**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
9. Running Cucumber from Command Line

10. Running Cucumber Parallel tests from Command Line

11. Cucumber Anti Patterns

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>

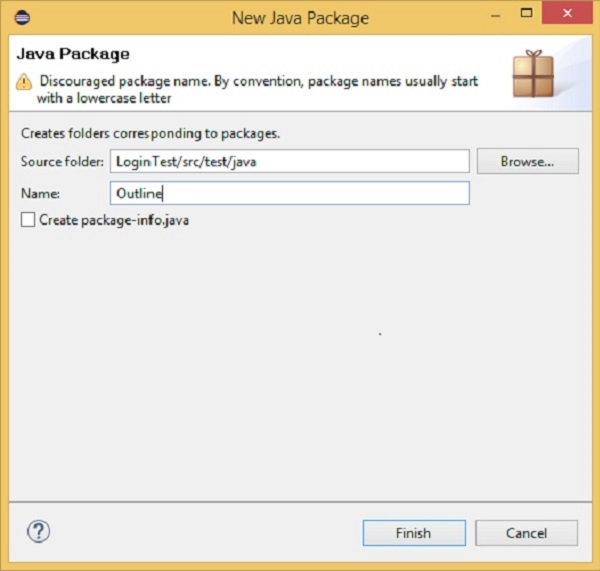
<version>4.10</version>

<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.

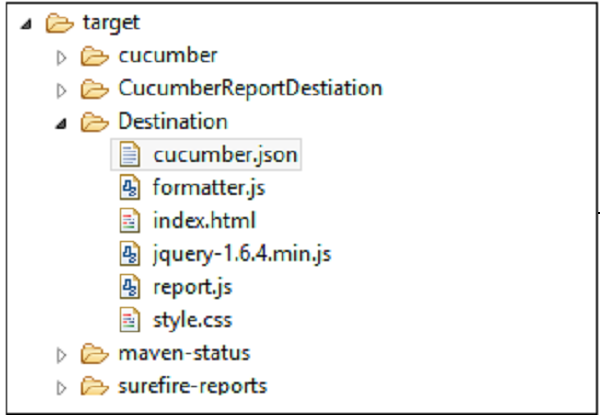
public class runTest { }

* 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.

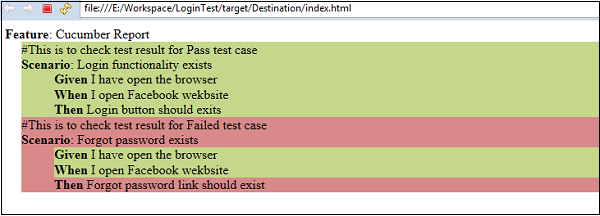
A folder named Destination will be created inside the target directory.



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)

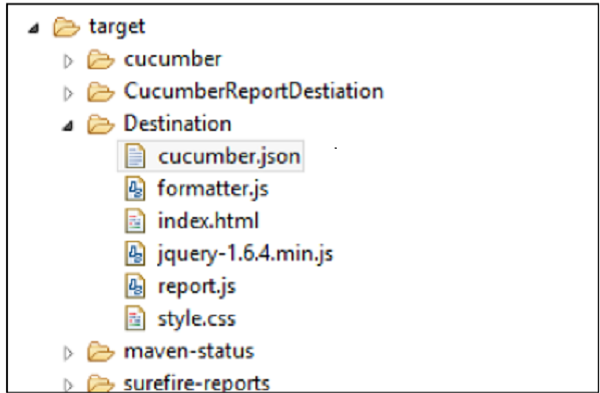
@Cucumber.Options( format={"json:target/Destination/cucumber.json"})

//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 −

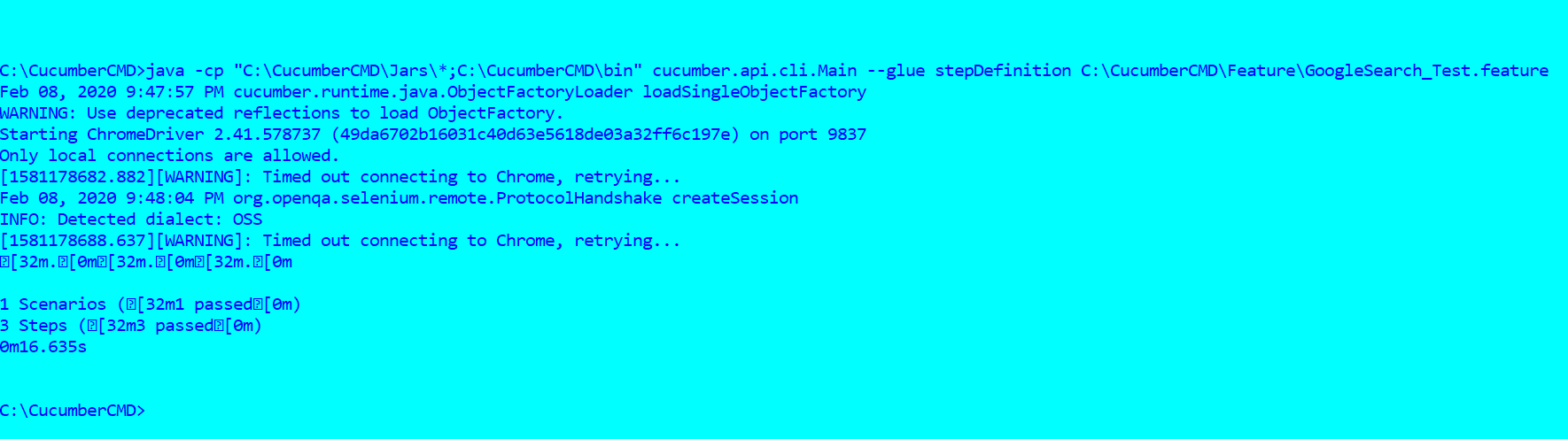


**Note** − JSON is less readable as compared to the HTML report format.

1. **Running Cucumber BDD from Command Line using ‘cucumber.api.cli.Main’**
2. Open CMD
3. Change to cucumber project location ‘cd C:\CucumberCMD’
4. Enter the below command

java -cp "C:\CucumberCMD\Jars\\*;C:\CucumberCMD\bin" cucumber.api.cli.Main --glue stepDefinition C:\CucumberCMD\Feature\GoogleSearch\_Test.feature

You will see the below output from CMD and the chrome browser is launched and the automation scenario is completed.

****

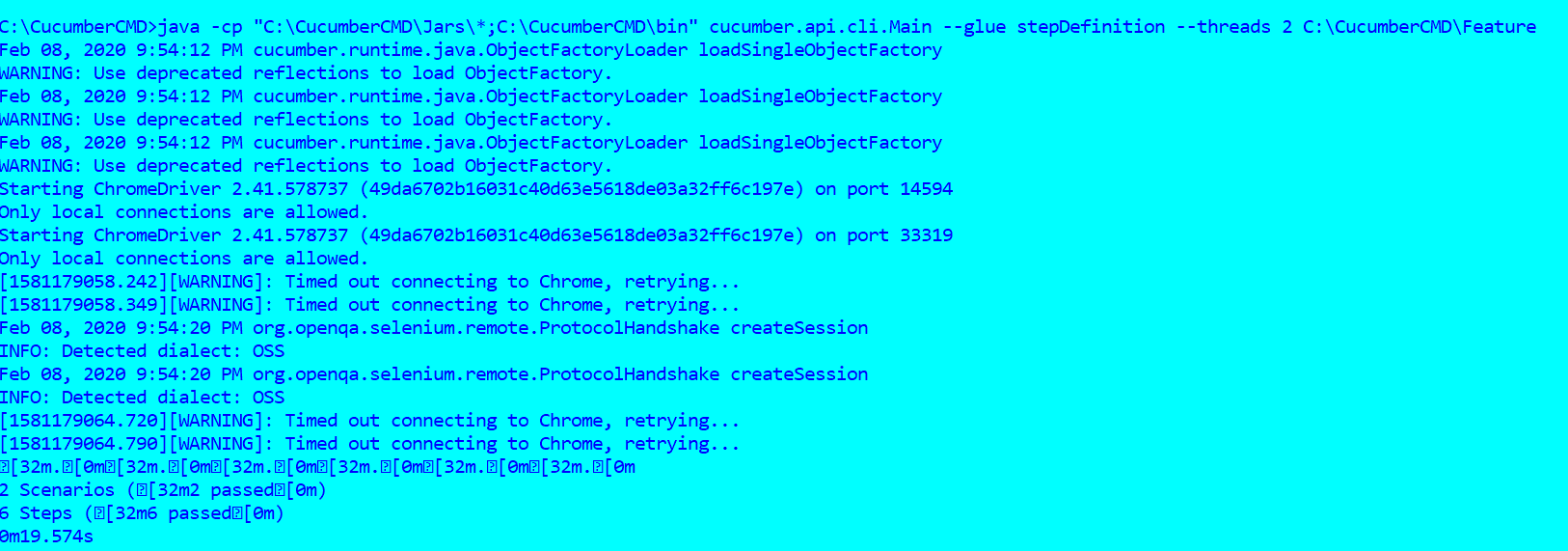
1. **Parallel Testing in CommandLine using ‘cucumber.api.cli.Main’**

The --threads option needs to be set to a value greater than 1 to run in parallel. When the parallel mode is used, the scenarios and rows in a scenario outline will be run in multiple threads.

The following lists of latest jars are required to enable parallel execution in command line using ‘cucumber.api.cli.Main’.

1. cucumber-core.jar [4.4.0](http://central.maven.org/maven2/io/cucumber/cucumber-core/4.4.0/cucumber-core-4.4.0.jar)
2. cucumber-java.jar [4.4.0](http://central.maven.org/maven2/io/cucumber/cucumber-java/4.4.0/cucumber-java-4.4.0.jar)
3. cucumber-java8.jar [4.4.0](http://central.maven.org/maven2/io/cucumber/cucumber-java8/4.4.0/cucumber-java8-4.4.0.jar)
4. gherkin.jar [5.1.0](http://central.maven.org/maven2/io/cucumber/gherkin/5.1.0/gherkin-5.1.0.jar)
5. cucumber-expressions.jar [7.0.2](http://central.maven.org/maven2/io/cucumber/cucumber-expressions/7.0.2/cucumber-expressions-7.0.2.jar)
6. datatable.jar [1.1.14](http://central.maven.org/maven2/io/cucumber/datatable/1.1.14/datatable-1.1.14.jar)
7. datatable-dependencies.jar [1.1.14](http://central.maven.org/maven2/io/cucumber/datatable-dependencies/1.1.14/datatable-dependencies-1.1.14.jar)
8. tag-expressions.jar [1.1.1](http://central.maven.org/maven2/io/cucumber/tag-expressions/1.1.1/tag-expressions-1.1.1.jar)
9. typetools.jar [0.5.0](http://central.maven.org/maven2/net/jodah/typetools/0.5.0/typetools-0.5.0.jar)
   * 1. Open CMD
     2. Change to cucumber project location ‘cd C:\CucumberCMD’
     3. Enter the below command

**java -cp "C:\CucumberCMD\Jars\\*;C:\CucumberCMD\bin" cucumber.api.cli.Main --glue stepDefinition --threads 2 C:\CucumberCMD\Feature**

****

**12. Cucumber Anti-Patterns**

Cucumber is designed to be a collaboration tool that creates a living documentation, which is possible to automate, of the behaviour you want in a system. You describe the wanted behaviour using a language called [Gherkin](https://cucumber.io/docs/reference#gherkin). If you are unfamiliar with Gherkin, it is probably a good idea to read up on it before you continue. The rest of the text assumes that you are familiar with Gherkin and that the format Given/When/Then isn't strange to you.

**Bad collaboration**

Cucumber is a collaboration tool and should be used for collaborating when driving the implementation of software.

When do you write feature files

One anti-pattern is that you write the feature files after you have written the code. This could arguable be the worst anti-pattern when using Gherkin. It is very common when you start using Cucumber and BDD.

What you really want is to write the Gherkin before you write the software. This will allow you to actually use the scenarios to drive the development and not the other way around, to document what you developed.

Why should you write the Gherkin before you write the software? Documenting the wanted behaviour is a way to make sure that everyone agrees on what the software is supposed to do. When you have reached that agreement, you have a common understanding of the problem, then it is a good time to start the implementation and write the code needed. Writing code before you have come to a common understanding is one way to make sure you implement something that you will have to change later.

Cucumber is a testing tool that allows you to test your assumptions and understanding of the problem before you actually write the code to solve the problem. It will make it clear if the people involved agree on the wanted behavior or if there is ambiguity that hasn't been discovered yet. Concrete examples before the implementation will make it clear that everyone has the same understanding.

A product owner or business analyst, a dev, and a tester will all have different perspectives on the problem. Understanding these different perspectives before writing the code increases the chance of creating something that actually is what the end user needs.

Business people create scenarios in isolation

One person, maybe the product owner or business analyst, writes the scenarios alone. The result will be Gherkin that doesn't represent everybody's understanding. The scenarios written by the product owner or business analyst alone tend to be un-testable.

To be able to automate the scenarios, they will have to be changed. This leads to Gherkin that isn't the Gherkin the product owner or business analyst originally wrote and therefore it doesn't represent what they wanted. They will loose interest and the Gherkin will be something devs or testers use instead.

The common understanding is lost.

There is also a chance that when the devs start to reformulate the Gherkin, they misunderstand what the product owner or business analyst actually meant and ends up with something that is incorrect.

Devs or testers writing scenarios without talking to business people

This is similar to the case where product owner or business analyst writes the scenarios alone. But in this case the examples tend to be unrealistic. Sometimes really dry. The examples are not talking about real users, [personas](https://en.wikipedia.org/wiki/Persona_%28user_experience%29), but instead talking about user1 and user2.

In this case the developers missed out on an opportunity to have realistic users, realistic data and ended up with something really boring and dry.

Too high level

If a scenario is on a too high level, you can't really trust it. You can probably not tell what it actually will do because it is too high level and vague.

An example could be

Given a bank account

When I withdraw some money

Then the balance should be the original balance minus the amount withdrawn

This is an example that expresses a business rule. It doesn't contain any concrete example. We don't know the original balance; we don't know how much that was withdrawn so we can't know the outcome. It is not concrete.

Another example is

Given the system is turned on

When it is used

Then it should work perfectly

This leaves a lot of freedom to the implementer. It also trusts that the implementer has understood the requirement perfectly and will not make any mistake. It is totally useless as an example that drives the development.

This is a trap that it is easier to fall into if the the product owner or business analyst writes the scenarios alone. With more people writing scenarios, more perspectives are taken into account. Vague examples can be made concrete, unclear examples can be questioned, and unclear rules can be found.

Finding the right balance between too vague and too many details is hard. This is something experience will teach you. Getting it perfect from the beginning is not something to expect from a team. It is hard even with lots of experience.

No living documentation

Cucumber is a documentation tool and can be used to create a good, living, documentation about a system.

When you read Gherkin and it is bad documentation

The litmus test could be: Take a scenario, show it to someone who isn't familiar with the domain and ask that person if they can describe the functionality the scenario represents. If that person can't describe what the system is supposed to do, then you have a case of bad documentation.

An example can be a long scenario, ten or more steps, and lots of [incidental details](https://www.vocabulary.com/dictionary/incidental) that don't describe an example of a business rule but rather a journey through the application. How it should be used rather than the behavior are we trying to verify.

Incidental details

Someone tried to tell a good story but instead overload the reader with too much information. The incidental details are often there because the author has written a test rather than a documentation of the desired behavior.

An example could be

When I sign up as Matt, my password is password, my password confirmation is password, and I check my bank balance

Then I find $100

The purpose with the scenario is to check the bank balance. In this case, it isn't relevant what your password is or how you login. It is there because the author needed the values when automating a test.

The problem with these incidental details is that they obscure the essence of the scenario. It is hard to understand what we really try to test. In this case, the reader would have to guess. Are we testing the password functionality or are we testing the bank bank balance? Or are we testing both in one scenario? All the details about passwords and bank balance makes it unclear what the purpose of the scenario is.

Hard to tell what you are testing

Are you testing one rule or are you testing several rules in the same scenario?

There are two problems here:

Testing multiple things at the same time

Not creating a clear documentation for checking if this is the desired behaviour

You may be forced to read between the lines to find out what the actual essence of the scenario is. If you are unlucky, the scenario might not have an essence. It is unfocused and aims at everything at the same time.

Bad name on a scenario

A good name on a scenario tells the reader what this is about. No name leaves the reader guessing.

A name that tries to summarize the entire behaviour of the scenario. Such as  
Scenario: Sign up, login, go to balance screen, check balance, logout  
is boring and you will loose the interest of the reader.

A good place to sum up the essence of a scenario is in its name. A name could be  
Scenario: Check balance

The name indicates that any details about something else are incidental. Any details about a password are incidental and should be removed. If a password is needed for the automation, it can be added in the automation layer.

A good naming algorithm could be to use the [Friends](https://en.wikipedia.org/wiki/Friends) metaphor. This is the one where ...

Here are a few examples:

This is the one where the balance is positive

This is the one where I have an overdraft on my account

This is the one where I change my password

The behavior is then described in the Given/When/Then of the scenario

Not using the narrative section of a feature

The first part in a [feature file](https://cucumber.io/docs/reference) is called the narrative section. You can write whatever you want here. You can leave it blank or add useful things.

Leaving the narrative section blank or adding a user story like the one below are two ways if applying an anti-pattern.

As a user,

I want to check my balance,

so I know what my balance is

This user story doesn't give you any extra information and is therefore pretty useless. Someone just filled out a template without really reflecting over why someone else wants this behaviour.

Instead, put useful information for the reader in the beginning of the feature. Describe the business rules; the abstract rules that we will describe examples for.

An example of a rule could be:  
After you have done a withdrawal, that withdrawal should show up in your list of transactions.

The format you describe the business rule in is not that important. Describe them using a bullet list if that's better. The examples that follows will make them concrete.

The narrative section is the place to put questions or uncertainties in as well.

Think of the feature file as living documentation where you put the rules discovered and unanswered questions, known unknowns. This is our current understanding of this feature. There is stuff we know and there is stuff we don't know yet.

Beginners’ mistakes

There are some mistakes commonly done by beginners.

Lots of user interface details

The system is used through its user interface. This sometimes leads to scenarios that talks about going to a specific url, click a specific link, find an element using a specific css selector etc. It doesn't really tell us what the purpose of using the program is. It is hard to understand why the user want to do this journey.

One problem with this is understanding the purpose. Another problem is that user interfaces changes a lot more frequently than the underlying domain logic.

UI trends change more often than the actual business. This will break your tests even if the business haven't changed. UI tests are very slow and brittle. It takes a long time to fire up a UI testing tool.

Testing through the UI will make you stuck at the top of the [testing pyramid](http://martinfowler.com/bliki/TestPyramid.html). All of your testing will be done through the user interface. This is a bad and painful place to be since it is slow and brittle.

Scenarios littered with UI details will make it impossible to test anything below the user interface.

Too many user interface details is also very poor documentation. It is hard to understand the business rule by understanding how the navigation through a system works. The navigation through a program doesn't tell us why we are using the program.

The problem you solve by using this program is not described using words from the domain, it is described using the generic terms of user interfaces. Not the domain of your business. You miss the nouns and verbs that describe your business problem that you would like to use deeper down in your code.

This anti-pattern usually comes from the fact that you wrote the scenarios after you actually implemented the solution.

Describing actions using the personal pronoun I

Most systems have behavior that is used by multiple users or actors. This means that you want to talk a bout a specific user, a persona, if you can. A persona will give you context about what the system should be able to do to support a specific category of users.

If a system supports a university, there is probably a difference between a 21 year old student and a 53 year old administrator. The role, student or administrator, gives you different contexts. The age may or may not give you valuable information.

Documenting boring scenarios

Some scenarios are very boring and perhaps just needed in the beginning of a project. An example is:

Given my bank account is empty

When I check my bank balance

Then it should be 0

You probably need certain scenarios in the beginning of the project or when you start developing a certain feature. Some of them will, however, be obvious after a while and covered by other tests.

Keeping all scenarios forever

Not all scenarios will bring value forever. You may delete scenarios after a while if you are certain that this case is covered in other test. But before you delete them, make sure that the behavior is covered somewhere else so you don't loose it.

Instead of deleting scenarios, you can rewrite them to document something more interesting. There are perhaps edge cases that can be covered with a rephrased scenario.

Keeping scenarios because they where written once is not a a good argument. Don't do that.

No clear separation between Given/When/Then

It happens that beginners have a hard time grasping the difference between Given, When and Then. The problem is that from a technical perspective, there is no difference, Cucumber will treat them all equal.

What is the difference then?

Given is the context - the past

When is an action that changes the system - the present

Then is the expected outcome - the near future

A metaphor could be: Going to the theater

You are sitting in your chair and the curtain is drawn. Behind the curtain there is a lot of stage workers preparing. Putting things in place. This is the Given. This is where you do the setting up of your system, creating initial objects, prepare the database, navigating to a specific page and so on.

The curtain is drawn apart and the act starts. Things start to happen, one thing lead to another. Actors start doing things and they say their lines. This is the When. This is triggering the important event that we want to happen in a specific context.

One thing on stage leads to another thing, the expected outcome. What happened in the story as a result of what an actor did. This is the Then. The observable change that we want to assert.

The Given and When is hopefully easier to grasp with this metaphor. The Then is perhaps less clear.

The past - the preparation is Given

The present - the action is the When

The near future - the expected outcome is the Then

Multiple When

That is a When is followed by one or many And.

There are cases when it can be ok. Say that two different persons does two different things in a When step, then it can be ok. This is an example of similar events.

When there are two completely different events, then it may be a bad idea to have an implicit When, expressed as an And. It can be an indicator that you should split this scenario into two different scenarios. Chances are that you describe two different behaviors in the same scenario.

**Double edged sword**

There are cases that may or may not be anti-patterns. They can be used in a bad way, but they also enable you to do really good things. They are like a double edged sword and depends on the context.

Scenario outline

Scenario outline is an example of something that easily can be overused and lead to slow tests. This is especially true if they contain a lot of UI details.

It is very easy to add a lot of scenarios when you use an outline. If the scenarios are slow, you will end up with a really slow test suit.

Using scenario outlines to verify complicated algorithms can be a valid use case. But probably not through the slow user interface.

An example can be to verify a complicated insurance algorithm where there are lots of data that will create different outcomes. In this case, it is important that you can reach the algorithm without going through the UI. This requires the system to be built so it is testable.

If the scenarios are fast, then it may be ok to use Scenario outlines. If they are slow, do not use scenario outlines.

Multiple Then in the same scenario

This is not necessarily an anti-pattern. An example could be when something is returned in a retail system.

The customer should be refunded

The inventory should be updated

The finance system should be updated

A message should be send to a warehouse that the returned item needs to be picked up

These are all connected. Verifying them in one go could be a good idea. At the same time, these outcomes are the result of four different business rules. Splitting them into four different scenarios will make the business rule much more explicit and will probably help you understand the system better. Splitting these will also allow the development team to implement the rules incrementally; they don't have to be implemented at the same time. This only applies to independent outcomes.

There are examples when it not possible to split a scenario. Withdrawing money from a bank account is such an example.

The customer should get their money from an ATM machine, say £50

The customer account should be debited £50

These outcomes are depending on each other and can't be divided into different scenarios. The bank will not appreciate if the customers gets money without the account being debited. The other way around is probably also true, most customers will not be happy if their account is debited but they didn't get any cash.

Holiday photos

The features can be seen as a reminder of what you talked about and agreed on in the session when you wrote them. They should contain enough details for you to remember the most important details.

They should serve the same purpose as a holiday photo, remind you of a great time.

Conclusion

There are many mistakes you can do when you use Cucumber and implement BDD. It is not the end of the world if you do them, but it is better to avoid them. You are now equipped with some knowledge about some of the anti-patterns that are possible to do. There are probably many more mistakes you will do, but that's life. You live and learn.

**References:**

1. Cucumber Anti Patterns Blog - <http://www.thinkcode.se/blog/2016/06/22/cucumber-antipatterns>

2. Cucumber Anti Patterns Documentation - <https://cucumber.io/docs/guides/anti-patterns/>

3. Running Cucumber project through Command Line using Java - <https://sqa.stackexchange.com/questions/31075/how-to-run-selenium-cucumber-eclipse-project-through-command-prompt>

4. Step by Step Running Cucumber from Command Line using Java - <https://dzone.com/articles/running-cucumber-from-the-command-line>

5. How to Run Cucumber BDD from command line using Java –

<https://www.youtube.com/watch?v=Z8gm4hFsivI>

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

7. Cucumber API Reference - <https://cucumber.io/docs/cucumber/api/>

8. Cucumber Test from Command Line - <https://www.toolsqa.com/selenium-cucumber-framework/run-cucumber-test-from-command-line-terminal/>

9. Running Cucumber from command line using Maven - <https://www.youtube.com/watch?v=4cF1Q7CISSE>

# 10. Parallel execution using ‘cucumber.api.cli.Main’ class - <https://grasshopper.tech/283/>

11. Getting updated ‘cucumber.api.cli.Main’ options - <https://stackoverflow.com/questions/43696842/usage-java-cucumber-api-cli-main-options-filedirlineline>

12. Latest Jars containing parallel execution ‘threads’ options - <http://grasshopper.tech/1162/>

13. **Parallel execution Cucumber documentation** - <https://cucumber.io/docs/guides/parallel-execution/>