|  |  |
| --- | --- |
| <strong>question</strong> <strong>mark</strong> | 3d human with a red <strong>question</strong> <strong>mark</strong> | Damián Navas ...  Behavior-driven development Training content  Adarsh Mehrotra & Rishi Khurana & Karanbir Singh | Abstract  It is envisaged that by end of this course, the readers will be familiar in writing Behavior-Driven Development (BDD) with Selenium-Java. BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing. |

Table of Contents

[1. Basics 2](#_Toc526431392)

[2. SetUp BDD with Cucumber-Maven & selenium for a java app 6](#_Toc526431393)

[3. Cucumber Project Structure 9](#_Toc526431394)

[4. Detailed Examples with ‘How to Run’ the Scenarios 12](#_Toc526431395)

Advanced usage

5. Hooks ….………………………………………………………………………………………………………………………………………. 15

6. Configuration ……………………………………………………………………………………………………….……………………… 18

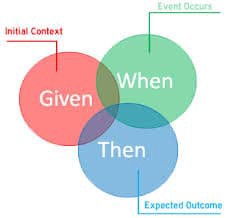
7. Capturing and saving screenshots for failed scenarios …………………………………………………………………. 20

[8. Summary 21](#_Toc526431396)

# Basics

* 1. *What is BDD*

BDD is a methodology for developing software through continuous example-based communication between developers, QAs and BAs. It involves Context (Given / the starting state), Event (When / what the user does) and Outcomes (Then / the expected results). This format/structure is called Gherkin. BDD is also referred to as Specification by Example.



* 1. *Why BDD*
     1. BDD offers better communication between developers, testers and domain experts
     2. The Given-When-then notations used in the BDD approach are closer to everyday language
     3. Tools targeting a BDD approach generally afford the automatic generation of technical and end user documentation from BDD "specifications"

* 1. *Salient Points with BDD*
     1. The terminology used to describe is "specifications of the product's behavior" rather than functional test
     2. The terminology used in BDD is explained below-
        1. **Cucumber** - Cucumber is a framework that understands Gherkin and runs the automated tests. Each step from documentation written in Gherkin should have underlying test code and test conditions.
        2. **Scenarios** – the tests/examples are called scenarios. The scenarios are a way of explaining (in plain English) how a given feature should behave in different situations or with different input parameters.
        3. **Gherkin** – the scenarios are written in English-like language called Gherkin.
        4. **Feature** file - Feature files are text files with .feature extension, which can be opened by any text editor as well as readable by any BDD-aware tool, such as Cucumber or JBehave.
        5. **Background** – ‘Background’ is a keyword used in Cucumber that defines a step or series of steps which are common to all the tests in the feature file

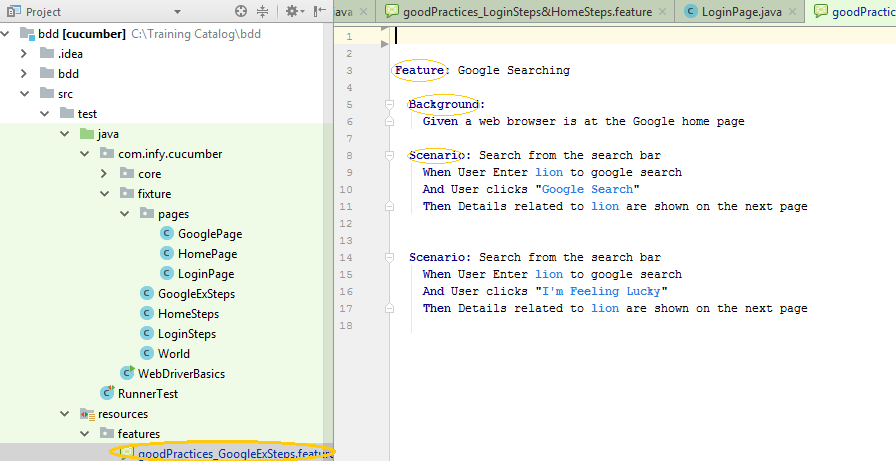


Fig.1 Shows .feature file with a Scenario and Background

* + 1. Best practices on writing a good Gherkin-
       1. Write all steps in third-person point of view
       2. Do not use partial phrases
       3. Good title to a scenario-One simple statement should be sufficient to concisely capture the intended behavior
       4. Don’t use the words “verify,” “assert,” or “should” in scenario titles. Also, “and” “or” and “but” in scenario means we are trying to test more than one scenario which is not advisable.

|  |  |
| --- | --- |
| Bad Example | Good Example |
| The user can add a cart from “Get Cart Added” button on the home page or from the “C” icon on the “BUY” page | We can create two different Scenarios: The user adds cart from “Get Cart Added” button on the home page / The user adds cart from the “C” icon on the “BUY” page |
| The “THANKS” page should be displayed after a successful logout | Logout displays the THANKS page |

* + - 1. Usage of ‘OR’ step for conditional logic does not work. When automated, every step is executed sequentially

For example- # BAD EXAMPLE!

**Feature**: Video game character Control

**Scenario**: Character jumps

**Given** a level is started

**When** the player pushes the "A" button

**Or** the player pushes the "B" button

**Then** Character jumps straight up

# Correct Example

**Feature**: Video game character Control

**Scenario**: Character jumps

**Given** a level is started

**When** the player pushes the "PUSH" button

**Then** Character jumps straight up

**Examples**: Buttons

| PUSH |

| A |

| B |

* + - 1. Proper business behavior is the most important trait and translating that to Gherkin is the most important trait to good BDD. This is illustrated with an example-

For example, let’s consider a test that searches for images of lion on Google. Here is an example of a test procedure that we need to translate to Gherkin-

* Open a web browser.
* Web browser opens successfully.
* Navigate to https://www.google.com/.
* The web page loads successfully and the Google image is visible.
* Enter “lion” in the search bar.
* Links related to “lion” are shown on the results page.
* Click on the “Images” link at the top of the results page.
* Images related to “lion” are shown on the results page.

# BAD EXAMPLE

**Feature**: Google Searching

**Scenario**: Google Image search shows pictures

**Given** the user opens a web browser

**And** the user navigates to "https://www.google.com/"

**When** the user enters "lion" into the search bar

**Then** links related to "lion" are shown on the results page

**When** the user clicks on the "Images" link at the top of the results page

**Then** images related to "lion" are shown on the results page

* This is not behavior-driven, it is still procedure-driven.
* **Given-When-Then** steps must appear in order and cannot repeat.
* A Given may not follow a When or Then, and a When may not follow a Then.
* The reason is simple: any single When-Then pair denotes an individual behavior.
* Since we have already mentioned that one scenario should cover one behavior, the above example should be split into two like the below-

**Feature**: Google Searching

**Scenario**: Search from the search bar

**Given** a web browser is at the Google home page

**When** the user enters "lion" into the search bar

**Then** links related to "lion" are shown on the results page

**Scenario**: Image search

**Given** Google search results for "lion" are shown

**When** the user clicks on the "Images" link at the top of the results page

**Then** images related to "lion" are shown on the results page

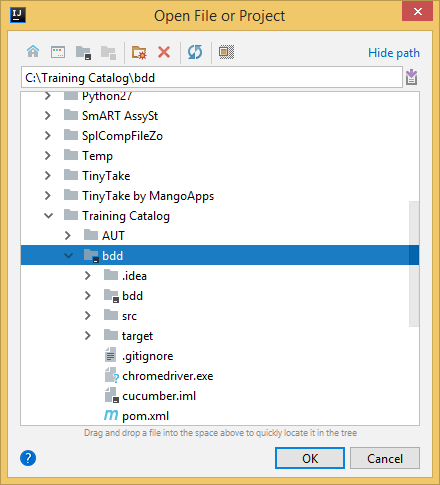
***WHAT I LEARNT***

*Basics of BDD*

*Best practice guidelines to minimize the feature smells (time dependency, inter-dependency, multi-scenario scenarios, overuse of variables)*

# SetUp BDD with Cucumber-Maven & selenium for a java app

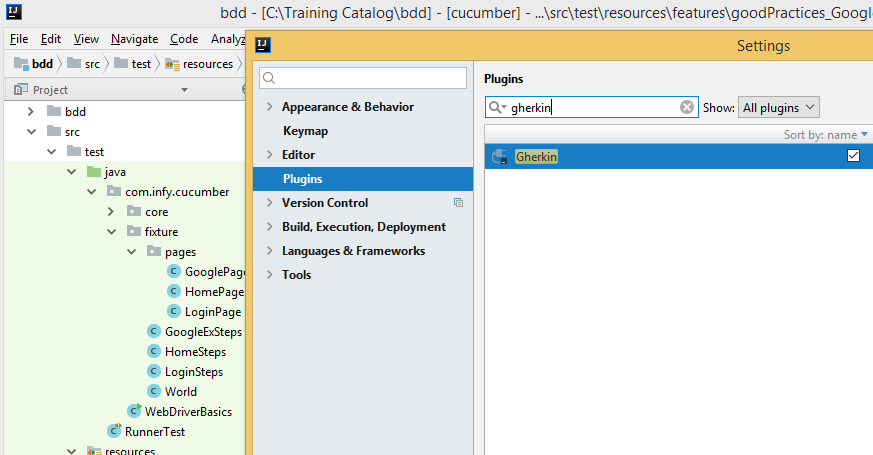
* 1. *Pre-requisites* 
     1. Install Intellij IDE (community edition 4.2)
     2. Install Java 1.8
     3. Import the project found [here](http://infygit.ad.infosys.com/eqs/bdd) and importing it by clicking File->Open



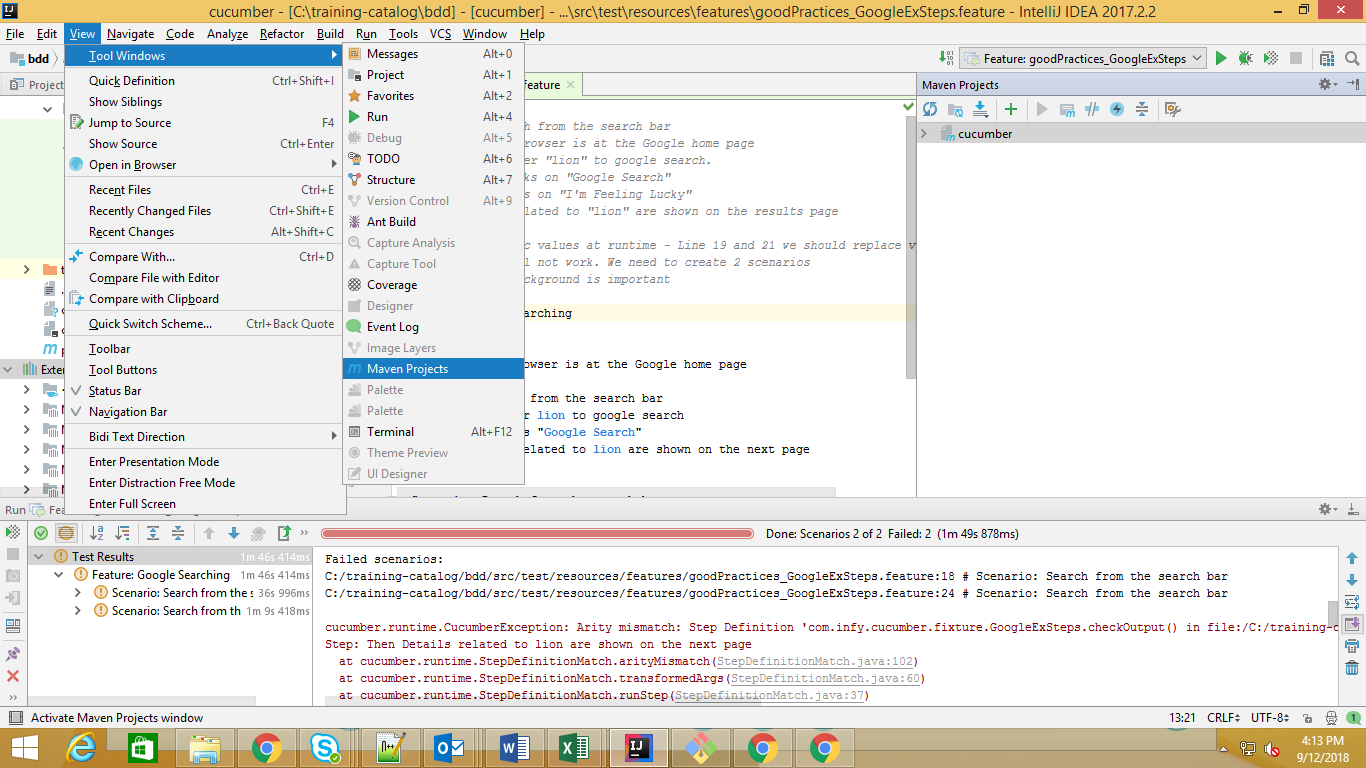
* + 1. If the JDK is not installed, then options with IntelliJ like ‘Run’, ‘Run With coverage’ will not appear. In that case, please install latest JDK from the [Oracle](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) website.
    2. It is assumed that Maven is already installed as Maven will be required to run the project. This can be tested by going to command prompt and checking for



* + 1. Install Gherkin plugin for IntelliJ that helps in intuitive styling. This can be achieved by clicking on File->Settings



* + 1. If there are Maven related dependency issues, kindly click to get Maven Projects. Then, click on the icon marked in RED to get latest. The system may need to connect on Internet for first time. —



* + 1. The set-up is complete

***WHAT I LEARNT***

*Dependencies required to run the project*

*Usage of best-of-the-breed plugins to accelerate Developer productivity*

*Running cucumber Features*

# Cucumber Project Structure

* 1. *Important Elements*
     1. POM.XML – There are certain dependencies & has all dependencies as GAVs (Groups, Artifact ID, Version) –

maven-compiler-plugin

cucumber-java

cucumber-junit

junit

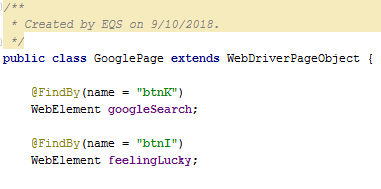
selenium-java

webdrivermanager

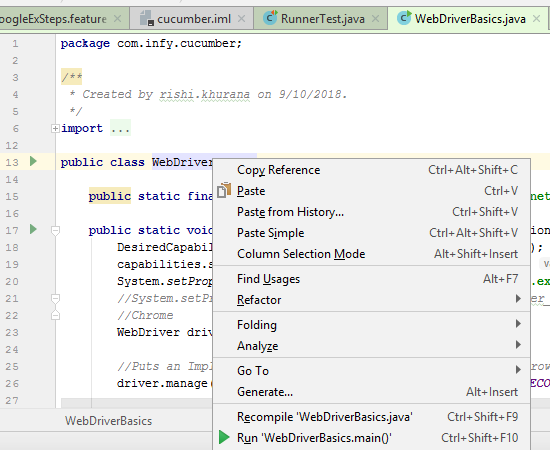
cucumber-picocontainer

* + 1. Chromedriver.exe – The browser instance of Chrome opens when the BDD tests execute. This is available locally and there is no dependency on external connection to run the Chrome.
    2. RunnerTest class - As Cucumber uses Junit, there is a need to bring in ‘RunnerTest’ class. This class uses the Junit annotation @RunWith(), so that jUnit identifies the ‘RunnerTest’ class. This is a starting point for Junit to start executing the tests. (Location - src🡪test🡪java)
    3. Resources Folder – has a folder called ‘features’ that contains all the feature files and chromedrover.exe to utilize the browser locally.
    4. ‘core folder’ - There are helper classes under ‘core’ folder namely-
       - 1. BrowserDriver: This helper class is created to instantiate different type of browsers. All logic associated with initializing browser is segregated in BrowserDriver.java.
         2. SeleniumDriverSuperPower: This is a public class, all the methods are static and can be invoked without creating an instance of class. One of major use case of this class is setting up of browsers.
         3. WebDriverPageObject: This class is extended in every page object class and methods are consumed.

For Example: Method with name setTextAndUnfocus() is used across to set text in a Web Element - text box.



* + 1. The ‘webdriverBasics’ class has been created for the sole purpose of educating the team on the nuances of using browser, navigation & other relevant methods on finding the right elements. The class can be directly run by right-clicking on ‘Run WebDriverBasics.main()’ –



* + 1. Pages folder – The features file *‘goodPractices\_GoogleExSteps.feature’* has been named so that the co-related class file *‘GoogleExSteps’* that contains the methods. As a best practice, the web-elements should not be in this class file but managed separately. The web-elements for *‘GoogleExSteps’* class has all web-elements under pages🡪 GooglePage. All classes under the ‘pages’ folder extend WebDriverPageObject. Here is a dependency tree depiction-

Features🡪pages🡪core

* + 1. PageFactory class – Used in classes under the ‘pages’ directory. This is another best practice on Selenium. The **PageFactory** Class in Selenium is an extension to the Page Object design pattern. It is used to initialize the elements of the Page Object or instantiate the Page Objects itself.

PageFactory.*initElements*(driver, **this**);

* + 1. Usage of cucumber-picocontainer – as we have seen from above examples, we create a scenario in [Gherkin](https://github.com/cucumber/cucumber/wiki/Gherkin) using steps. The step may be dependent on previous step. We should have some mechanism to share state between these steps. The elegant way to share state between steps is via picoContainer. We define a constructor that requires an object that PicoContainer can create and inject.
* Add the following to POM.XML

<**dependency**>  
 <**groupId**>info.cukes</**groupId**>  
 <**artifactId**>cucumber-picocontainer</**artifactId**>  
 <**version**>1.2.5</**version**>  
 <**scope**>test</**scope**>  
</**dependency**>

* Create a class called World that contains list of all the ‘Pages’. If you look into any stepdef class, say ‘LoginSteps’; this has constructor for the class-

**public** LoginSteps(World world){  
 **this**.**world** = world;  
}

* When loginFuntionality.feature calls ‘LoginSteps’ class, we can call methods of HomePage class by world.HomePage.<>-

@Then(**"^Home page should be displayed$"**)  
**public void** verify\_user\_login\_Is\_Success\_or\_not() **throws** Throwable {  
 **world**.**homePage**.verify();  
}

* + 1. The current set-up initiates CHROME as the local web-driver. This can be referred in ‘BrowserDriver’ class. The switch-case statement conveys that same set-up for Firefox or any other browser as has been done for CHROME.

***WHAT I LEARNT***

*Folder set-up with Cucumber-jUnit*

*Best practices on using Selenium classes*

# Detailed Examples with ‘How to Run’ the Scenarios

* 1. *Details of Examples*
     1. Selenium webdriver basics

Feature file- NA

Test class- WebDriverBasics

Run the classfile to open the webpage. The website is launched and the browser can be maximized. We use ‘driver.findElement’ to use ‘MathCalculator’. Further, ‘PercentCalculator’ is invoked. The values can be entered using .sendKeys and then this clicks ‘Calculate’ button to get the result. The result is also published in the logs and then browser is closed.

* + 1. Understanding & running the feature file ‘Google Searching feature’

Feature file- goodPractices\_GoogleExSteps.feature

Test class- GoogleExSteps class

When we right-click and run the file, the following flow kicks off-

The Scenario’s first line starts getting executed-

**Scenario:** Search from the search bar  
 **When** User Enter **lion** to google search

where the control passes to GoogleExSteps.java. We can press CTRL+ and click on ‘Enter lion to google search to navigate to the below-

@When(**"^User Enter ([^\"]\*) to google search$"**)  
**public void** enter(String textToSearch){  
 **world**.**googlePage**.enterSearchText(textToSearch);  
}

As explained earlier, the ‘World’ class has all the details of all pages so the googlePage ‘enterSearchtext’ method gets called-

**public void** enterSearchText(String textToSearch){  
 setTextAndUnfocus(**searchTextBox**,textToSearch);  
}

SinceGooglePage.java extends WebDriverPageObject, the following method gets called –

**public void** setTextAndUnfocus(WebElement webElement, String text) {  
 webElement = setText(webElement, text);  
 webElement.sendKeys(Keys.***TAB***);  
}

The webElement contains Chromedriver details:chrome, xpath detail and text as ‘lion’. webElement.sendKeys goes ahead and enters the text called ‘lion’ in the Chrome’s google search page.

The second scenario will run follow similar pattern as above in execution-

**Scenario:** Search from the search bar  
 **When** User Enter **lion** to google search  
 **And** User clicks "**I'm Feeling Lucky**"  
 **Then** Details related to **lion** are shown on the next page

* + 1. Understanding and running the feature file ‘goodPractices\_LoginSteps&HomeSteps’

Feature file- goodPractices\_LoginSteps&HomeSteps.feature

Test class- LoginSteps-> LoginPage class & HomeSteps->HomePage class

Let us look at the scenario in the feature file which is-

**Scenario Outline:** As a DevOps Engineer, I want to login to timespoints.com.  
**Given** User navigates to application and login with username "**<username>**" and password "**<password>**"  
**Then** Home page should be displayed.  
**And** Log out from the application

In the above scenario, the username and password are placeholders where the values will come from the tag like below-

**Examples:  
 |***username* **|***password* **|  
 |satpalsahab@gmail.com |guess@123|  
 |satpalsahab@gmail.com |guess@123|**

Coming back to the scenario, click on the below with CTRL key to navigate to the declaration –

**Given** User navigates to application and login with username "**<username>**" and password "**<password>**"

The control flows to the ‘LoginSteps’ class at the following method-

@When(**"^User navigates to application and login with username \"(.\*)\" and password \"(.\*)\"$"**)  
**public void** user\_login\_to\_timespoints(String username, String password) **throws** Throwable {  
//WebDriver object is been initialized here

**driver** = SeleniumDriverSuperPower.*setUpDriver*();

**//**To open an URL via initialized driver object, one way is to use the GET method.  
 **driver**.get(***APP\_URL***);

// The login page is initialized here.  The login method can be invoked to login to an application.  
 **world**.**loginPage** = **new** LoginPage(**driver**);

//Invoking login method of loginPage object to access an application.   
 **world**.**homePage** = **world**.**loginPage**.login(username,password);  
}

The login method on ‘loginPage’ class gets called that calls ‘setTextAndUnfocus’ under WebDriverPageObject.java. the username and password passed on under ‘Examples’ on the feature file get typed on text boxes and then page submitted for Login.

The other lines on the Scenario follow similar flow.

# Hooks

* 1. What are Cucumber Hooks ?

Cucumber hook are blocks of code which allows to manage the code workflow and helps in reducing the code redundancy. It is an unseen step which facilitates to execute some piece of code before and after each scenario. Cucumber hooks can be defined anywhere in your project or step definition layers, using the methods **@Before** and **@After**.

* 1. Why to use Cucumber Hooks?

In order to bring optimization, hooks can be utilized. Cucumber supports only two types of hooks (Before and After). Method/function/piece of code, defined within Before and After hooks, always run, even if the scenario gets passed or failed.

While working on different scenarios, you might have encountered situation where you need to perform various prerequisite steps before testing any scenario. For e.g.

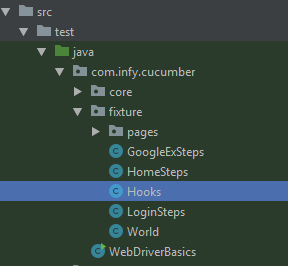
* Starting a webdriver
* Setting up browser cookies
* Opening landing page of your application under test

In the same way, there are situations where we want to perform some operations after test execution. For e.g.

* Taking screenshot for failed scenarios
* Killing the webdriver
* Clearing the test data from database
* Closing DB connections
  1. How to use Cucumber Hooks?

For better understanding, please refer to Hooks.java class in the training project.

Here is a dependency tree depiction



In Hooks class, a function with name **closeBrowser** is defined with @After annotation. The purpose of this function is to kill driver instance. **After hook will execute** function closeBrowser each time after test execution, **even in case of test failure**.

import cucumber.api.java.After;

public class Hooks {  
  
 @After  
 public void closeBrowser(Scenario scenario) {  
 WebDriver driver = BrowserDriver.*getDriverObject*();

try{  
 System.*out*.println("Calling After hooks");  
 driver.close();  
 driver.quit();  
 } catch (Exception ex){ System.*err*.println("Exception while closing browser");  
 }  
 }  
}

NOTE:   
Please make sure that the package import statements should be **import cucumber.api.java.After; and import cucumber.api.java.Before;**

# Configuration

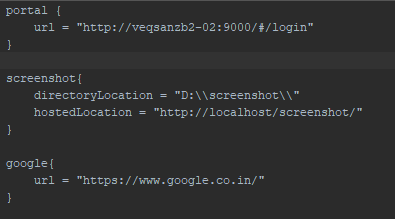
* 1. About Typesafe config

In order to store the configurable parameters of an application, files with .**conf** extension are mainly used in Java related technologies. The separation of configuration from code is a good practice that makes our system customizable as we can load different configuration values at run time. The Typesafe library limits itself to config files and has nice support for merging configurations. Let’s dive a little deeper to understand the concept.  
  
Add following dependency to POM.xml

<dependency>  
 <groupId>com.typesafe</groupId>  
 <artifactId>config</artifactId>  
 <version>1.3.4</version>  
</dependency>

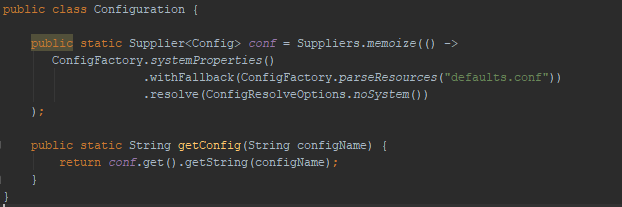
* 1. Loading Typesafe config from a resource or configuration files

To define configuration values, we need to create a file with .conf extension. Let’s call it default.conf.



We can now parse the file and to get the obtained configuration in our script:

To achieve the same, we have a created a generic class named Configuration to load all values. By default, the ConfigFactory looks for a configuration file called application.conf. but to make everything explicit, it would be good to specify config name which is being loaded



This aforementioned code will setup two global configurations into single conf object

* System: This will have all JVM properties as well as system level properties
* Properties: All the configurations defined in default.conf will be merged

So to say, when searching for any property, it will first search the system properties, then traverse down the fallbacks until a value is found.

Also, the concept of Memoization increases efficiency by reducing the number of calls to the expensive code. Memoization is similar to caching with regards to memory storage. However, caching is a more generic term that addresses the problem at the level of class instantiation, object retrieval, or content retrieval,whereas**,** memoization solves the problem at the level of method/function execution.

When we want to execute the memoized method, we can simply call the *get* method of the returned *Supplier*. Depending on whether the method’s return value exists in memory, the get method will either return the in-memory value or execute the memoized method and pass the return value to the caller.

6.3 Fetching values from configuration values

To load configuration values, we can call static function named getConfig. This function has one string parameter and value can be like a.b.c . For E.g.

Portal url can be fetched from default.config with single line of code.

Configuration.*getConfig*("portal.url");

# Capturing and saving screenshot for failed scenarios

* 1. How to check scenario is failed?

Whenever any scenario fail, we need to know where was the error in the script. Solution for this is to capture a screenshot of a webpage when the test fails and it would be easy to identify where exactly the script got failed by seeing the screenshot.

To achieve this, we have implemented @After hook so that whenever any scenario fails screenshot is captured.

@After  
public void closeBrowser(Scenario scenario) {  
 WebDriver driver = BrowserDriver.*getDriverObject*();  
 try{ if (scenario.isFailed()){  
 try {  
 if (driver != null)  
 {  
 String screenShotDir = Configuration.*getConfig*("screenshot.directoryLocation");  
 String screenShotId = Long.*toString*(System.*nanoTime*());   
 BrowserDriver.*saveScreenShot*(driver, new File(screenShotDir + screenShotId +".png"));  
 }  
 } catch (Exception ex) {  
 }  
 }  
  
 System.*out*.println("Calling After hooks");  
 driver.close();  
 driver.quit();  
 }catch (Exception ex){ System.*err*.println("Exception while closing browser");  
 }  
}

In aforementioned code, we first verify that whether scenario is failed or not with the following statement



If this statement returns true, we proceed further to check whether this is a selenium specific scenario or not, if driver object is not null, it means that, screenshot for the application under test can be captured.

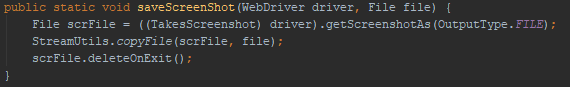
After that, we figure out the directory location by calling getConfig method of the Configuration class. This function returns the directory location path where screenshot file or screenshot is supposed to be saved.



To make all screenshot filenames unique, we call **nanoTime()** method of Java System class which returns high resolution time source in nanoseconds of running Java virtual machine.

* 1. Capturing and saving screenshot

To make the solution more generic, code to capture and save screenshot can be placed into separate method



This method has two arguments

* WebDriver: This is selenium WebDriver object
* Screenshot filename: Screenshot file path to be saved.

Selenium provides an interface named **TakesScreenshot** which helps to take screenshot during execution of automation script. TakesScreenshot interface has a method named **getScreenshotAs** which can capture a screenshot and store it in different ways in specified locations.

**OutputType** is an interface which provides you options to take screenshot in different type such as FILE, BASE64, BYTES and class. The most used type is FILE.

When webdriver takes a screenshot, it is stored in temp folder which is cleaned up as soon as execution is over. So we cannot find it after execution. This is the reason we need to copy screenshot file from temp folder to a permanent folder.

# Summary

The above examples will help create a basic infrastructure on writing Behavior-driven tests using Cucumber. Also, the application of BDD principles at the coding level will help us write more maintainable and better documented code.