BDD with Cucumber – User Handbook

Learn Automation Testing from Scratch

BDD with Cucumber – User Guide

Trainer – Haradhan Pal

Agenda

- Introduction to TDD and BDD
- Cucumber Introduction
- Advantages of Cucumber Over Other Tools
- Prerequisite required for using Cucumber with Selenium
- Developing one simple Cucumber Framework from Scratch
- Add Code inside Step Definitions Class
- Create Multiple Scenarios
- Implements undefined Steps
- Add Background
- Create Multiple Feature Files
- Create Multiple Scenarios
- Scenario Outline
- Adding Examples
- Cucumber Tags
- Execute Scenarios with Tags
- Exclude Scenarios with Tags
- OR & AND Tags in Cucumber
- Cucumber Hooks
- Tagged Hooks In Cucumber
- Setting Order or Priority of Cucumber Hooks
- Cucumber Report, HTML Report, JSON Report, XML Report
- Extent Report
- Steps to Generate Extent Report using Cucumber
- Maintain Backup of All Generated Extent Reports

TDD Overview

TDD (Test-Driven Development) is an iterative development process. Each iteration starts with a set of tests written for a new piece of functionality. These tests are supposed to fail during the start of iteration as there will be no application code corresponding to the tests. In the next phase of the iteration, Application code is written with an intention to pass all the tests written earlier in the iteration. Once the application code is ready tests are run.

This helps us in many ways:

- User write the application code based on the tests. This gives a test first environment for development and the generated application code turns out to be bug-free.
- With each iteration, user write tests and as a result, with each iteration, user get an automated regression pack. This turns out to be very helpful because with every iteration user can be sure that earlier features are working.
- These tests serve as documentation of application behavior and reference for future iterations.

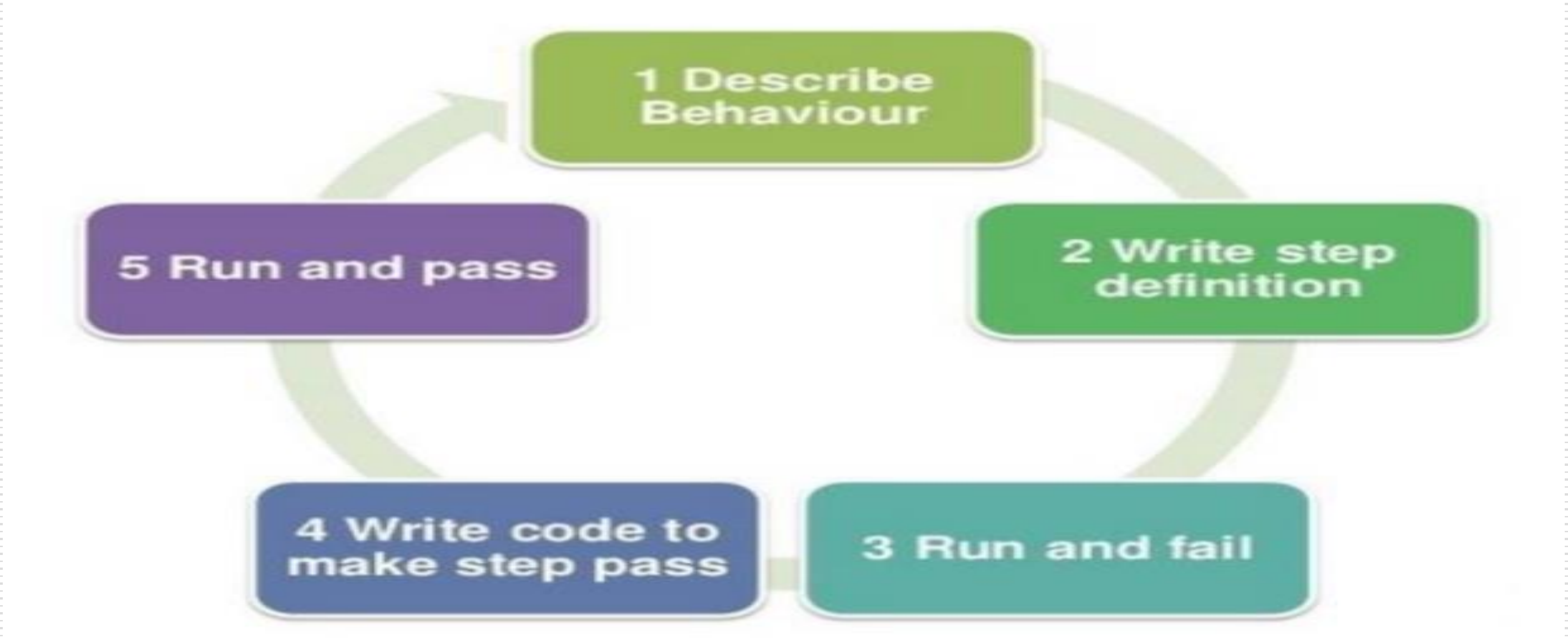
Behavior Driven Development

Behavior Driven testing is an extension of TDD. Like in TDD in BDD also user write tests first and the add application code. BDD gives user 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.

The major difference between TDD and BDD are:

- Tests are written in plain descriptive English type grammar
- Tests are explained as behavior of application and are more user-focused
- Using examples to clarify requirements
- This difference brings in the need to have a language that can define, in an understandable format.

Features of BDD



- Shifting from thinking in "tests" to thinking in "behavior"
- Collaboration between Business stakeholders, Business Analysts, QA Team and developers
- Driven by Business Value
- Extends Test-Driven Development (TDD) by utilizing natural language that non-technical stakeholders can understand
- BDD frameworks such as Cucumber or JBehave are an enabler, acting a "bridge" between Business & Technical Language
- BDD is popular and can be utilized for Unit level test cases and for UI level test cases.
- The scenarios are written based on the expected behavior of the software, and it is tested to check if it matches said scenarios.
- These scenarios are documented using a Domain Specific Language such as Gherkin. In each test scenario, natural language constructs constituting small English-like phrases are used to describe the behavior and expected outcome of an application. This is done using a dedicated software tool like Cucumber, that allows the execution of automated acceptance tests written in Gherkin.

Introduction to Cucumber

- A cucumber is a tool based on Behavior Driven Development (BDD) framework which is used to write acceptance tests for the web application. It allows automation of functional validation in easily readable and understandable format (like plain English) to Business Analysts, Developers, Testers, etc.
- Cucumber feature files can serve as a good document for all. There are many other tools like JBehave which also support BDD framework. Initially, Cucumber was implemented in Ruby and then extended to Java framework. Both the tools support native JUnit.
- Cucumber reads the code written in plain English text (Gherkin Language) in the feature file
- It finds the exact match of each step in the step definition (a code file).
- 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.

Prerequisite required for using Cucumber

User need the following items before starting cucumber:

Step 1: Download and install the Java platform on user machine

Step 2: Download and install Eclipse IDE

Step 3: Download Cucumber Eclipse Plugin:

a. In the eclipse, navigate to Help > Install New Software. Copy the URL "http://cucumber.github.io/cucumber-eclipse/update-site/" and press Enter.

b. User would see a checkbox named "Cucumber Eclipse Plugin", Select the checkbox 'Cucumber Eclipse Plugin'.

c. Click 'Next'

d. Again click 'Next' and Accept the license terms.

e. Click Finish

f. Click 'Install anyway'

g. Click 'Restart Now'

Step 4: Create a Maven Project in Eclipse

Step 5: Open the pom.xml file in eclipse and the below dependency after navigating to Maven Repository "https://mvnrepository.com/"

a. cucumber-java

b. cucumber-core

c. cucumber-junit

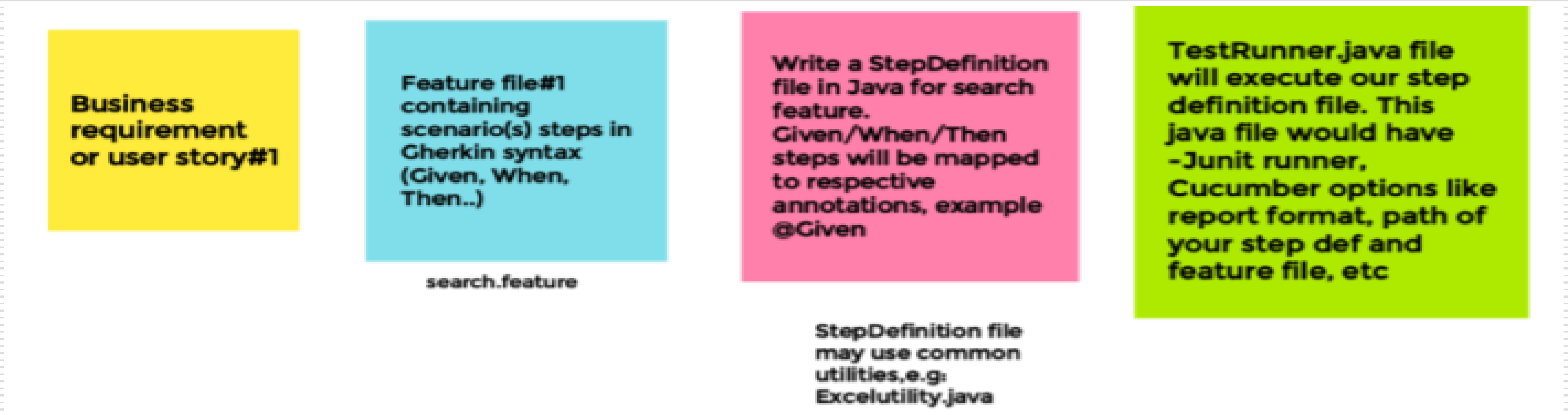
d. cucumber-picocontainer

e. cucumber-gherkin

Note: io.cucumber is the new version and info.cukes are the older version. Wherever user can see both io.cucumber and info.cukes, they need to go with the version io.cucumber as its artifact id.

Cucumber Framework

Cucumber framework mainly consists of three major parts – Feature File, Step Definitions, and the Test Runner File.


Business requirement or user story#1
Feature file#1 containing scenario(s) steps in Gherkin syntax (Given, When, Then..)
search.feature
Write a StepDefinition file in Java for search feature. Given/When/Then steps will be mapped to respective annotations, example @Given
StepDefinition file may use common utilities,e.g: Excelutility.java
TestRunner.java file will execute our step definition file. This java file would have -Junit runner, Cucumber options like report format, path of your step def and feature file, etc

1. Feature File

A standalone unit or a single functionality (such as a login) for a project can be called a Feature. Each of these features will have scenarios that must be tested using Selenium integrated with Cucumber. A file that stores data about features, their descriptions, and the scenarios to be tested, is called a Feature File.

Cucumber tests are written in these Feature Files that are stored with the extension – ".feature". A Feature File can be given a description to make the documentation more legible.

Example:

The Login function on a website

Feature File Name: userLogin.feature

Description: The user shall be able to login upon entering the correct username and password in the correct fields. The user should be directed to the homepage if the username and password entered are correct.

Keywords such as GIVEN, WHEN, and THEN used to write the test in Cucumber are called Annotations.

GIVEN user navigates to login page by opening Firefox

WHEN user enters correct <username> AND <password> values

THEN user is directed to the homepage

Cucumber Framework – Contd.

2. Step Definitions

Now that the features are written in the feature files, the code for the related scenario has to be run. To know which batch of code needs to be run for a given scenario, Steps Definitions come into the picture. A Steps Definitions file stores the mapping data between each step of a scenario defined in the feature file and the code to be executed.

Step Definitions can use both Java and Selenium commands for the Java functions written to map a feature file to the code.

Example:

```
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
public class Steps
{
@Given("^user navigates to the login page by opening Firefox$")
//Code to Open Firefox Browser and launch the login page of application to define the GIVEN step of the feature
@When("^user enters correct username and password values$")
//take inputs for username and password fields using find element by xpath. Put the correct username and password values as inputs to define the WHEN step of the feature
@Then ("^user gets directed to homepage$”)
//Direct to the Homepage of the application as a result of correct username and password inputs in the WHEN step. This would define the THEN step of the feature
```

3. Test Runner File

To run the test, user needs a Test Runner File, which is a JUnit Test Runner Class containing the Step Definition location and the other primary metadata required to run the test.

The Test Runner File uses the @RunWith() Annotation from JUnit for executing tests. It also uses the @CucumberOptions Annotation to define the location of feature files, step definitions, reporting integrations, etc.

Background in Cucumber

- Most of the time user will find that several scenarios in the same feature start with a common context. Cucumber provides a mechanism for this, by providing a Background keyword, where user can specify steps that should be run before each scenario in the feature.

- A Background is much like a scenario containing a number of steps. But it runs before each and every scenario were for a feature in which it is defined.

- For example, to purchase a product on any E-Commerce website, user need to do the following steps:

  1. Navigate to Login Page
  2. Submit UserName and Password

- After these steps only user will be able to add a product to cart for checkout and able to perform the payment. Hence, login part might be common across all scenarios. So instead of writing them again and again for all tests, user can move it under the Background keyword.

Scenario Outline & Examples in Cucumber

- Scenario includes all the possible circumstances of the feature and test scripts for these circumstances. The keyword "Scenario" represents a scenario in Gherkin language. One feature can have multiple scenarios, and each scenario consists of one or more steps.
- Cucumber Scenario Outline in Gherkin Based on Gherkin Reference, the Scenario Outline keyword can be used to repeat the same steps with different values or arguments being passed to the step definitions. This is helpful if user want to test multiple arguments in the same scenario.
- When to use scenario outlines user want their scenario outlines to describe behaviors or examples that are important to the business, not detail every boundary or test case. Adding too many rows simply creates more work and obscures where the value is to the business.
- All scenario outlines are followed by Examples part that contains the set of data that has to be passed during the execution of tests. Pipe symbol (|) is used to specify one or more parameter values in a feature file.
- In a single execution, Scenario is executed only once whereas Scenario outline (For similar data trace) can be executed multiple times depending upon the data provided as Example.

Tags in Cucumber

- It might look very simple when user 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, user may have 10, 20, or may be a greater number of scenarios in a single feature file. They may represent different purpose (Smoke/Integration/End to End/Regression test), different prospective (Developer/QA/BA), different status (Ready for execution/Work in progress/defect fixes), etc.
- Cucumber has already provided a way to organize user scenario execution by using tags in feature file. User can define each scenario with a useful tag. Later, in the runner file, they can decide which specific tag (and so as the scenario(s)) user want Cucumber to execute. Tag starts with "@". After "@" user can have any relevant text to define their tag.
- Suppose, there are two or more scenarios in a feature file. User 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.
- There is no limit in defining tags within the feature file. Based on user need, they can derive tags to be used and scenarios to be executed.
- Tag can also be defined at a feature level. Once user 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, user can use more than one tag for the single feature. Whenever Cucumber finds an appropriate call, a specific scenario will be executed.
- For excluding tag, user can use "not " in JUnit runner class to exclude smoke test scenario. It will look like the following.

```
@RunWith(Cucumber.class)
@Cucumber.Options(plugin = ("pretty"), tags = (“not @SmokeTest"))
```

Tags using AND/OR in Cucumber

- Sometimes, requirements are complicated, it will not always simple like executing a single tag. It can be complicated like executing scenarios that are tagged either as @SmokeTest or @RegressionTest. It can also be like executing scenarios that are tagged both as @SmokeTest and @RegressionTest. Cucumber tagging gives us the capability to choose what we want with the help of ANDing and ORing.
- Tags that separated with "or" are ORed. OR means scenarios that are tagged either as @SmokeTest OR @RegressionTest.
- Tags which are passed with "and" separator are ANDed. When both the tags will match only, scenarios will be executed.

Hooks in Cucumber

- Multiple prerequisites are required at the start of test execution for most of the times. Right from setting up the web driver, browser settings to cookies, navigating to the specific URL, etc.
- Similarly, some steps must be performed after executing the test scenarios, like quitting driver, clearing browser cookies, generating reports, etc. Such cases can be easily handled using one particular type of Cucumber annotations, namely Cucumber Hooks.
- Cucumber hooks are blocks of code that runs before or after each scenario. It can be defined anywhere in project or step definition layers using methods @Before, @After. Cucumber hooks Annotations allow us to manage better code workflow and help in reducing code redundancy. Cucumber hooks are used in a situations where prerequisite steps before testing any test scenario is performed.
- User might have worked across different TestNG annotations, like BeforeTest, BeforeMethod, BeforeSuite, AfterTest, AfterMethod, AfterSuite, etc. Unlike TestNG, Cucumber provides only two hooks, i.e., Before and After.
- @Before Hook: It will execute before every scenario. Example

```
@Before
public void setUp() {
        System.out.println("Starting the test");
}
```

- @After Hook: It will execute after every scenario.

```
@After
public void tearDown () {
          System.out.println("Closing the test");
}
```

Tagged Hooks in Cucumber

- Tagged Hooks are basically the problem solvers when user need to perform different Before and After actions for different scenarios. To explain in a simpler way, think user have 5 different tags like Sanity, Regression, Integration etc., which need to be tested with different URLs. This is where Tagged hooks help you achieve that.
- Now we will create the hooks file and define the tagged hooks in it. This can be done using the @Before or the @After hook followed by the tag name of the scenario as shown in the code below:

```
@Before("@Smoke")
  public void beforeSmoke(){
     System.out.println("This will be executed before any smoke test cases");
  }
     @After("@Smoke")
  public void afterSmoke(){
     System.out.println("This will be executed after any smoke test cases");
  }
 @Before("@Integration")
  public void beforeIntegration(){
     System.out.println("This will be executed before any Integration test cases");
  }
     @After("@Integration")
  public void afterIntegration(){
     System.out.println("This will be executed after any Integration test cases");
  }
```

Setting Order or Priority of Cucumber Hooks

- In case user have already worked on TestNG, they must be familiar with the priority of tests and execution order. Similarly, Cucumber hooks can also be executed as per order.
- Let's, consider an example with a hooks file consisting of two @After hooks and two @Before hooks. User will set the order of the hooks as per our requirement by simply specifying the order as in the hooks file code below:

```
@Before(order=0)
    public void setUp() {
        System.out.println("This will execute first--Before");
    }
    @After(order=1)
    public void tearDown() {
        System.out.println("This will execute first--After");
    }
    @Before(order=1)
    public void setUpTwo() {
        System.out.println("This will execute second--Before");
    }
    @After(order=0)
    public void afterOne() {
        System.out.println("This will execute second--After");
    }
```

Reports in Cucumber

- During test execution (Either Manual or Automation), it is always required to understand the out put of the execution. The execution output should display in specific format, which immediately depicts the overall results of the execution. Hence, our framework also should have the same capability to create output or generate test execution reports.
- It is essential to know, how better user can generate our Cucumber test reports. As Cucumber is a BDD framework, it does not have a fancy reporting mechanism. In order to achieve this, Cucumber itself has provided a nice feature to generate reports.
- Cucumber.io has developed a free cloud-based service for sharing reports. Cucumber reporting service allows to configure cucumber to publish results to the cloud which can be accessed from browser throughout the organization.
- Cucumber uses reporter plugins to produce reports that contain information about which scenarios have passed or failed.

- Pretty Report: The first plugin, is Pretty. This provides more verbose output. To implement this, just specify plugin = "pretty" in CucumberOptions.

@CucumberOptions(plugin = ("pretty"))

- Monochrome Mode Reporting: If the monochrome option is set to false, then the console output is not as readable as it should be. In case the monochrome is not defined in Cucumber Options, it takes it as false by default.

@CucumberOptions(monochrome = true);

HTML, JSON, XML and JUNIT Reports in Cucumber

- HTML Report : To generate custom report in html format, include below plugin in Runner class,

plugin = {
   "pretty", “html:target/html-reports/report.html” }

After execution Cucumber will generate HTML report under target/html-reports/ folder.

- JSON Report: To generate custom report in json format include below plugin in Runner class,

plugin = {
   "pretty", "json:target/json-reports/report.json‟ }

- XML Report: To generate custom report in xml format include below plugin in Runner class,

plugin = {
   "pretty", "junit:target/xml-reports/report.xml‟ }

- JUNIT Report : In case of using junit, reports can be generated in a separate file depends on runner and by providing below plugin details in runner class,

plugin = { “pretty”,
 “html:target/html-reports/report.html”,
 “junit:target/junit-reports/”
}

Extent Report

- In the previous video you already learnt how to generate reports in HTML, JSON, XML and JUNIT Format. However, these reports might not be appropriate sometime to share with respective stakes holder as demand is complete exhaustive reports with proper graphs, pie-chart and more granular details.

- Extent Report is a powerful, open-source library used in testing automation frameworks for generating beautiful and user-friendly HTML reports. It allows the user to customize the report template by using custom CSS, JSON, or XML. It can integrate with almost all the major testing frameworks like JUnit, TestNG, etc. Extent reports are HTML-based documents that can carry detailed information about the test executed along with custom logs, screenshots and use a pie chart to represent an overview of the test.

- Extent reports cannot directly integrate with the Cucumber framework. However, in case user want to generate extent reports for their Cucumber features, user will need to use some adapter plugin. Additionally, this plugin will allow the Extent report to recognize and capture scenarios and features present in the framework. It is where the grasshopper cucumber adapter plugin comes into the picture. The plugin is built on top of the Extent Report and allows user to quickly generate extent reports for their cucumber framework. Moreover, this plugin helps in reducing the pain of implementing the reporting in user framework.

Steps to Generate Extent Report using Cucumber

a. Add grasshopper extent report adapter plugin to the Maven project:

Navigate to the "https://mvnrepository.com/", type "extentreports" in the search box and select latest "ExtentReports Cucumber7 Adapter". Add the maven dependency in the Maven project pom.xml file.

b. Add Extent Report Library to the project:

Again, navigate to the "https://mvnrepository.com/", type "extent-pdf-report" in the search box and select latest "Extent PDF Report". Add the maven dependency in the Maven project pom.xml file.

c. Create a file named “extent.properties” in same folder where feature files were located. Add the following lines to the file.

```
extent.reporter.spark.start=true
extent.reporter.spark.out=target/automation-report.html
```

d. Add the following lines in user TestRunner.java class

```
plugin = { "pretty",
           "com.aventstack.extentreports.cucumber.adapter.ExtentCucumberAdapter:" }
```

After completing all the setup and configuration run TestRunner class and then refresh the project. A beautiful HTML report will be generated in the target folder.

How to Maintain Backup of All Reports

- As of now we have learned how user can generate an extent report in Cucumber Junit. in case of multiple times execution, it will keep on overriding the previous report once the new report creates. Usually, user need to maintain the backup of all the reports generated from previous tests. Hence, user need to save each report with a different report name or folder name.
- With the Extent reporter plugin adapter, it's pretty easy to create reports in the different folder names. User need to add two settings to their extent.properties file; basefolder.name and basefolder.datetimepattern. The values assigned to these will merge to create a folder name. Consequently, a report will generate inside that. The value for the basefolder.datetimepattern should be in a valid date-time format.

```
basefolder.name=target/ExtentReport
basefolder.datetimepattern=dd-MMM-YY HH-mm-ss
```