTable table){
22.     //Initialize data table
23.     List<list> data = table.raw();
24.     System.out.println(data.get(1).get(1));
25.
26.     //Enter data
27.     driver.findElement(By.name("firstname")).sendKeys(data.get(1).get(1));
28.     driver.findElement(By.name("lastname")).sendKeys(data.get(2).get(1));
29.     driver.findElement(By.name("registered_email__")).sendKeys(data.get(3).get
    (1));
30.     driver.findElement(By.name("registered_email_confirmation__")).
31.       sendKeys(data.get(4).get(1));
32.     driver.findElement(By.name("registered_passwd__")).sendKeys(data.get(5).g
    et(1));
33.
34.     Select dropdownB = new Select(driver.findElement(By.name("birth_day")));

35.     dropdownB.selectByValue("12");
36.
37.     Select dropdownM = new Select(driver.findElement(By.name("birth_month
    ")));
```

```
38.      dropdownM.selectByValue("7");
39.
40.      Select dropdownY = new Select(driver.findElement(By.name("birth_year")))
    ;
41.      dropdownY.selectByValue("1992");
42.
43.      driver.findElement(By.className("_59mt")).click();
44.      // Click submit button driver.findElement(By.name("websubmit")).click();
45.   }
46.
47.   @Then("^User registration should be successful$")
48.   public void User_registration_should_be_successful() {
49.      if(driver.getCurrentUrl().equalsIgnoreCase("https://www.javaTpoint.com/"))
    {
50.         System.out.println("Test Pass");
51.      } else {
52.         System.out.println("Test Failed");
53.      }
54.      driver.close();
55.   }
56. }
```

Step 8

After creating the step definition file now, we need to create a runner class file.

- Create a runner class inside the package dataTable with extension "**.java**" and named as RunTest.java.
- Inside runner class **RunTest.java**, write the following code.

```
1. package dataTable;
2. import org.junit.runner.RunWith;
3. import cucumber.junit.Cucumber;
4. @RunWith(Cucumber.class)
5. @Cucumber.Options(format = {"pretty", "html:target/cucumber"})
6. public class RunTest { }
```

Save this file, and run the test by using the following options:

- Select the runner class i.e., RunTest.java file inside your package.
- Right-click on it, and select the option, **Run as → JUnit**.

If your execution is successful, you will observe the following things:

- **JavaTpoint** website gets loaded.
- We will see the home page or the page provided by the respective website.

Data can be entered on the registration page. What is Hook in Cucumber?

In Cucumber, the **hook is the block of code** which can be defined with each scenario in step definition file by using the annotation **@Before** and **@After**. These @Before and @After annotations create a block in which we can write the code.

Cucumber **hook** facilitates us to handle the code workflow better and also helps us to reduce code redundancy.

Syntax:

```
1. @Before setup ()
2. {
3. logic
4.
5.    } @
6.
7. Scenario
8. Given
9. When
10. And
11. Then
12.
13. @After cleanup (){
14.       logic
15.    }
```

As per the code logic, hook's job is to start and close the web driver session after a specific function/method. Hence, in actual, it is not relevant to any function/method or scenario

The Need of Hook

At the time of testing, we may encounter circumstances where we need to perform some conventional prerequisite steps before the testing of the test scenario.

Consider the following prerequisite to understand the kind of prerequisites which may encounter at the time of testing:

- To Start a web driver
- Set up of Data Base connections
- Set up of test data
- Set up of browser cookies
- Navigation to a certain page

Similarly, there are always some prerequisite steps which may encounter after testing:

- To stop the web driver

- To Close DB connections
- To Clear the test data
- To Clear browser cookies
- To Log out from the application
- Printing reports or logs
- Taking the screenshots of error

In order to handle these types of conventional prerequisite steps, using cucumber hook is the best option.

Hook Annotations

Unlike TestNG Annotations, the cucumber supports only two hooks:

- **@Before**
- **@After**

@Before

As the name suggests, we can use the **@Before** hook with the function/method after which we need to start web driver.

@After

As the name suggests, we can use the **@After** hook with the function/method after which we need to close the web driver.

Let's understand this notion better with an example of a step definition file.

Example:

Here is an instance of a step definition file of a **Maven testing project**. This project is created for the testing of web application javaTpoint.

In order to use the hook, we created the step definition file named **hookTest.java** under the package **javatpointTest**.

```
1. package javatpointTest;
2.
3. import org.openqa.selenium.By;
4. import org.openqa.selenium.WebDriver;
```

```
5.  import org.openqa.selenium.firefox.FirefoxDriver;
6.
7.  import cucumber.annotation.en.Given;
8.  import cucumber.annotation.en.Then;
9.  import cucumber.annotation.en.When;
10.
11. public class hookDemo {
12.   WebDriver driver = null;
13.
14.   @Before public void setUp(){
15.     driver = new FirefoxDriver();
16.   }
17.
18.   @Given("^User navigates to javatpoint$")
19.   public void goToFacebook() {
20.     driver.navigate().to("https://www.javatpoint.com/");
21.   }
22.
23.  @When("^ user enter Username as \"([^\"]*)\" and Password as \"([^\"]*)\"$"
     ")
24.   public void User_enter_Username_and_Password(String arg1, String arg2) {

25.     driver.findElement(By.id("emailAddress")).sendKeys(arg1);
26.     driver.findElement(By.id("password")).sendKeys(arg2);
27.     driver.findElement(By.id("u_0_v")).click();
28.   }
29.
30.   @Then("^login should be unsuccessful$")
31.   public void validateRelogin() {
32.     if(driver.getCurrentUrl().equalsIgnoreCase(
33.       "https://www.javatpoint.com/login.php?login_attempt=1&lwv=110")){
34.         System.out.println("Test Pass");
35.     } else {
36.       System.out.println("Test Failed");
37.     }
38.     driver.close();
39.   }
```

```
40.
41.    @After public void cleanUp(){
42.        driver.close();
43.    }
44. }
```

When we execute this code, the following will be the sequence of execution:

- At the beginning, **@Before** annotation will set up the web driver and other required prerequisites to execute the test.
- After setting up web driver and other prerequisites, the **Given** statement will be executed.
- After the execution of the **Given** statement, the **When** statement will be executed.
- After the execution of the **When** statement, the **Then** statement will be executed.
- Now at the last, **@After** hook will **close** the web driver and do the cleanup process.

Since we know that, to execute step definition file, we should have a complete **Maven testing project** so first create it in eclipse.

Tagged Hooks

The hook can also we used with tag. We can use **@before** and **@after** hooks with a specific test.

Example:

```
1. @Before ('@RegressionTest)
2.
3. @After ('@RegressionTest)
```

We can also use the same concept of the hook with logical and/or operator.

Example:

```
1. @Before ('@RegressionTest, @SmokeTest)
2.
3. @ After ('@RegressionTest, @SmokeTest)
```

-
- Submit button will be clicked.