Author: Dennys Barros
Email: dennys.sbarros@gmail.com
LinkedIn: https://www.linkedin.com/in/dennys-barros-05425349/

Technologies used:

- Java
- Spring Boot
- JUnit
- Used a Macbook (Catalina 10.15.7)
- IntelliJ

Requirements:

1. Maven needs to be installed (so it can be executed via command line)

How to run it (using IDE):

1. Import the project into a preferred IDE
2. Find the pom.xml file > right click > Maven > Reload projects (it will download and configure all needed dependencies)
3. To run the feature itself, go to src > main > java > Runner.java and run it
4. To run the tests, go to src > test > java > BlogPostsServiceTest.java and run it

If you want to run it via command line:

1. Go to the source project folder via command line
2. *mvn spring-boot:run* (to run the application)
3. *sh runTests.sh* (to run the tests)

The report of the test execution will be saved at *<src_folder>/testReport/surefire-report.html*

Code structure:

```
src/
  main/
    java/
      model/
        Address.java
        BlogPosts.java
        Company.java
        GeoLocation.java
        Users.java
      services/
        BlogPostsService.java
      utils/
        Constants.java
      Runner.java
test/
    java/
      BlogPostsServiceTest.java
```

Packages definition:

- **model -** class which specifies Users and BlogPosts objects. The model was based on the structure of the payload from https://jsonplaceholder.typicode.com/posts and https://jsonplaceholder.typicode.com/users

- **services -** implements the methods to perform GET requests to https://jsonplaceholder.typicode.com and manipulate the response in order to get what was asked by the User Story

- **utils -** contains a Constants class

- **tests -** contains the tests of the application.

Implementation decisions:

I decided to use Spring since it is a well-known framework to create and manipulate API services.

RestTemplate web client was used to perform the HTTP requests. The methods exchange() and getForObject() helped me to store the API response into BlogPosts and Users objects. With these objects, I was able to manipulate the data and retrieve only the user who wrote a blog based on its title, and then get all posts from that user.

Five methods were created in BlogPostsService.java to perform what was asked by the user story.

I decided to create a main method to run it and confirm the application is running correctly.

The class BlogPostsServiceTest.java contains the tests I created to make the implementation more robust:

The name convention for the tests followed the pattern:

- MethodName_StateUnderTest_ExpectedBehavior

The following tests focused more on the response status of the request:

- getEntity_existingPostEndPoint_statusOK()
- getEntity_existingUserEndPoint_statusOK()
- getEntity_postDoesntExist_status404()
- getUserById_inexistentUser_404NotFound()

The following tests focused on the results from the API:

- getUserById_userId3_nameClementine()
- getUserIdByPostTitle_eumEtEstOccaecati_1()
- getUserIdByPostTitle_titleDoesntExist_0()

To also confirm the results of the tests were correct, I double-checked the results using Postman.

More tests could be created to test the API results, but the methods would be quite similar. Also, as we are consuming an external API, unit tests with mocked data could be created (a developer would create such tests). The QA could help developers by suggesting scenarios to be covered and reviewing the tests.