WIREMOCK

Mock class which defines a server, it could be used with httpClient to simulate the whole process of request, response and testing and check how it works.

```
WireMockServer wireMockServer = new WireMockServer();
```

```
wireMockServer.start();
configureFor("localhost", 8080);
stubFor(get(urlEqualTo("/baeldung")).willReturn(aResponse().withBody("Welcome to Baeldung!")));
 :
 : //you can also mock a client trying to connect to the mocked server
 :
wireMockServer.stop();
```

Find out more about wiremock here: https://www.baeldung.com/introduction-to-wiremock

CUCUMBER

It is based on the execution of Gherkin files, this kind of files significantly simplifies the way in which we test APIs by using plain text and logical orders to define what we want to test.

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

Find out more about the Gherkin syntax here: https://cucumber.io/docs/gherkin/reference/

This file will have the *.feature* extension, a plain text file with that extension will be saved in *src/main/resources*, this process will be easier once we get into Karate, creating a maven project with the Karate artefact will organise the directories with everything needed. Another option will be to add manually the files and the dependencies to our current project.

Case of the cucumber dependencies:

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
```

```
<!--JUnit dependencies-->

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>1.2.4</version>
</dependency>
```

In the *.feature* file each scenario will behave as a block testing something, in cucumber there is no parallel execution among scenarios. The orange words within the scenario are called keywords and with them you can choose as the developer what the scenario is going to test and how is going to test it (to find out what any word does you may find this useful: https://cucumber.io/docs/gherkin/reference/#examples). Everything that goes after a keyword is used as a little description but is also used to match with methods called *step definitions*, this is considered as the main disadvantage of cucumber because it takes some time to the developer to write this methods, Karate solves this problem making it easier avoiding this step definitions.
Should we know that for testing HTTP requests and responses Karate is a way simpler tool than Cucumber. However, the fact that the developer could write the step definitions allows more personalization of what you want to test as well as a more natural way to address complex testing problems.

For example:

```
Given I have registered a course in Bandung
```

Its step definition:

```
@Given("I have registered a course in Bandung")
public void verify Account() {
    // method implementation
}
```

For a real example of an implementation of cucumber with a get and a post request you may want to go over this post: https://www.baeldung.com/cucumber-rest-api-testing

KARATE

Karate allows developers to write tests of HTTP requests and checks if they get the expected response.

Karate derives from Cucumber, one of its most important differences is that the *.feature* files do not need to have *steps definitions* associated with each keyword in every scenario.

Tutorial to build your own project

First of all we may want to include the dependencies of karate in our current project:

```
<dependency>
    <groupId>com.intuit.karate</groupId>
    <artifactId>karate-apache</artifactId>
    <version>0.6.0</version>
</dependency>
```

```
<dependency>
    <groupId>com.intuit.karate</groupId>
    <artifactId>karate-junit4</artifactId>
    <version>0.6.0</version>
</dependency>
```

Another way might be to create a new maven project with the archetype of karate. First, open your terminal and go to your workspace. Once in there, manually create the archetype by using this command:

```
mvn archetype:generate \
-DarchetypeGroupId=com.intuit.karate \
-DarchetypeArtifactId=karate-archetype \
-DarchetypeVersion=0.9.4 \
-DgroupId=com.yapiko \
-DartifactId=YapikoKarate
```

The directory structure will be similar to this one but changing users for companies:

```
src/test/java
    |
    +-- karate-config.js
    +-- logback-test.xml
    |
    \-- examples
        \-- companies
        |   |
        |   +-- companies.feature
        |   +-- CompaniesRunner.java
        |
        +-- ExamplesTest.java (this one is useless)
```

Like in Cucumber in order to run our tests a *valueRunner* class will be needed. Instead of using an actual API REST we are going to mock one with WireMock, thus practising the concepts previously explained.

First of all, we need to add the dependencies:

```
...
<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <java.version>1.8</java.version>
   <maven.compiler.version>3.6.0</maven.compiler.version>
   <karate.version>0.6.2</karate.version>
   <wiremock.version>2.14.0</wiremock.version>
   <junit.version>4.12</junit.version>
</properties>
<dependencies>
   <dependency>
       <groupId>com.intuit.karate</groupId>
       <artifactId>karate-apache</artifactId>
       <version>${karate.version}</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>com.intuit.karate</groupId>
       <artifactId>karate-junit4</artifactId>
       <version>${karate.version}</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>com.github.tomakehurst</groupId>
       <artifactId>wiremock</artifactId>
       <version>${wiremock.version}</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>${junit.version}</version>
       <scope>test</scope>
   </dependency>
</dependencies>
...
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19.1</version>
   <configuration> <!--you must install this configuration to make it run-->
     <testFailureIgnore>true</testFailureIgnore>
    </configuration>
</plugin>
```

Once in here, let's complete the runner class according to how we want our mock server to answer, that will depend on the kind of request we want to test, **in this first case we will be testing a GET method**.

```java
package examples.companies;

import com.github.tomakehurst.wiremock.WireMockServer;
import com.intuit.karate.junit4.Karate;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;

import static com.github.tomakehurst.wiremock.client.WireMock.aResponse;
import static com.github.tomakehurst.wiremock.client.WireMock.configureFor;
import static com.github.tomakehurst.wiremock.client.WireMock.get;
import static com.github.tomakehurst.wiremock.client.WireMock.stubFor;
import static com.github.tomakehurst.wiremock.client.WireMock.urlEqualTo;

@RunWith(Karate.class)
public class CompaniesRunner {

   private static final WireMockServer wireMockServer = new WireMockServer();
   private static final String URL = "/companies";

   @BeforeClass
   public static void setUp() {
       wireMockServer.start(); //starting the server
       configureFor("localhost", 8080);
       stubForGetAllCompanies();
   }

   private static void stubForGetAllCompanies() {
       stubFor(get(urlEqualTo(URL))
               .willReturn(aResponse()
                       .withStatus(200)
                       .withHeader("Content-Type", "application/json")
                       .withBody(getAllCompanies())));
   }

   private static String getAllCompanies() {
       return "[" + getParadigmaDigitalCompany() + ", " + getMinsaitCompany() + "]";
   }

   private static String getParadigmaDigitalCompany() {
       return "{" +
               " \"cif\":\"B84946656\"," +
               " \"name\":\"Paradigma Digital\"," +
               " \"username\":\"paradigmadigital\"," +
               " \"email\":\"info@paradigmadigital.com\"," +
               " \"address\":{" +
               "    \"street\":\"Atica 4, Via de las Dos Castillas\"," +
               "    \"suite\":\"33\"," +
               "    \"city\":\"Pozuelo de Alarcon, Madrid\"," +
               "    \"zipcode\":\"28224\"" +
               " }," +
               " \"website\":\"https://www.paradigmadigital.com\"" +
               "}";
   }
```

```
    private static String getMinsaitCompany() {
        return "{" +
                " \"cif\":\"B82627019\"," +
                " \"name\":\"Minsait by Indra\"," +
                " \"username\":\"minsaitbyindra\"," +
                " \"email\":\"info@minsait.com\"," +
                " \"address\":{" +
                "    \"street\":\"Av. de Bruselas\"," +
                "    \"suite\":\"35\"," +
                "    \"city\":\"Alcobendas, Madrid\"," +
                "    \"zipcode\":\"28108\"" +
                " }," +
                " \"website\":\"https://www.minsait.com\"" +
                "}";
    }

    @AfterClass
    public static void tearDown() {
        wireMockServer.stop();
    }
}
```

Mocking is good as a proof of concept and a way to start learning this new way of testing but in real life cases this runner class will contain something similar to a microservice or could be even empty if we are testing a third party API.

Finally, we will test the service in the *.feature* file, you can obviously extend the grade of complexity of these tests but this might be enough to get the concept for now:

```
#Sample Feature Definition Template
Feature: testing company mock web services

 Background:
  * url 'http://localhost:8080'

 Scenario: get all companies

  Given path 'companies'
  When method get
  Then status 200
  And match $ == '#[2]'
   # checks that the body of the response  has two elements, as we have stubbed for two companies
  And match each $ contains {name: '#notnull'}
  And match each $ contains {email: '#notnull'}
  #checks if each element has this fields and are not null
```

The best way to run these is using the following command on the console:

```
mvn test -Dtest=CompaniesRunner
```

The fact of running the test in the terminal will allow us to isolate the execution of the concrete *.feature* we are interested in.

By now, you may be wondering how the runner class and the *.feature* communicate with each other, this is simple, every time you execute a runner class karate is internally in charge of running all the *.feature* reachable by this runner class. Well, here, it is pretty important to be careful with the folder hierarchy of your project and its naming conventions. For example, if you have a runner class and directories at the same level which contain a *.feature* and its respective runner class, the execution of the high level runner class will implicate the execution of all the *.feature* underneath it.

It is important to understand this problem, but this has a solution *"@tags"* which allow you to select the scenarios you want to run, thus avoiding to create too many *.feature* files which could be tested with the same runner.

```
mvn test -Dkarate.options="--tags @test1" -Dtest=CompaniesRunner
```

Find out more here: https://github.com/intuit/karate#test-suites

After typing that command, we will find some kind of table in the execution telling us the status of the tests. Furthermore, Karate creates and *.html* which eases the visualization of the tests.

You will find a complete example with the get and post methods in this repository: https://github.com/YMRodriguez/Karate_yapiko

REFERENCES:
A collection of references deployed along the document

- https://www.baeldung.com/introduction-to-wiremock post with several examples of the different functionalities of wiremock
- https://www.paradigmadigital.com/dev/probar-servicios-web-facil-practicas-karate/ hands on example of a karate api testing (spanish)
- https://www.baeldung.com/karate-rest-api-testing hands on example in english of a karate api testing
- https://www.baeldung.com/cucumber-rest-api-testing hands on example of cucumber api testing
- http://get.mocklab.io/?utm_source=wiremock.org&utm_medium=primary-nav&utm_campaign=mocklab interesting tool