**DAY 1 - CUCUMBER BASICS AND ADVANCED**

**TABLE OF CONTENTS**

1. Cucumber - Overview
2. Cucumber - Annotations
3. Cucumber - Scenario Outline
4. Cucumber - Tags
5. Cucumber - Data Tables
6. Cucumber - Comments
7. Cucumber - Hooks
8. Cucumber – Reports

1. Cucumber - Overview

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

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

Example

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

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

**How it Works**

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



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

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

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

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

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

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

Cucumber supports over a dozen different software platforms like −

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

**Advantages of Cucumber Over Other Tools**

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

**Cucumber - Annotations**

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

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

Scenario:

GIVEN I am a Facebook user

WHEN I enter my

AND I enter my

THEN login should be successful.

BUT home page should not be missing.

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

Background:

Go to Facebook home page.

**Example Scenario**

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

Step 1

Create a Maven Test Project named as AnnotationTest.

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

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>2.47.1</version>

</dependency>

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

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

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

Step 2

Create a package named Annotation under src/test/java

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

Step 3

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

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

|  |
| --- |
| Feature: annotation  #This is how background can be used to eliminate duplicate steps  Background:  User navigates to Facebook Given  I am on Facebook login page  #Scenario with AND  Scenario:  When I enter username as "TOM"  And I enter password as "JERRY"  Then Login should fail  #Scenario with BUT  Scenario:  When I enter username as "TOM"  And I enter password as "JERRY"  Then Login should fail  But Relogin option should be available  Step 4  Create a step definition file.   * Select and right-click on the package outline. * Click on ‘New’ file. * Give the file name as annotation.java * Write the following text within the file and save it.   package Annotation;  import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;  import cucumber.annotation.en.Given;  import cucumber.annotation.en.Then;  import cucumber.annotation.en.When;  public class annotation {  WebDriver driver = null;  @Given("^I am on Facebook login page$")  public void goToFacebook() {  driver = new FirefoxDriver();  driver.navigate().to("https://www.facebook.com/");  }      @When("^I enter username as \"(.\*)\"$")  public void enterUsername(String arg1) {  driver.findElement(By.id("email")).sendKeys(arg1);  }    @When ("^I enter password as \"(.\*)\"$")  public void enterPassword(String arg1) {  driver.findElement(By.id("pass")).sendKeys(arg1);  driver.findElement(By.id("u\_0\_v")).click();  }  @Then("^Login should fail$")  public void checkFail() {  if(driver.getCurrentUrl().equalsIgnoreCase(  "https://www.facebook.com/login.php?login\_attempt=1&lwv=110")){  System.out.println("Test1 Pass");  } else {  System.out.println("Test1 Failed");  }  driver.close();  }    @Then("^Relogin option should be available$")  public void checkRelogin() {  if(driver.getCurrentUrl().equalsIgnoreCase(  "https://www.facebook.com/login.php?login\_attempt=1&lwv=110")){  System.out.println("Test2 Pass");  } else {  System.out.println("Test2 Failed");  }  driver.close();  }  } |

**Step 5**

Create a runner class file.

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

package Annotation;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

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

public class runTest {

}

Step 6

Run the test using option −

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

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

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

Cucumber - Scenario Outline

**Scenario outline** basically replaces variable/keywords with the value from the table. Each row in the table is considered to be a scenario.

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

**Scenario:**

Given user navigates to Facebook

When I enter correct username and password

Then login should be successful

**Scenario:**

Given user navigates to Facebook

When I enter correct email address and password

Then login should be successful

**Scenario:**

Given user navigates to Facebook

When I enter correct phone number and password

Then login should be successful

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

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

Example

Let’s create an example of a scenario outline −

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

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

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>2.47.1</version>

</dependency>

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

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.0.2</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

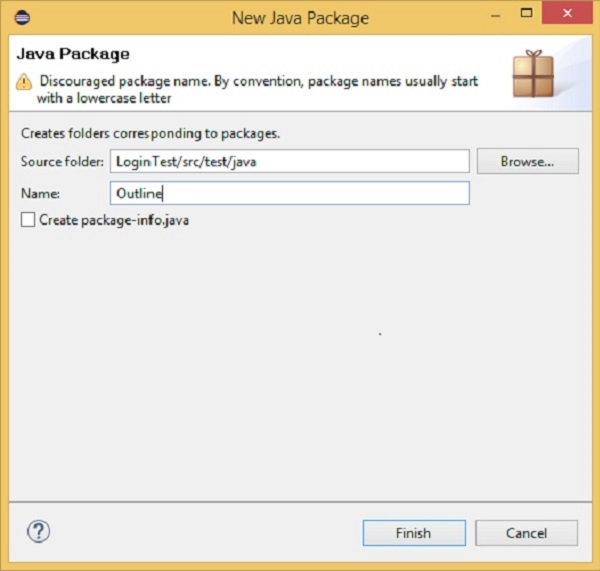
<version>4.10</version>

<scope>test</scope>

</dependency>

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

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



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

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

**Feature** − Scenario Outline

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

Given user navigates to Facebook

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

Then login should be unsuccessful

**Example** −

| username | password |

| username1 | password1 |

| username2 | password2 |

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

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

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

package Outline;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class stepdefinition {

WebDriver driver = null;

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

public void goToFacebook() {

driver = new FirefoxDriver();

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

}

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

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

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

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

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

}

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

public void validateRelogin() {

if(driver.getCurrentUrl().equalsIgnoreCase(

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

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

} else {

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

}

driver.close();

}

}

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

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

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

package Outline;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

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

public class runTest { }

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

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

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

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

Cucumber - Tags

It looks simple when we just have one, two, or maybe five scenarios in a feature file. However, in real life it does not happen. For each feature under test, we may have 10, 20, or may be more number of scenarios in a single feature file. They may represent different purpose (Smoke test/Regression test), different prospectives (Developer/QA/BA), different status (Ready for execution/Work in progress), etc. How to manage execution for such a mass?

For this, Cucumber has already provided a way to organize your scenario execution by using tags in feature file. We can define each scenario with a useful tag. Later, in the runner file, we can decide which specific tag (and so as the scenario(s)) we want Cucumber to execute. Tag starts with “@”. After “@” you can have any relevant text to define your tag. Let’s understand this with an example.

Suppose, there are two or more scenarios in a feature file. We want to execute only one scenario as part of smoke test. So first thing is to identify that scenario and second is to tag it with “@SmokeTest” text at the beginning of the scenario. Let’s take a deep look at it −

**Step 1** − Create a Maven project named as **cucumberTag**.

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

**Step 3**− Create a feature file named **cucumberTag.feature**.

Write the following text within the file and save it. This feature file contains two scenarios where only one has been marked as **SmokeTest** tag.

**Feature** − Cucumber Tag

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

Given user navigates to Facebook

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

Then login should be unsuccessful

Examples

| username | password |

| username1 | password1 |

| username2 | password2 |

#following scenario has been tagged as SmokeTest and this should get executed. @SmokeTest

**Scenario:**

Given user navigates to Facebook

When I enter Username as "<>" and Password as "<>"

Then the user should be redirected to login retry

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

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

package cucumberTag;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class cucumberTag {

WebDriver driver = null;

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

public void goToFacebook() {

driver = new FirefoxDriver();

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

}

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

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

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

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

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

}

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

public void validateRelogin() {

if(driver.getCurrentUrl().equalsIgnoreCase(

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

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

} else {

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

}

driver.close();

}

@Then("^User should be redirected to login retry$")

public void loginRetry() {

if(driver.getCurrentUrl().equalsIgnoreCase(

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

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

} else {

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

}

driver.close();

}

}

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

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

package cucumberTag;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

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

public class runTest { }

* Run the test option.
* Right-click and select the option **‘Run as’**.
* Select JUnit test.

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

* Facebook opens in a new Firefox web-browser instance.
* No value will be provided to the username and the password field.
* Login will be clicked.
* Login retry page will be loaded.

There is no limit in defining tags within the feature file. Based on your need, you can derive tags to be used and scenarios to be executed.

There are mainly two types of tag −

* **Default tag** − Default tag has their predefined meanings. Example @Dev,@Ignore
* **Custom tag** − Custom tag provides you full flexibility to choose appropriate text for defining your tag.

Tag can also be defined at a feature level. Once you define a tag at the feature level, it ensures that all the scenarios within that feature file inherits that tag. Depending on the nature of the scenario, we can use more than one tag for the single feature. Whenever Cucumber finds an appropriate call, a specific scenario will be executed.

Cucumber also provides a way to inverse the choice of tags. Consider that out of 25 defined scenarios, 10 are marked as smoke test. We are required to execute only regression test scenarios.

For this, we can use “~” in JUnit runner class to exclude smoke test scenario. It will look like the following.

@RunWith(Cucumber.class)

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

tags = {"~@SmokeTest"})

public class runTest { }

While defining multiple tags, we can also define logical or/and logical and operation.

* Defining logical or in runner class − **@dev,@wip** − It says that scenarios matching any of this tag needs to be executed.
* Defining logical or in runner class − **[@dev,~@wip] −** It says that scenarios matching both these tag needs to be executed.

Cucumber - Data Tables

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

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

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

**Feature** − New user registration.

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

Given I am on a new user registration page.

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

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

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

Given I am on a new user registration page

When I enter valid data on the page

| Fields | Values |

| First Name | Tom |

| Last Name | Kenny |

| Email Address | someone@someone.com |

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

| Password | Password1 |

| Birthdate | 01 |

Then the user registration should be successful.

Example

Let’s automate an example of a data table.

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

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

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>2.47.1</version>

</dependency>

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

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

**Step 3** − Create a Feature file.

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

**Feature** − Data table

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

**Scenario:**

Given I am on the new user registration page

When I enter invalid data on the page

| Fields | Values |

| First Name | Tom |

| Last Name | Kenny |

| Email Address | someone@someone.com |

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

| Password | Password1 |

| Birthdate | 01 |

Then the user registration should be unsuccessful

* Save the file.

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

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

package dataTable;

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.firefox.FirefoxDriver;

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

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

import cucumber.table.DataTable;

public class stepdefinition {

WebDriver driver = null;

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

public void goToFacebook() {

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

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

}

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

public void enterData(DataTable table){

//Initialize data table

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

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

//Enter data

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

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

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

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

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

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

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

dropdownB.selectByValue("15");

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

dropdownM.selectByValue("6");

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

dropdownY.selectByValue("1990");

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

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

}

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

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

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

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

} else {

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

}

driver.close();

}

}

* Save the file.

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

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

package dataTable;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

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

public class runTest { }

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

You may observe the following things upon successful execution.

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

Cucumber - Comments

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

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

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

Example

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

Feature: annotation

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

Background:

User navigates to Facebook

Given I am on Facebook login page

#Scenario with AND

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

#Scenario with BUT

Scenario:

When I enter username as "TOM"

And I enter password as "JERRY"

Then Login should fail

But Relogin option should be available

Cucumber - Hooks

Cucumber **hook** 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.

To understand this notion better, let’s take an example of a feature file and a step definition file.

The highlighted portion in the given statement actually does the job of setting up the webdriver and ending the webdriver session. So, it is actually not relevant to the essence of “Given statement”, and it is more like a setup for the test. Also if we think with a broader prospective, then in case of multiple scenarios for this feature, this webdriver setup and cleanup will run with each given statement. Logically, it makes sense to have the setup and cleanup executed only once.

So to bring optimization, hooks can be utilized. More often we use two types of hooks: “Before” hook and “After” hook. Method/function/piece of code, defined within Before and After hooks, always run, even if the scenario gets passed or failed.

As the name suggests, before hook gets executed well before any other test scenarios, and after hook gets executed after executing all the scenarios.

Hooks are defined within the step definition file only.

Let’s automate an example of before and after hook.

**Step 1** − Create Maven project as hookTest, add necessary dependency in pom.xml.

**Step 2** − Create a Java package named as hookTest under **src/test/java**

**Step 3** − Create a step definition file named as **hookTest.java** under the package.

package hookTest;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class hookTest {

WebDriver driver = null;

@Before public void setUp(){

driver = new FirefoxDriver();

}

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

public void goToFacebook() {

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

}

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

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

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

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

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

}

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

public void validateRelogin() {

if(driver.getCurrentUrl().equalsIgnoreCase(

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

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

} else {

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

}

driver.close();

}

@After public void cleanUp(){

driver.close();

}

}

**Step 4** − Create a feature file named “hookTest.feature” under the package.

**Feature** − Scenario Outline.

**Scenario Outline** − Hook Test

Given user navigates to Facebook

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

Then login should be unsuccessful

Examples

| username | password |

| username1 | password1 |

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

* Create the runner class as runTest.java inside the package dataTable (see the section scenario outline for more detailed steps).
* Write the following code.

package hookTest;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

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

public class runTest { }

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

So now when we run this, following will be the sequence of execution.

* Before hook − Set up the webdriver and other prerequisites to run the test.
* Given statement
* When statement
* Then statement
* After hook − Close the webdriver and do the cleanup process.

Cucumber - Reports

We do test execution in order to understand the stability of a product, so be it manual test or an automated test, it is very important to generate a concise report that can depict the stability of a product. Hence, while we are automating our test scenario with Cucumber, it is essential to know, how better we can generate our Cucumber test reports. As we know that Cucumber is a BDD framework, it does not have a fancy reporting mechanism.

In order to achieve this, we need to integrate Cucumber with other open source tool like Ant/Junit. Here, we will take examples of JUnit further because, it provides support for Java language.

Let’s look into the details of different report format, which is available and easy to use −

Pretty Format (HTML Report)

Pretty Format generates the Cucumber test report in the HTML format, i.e. an HTML file. It is the most readable report format. It generates the report in the same way as it is a feature file, so tracing is also made easy. Also, you can specify the location where you want this report to be placed after the test execution. It can be −

* **Local Directory** − We can specify target directory for report as any local directory of the machine where the test will run.
* **Server Directory** − Also we have a provision to specify a target directory as any directory on the server, which is publically accessible. This generally helps when we want our clients/stakeholders to view the test results at any given point of time.

Example

Let’s automate an example of a pretty format.

**Step 1** − Create a Maven project named **cucumberReport** in Eclipse.

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

**Step 3** − Create a feature file named **cucumberReport.feature**

Write the following text within the file and save it.

**Feature** − Cucumber Report

#This is to check test result for Pass test case

**Scenario:** Login functionality exists

Given I have opened the browser

When I open Facebook website

Then Login button should exist

#This is to check test result for Failed test case

**Scenario:** Forgot password exists

Given I have open the browser

When I open Facebook website

Then Forgot password link should exist

**Note** − Here scenario first will pass, whereas the second scenario will fail. So that we can witness how the pass and failed report looks like.

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

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

package CucumberReport;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.annotation.en.Given;

import cucumber.annotation.en.Then;

import cucumber.annotation.en.When;

public class cucumberReport {

WebDriver driver = null;

@Given("^I have open the browser$")

public void openBrowser() {

driver = new FirefoxDriver();

}

@When("^I open Facebook website$")

public void goToFacebook() {

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

}

@Then("^Login button should exits$")

public void loginButton() {

if(driver.findElement(By.id("u\_0\_v")).isEnabled()) {

System.out.println("Test 1 Pass");

} else {

System.out.println("Test 1 Fail");

}

}

@Then("^Forgot password link should exist$")

public void forgotPWD() {

if(driver.findElement(By.id("")).isEnabled()) {

System.out.println("Test 1 Pass");

} else {

System.out.println("Test 1 Fail");

}

}

}

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

* Create a runner class named **runTest.java** inside the package.
* Write the following code. Save the file.

package CucumberReport;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

@Cucumber.Options(

format = {"pretty", "html:target/Destination"} )

//Specifying pretty as a format option ensure that HTML report will be generated.

//When we specify html:target/Destination - It will generate the HTML report

inside the Destination folder, in the target folder of the maven project.

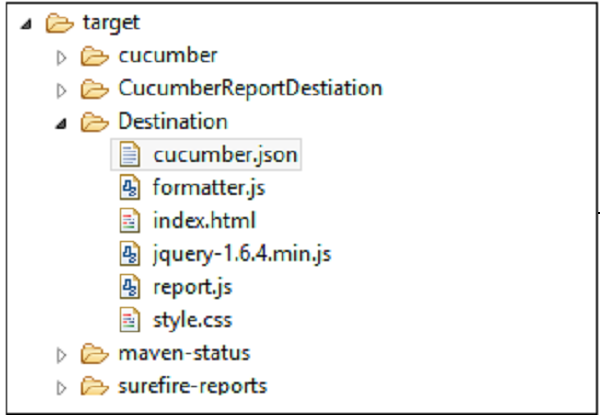
public class runTest { }

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

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

Both the scenario will get executed one by one.

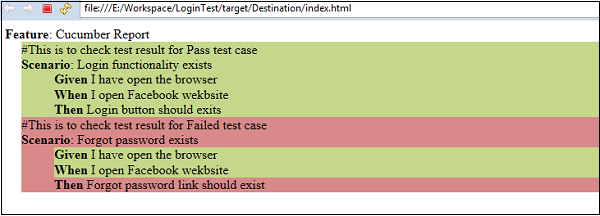
A folder named Destination will be created inside the target directory.



The report will be there named as “Index.html”.

Open Index.html with web browser.

You will see the report mentioned in the following image −



It exactly highlights the color of failed scenario. Moreover, you will see highlight for failed step in that scenario. This makes the debugging very easy.

**JSON Report**

By now we have seen how easy HTML report is. However, if we want to pass on this report information to any other application, that’s kind of tricky in case of HTML reports. Here comes the need of another reporting format. JSON-Java script object notation is another format for generating Cucumber test reports. JSON is an object containing a lot of information stored in text format. JSON reports bring a different value to the table. JSON report can also be used as a payload of information to be transferred between different servers. Further, it can be used to be displayed as a web page. In a nutshell, JSON reports can be used by other application.

**What is payload information?** When data is sent over the Internet, each unit transmitted includes both header information and the actual data being sent. The header identifies the source and destination of the packet, while the actual data is referred to as the payload. In order to generate a JSON report, we just need to make a change in the runner file.

* Change the format option in the runner file as follows.

package CucumberReport;

import org.junit.runner.RunWith;

import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)

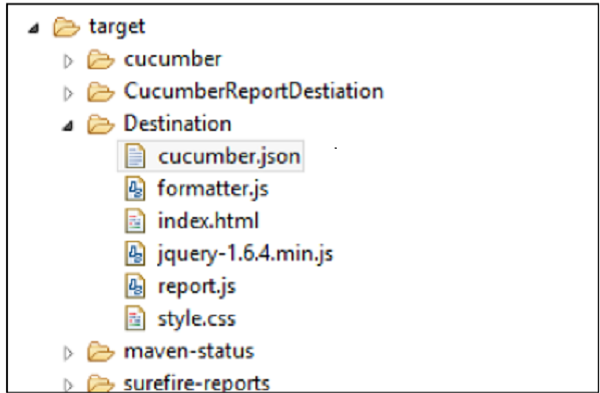
@Cucumber.Options( format={"json:target/Destination/cucumber.json"})

//When we specify json:target/Destination/cucumber.json - It will generate the JSON

report inside the Destination folder, in the target folder of the maven project.

public class runTest {}

* Run the test using option −
  + Select **runTest.java** file from package explorer.
  + Right click and select option **Run as**.
  + Select JUnit test.
* You will observe the following things when you run this class file.
  + Both the scenario will get executed one by one.



* The report will be there named as **cucumber.json** (as provided in runner class).
* Open **cucumber.json** file with the text editor.
* You will see the report mentioned in the following screenshot after placing line breaks −



**Note** − JSON is less readable as compared to the HTML report format.

**References:**

1. Cucumber Quick Guide - <https://www.tutorialspoint.com/cucumber/cucumber_quick_guide.htm>