**What is BDD (Behavior Driven Development) Testing?**

BDD (Behavior-driven development) Testing is a technique of agile software development and is as an extension of TDD, i.e., Test Driven Development. In BDD, test cases are written in a natural language that even non-programmers can read.

**TDD vs BDD:**

**TDD**:

In this technique, a QA engineer starts designing and writing test cases for every small functionality of an application. This technique attempts to answer a simple question – Is the code valid? The main intention of this technique is to modify or write a fresh code only when the test fails.

TDD is focused on testing smaller pieces of functionality in isolation

Only technical persons can understand TDD

**BDD**:

In BDD, tests are mainly based on systems behavior. This approach defines various ways to develop a feature based on its behavior. In most cases, the Given-When-Then approach is used for writing test cases.

BDD is designed to test an application’s behavior from the end user’s standpoint

Both technical & non-technical persons can understand BDD

**BDD Tools or Frameworks:**

The following all tools are support BDD Approach

* Cucumber (Ruby framework)
* Spec Flow (.NET framework)
* Behave (Python framework)

**What is cucumber**:

* Cucumber is a testing tool or framework that supports Behavior Driven Development (BDD).
* It offers a way to write tests that anybody can understand, regardless of their technical knowledge.
* In BDD, users (business analysts, product owners) first write scenarios or acceptance tests that describe the behavior of the system from the customer’s perspective, for review and sign-off by the product owners before developers write their codes.
* Cucumber framework uses Ruby programming language.
* Cucumber Framework supports other language as well beyond Ruby like Java, Scala, Groovy etc.
* Cucumber testing tool supports only web environment
* It is open source

**what is feature file in cucumber**

* This is a file where you will describe your tests/scenarios & requirements  in Descriptive language (Like Gherkin).
* A feature usually contains a list of scenarios to be tested for that feature.

**Difference between Feature, Scenario & Scenario outline?**

* Feature: A feature would describe the current test script which has to be executed.
* Scenario: Scenario describes the steps and expected outcome for a particular test case.
* Scenario Outline: Same scenario can be executed for multiple sets of data using scenario outline.

Gherkin keywords

1. Feature
2. Background
3. Scenario
4. Given
5. When
6. Then
7. And
8. But
9. ' \* '
10. Scenario Outline
11. Examples

**Feature**: Keyword

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

**Feature**: Login Action Test

Or

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

Or

**Feature**: Login Action Test

This feature will test a Login and Logout functionality

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

**Note**: Description is not a keyword of Gherkin.

Example :

***Feature File***

Feature: LogIn\_Feature

In order to access my account As a user of the website I want to log into the website

Scenario: Successful Login with Valid Credentials

Given User is at the Home Page

And Navigate to Login Page

When User enter Username and Password

And Click on the Login button

Then Successful Login message should display

Scenario: Successful Logout

When User Logout from the Application

Then Successful Logout message should display

**Background: Keyword**

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

* Navigate to Home Page
* Click on the Login link
* Enter Username and Password
* Click on Submit button

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

**Feature**: Add to Cart

This feature will test functionality of adding different products to the User basket from different flow

**Background**: User is Logged In

**Scenario**: Search a product and add the first result/product to the User basket

**Given** User searched for Lenovo Laptop

**When** Add the first laptop that appears in the search result to the basket

**Then** User basket should display with 1 item

**Scenario: Keyword**

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

**Scenario**: Search a product and add the first result/product to the User basket

Or

**Scenario**: Successful Login with Valid Credentials

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

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

**Given Keyword**

**Given** defines a precondition to the test. For e.g. In the shopping website, assume that the Login page link is only present on the Home Page, so the precondition for clicking the Login link is that the user is at the Home Page. If user is not at the Home Page, user would not be able to enter Username & Password. This precondition can be expressed in Gherkin like this:

**Scenario**: Successful Login with Valid Credentials

**Given** User is on Home Page

**When** User Navigate to Login Page

**When Keyword**

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

**Scenario**: Successful Login with Valid Credentials

**Given** User is on Home Page

**When** User Navigate to Login Page

Here user is performing some action using When keyword, clicking on the Login link. We can see that when defines the action taken by the user. It's the event that will cause the actual change in state of the application.

**Then Keyword**

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

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Successful Login with Valid Credentials       **Given** User is on Home Page

**When** User Navigate to Login Page

**And** User enters Username and Password

**Then** Message displayed Login Successfully

Here we can see that **Then** is the outcome of the steps above. The reader of this test would easily be able to relate to Then step and would understand that when the above conditions are fulfilled then the Then step will be executed.

**And Keyword**

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

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Successful Login with Valid Credentials

**Given** User is on Home Page

**When** User Navigate to Login Page

**And** User enters Username and Password

**Then** Message displayed Login Successfully

Or

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Successful Login with Valid Credentials

**Given** User is on Home Page

**And** Login Link displayed

**When** User Navigate to Login Page

**And** User enters Username and Password

**Then** Message displayed Login Successfully **And** Logout Link displayed

Here you would see that And is being used to add more details to the Given step, it's simply adding more conditions. We have just added three conditions. Use it when you have specified more than one condition. **And is used to add more conditions to Given, When and Then statements**.

**But Keyword**

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

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Unsuccessful Login with Invalid Credentials

**Given** User is on Home Page

**When** User Navigate to Login Page

**And** User enters Username and Password

**But** The user credentials are wrong

**Then** Message displayed Wrong Username & Password

Here you can see how adding **But** has helped define a negative test, in this test we will try to test failure conditions. Where a wrong credentials are a failure condition.

**\* Keyword**

This keyword is very special. This keyword defies the whole purpose of having Given, When, Then and all the other keywords. Basically Cucumber doesn't care about what Keyword you use to define test steps, all it cares about what code it needs to execute for each step. That code is called a **step definition** and we will discuss about it in the next section. At this time just remember that all the keywords can be replaced by the \* **keyword** and your test will just work fine. Let's see with example, we had this test earlier:

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Successful Login with Valid Credentials

**Given** User is on Home Page

**When** User Navigate to Login Page

**And** User enters Username and Password       Then Message displayed Login Successfully

**Using \* Keyword**

**Feature**: Login Action Test

Description: This feature will test a Login and Logout functionality

**Scenario**: Successful Login with Valid Credentials

\* User is on Home Page

\* User Navigate to Login Page

\* User enters Username and Password

\* Message displayed Login Successfully

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

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

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

**What is Step Definition?**

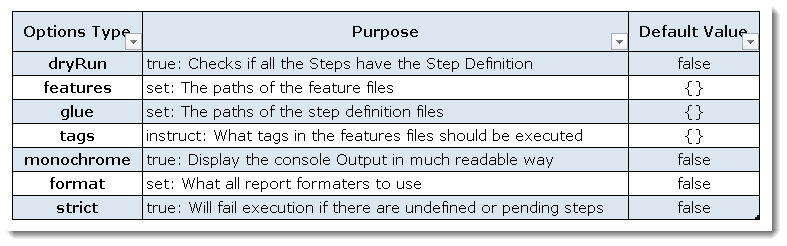
A Step Definition is a small piece of code with a pattern attached to it or in other words a Step Definition is a java method in a class with an annotation above it. An annotation followed by the pattern is used to link the Step Definition to all the matching Steps, and the code is what Cucumber will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

**What is Cucumber Options?**

**@CucumberOptions** are used to set some specific properties for the Cucumber test.

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

Following Main Options are available in Cucumber:



**Dry Run**

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

Graphical user interface, text, application

Description automatically generated

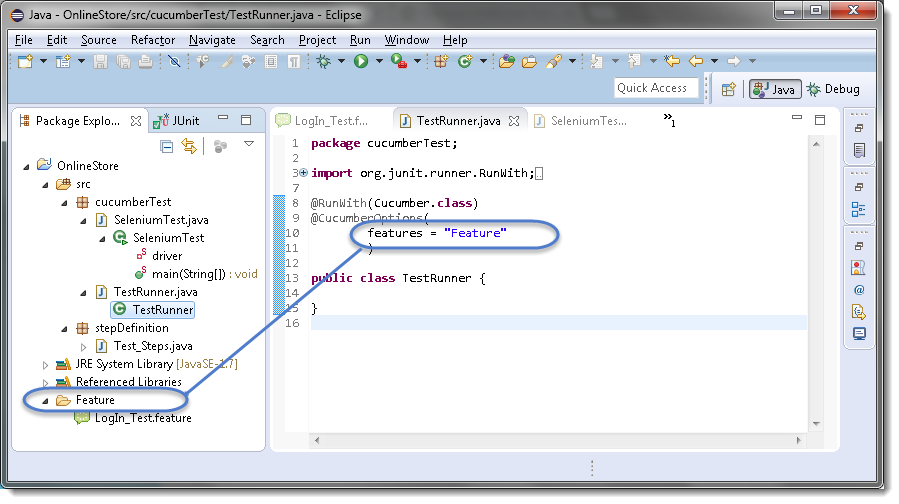
**Features**

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

**features = "Feature"**

Or if the Feature file is in the deep folder structure

**features = "src/test/features"**



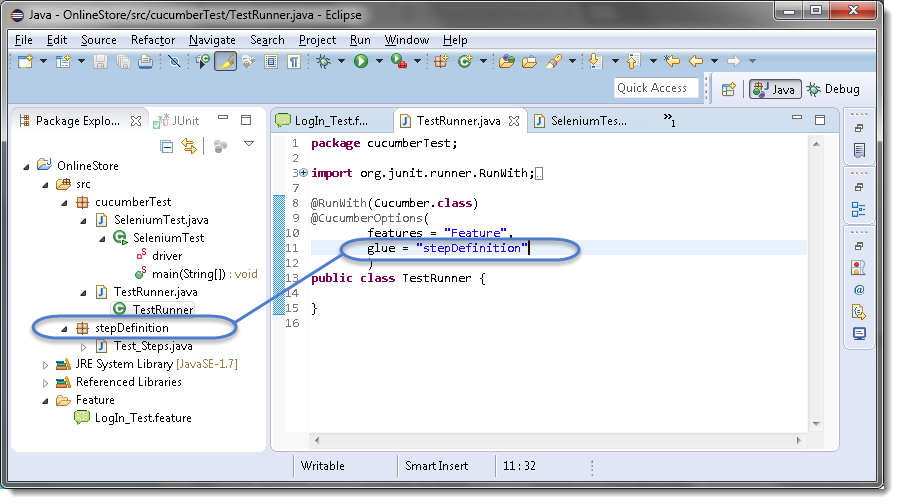
**Glue**

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

**glue = "stepDefinition"**

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

**glue = "src/test/stepDeinition"**



**Format**

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

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

**format = {"pretty"}**

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

**format = {"html:Folder\_Name"}**

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

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

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

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

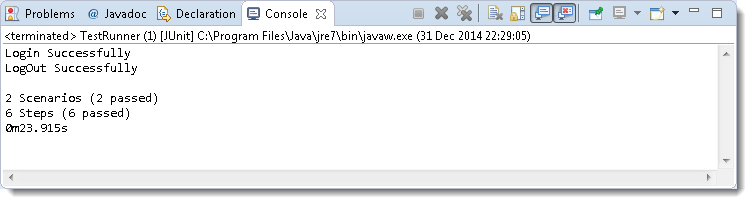
**Monochrome**

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

Graphical user interface, text, application

Description automatically generated

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



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



**Data Driven Testing in Cucumber:**

[**https://www.toolsqa.com/cucumber/data-driven-testing-in-cucumber/**](https://www.toolsqa.com/cucumber/data-driven-testing-in-cucumber/)

**Data Driven Testing Using Examples Keyword**

[**https://www.toolsqa.com/cucumber/data-driven-testing-using-examples-keyword/**](https://www.toolsqa.com/cucumber/data-driven-testing-using-examples-keyword/)

**Difference between Scenario Outline & Data Table**

[**https://www.toolsqa.com/cucumber/data-tables-in-cucumber/**](https://www.toolsqa.com/cucumber/data-tables-in-cucumber/)

**Data Tables in Cucumber**

[**https://www.toolsqa.com/cucumber/data-tables-in-cucumber/**](https://www.toolsqa.com/cucumber/data-tables-in-cucumber/)

**Maps in Data Tables**

[**https://www.toolsqa.com/cucumber/maps-in-data-tables/**](https://www.toolsqa.com/cucumber/maps-in-data-tables/)

**Cucumber Hooks:**

1. **Cucumber Tags**

[**https://www.toolsqa.com/cucumber/cucumber-tags/**](https://www.toolsqa.com/cucumber/cucumber-tags/)

1. **Cucumber Hooks**

[**https://www.toolsqa.com/cucumber/cucumber-hooks/**](https://www.toolsqa.com/cucumber/cucumber-hooks/)

1. **Tagged Hooks in Cucumber**

[**https://www.toolsqa.com/cucumber/tagged-hooks-in-cucumber/**](https://www.toolsqa.com/cucumber/tagged-hooks-in-cucumber/)

1. **Execution Order of Hooks**

[**https://www.toolsqa.com/cucumber/execution-order-hooks/**](https://www.toolsqa.com/cucumber/execution-order-hooks/)

1. **Background in Cucumber**

[**https://www.toolsqa.com/cucumber/background-in-cucumber/**](https://www.toolsqa.com/cucumber/background-in-cucumber/)

Parallel Execution & Reports :

<https://youtu.be/jRihi74zJFw>

<https://qaautomation.expert/2021/05/05/parallel-testing-in-cucumber-with-junit/>