Behavior Driven Development

* Behavior Driven Development is derived from Test Driven Development as a software development methodology.
* As a part of the Test Driven Development approach, the developer creates tests as a part of the acceptance criteria first. The developer will make sure the test is passed and will commit changes in the code, if required.
* Test-driven development is a way to ensure that the system meets its requirements by providing 100% test coverage.
* BDD was introduced to reduce the time to test, less code with more collaboration.

Behavior-driven development (BDD) is an agile software development methodology that prioritizes collaboration among developers, testers, and non-technical stakeholders to ensure customer-centric outcomes. In BDD, requirements are defined in terms of desired behaviors and user stories, enabling clearer communication and a focus on delivering business value.

# Advantages of BDD

1. Collaboration within the team
2. Clear requirements
3. Focus on behavior
4. Automation testing
5. Early outcome
6. Better code quality

# BDD tools and framework

1. Cucumber
2. SpecFlow
3. Behave
4. JBehave
5. RSpec
6. Robot Framework
7. Karate
8. Gauge

# What is BDD testing?

Behavior-Driven Development (BDD) testing is a software development approach that focuses on the system's behavior from the end user’s perspective. It ensures that the software meets both business requirements and user expectations.

# Rules of BDD

## Rule 1 – Gherkins’ golden rule

Treat other readers as you would want to be treated. Write feature files that are easy to understand.

Example:

Scenario: Buy cookies

Given: I want cookies

When: I buy cookies

Then: I get them

This scenario lacks clarity and specifications

Scenario: Adding Chocolate cookies to cart

Given: I am on the ByCookiesOnline home page

When: I search for “Chocolate Cookies”

And: I click on Add to Cart

Then: The cookies are added to my cart

## Rule 2 - The cardinal rule of BDD

The cardinal rule of BDD is a one-to-one rule: One scenario should cover exactly one single, independent behaviour.

By focusing on one behaviour at a time,

1. Causes less confusion
2. Easy to identify test failures
3. Efficiency and faster test cycles

# Overview of Cucumber

The cucumber tool is based on the Behaviour Driven Development framework that acts as the bridge between the following people:

1. Software Engineer and Business Analyst.
2. Manual Tester and Automation Tester.
3. Manual Tester and Developers.

Cucumber makes it easy for anyone to understand the application code. Cucumber uses Gherkin which is readable and understandable.

# How to Install Cucumber

Steps to install the Cucumber plugin in Eclipse:

* In Eclipse, click on Help 🡪 Install New Software
* Click on Add button. Enter Name as “Cucumber” and give the below URL
* [**https://cucumber.github.io/cucumber-eclipse-update-site-snapshot/**](https://cucumber.github.io/cucumber-eclipse-update-site-snapshot/)
* Select the cucumber check box and click on Next until finish.
* Cucumber will be successfully installed.

# Features and Step Definitions

Cucumber framework has three major parts

* **Feature file**

Behavior of the application is given in feature file in plain English text with help of Gherkin Language. Gherkin is used to define executable steps with the help of keywords. Keywords which hold a specific meaning and would help and defined the meaning of a scenario. Sample keywords involved are GIVEN, WHEN, THEN & AND where

* GIVEN is a precondition of the application
* WHEN Trigger point for the scenario more like steps to be executed.
* THEN is the outcome more like expected Result
* AND bridge between two statements.
* **Steps and Step Definitions**

Step definitions file would be consisting of a code which defines the steps for annotations created in feature file.

* **Test Runner file.**

Test Runner file will have all the metadata information required for test execution. It makes use of ‘@RunWith()’ annotation from JUnit framework for execution. Secondarily you would be making use of ‘@CucumberOptions’ annotation to define the location of feature files, step definitions location through glue, what reporting integrations to use etc.

# Building Scripts using Gerkins(updating the Feature File)

Gerkins language uses the following keywords:

GIVEN

WHEN

THEN

AND

Below is a sample Feature file for validating Login functionality

**Feature**: Walmart Login

Verify user is able to Login to Walmart

**Scenario**: Login with correct username and password

Given the Login Page

When Correct UserName and Password is entered

And Click on Sign In Button

Then Login is successful

**Scenario**: Login with Incorrect username and password

Given the Login Page

When Incorrect UserName and Password is entered

And Click on Sign In Button

Then Application throws correct error message.

## Commonly used Gherkin commands

 1. Feature.

2. Scenario outline.

3. Given, When, Then, And (Steps)

4. Background.

5. Scenario

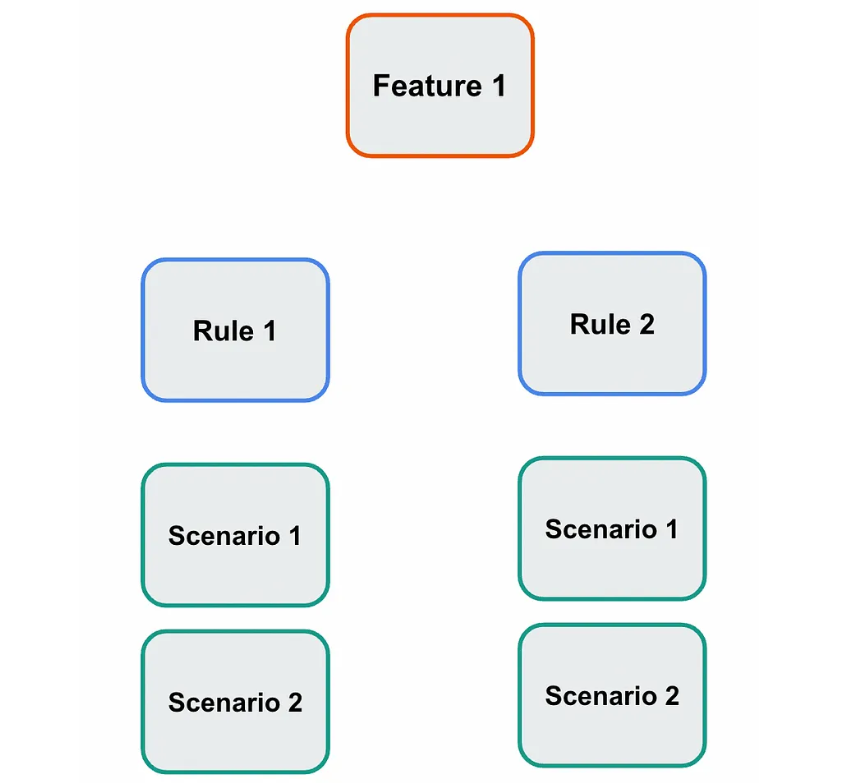
6. Rule

7. Example

### Rule

keyword **Rule** on gherkins introduced in **Cucumber version 6.0.0** which allows users to group related test scenarios or provide restrictions on the scope of the story. This will help organize and manage sets of scenarios by grouping them based on common characteristics or behaviours.

Structure using rule:



Example:

Let us say there is phone number field in user registration page which accepts numbers from 8 – 12 digits.

Now this a rule that can have multiple scenarios

Feature: User Registration

Rule: Phone number must be between 8 -12 digits

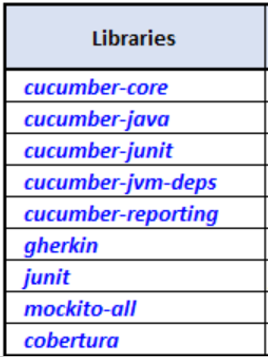
Scenario 1: Ensure phone number of 10 digits is accepted

Scenario 2: Ensure error is thrown when phone number of 7 digit is entered

Scenario 3: Ensure error is thrown when phone number of 13 digit is entered

**Using Cucumber and Maven together**

1. Create a Maven Project
2. Add the following dependencies in pom.xml file (copy from [www.mvnrepository.com](http://www.mvnrepository.com))



1. Right click on the project 🡪 select Maven 🡪 Update Project
2. Create a Feature file:

Create a package with the name “feature”

Right click on the create package 🡪 New 🡪 File 🡪 Give name as “test.feature”

1. Create a package with the name SeleniumGlueCode and a new step definition file (normal java class)
2. To run the code, we need to create a jUnit runner class. Right click on “runner” package 🡪 new 🡪 Class 🡪 TestRunner

Sample Dependecy:

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.18.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/junit/junit -->

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>7.15.0</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-core -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-core</artifactId>

<version>7.17.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-jvm-deps -->

<dependency>

<groupId>io.cucumber</groupId>

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

<version>1.0.6</version>

<scope>provided</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/net.masterthought/cucumber-reporting -->

<dependency>

<groupId>net.masterthought</groupId>

<artifactId>cucumber-reporting</artifactId>

<version>5.7.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/io.cucumber/gherkin -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>gherkin</artifactId>

<version>28.0.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/net.sourceforge.cobertura/cobertura -->

</dependencies>

## Sample Feature, step definition and runner

Feature: Login

Verify user is able to Login to eComm application

@tag1

Scenario: Login as an authenticated User

Given Login page of the application

When user enters correct UserName and password

And clicks on Sign In Button

Then Account home page is displayed

-------------------Step Def---------------------

WebDriver driver;

@Given("Login page of the application")

**public** **void** login\_page\_of\_the\_application() {

driver = **new** ChromeDriver();

driver.get("https://magento.softwaretestingboard.com/customer/account/login");

driver.manage().timeouts().implicitlyWait(Duration.*ofSeconds*(10));

}

@When("user enters correct UserName and password")

**public** **void** user\_enters\_correct\_user\_name\_and\_password() {

driver.findElement(By.*id*("email")).sendKeys("peterparker123@yopmail.com");

driver.findElement(By.*id*("pass")).sendKeys("Abcd@123");

}

@When("clicks on Sign In Button")

**public** **void** clicks\_on\_sign\_in\_button() {

driver.findElement(By.*name*("send")).click();

}

@Then("Account home page is displayed")

**public** **void** account\_summary\_page\_is\_displayed() {

String a = driver.findElement(By.*xpath*("//strong[text()='Account Information']")).getText();

Assert.*assertEquals*(a, "Account Information");

}

------------------Runner class--------------------------------

@RunWith(Cucumber.**class**)

@CucumberOptions(features="src/test/java/features",glue={"stepDefinations"})

**public** **class** TestRunner {

}

# Capturing Arguments

https://demowebshop.tricentis.com/register

User can send arguments through scenario steps.

Say for Example, for user registration, if the step needs to be parameterized for **Gender**

Feature file will look like this:

@tag2

Scenario: Verify user registration

Given I open the application url

when I click on Register link

And I select the gender as "Female"

And I enter all other mandatory fields

And I click on Register button

Then Registration must be successful

Step Definition:

@When("I select the gender as {string}")

**public** **void** selectGender(String gender) {

}

# **Background Feature**

The background feature can be used when you have to execute certain steps before executing each scenario when multiple scenarios are in the feature file.

Feature: Login

Verify user is able to Login to eComm application

Background: Launch application

Given I open Chrome Browser

And I navigate to application URL

@tag1

Scenario: Login as an authenticated User

Given I enter correct UserName as "peterparker123@yopmail.com" and password as "Abcd@123"

And I click on Sign In Button

Then Account home page is displayed

# Using Scenario Outline

Scenario outlines are used to run the code with multiple test data for multiple iterations

Multiple rows of test data are given in the Examples block of feature File. Below is a sample Feature file with Scenario Outline

Scenario Outline: Verify Search Functionality

Given I enter correct UserName as <userName> and password as <password>

And I click on Sign In Button

Then Account home page is displayed

When I enter <product> in the search box

And I click on search icon

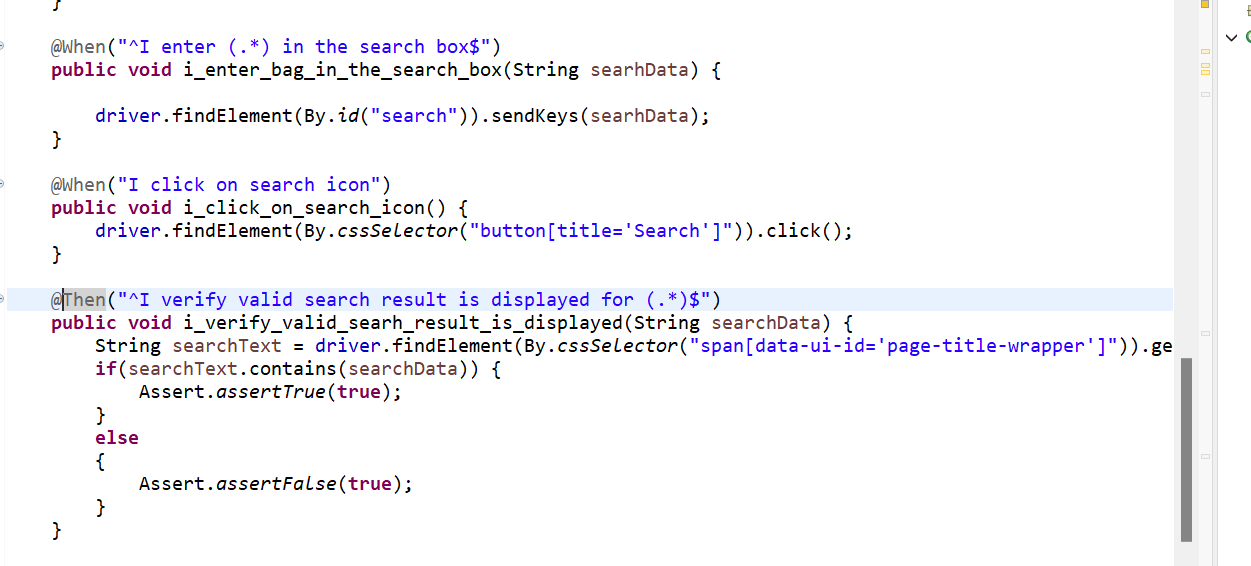
Then I verify valid search result is displayed

Examples:

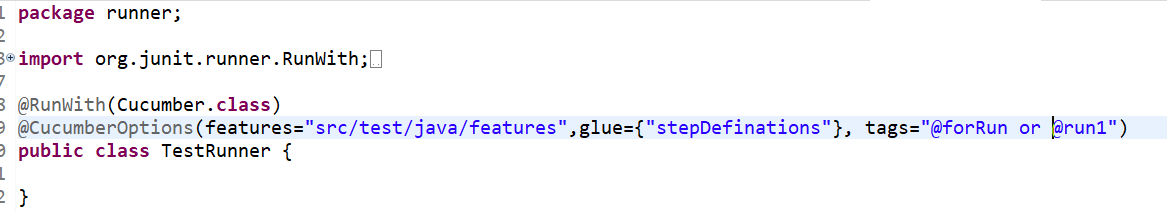
|userName | password | product |

|peterparker123@yopmail.com | Abcd@123 | Bag |

|peterparker123@yopmail.com | Abcd@123 | Jackets |



## Test Runner with Tags



# DocStrings in Cucumber

Docstrings allow you to specify a larger amount of text that could not fit on a single line. It parses big data as one piece.

This is a helpful solution when we have plenty of text to enter in multiple lines. For example, if you need to represent the exact content of an email or contact us, you could use Doc String.

DocString should be written within pair of triple quotes (""") or using three backticks (```).

Say, we want to type the below text in the application:

Hello,

How are you! This text will be entered

in the same format.

Thanks!

John

For this docstring can be used.

-----------------------------Feature file-----------------------

@f2

Scenario: Type email

Given I open the application URL

When I enter the below message

"""

Hello,

How are you! This text will be entered

in the same format.

Cheers,

John

"""

Then I validate above message is displayed

"""

Hello,

How are you! This text will be entered

in the same format.

Cheers,

John

"""

---------------------------------Step Defination------------------------

@Given("I open the application URL")

**public** **void** i\_open\_the\_application\_url() {

driver = **new** ChromeDriver();

driver.get("http://autopract.com/selenium/form4/");

}

@When("I enter the below message")

**public** **void** i\_enter\_the\_below\_message(String docString) {

driver.findElement(By.*cssSelector*("#Message")).sendKeys(docString);

}

@Then("I validate above message is displayed")

**public** **void** i\_validate\_above\_message\_is\_displayed(String docString) {

String s = driver.findElement(By.*cssSelector*("#Message")).getAttribute("value");

Assert.*assertEquals*(s, docString);

}

# Data tables in cucumber

*DataTables* are also used to handle large amounts of data. Data Table is a data structure provided by Cucumber. It is a bridge which passes values from the feature files to the parameters of the Step Definitions. It can handle large amounts of data and can be passed as one-dimensional, two-dimensional data and in the form of key-value pair.

**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 needs 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

Let's understand the data table with an instance of a registration form, which is a web application. For this registration form, we can create a [feature file](https://www.javatpoint.com/feature-file-in-cucumber-testing), and later we will create a feature file with data table.

The registration form contains the following parameters:

* First Name
* Last Name
* Email
* Password
* Confirm Password

**By using the normal method:**

@tag

Feature: New user registration.

Scenario: Verification of successful registration when the inputs are correct.

Given user on the user registration page

When user enters a valid user name

And valid e-mail address

And valid password

And valid confirmation password

And valid Birth-date

And valid Gender

And valid phone number

Then user registration should be successful

**Replacing with data tables**

Given the user on the user registration page.

When user enter valid data on the page

| Fields | Values |

| First Name | User Name |

| Last Name | User Last Name |

| Email Address | someone@gmail.com |

| Password | abcd@123 |

| ConfirmPassword | abcd@123 |

Then the user registration should be successful.

-------------------Step definition------------------------

WebDriver driver;

@Given("I am on the user registration page")

**public** **void** the\_user\_on\_the\_user\_registration\_page() {

driver = **new** ChromeDriver();

driver.get("https://magento.softwaretestingboard.com/customer/account/create/");

}

@When("I enter valid data on the page")

**public** **void** user\_enter\_valid\_data\_on\_the\_page(DataTable dataTable) {

//Initialize data table

List<List<String>> data = dataTable.asLists();

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*("email")).sendKeys(data.get(3).get(1));

driver.findElement(By.*name*("password")).sendKeys(data.get(4).get(1));

driver.findElement(By.*name*("password\_confirmation")).sendKeys(data.get(5).get(1));

driver.findElement(By.*cssSelector*("button[title='Create an Account']")).click();

}

@Then("The user registration should be successful")

**public** **void** the\_user\_registration\_should\_be\_successful() {

System.***out***.println("User registration success");

}

# Transformer in cucumber

# Grouping features in sub folder

Features can be grouped in subfolders and paths for the same can be provided. It will run all the files under the folder mentioned in the test runner class.

# Filtering Scenarios

Scenarios can be filtered based on the tags.

By adding tags to the scenario and mentioning the tags in the runner file, it will pick the scenarios with the tag and run them.

‘And’ or ‘or’ keywords can also be used for filtering.

We can also run specific scenarios by giving the link number of the scenario we want to run.

Example:

@RunWith(Cucumber.**class**)

@CucumberOptions(features="src/test/java/features/test1.feature:9",glue={"stepDefinations"})

**public** **class** TestRunner {

}

Where ‘9’ is the line number of the scenario that should be run.

# Cucumber Hooks

Cucumber supports ***hooks***, which are blocks of code that run ***before*** or ***after*** each scenario. You can define them anywhere in your project or step definition layers, using the methods ***@Before*** and ***@After***.

Cucumber **hook** allows us to better manage the code workflow and helps us to reduce the code redundancy.

Hooks are mainly used to perform precondition steps.

Example:

* Starting a webdriver
* Setting up DB connections
* Setting up test data
* Navigating to certain page
* or anything before the test

In the same way, there are always after steps as well of the tests like:

* Killing the webdriver
* Closing DB connections
* Clearing the test data
* Clearing browser cookies
* Logging out from the application
* Printing reports or logs
* Taking screenshots on error
* or anything after the test

Example:

**public** **class** Hooks {

**static** WebDriver *driver*;

@Before

**public** **void** open\_browser() {

*driver* = **new** ChromeDriver();

}

@After

**public** **void** close\_browser() {

*driver*.close();

}

}

# Setting up profiles

Cucumber profiles are not available on Cucumber-JVM. However, it is possible to set configuration options using [Maven profiles](https://maven.apache.org/guides/introduction/introduction-to-profiles.html).

Pom.xml profile

<profiles>

<profile>

<id>dev</id>

<properties>

<base.url>https://www.google.com</base.url>

</properties>

</profile>

<profile>

<id>qa</id>

<properties>

<base.url>https://magento.softwaretestingboard.com/customer/account/create/</base.url>

</properties>

</profile>

</profiles>

-----------------------build plugin management------------------

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M4</version>

<configuration>

<systemPropertyVariables>

<base.url>${base.url}</base.url>

</systemPropertyVariables>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

-------------------------In step definition-----------------------

String s = System.*getProperty*("base.url");

Hooks.*driver*.get(s);

# Cucumber – Maven – TestNG – Selenium Integration using Page object modeling

* 1. Create page classes in Main package
  2. Create features in test/resources folder
  3. Create step definition package for the steps
  4. Create runner class for running testNG

Below is the sample runner class:

**package** runner;

**import** io.cucumber.testng.AbstractTestNGCucumberTests;

**import** io.cucumber.testng.CucumberOptions;

@CucumberOptions(tags = "", features = {"src/test/java/features"}, glue = {"stepDefinations"})

**public** **class** Runner **extends** AbstractTestNGCucumberTests {

}

# Reports

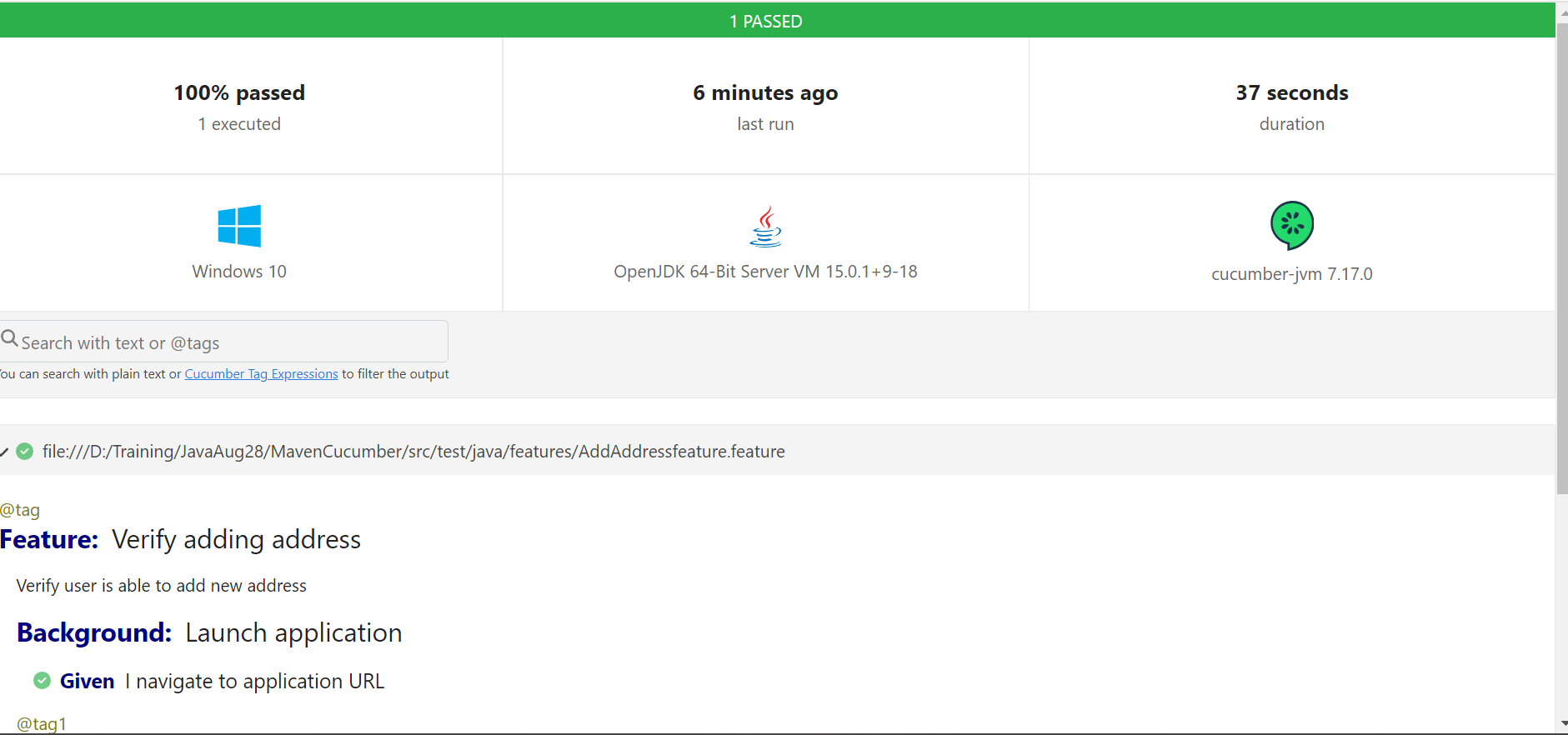
## Generating HTML Report

1. In the testrunner.java class, add a plugin inside @CucumberOptions to

plugin = { "pretty", "html:target/cucumber-reports/dsalgo.html"}, dryRun=**false**,

monochrome = **true**

1. Now save the Runner.java class and execute it. On execution, you will see that the folder htmlreports is created inside the target folder.
2. Access the folder and look for the index.html file, which contains the test results in HTML format.



# Using Extent Report

Step 1 - To generate Extent Reports, we need to add the below dependencies from [www.mvnreporsitories.com](http://www.mvnreporsitories.com) to POM.xml file

<dependency>

<groupId>tech.grasshopper</groupId>

<artifactId>extentreports-cucumber7-adapter</artifactId>

<version>1.14.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.aventstack/extentreports -->

<dependency>

<groupId>com.aventstack</groupId>

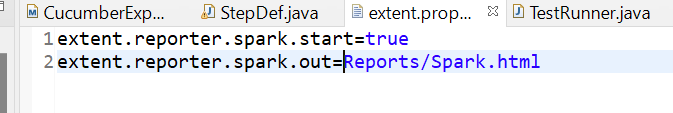
<artifactId>extentreports</artifactId>

<version>5.1.2</version>

</dependency>

Step 2 - ***Create extent.properties file in src/test/resources***

We need to create the extent.properties file at the ***src/test/resources*** folder for the grasshopper extent report adapter to recognize it.

****

**Step 3 – add the below plugin in runner file**

@RunWith(Cucumber.**class**)

@CucumberOptions(features="src/test/java/features/feature2.feature",glue={"stepDefinations"}, plugin = {"com.aventstack.extentreports.cucumber.adapter.ExtentCucumberAdapter:"})

**public** **class** TestRunner {

}

# Maven Report

Reports can also be generated using Maven Surefile Plugin. We need to add below plugins for generating a surefire report.

maven-surefire-report-plugin

maven-site-plugin

org.apache.maven.plugins

These should be added in **reporting** tag for them to work.

<reporting>

<plugins>

<!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-surefire-report-plugin -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-report-plugin</artifactId>

<version>3.5.0</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-site-plugin</artifactId>

<version>3.12.1</version>

</plugin>

<plugin>

<!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-project-info-reports-plugin -->

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-project-info-reports-plugin</artifactId>

<version>3.6.1</version>

</plugin>

</plugins>

</reporting>

Mvn site

Note – Runner class should have contain the word ‘test’

In order to generate the report, we have to run the below command in Maven 🡪 build

Surefire-report:report

And then run as maven test