BDD

Lesson – 4 : Cucumber – Introduction

Capgemini

Lesson Objectives

In this lesson, you will learn:

- Cucumber Framework
- How it Works
- Advantages of Cucumber
- Feature File
- Steps Definitions
- Cucumber Options
- *Scenario Outline*
- *Data Tables in Cucumber*

Cucumber Framework

- Cucumber is one such open source tool, which supports behavior driven development
- Cucumber is a software requirements and testing tool that enables a style of development that builds on the principles of test-driven development
- Cucumber reads the code written in plain English text (Language Gherkin) in the feature file
- It finds the exact match of each step in the step definition
- Cucumber supports over a dozen different software platforms like –
  - Ruby on Rails
  - Selenium
  - PicoContainer
  - Spring Framework
  - Watir

How it Works


1 Describe Behaviour
2 Write step definition
3 Run and fail
4 Write code to make step pass
5 Run and pass

Advantages of Cucumber

- 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.

Feature Files

- A **Feature** can be defined as a standalone unit or functionality of a project.
- Cucumber tests are written in **feature files**.
- A **feature** usually contains a list of scenarios to be tested for that feature
- It is advisable that there should be a separate feature file, for each feature under test.
- There are no logic details written in the feature file.
- The extension of the feature file needs to be ".feature".
- For Example:

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

Feature Files

Example

Feature: Login functionality for a social networking site. The user should be able to login into the social networking site if the username and the password are correct. The user should be shown the error message if the username and the password are incorrect. The user should be navigated to home page, if the username and password are correct.

In Feature file it will be written with Gherkin keywords as below:

Feature: Login functionality for a social networking site. **Given** I am a social networking site user. **When** I enter username as username1. **And** I enter password as password1. **Then** I should be redirected to the home page of the site.

The above-mentioned scenario is of a feature called user login.

All the words highlighted in bold are Gherkin keywords.

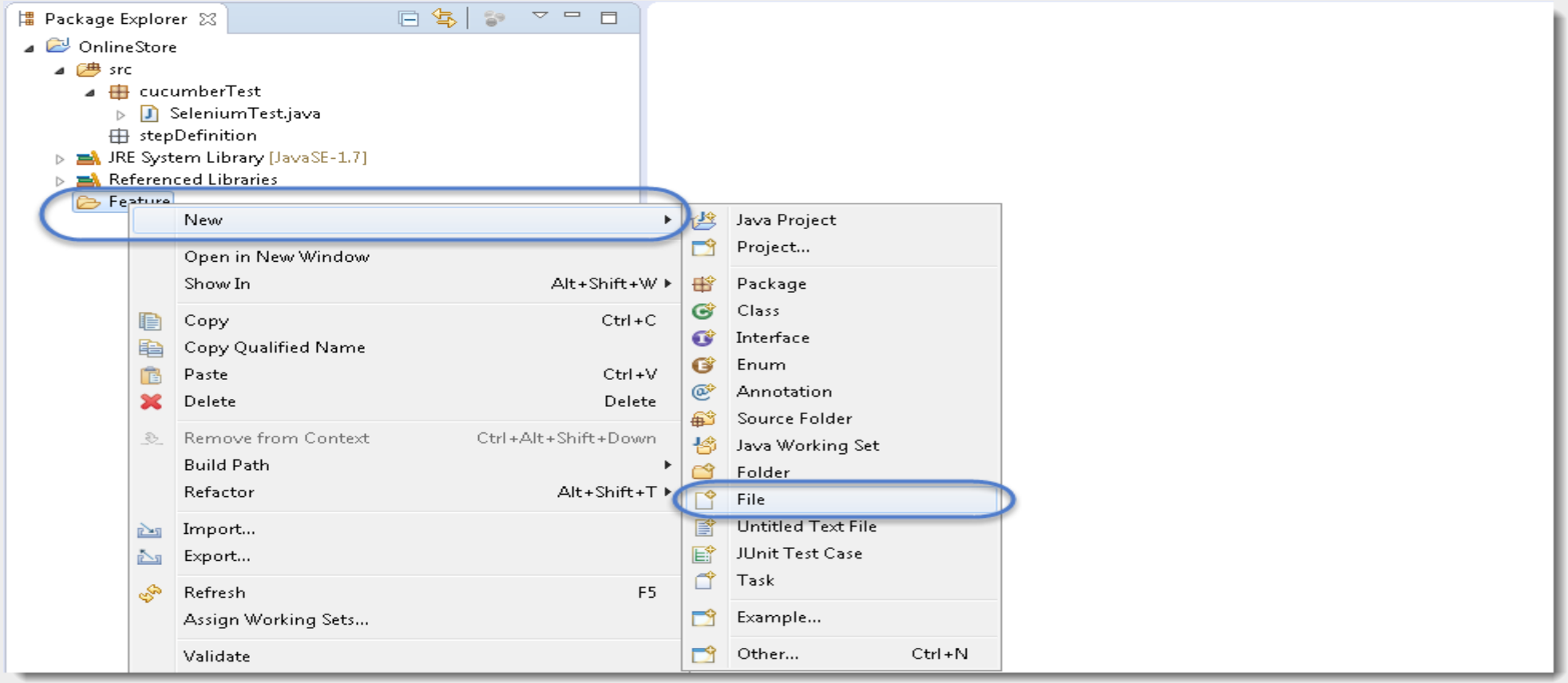
Feature Files

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

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

Feature File

1) On the ***Feature*** folder *Right click* and select ***New > File***

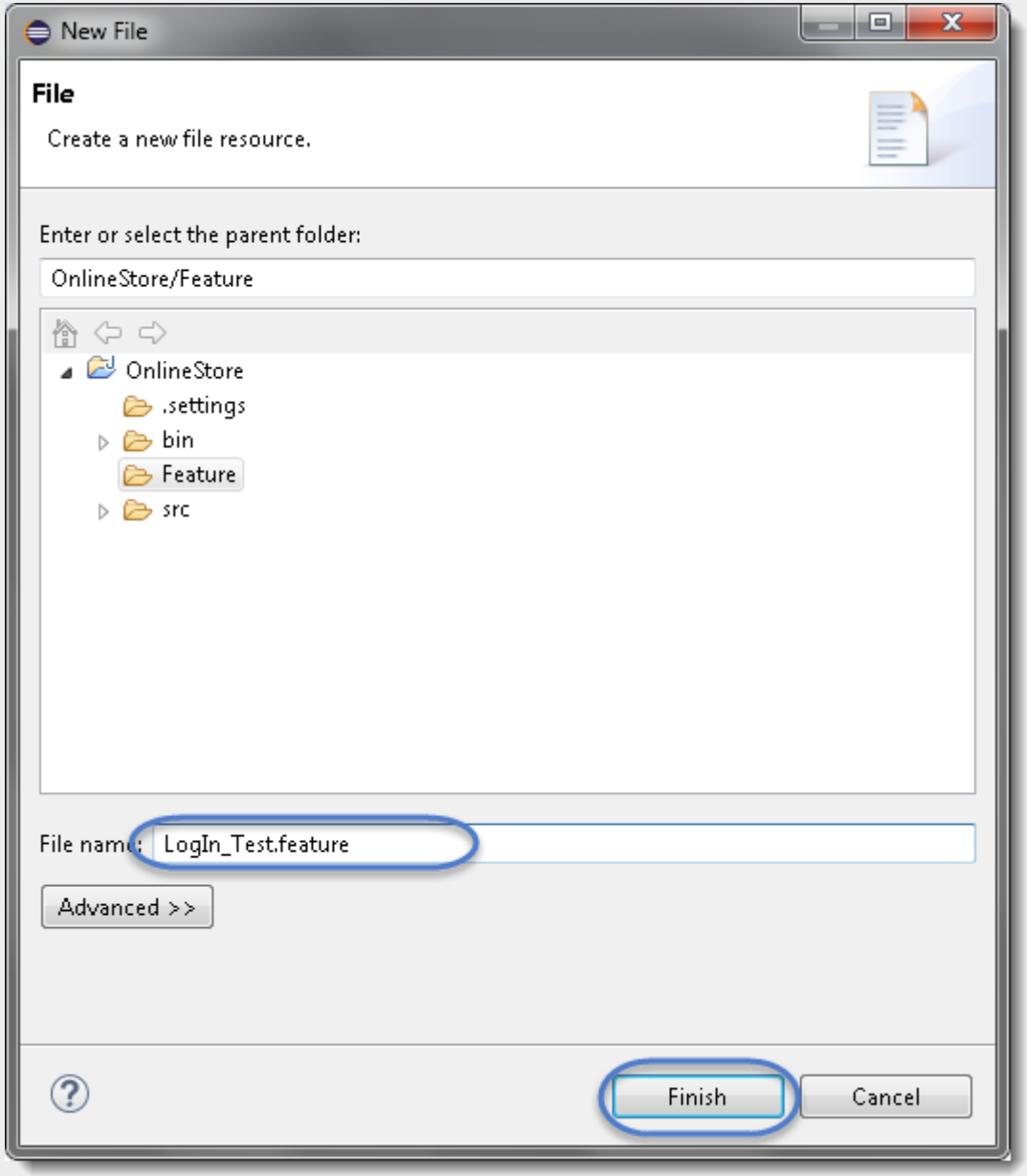


Feature File

2) In order for Cucumber to automatically detect the stories (or ***features***, as they're known in *Cucumber*), you need to make sure that they carry the '***.feature***' file extension.

For example, in this case, I've named my user story '***LogIn_Test.feature***'.

Every '*.feature*' file conventionally consists of a single feature



Feature File

3) Write the first cucumber script. In BDD terms the scenario would look like the following.

Cucumber Test Script

```
1   Feature: Login Action
2
3   Scenario: Successful Login with Valid
4   Credentials
5   Given User is on Home Page
6   When User Navigate to LogIn Page
7   And User enters UserName and Password
8   Then Message displayed Login Successfully
9
10  Scenario: Successful LogOut
11  When User LogOut from the Application
    Then Message displayed LogOut
    Successfully
```

Steps Definitions

- Steps definition file stores the mapping between each step of the scenario defined in the feature file with a code of function to be executed.
- When Cucumber executes a step of the scenario mentioned in the feature file, it scans the step definition file and figures out which function is to be called.
- So with each function, whatever code you want to execute with each test step (i.e. GIVEN/THEN/WHEN), you can write it within Step Definition file.
- Make sure that code/function has been defined for each of the steps.
- This function can be Java functions, where we can use both Java and Selenium commands in order to automate our test steps.

Steps Definitions

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

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

Steps Definitions

Problems | Javadoc | Declaration | Console

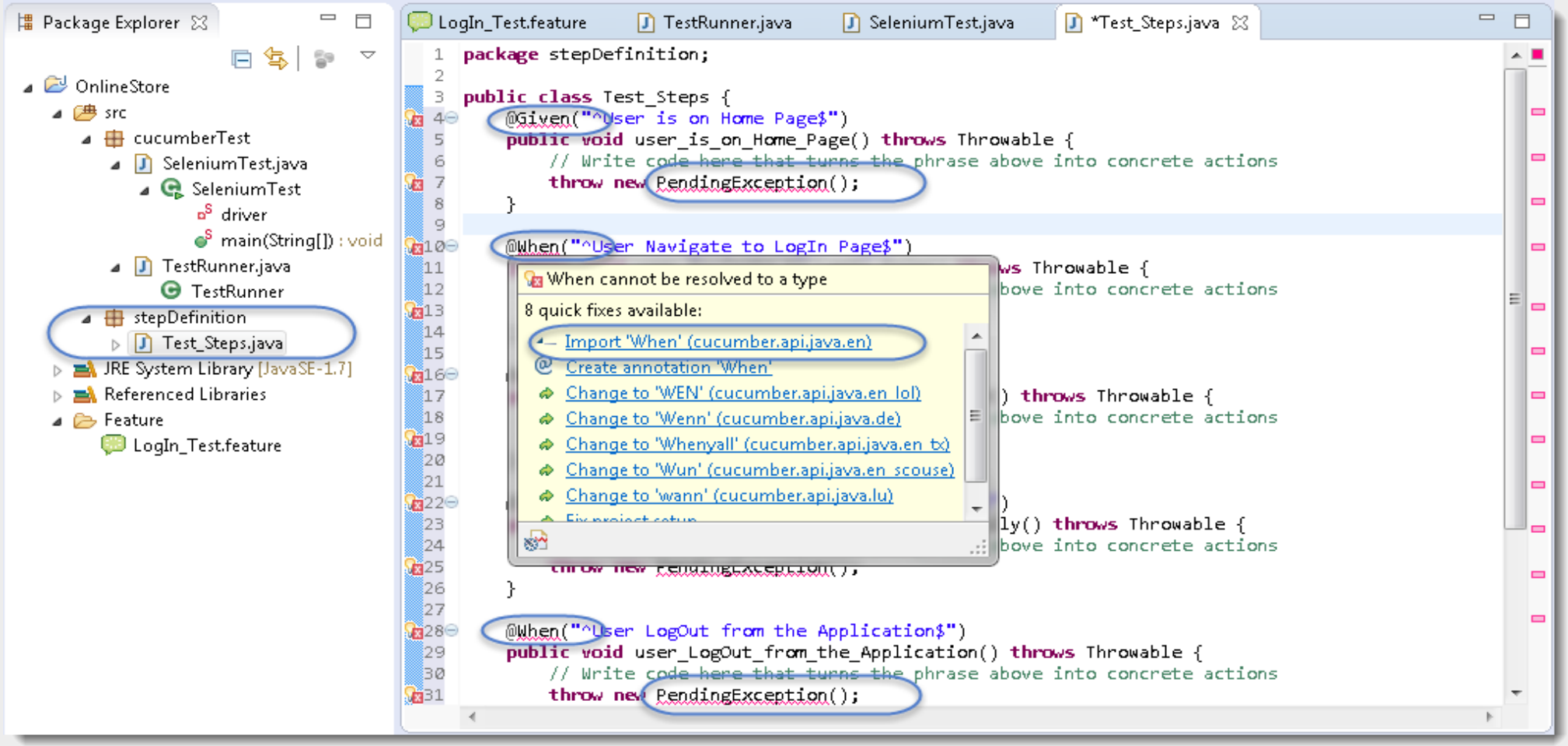
<terminated> TestRunner (1) [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe (31 Dec 2014 17:42:56)

```
2 Scenarios (0[33m2 undefined0[0m)
6 Steps (0[33m6 undefined0[0m)
0m0.000s

You can implement missing steps with the snippets below:

@Given("^User is on Home Page$")
public void user_is_on_Home_Page() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^User Navigate to LogIn Page$")
public void user_Navigate_to_LogIn_Page() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^User enters UserName and Password$")
public void user_enters_UserName_and_Password() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^Message displayed Login Successfully$")
public void message_displayed_Login_Successfully() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^User LogOut from the Application$")
public void user_LogOut_from_the_Application() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^Message displayed Logout Successfully$")
public void message_displayed_Logout_Successfully() throws Throwable {
```

Steps Definitions

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

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



Cucumber Options

- ***@CucumberOptions*** are like property file or settings for your test.
- Basically
- *@CucumberOptions* enables us to do all the things that we could have done if we have used cucumber command line.
- This is very helpful and of utmost importance if we are using IDE such eclipse only to execute our project.
- ***TestRunner Class***

```
package cucumberTest;

import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(
features = "Feature"
,glue={"stepDefinition"}
)

public class TestRunner {
}
```

Cucumber Options

- Following Main Options are available in Cucumber:

Options Type	Purpose	Default Value
dryRun	true: Checks if all the Steps have the Step Definition	false
features	set: The paths of the feature files	{}
glue	set: The paths of the step definition files	{}
tags	instruct: What tags in the features files should be executed	{}
monochrome	true: Display the console Output in much readable way	false
format	set: What all report formaters to use	false
strict	true: Will fail execution if there are undefined or pending steps	false

Cucumber Options

Dry Run

- ***dryRun*** option can either set as ***true*** or ***false***.
- If it is set as *true*, it means that
- *Cucumber* will only checks that every *Step*mentioned in the *Feature File* have corresponding code written in *Step Definition* file or not.
- So in case any of the function is missed in the *Step Definition* for any *Step* in *Feature File*, it will give us the message.

Monochrome

- This option can either set as ***true*** or ***false***.
- If it is set as *true*, it means that the *console output* for the *Cucumber test* are much more readable.
- And if it is set as *false*, then the *console output*is not as readable as it should be.

Cucumber Options

Features

- ***Features Options*** helps *Cucumber* to locate the *Feature file* in the project folder structure.
- All we need to do is to specify the folder path and *Cucumber* will automatically find all the `***.features***` extension files in the folder.
- It can be specified like:

  features = "Feature"
- *Or if the Feature file is in the deep folder structure*

  features = "src/test/features"

Glue

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

  glue = "stepDefinition"
- *Or if the Step Definition file is in the deep folder structure*

  glue = "src/test/stepDeinition"

Cucumber Options

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 colours 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"}***

Scenario Outline

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

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

Examples:

username	***password***
testuser_1	***Test@153***
testuser_2	***Test@153***

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

Scenario Outline

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

And User enters <username> and <password>

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

The complete code will look like this:

```
1   Feature: Login Action
2
3   Scenario Outline: Successful Login with Valid Credentials
4   Given User is on Home Page
5   When User Navigate to LogIn Page
6   And User enters "<username>" and "<password>"
7   Then Message displayed Login Successfully
8   Examples:
9      | username   | password |
10     | testuser_1 | Test@153 |
11     | testuser_2 | Test@153 |
```

Scenario Outline

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

```
package cucumberTest;

import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(
features = "Feature"
,glue={"stepDefinition"}
)

public class TestRunner {

}
```

4) There are no changes in ***Test_Steps*** file

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

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 amount of data.
- They are quite powerful but not the most intuitive as you either need to deal with a ***list of maps*** or a ***map of lists***.
- Most of the people gets confused with Data tables & Scenario outline, but these two works completely differently.

Difference between Scenario Outline & Data Table

Scenario Outline:

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

Test Data:

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

Data Tables in Cucumber

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

```
Scenario: Successful Login with Valid Credentials
Given User is on Home Page
When User Navigate to LogIn Page
And User enters Credentials to LogIn
    | testuser_1 | Test@153 |
Then Message displayed Login Successfully
```

- 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.

Data Tables in Cucumber

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

The implementation of the above step will belike **this**:

```
@When("^User enters Credentials to LogIn$")
public void user_enters_testuser__and_Test(DataTable usercredentials) throws Throwable {

//Write the code to handle Data Table
List<List<String>> data = usercredentials.raw();

//This is to get the first data of the set (First Row + First Column)
driver.findElement(By.id("log")).sendKeys(data.get(0).get(0));

//This is to get the first data of the set (First Row + Second Column)
    driver.findElement(By.id("pwd")).sendKeys(data.get(0).get(1));

    driver.findElement(By.id("login")).click();
}
```

Summary

In this lesson, you have learnt :

- Cucumber Framework
- How it Works
- Advantages of Cucumber
- Feature File
- Steps Definitions
- Cucumber Options
- *Scenario Outline*
- *Data Tables in Cucumber*

Review Question

Question 1: