**Cucumber**

**BDD** in a nutshell Behaviour-Driven Development (BDD) is a set of practices that enables software teams to produce more valuable software with fewer bugs. BDD can lead **to reduced development and maintenance costs and shorter time to market.**

The two main practices of BDD are **Specification by Example** and **Test-Driven Development(TDD).**

(BDD- TDD+ Incremental Requirement Analysis)

**Specification by Example:**

Specification by Example (SbE) is a technique that enables product owners, business analysts, testers and programmers to eliminate common misunderstandings about business requirements. SbE uses examples in conversations to illustrate business rules and behaviour of the software to be built.

**Test-Driven Development:**

Test-Driven Development (TDD) in the context of BDD turns examples into human-readable executable specifications. **Programmers use these specifications as a guide to implement increments of new functionality.** The result is a lean codebase and a suite of automated regression tests that keeps maintenance costs low throughout the lifetime of the software.

**Cucumber** lets you keep specifications, automated tests and documentation in the same place - a single source of truth that never gets out of sync.

Other popular BDD f/ws: JBehave, Jasmine etc

Login – 10 testcases-

basic code – run 10 test cases – 6 of them , 4 test cases failed

may be +3 testcases based on above failure

modify code- run 10 test cases – 9 of them, 1 test case failed

modify code- 10 testcases passed

Tests are driving the devlopement – TestDrivenDevelopment(TDD)

**Cucumber flow:**

1. **Runner class** – where we configure path to feature files and step definitions

Runner uses JUnit framework.

1. **Feature file:**

file which has extension .feature

File where you describe the requirements as small scenarios in **simple english language(Gherkin language**- internally uses Ruby)

We use simple English keywords like **given, when, then** to explain the scenarios

based on **gherkin** language

A single feature file can have more than one scenario.

UserStory or **feature**

eg: Customer Registration

username, pwd, email, phone(mandatory)

age,DOB, address(optional)

Testcases or **scenarios**:

ValidScenario with required fields

ValidScenario with required fields and optional fields

InvalidScenario without required fields

InvalidScenario for validations- password character

……

**3.StepDefinition classes:**

**Every feature file will have corresponding step definition class**

**It’s a Java class which is connected with respective feature file**

**When you run/execute a feature file- it executes the logic inside stepdefinition**

**class**

**Setup:**

1. **Install cucumber eclipse plugin:**

Update Site- copy below link to eclipse- install new software- add-

<https://cucumber.io/cucumber-eclipse/update-site>

1. **Create new maven project and add required dependencies**

**Cucumber can be used or integrated with any automation tools:**

Web automation using WebDriver

API automation using RestAssured or HttpClient

Mobile Automation using Appium

**Cucumber Before annotation runs before every scenario.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**LoginScenario**

**enter username\enter pwd**

**click submit**

**execute testcases**

**test cases fail**

**7testcases will pass**

**8 testcases fail**

**12 , 3**

**15 testcases passed**

**tests are driving development**

**TDD- Test Driven Development**

**BDD- Behaviour driven development**

**feature file-**

**requirements which are split into scenarios**

**Feature – Gherkin**

**added cucumber eclipse plugin**

**added maven dependencies after creating maven project**

**we created RunCukesTest- config class**

**we create feature file which is connected stepdefintion(java class)**

**you can integrate cucumber with webdriver/restassured/httpclient/appium**

**It depends on what kind of testing we want to do- UIAutomation/APIAutomation/MobileAutomation**

**RunCukesTest:**

**We use Junit Runner which runs Cucumber class with certain options(CucumberOptions)**

**\*\*\*\*\*\*\*\***

**CucumberOptions- feature, glue(step definitions),dryRun,strict, monochrome,plugins**

**feature- we configure feature files folder path so that cucumber can run all features**

**glue-package path of step definitions**

**dryRun- true- features are run just to generate step definitions skeletons**

**dryRun- false- features are run to execute step definition methods or actual testcases are executed.**

**RunCukesTest->feature->stepdefinitons->Page**

**RunCukesTest->feature->stepdefinitons->APIs**

**Feature file:**

**Feature- brief description about user story(eg- login, signup, search)**

**Scenario- divide the user story into small scenarios using gherkin keywords:**

**Given**

**When**

**Then**

**And**

**But**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Gherkin

Cucumber executes your .feature files, and those files contain executable specifications written in a language called Gherkin.

Gherkin is plain-text English (or one of 60+ other languages) with a little extra structure. Gherkin is designed to be easy to learn by non-programmers, yet structured enough to allow concise description of examples to illustrate business rules in most real-world domains.

Here is a sample Gherkin document:

Feature: Refund item

Scenario: Jeff returns a faulty microwave

Given Jeff has bought a microwave for $100

And he has a receipt

When he returns the microwave

Then Jeff should be refunded $100

In Gherkin, each line that isn't blank has to start with a Gherkin keyword, followed by any text you like. The main keywords are:

* Feature
* Scenario
* Given, When, Then, And, But (Steps)
* Background
* Scenario Outline
* Examples

There are a few extra keywords as well:

* """ (Doc Strings)
* | (Data Tables)
* @ (Tags)
* # (Comments)

### Feature

A .feature file is supposed to describe a single feature of the system, or a particular aspect of a feature. It's just a way to provide a high-level description of a software feature, and to group related scenarios.

A feature has three basic elements---the Feature: keyword, a name (on the same line) and an optional (but highly recommended) description that can span multiple lines.

Cucumber does not care about the name or the description---the purpose is simply to provide a place where you can document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

Here is an example:

Feature: Refund item

Sales assistants should be able to refund customers' purchases.

This is required by the law, and is also essential in order to

keep customers happy.

Rules:

- Customer must present proof of purchase

- Purchase must be less than 30 days ago

In addition to a name and a description, features contain a list of [Scenarios](https://cucumber.io/docs/reference" \l "scenario) or [Scenario Outlines](https://cucumber.io/docs/reference" \l "scenario-outline), and an optional [Background](https://cucumber.io/docs/reference" \l "background).

### Descriptions

Some parts of Gherkin documents do not have to start with a keyword.

On the lines following a Feature, Scenario, Scenario Outline or Examples you can write anything you like, as long as no line starts with a key a keyword.

### Scenario

A scenario is a concrete example that illustrates a business rule. It consists of a list of [steps](https://cucumber.io/docs/reference" \l "steps).

You can have as many steps as you like, but we recommend you keep the number at 3-5 per scenario. If they become longer than that they lose their expressive power as specification and documentation.

In addition to being a specification and documentation, a scenario is also a test. As a whole, your scenarios are an executable specification of the system.

Scenarios follow the same pattern:

* Describe an initial context
* Describe an event
* Describe an expected outcome

This is done with steps.

### Steps

A step typically starts with Given, When or Then. If there are multiple Given or When steps underneath each other, you can use And or But. Cucumber does not differentiate between the keywords, but choosing the right one is important for the readability of the scenario as a whole.

#### Given

Given steps are used to describe the initial context of the system---the scene of the scenario. It is typically something that happened in the past.

When Cucumber executes a Given step it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to the test database.

It's ok to have several Given steps (just use And or But for number 2 and upwards to make it more readable).

#### When

When steps are used to describe an event, or an action. This can be a person interacting with the system, or it can be an event triggered by another system.

It's strongly recommended you only have a single When step per scenario. If you feel compelled to add more it's usually a sign that you should split the scenario up in multiple scenarios.

#### Then

Then steps are used to describe an expected outcome, or result.

The [step definition](https://cucumber.io/docs/reference" \l "step-definitions) of a Then step should use an assertion to compare the actual outcome (what the system actually does) to the expected outcome (what the step says the system is supposed to do).

### Background

Occasionally you'll find yourself repeating the same Given steps in all of the scenarios in a feature file. Since it is repeated in every scenario it is an indication that those steps are not essential to describe the scenarios, they are incidental details.

You can literally move such Given steps to the background by grouping them under a Background section before the first scenario:

Background:

Given a $100 microwave was sold on 2015-11-03

And today is 2015-11-18

### Scenario Outline

When you have a complex business rule with severable variable inputs or outputs you might end up creating several scenarios that only differ by their values.

Let's take an example from [feed planning for cattle and sheep](http://www.nutrientmanagement.org/assets/12028):

Scenario: feeding a small suckler cow

Given the cow weighs 450 kg

When we calculate the feeding requirements

Then the energy should be 26500 MJ

And the protein should be 215 kg

Scenario: feeding a medium suckler cow

Given the cow weighs 500 kg

When we calculate the feeding requirements

Then the energy should be 29500 MJ

And the protein should be 245 kg

# There are 2 more examples - I'm already bored

If there are many examples, this becomes tedious. We can simplify it with a Scenario Outline:

Scenario Outline: feeding a suckler cow

Given the cow weighs <weight> kg

When we calculate the feeding requirements

Then the energy should be <energy> MJ

And the protein should be <protein> kg

Examples:

| weight | energy | protein |

| 450 | 26500 | 215 |

| 500 | 29500 | 245 |

| 575 | 31500 | 255 |

| 600 | 37000 | 305 |

This is much easier to read.

Variables in the Scenario Outline steps are marked up with < and >.

#### Examples

A Scenario Outline section is always followed by one or more Examples sections, which are a container for a table.

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

Each of the rows below will create a new Scenario, filling in the variable values.

#### Scenario Outlines and UI

Automating Scenario Outlines using UI automation such as Selenium WebDriver is considered a bad practice.

The only good reason to use Scenario Outlines is to validate the implementation of business rule that behaves differently based on several input parameters.

Validating a business rule through a UI is slow, and when there is a failure it is difficult to pinpoint where the error is.

The automation code for Scenario Outlines should communicate directly with the business rule implementation, going through as few layers as possible. This is fast, and errors become easy to diagnose fix.

### Step Arguments

In some cases you might want to pass a larger chunk of text or a table of data to a step---something that doesn't fit on a single line.

For this purpose Gherkin has [Doc Strings](https://cucumber.io/docs/reference" \l "doc-strings) and [Data Tables](https://cucumber.io/docs/reference" \l "data-tables).

#### Doc Strings

Doc Strings are handy for passing a larger piece of text to a step definition. The syntax is inspired from Python's [Docstring](http://www.python.org/dev/peps/pep-0257/) syntax.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:

Given a blog post named "Random" with Markdown body

"""

Some Title, Eh?

===============

Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,

consectetur adipiscing elit.

"""

In your [Step Definition](https://cucumber.io/docs/reference" \l "step-definitions), there’s no need to find this text and match it in your pattern. It will automatically be passed as the last parameter in the step definition.

Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be de-indented according to the opening """. Indentation beyond the column of the opening """ will therefore be preserved.

#### Data Tables

Data Tables are handy for passing a list of values to a step definition:

Given the following users exist:

| name | email | twitter |

| Aslak | aslak@cucumber.io | @aslak\_hellesoy |

| Julien | julien@cucumber.io | @jbpros |

| Matt | matt@cucumber.io | @mattwynne |

Just like [Doc Strings](https://cucumber.io/docs/reference" \l "doc-strings), Data Tables will be passed to the [Step Definition](https://cucumber.io/docs/reference" \l "step-definitions) as the last argument.

The type of this argument will be DataTable. See the API docs for more details about how to access the rows and cells.

### Tags

Tags are a way to group Scenarios. They are @-prefixed strings and you can place as many tags as you like above Feature, Scenario, Scenario Outline or Examples keywords. Space character are invalid in tags and may separate them.

Tags are inherited from parent elements. For example, if you place a tag above a Feature, all scenarios in that feature will get that tag.

Similarly, if you place a tag above a Scenario Outline or Examples keyword, all scenarios derived from examples rows will inherit the tags.

You can tell Cucumber to only run scenarios with certain tags, or to exclude scenarios with certain tags.

Cucumber can perform different operations before and after each scenario based on what tags are present on a scenario.

See [tagged hooks](https://cucumber.io/docs/reference" \l "tagged-hooks) for more details.

### Comments

Gherkin provides lots of places to document your features and scenarios. The preferred place is [descriptions](https://cucumber.io/docs/reference" \l "descriptions). Choosing good names is also useful.

If none of these places suit you, you can start a line with a # to tell Cucumber that the remainder of the line is a comment, and shouldn't be executed.

## Step Definitions

Cucumber doesn't know how to execute your scenarios out-of-the-box. It needs Step Definitions to translate plain text Gherkin steps into actions that will interact with the system.

When Cucumber executes a [Step](https://cucumber.io/docs/reference" \l "steps) in a [Scenario](https://cucumber.io/" \l "scenario) it will look for a matching Step Definition to execute.

A Step Definition is a small piece of code with a pattern attached to it. The pattern is used to link the step definition to all the matching [Steps](https://cucumber.io/docs/reference" \l "steps), and the code is what Cucumber will execute when it sees a Gherkin Step.

To understand how Step Definitions work, consider the following Scenario:

Scenario: Some cukes

Given I have 48 cukes in my belly

The I have 48 cukes in my belly part of the step (the text following the Given keyword) will match the Step Definition below:

ruby

Given {^I have (\d+) cukes in my belly$} {cukes} {

puts "Cukes: $quantity"

}

Given(/I have (\d+) cukes in my belly/) do |cukes|

puts "Cukes: #{cukes}"

end

@Given("I have (\\d+) cukes in my belly")

public void I\_have\_cukes\_in\_my\_belly(int cukes) {

System.out.format("Cukes: %n\n", cukes);

}

<?php

/\*\*

\* @Given /^I have (\d+) cukes in my belly"$/

\*/

public function iHaveCukesInMyBelly($cukes)

{

print "Cukes: {$cukes}";

}

?>

Given(~'^I have (\\d+) cukes in my belly') { int cukes ->

println "Cukes: " + cukes

}

Given(/^I have (\d+) cukes in my belly$/, function (cukes) {

console.log("Cukes: " + cukes);

});

(Given #"^I have (\d+) cukes in my belly$" [cukes]

(println (str "Cukes: " cukes)))

GIVEN("^I have (\\d+) cukes in my belly$") {

REGEX\_PARAM(int, cukes);

USING\_CONTEXT(MyAppCtx, context);

std::cout << "Cukes: " << cukes;

}

[Given(@"^I have (\d+) cukes in my belly$")]

public void iHaveCukesInTheBelly(int cukes)

{

Console.WriteLine(string.Format("Cukes: {0}, world!"));

}

let [<Given>] ``^I have (\d+) cukes in my belly$``(cukes:int) =

printfn String.Format("Cukes: {0}", cukes)

Given("^I have (%d+) cukes in my belly$", function (cukes)

print("Cukes: " .. cukes)

end)

@Given('^I have (\d+) cukes in my belly$')

def i\_have\_cukes\_in\_my\_belly(self, cukes):

print "Cukes: " + cukes

Given("""^I have (\d+) cukes in my belly$"""){ (cukes:Int) =>

println(s"Cukes: $cukes")

}

Given {^I have (\d+) cukes in my belly$} {cukes} {

puts "Cukes: $quantity"

}

Given(/I have (\d+) cukes in my belly/) do |cukes|

puts "Cukes: #{cukes}"

end

When Cucumber matches a Step against a pattern in a Step Definition, it passes the value of all the capture groups to the Step Definition's arguments.

Capture groups are strings (even when they match digits like \d+). For statically typed languages, Cucumber will automatically transform those strings into the appropriate type. For dynamically typed languages, no transformation happens by default, as there is no type information.

Cucumber does not differentiate between the five step keywords Given, When, Then, And and But.

## Reports

Cucumber can report results in several different formats, using formatter plugins. The available formatters plugins are:

* [Pretty](https://cucumber.io/docs/reference" \l "pretty)
* [HTML](https://cucumber.io/docs/reference" \l "html)
* [JSON](https://cucumber.io/docs/reference" \l "json)
* [Progress](https://cucumber.io/docs/reference" \l "progress)
* [Usage](https://cucumber.io/docs/reference" \l "usage)
* [JUnit](https://cucumber.io/docs/reference" \l "junit)
* [Rerun](https://cucumber.io/docs/reference" \l "rerun)

Note that some Cucumber implementations might not provide all of these formatter plugins, and some implementations might provide additional ones.

### Pretty

Prints the [Gherkin](https://cucumber.io/docs/reference" \l "gherkin) source to STDOUT along with additional colours and stack traces for errors.

### HTML

Generates a HTML file, suitable for publishing.

### JSON

Generates a JSON file, suitable for post-processing to generate custom reports.

### Progress

This report prints results to STDOUT, one character at a time. It looks like this:

....F--U.......

### Usage

Prints statistics to STDOUT. Programmers may find it useful to find slow or unused Step Definitions.

### JUnit

Generates XML files just like [Apache Ant](https://ant.apache.org/)’s [junitreport](https://ant.apache.org/manual/Tasks/junitreport.html) task. This XML format is understood by most [Continuous Integration](http://en.wikipedia.org/wiki/Continuous_integration) servers, who will use it to generate visual reports.

### Rerun

The rerun report is a file that lists the location of failed Scenarios. This can be picked up by subsequent Cucumber runs:

cucumber @rerun.txt

This is useful while fixing broken scenarios, as only the scenarios that failed in the previous run will be run again. This can reduce time spent fixing a bug when running all scenarios is time-consuming.

If you are looking for a way to automatically rerun non-deterministic, or flickering scenarios in the same Cucumber run, the rerun report will not help you. It's meant to be used in a workflow where your scenarios are failing deterministically, and where you change scenarios or the system to make them pass between each Cucumber run.

If you have non-deterministic scenarios you have a deeper problem that Cucumber can't solve. You have to determine the root cause of the non-determinism, and address that yourself.

## Report attachments

Text, images and even video can be embedded into certain reports via an API that is available in [Step Definitions](https://cucumber.io/docs/reference" \l "step-definitions)and [Hooks](https://cucumber.io/docs/reference" \l "hooks).

This can make it easier to diagnose failures. Some [reports](https://cucumber.io/docs/reference" \l "reports) will ignore embedded data while others will include it.

### Screenshots

The recommended approach is to embed images in an [After Hook](https://cucumber.io/docs/reference" \l "after) if the Scenario fails:

ruby

# TODO

# This example assumes the use of Capybara with Selenium WebDriver

After do |scenario|

if(scenario.failed?)

page.driver.browser.save\_screenshot("html-report/#{scenario.\_\_id\_\_}.png")

embed("#{scenario.\_\_id\_\_}.png", "image/png", "SCREENSHOT")

end

end

// This example assumes the use of Selenium WebDriver

@After

public void embedScreenshot(Scenario scenario) {

try {

byte[] screenshot = webDriver.getScreenshotAs(OutputType.BYTES);

scenario.embed(screenshot, "image/png");

} catch (WebDriverException somePlatformsDontSupportScreenshots) {

System.err.println(somePlatformsDontSupportScreenshots.getMessage());

}

}

// TODO

// TODO

// TODO

; TODO

// TODO

// TODO

// TODO

-- TODO

# TODO

// TODO

# TODO

# This example assumes the use of Capybara with Selenium WebDriver

After do |scenario|

if(scenario.failed?)

page.driver.browser.save\_screenshot("html-report/#{scenario.\_\_id\_\_}.png")

embed("#{scenario.\_\_id\_\_}.png", "image/png", "SCREENSHOT")

end

end

### Text

Text can be embedded into the report from both [Step Definitions](https://cucumber.io/docs/reference" \l "step-definitions) and [Hooks](https://cucumber.io/docs/reference" \l "hooks):

ruby

# TODO

When /I do the deed/ do

puts 'This goes into the report'

Kernel.puts 'This goes to STDOUT, but not into the report'

end

@Before

public void keepScenario(Scenario scenario) {

this.scenario = scenario;

}

@When("I do the deed")

public void do\_the\_deed() {

scenario.write("This goes into the report");

System.out.println("This goes to STDOUT, but not into the report");

}

// TODO

// TODO

// TODO

; TODO

// TODO

// TODO

// TODO

-- TODO

# TODO

// TODO

# TODO

When /I do the deed/ do

puts 'This goes into the report'

Kernel.puts 'This goes to STDOUT, but not into the report'

end

## Integrating Cucumber with Maven

Maven has a lot of advantages over other build tools, such as dependency management, lots of plugins and the convenience of running integration tests. So let's also integrate our framework with Maven. Maven will allow our test cases to be run in different flavors, such as from the Terminal, integrating with Jenkins, and parallel execution.

Now we have integrated Cucumber with Maven, running Cucumber from the Terminal will not be a problem. Running any test framework from the Terminal has its own advantages, such as overriding the run configurations mentioned in the code.

So how do we run Cucumber test cases from the Terminal? Let's find out in our next section.

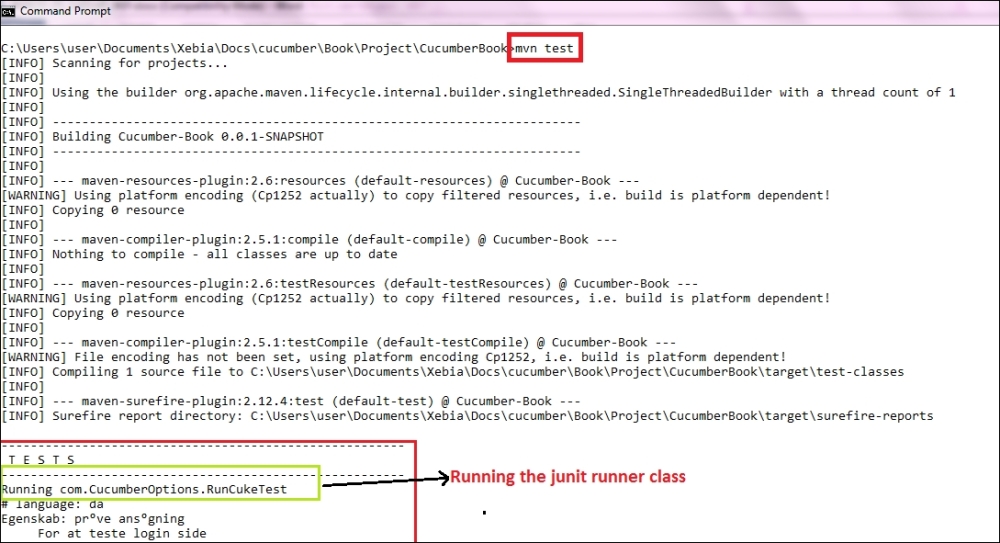
### How to do it…

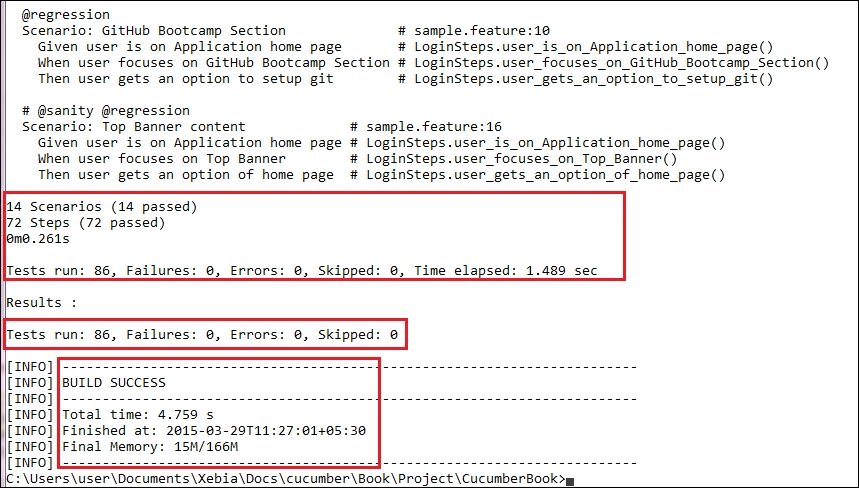
1. Open the command prompt and cd until the project root directory.
2. First, let's run all the Cucumber Scenarios from the command prompt. Since it's a Maven project and we have added Cucumber in test scope dependency and all features are also added in test packages, run the following command in the command prompt:

Copy

**mvn test**

This is the output:





1. The previous command runs everything as mentioned in the JUnit Runner class. However, if we want to override the configurations mentioned in the Runner, then we need to use following command:

Copy

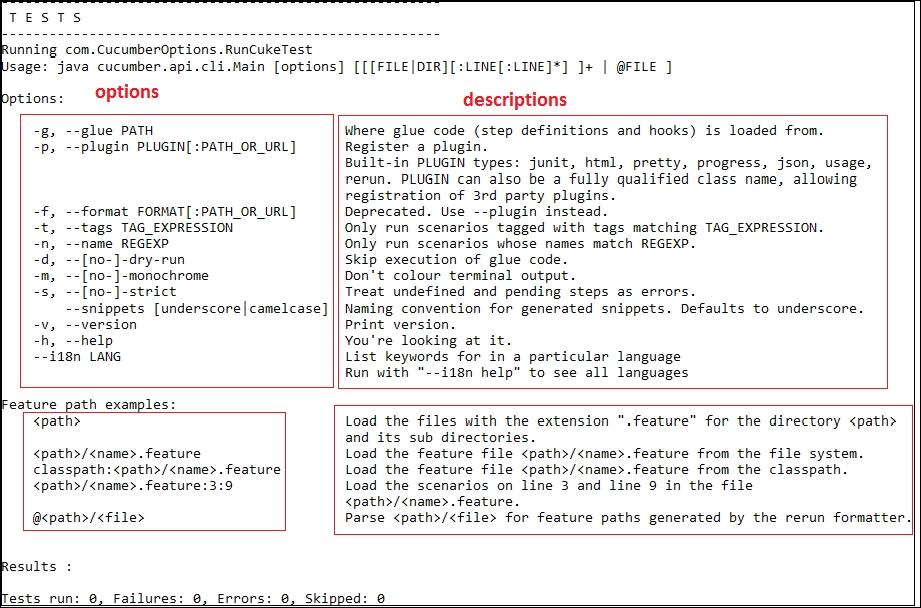
**mvn test –DCucumber.options="<<OPTIONS>>"**

1. If you need help on these Cucumber options, then enter the following command in the command prompt and look at the output:

Copy

**mvn test -Dcucumber.options="--help"**

This is the output:



## Overriding Options from the Terminal

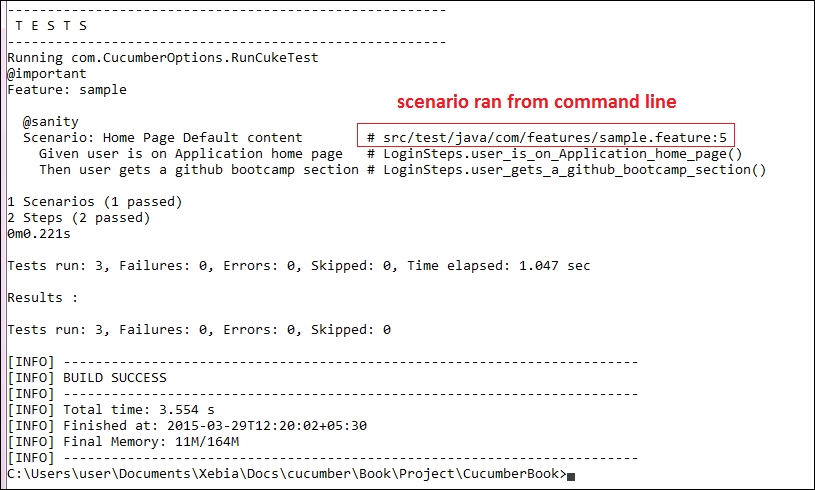
When it is necessary to override the options mentioned in the JUnit Runner, then we need Dcucumber.options from the Terminal. Let's look at some of the practical examples.

### How to do it…

1. If we want to run a Scenario by specifying the filesystem path, run the following command and look at the output:

Copy

**mvn test -Dcucumber.options="src/test/java/com/features/sample.feature:5"**



In the preceding code, "5" is the Feature file line number where a Scenario starts.

1. If we want to run the test cases using Tags, then we run the following command and notice the output:

Copy

**mvn test -Dcucumber.options="--tags @sanity"**

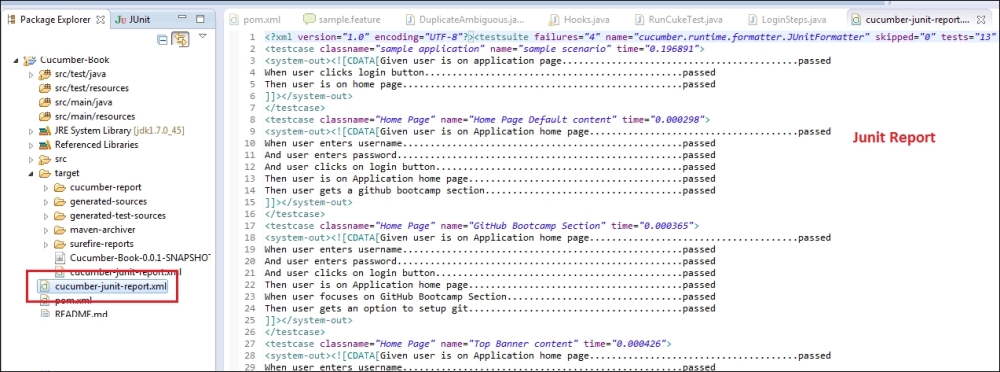
The following is the output of the preceding command:



1. If we want to generate a different report, then we can use the following command and see the JUnit report generate at the location mentioned:

Copy

**mvn test -Dcucumber.options="--plugin junit:target/cucumber-junit-report.xml"**



### How it works…

When you override the options with -Dcucumber.options, you will completely override whatever options are hardcoded in your @CucumberOptions. There is one exception to this rule, and that is the --plugin option. This...

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Did you handle duplicate step definitions**

**When there is a duplicate step definition – we get DuplicateStepDefintionFoundError**

**AmbigousStepDefitionError**

**IN order to figure out these issues before creating skeleton- we can run cucumberoptions with strict=true**

**Topics:**

**RunCukesTest**

**Config- feature, glue, dryRun, strict, plugin**

**Features**

**StepDefintions**

**Examples**

**ScenarioOutline**

**DataTable**

**DocStrings**

**Tags**

**Duplicate Step Definitions**

**Reports-plugins**

**Cucumber-Maven – run test from terminal**

**Ref:**

https://cucumber.io/docs/reference#gherkin

https://cucumber.io/docs/reference/jvm#java

https://www.packtpub.com/mapt/book/web\_development/9781785286001/2/ch02lvl1sec22/identifying-duplicate-and-ambiguous-step-definitions