HW1: Mid-term assignment report

Gonçalo Matos [92972], v2021-05-14

1 Introduction

1.1 Overview of the work

This report presents the outcomes of the Software Quality And Tests midterm individual project, in which I developed a Spring Boot application for meteorology and UV index alerts dissemination gotten from Instituto Português do Mar e da Atmosfera public API.

The main functional component of the app is the API, which mimics the external one, but it also provides a simple web page to test the previous and analyse its usage statistics. Despite that and because this work focus was to apply the expertise acquired on the curricular unit, most of the effort was put in developing tests to cover all the levels in the right proportion.

1.2 Current limitations

With a due date in the middle of other subjects projects, it was not easy to find time to develop this project. Even though I think I have achieved what I was proposed, if I had more time I would like to

redesign the web page interface and improve the API endpoints return by doing a better processing of the external API data.

2 Product specification

2.1 Functional scope and supported interactions

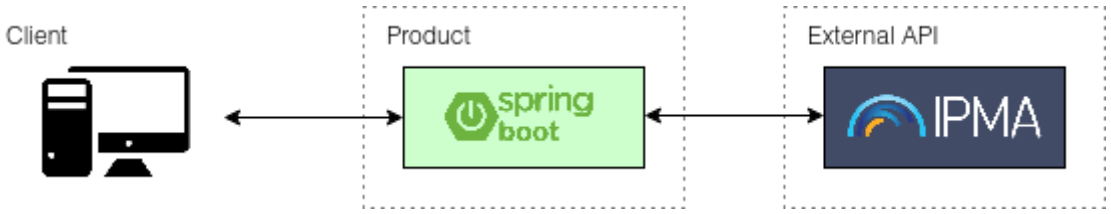
Like I have introduced before, the product has two forms of interaction: the web page (available at the root path - /) and the API (available at /api).

The web page as it is, although it provides all the alerts information, has a poor design and shows debug information that is not required by an end user. It's main purpose is to be used by a developer. However, with some minor changes it was suitable for all the audiences.

The API provides the same information, but through a REST endpoint, which allows for direct calls from end users, but as it is not common for anyone to consume an API directly, mainly, it allows for consumption by any service that would like to integrate the alerts, like a weather application. In the last scenario, the end user would be anyone with a phone and internet access, but there would be a developer (that builds the app that consumes the product API) as a middle man.

2.2 System architecture

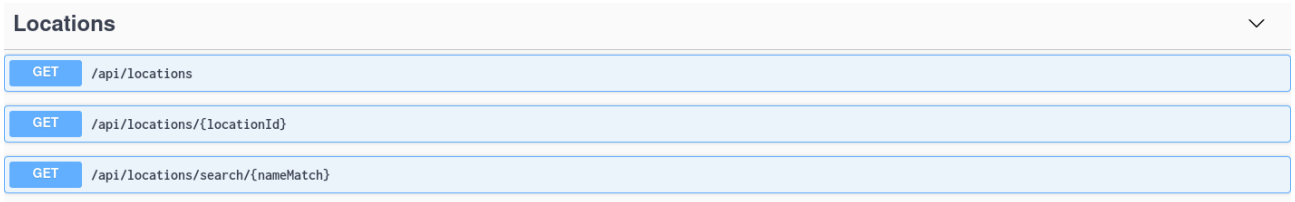
The system was implemented in Java, using Spring Boot framework. It essentially mimics some endpoints of IPMA public API, where it gets all the alerts information. To avoid repeated requests in a short timespan it also maintains an in memory cache, that it checks every time a client asks for any data.



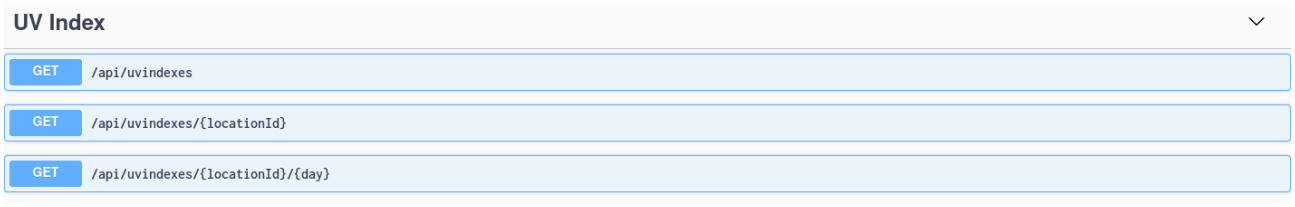
2.3 API for developers

The API has three main services:

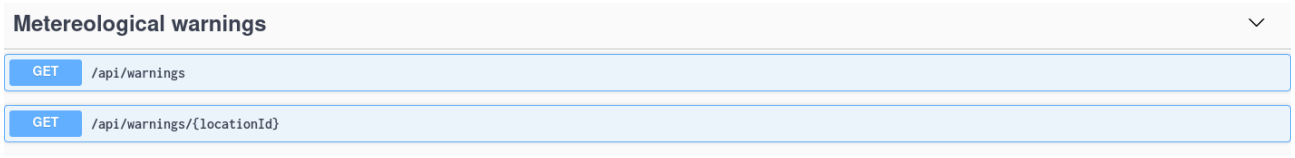
1. Locations, that provides the information about the locations the API covers;



2. UV Index, that provides information about the ultraviolet index and allows filtering by location and day;



3. Meteorological warnings, that provides the warnings and allows filtering by location;



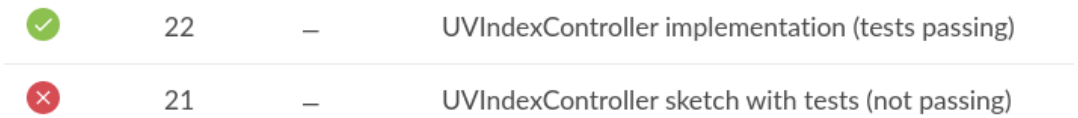
All of the endpoints return a JSON object with a data attribute which consists of an array with objects that vary according to the service being consumed. However, they all have in common 5 attributes that return information about the cache usage. Each one of the three services has an independent cache, so the metrics for each of them are also independent. Below is an example of an object returned by an endpoint, with all the mentioned attributes.

```
{
  "data": [
  ],
  "cacheHits": 0,
  "cacheMisses": 0,
  "cacheExpired": 0,
  "cacheSize": 0,
  "requests": 0
}
```

3 Quality assurance

3.1 Overall strategy for testing

During the development process I have tried to follow a TDD approach. However, as the time was short I did not manage to apply it 100% of the time. Below is an example of this approach, first I developed the test and then I have implemented the code to make it pass.



To build the test battery (71 tests) I have used several testing tools, which allowed me to cover all the scenarios: Mockito, JUnit, AssertJ, Selenium, SonarQube, Jacoco, SonarLint inspection tool plugin for IntelliJ.

The tests directory was structured by subfolders that allow for a better understanding.

3.2 Unit and integration testing

For every API endpoint that I have created, I have implemented the following unit and integration tests:

- Unit test for the controller with mocking of the service (controller folder);
- Unit test for the service, with and without (2 for each service) mocking of the external API (service folder);
- Integration test to validate the application API (API folder), with the full web context.

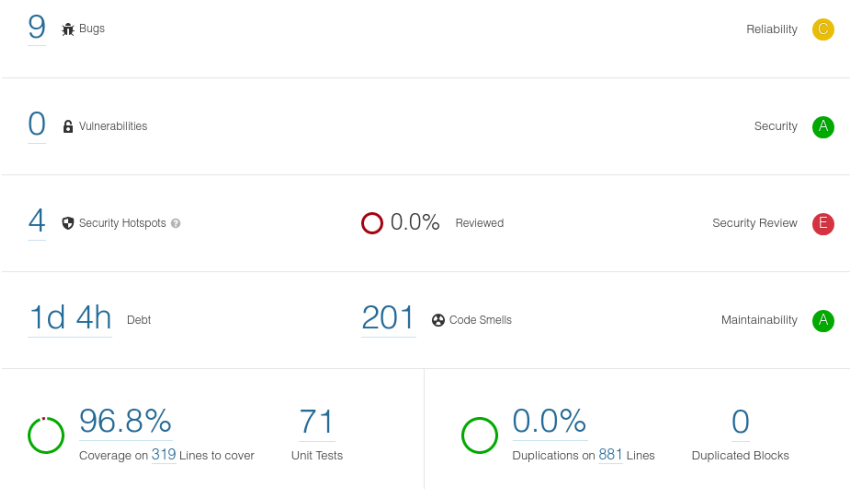
As I have developed the cache class from scratch, I have also implemented a Unit Test for it (cache folder).

3.3 Functional testing

For functional testing I have used Selenium automation ecosystem, which allowed me to replicate the user interaction with the web page and assure that it meets the expected requirements (folder views). To make the code more legible, avoid duplication and ease maintenance, I have followed the Page Object Model pattern defining the test at views.ViewSeleniumTest and its implementation at views.ViewObject.

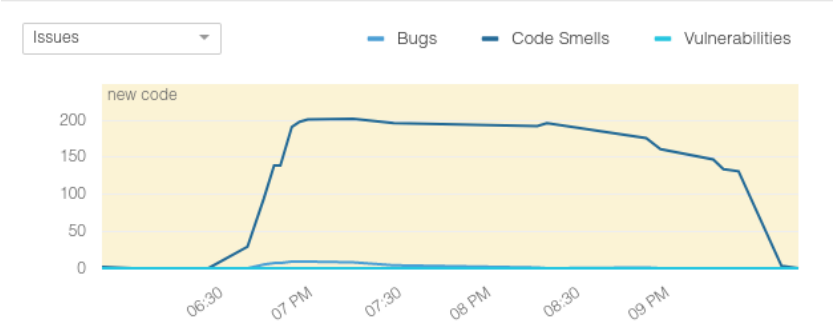
3.4 Static code analysis

As a human programmer I am not free of mistakes. To help me out finding the bugs and code smells that I have introduced by mistake I used SonarQube. Actually, I have used System.out.println method a lot to make sure I started with a bad score. I have only looked at my score when I finished the code development process and it showed some bugs and security hotspots and lots of code smells which summed up a debt of more than a day! The only good metric was the coverage, as I had all the tests implemented.



Even though this project does not handle sensitive data, I proposed myself the goal of achieving the SonarQube default quality gate, which asks for a coverage >= 80%, duplicated lines <= 3%, maintainability, reliability and security ratings of A and 100% of security hotspots reviewed.

It did not reveal a task as difficult as the 1 day of debt could indicate. Most of the errors were common to all the classes, as the code was very similar for each of the 3 endpoints that the system provides (both locations, UV and meteorology alerts have models with JSON mapping properties, controllers, services, and similar tests for all the previous). Eventually, with help of the IntelliJ replace all function and SonarLint inspection tool plugin I managed to correct all of them and not only achieve the quality gate, but to have the maximum score in all metrics (except for the coverage, which is difficult to achieve 100%). I have started by the security problems, then the bugs and finally the code smells, from the criticals to the minors. The evolution of my code issues number is visible in the graphic below.

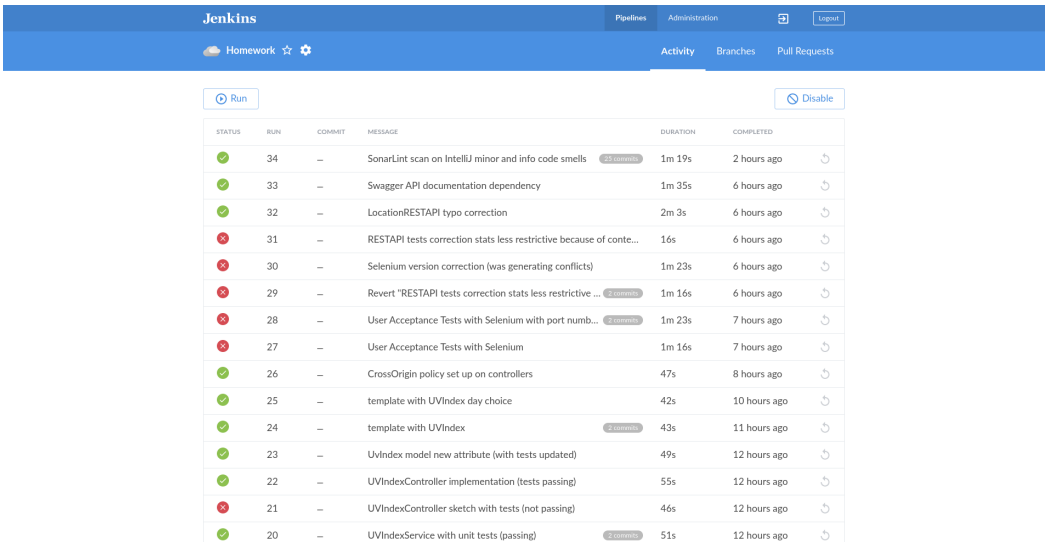


Despite that most of them were common mistakes, like extra semicolons, the premetitated System.out.println, some of the problems that this tool warned me about were new to me. Here are some examples:

- I was logging a String with String.format. This is not a good practice as the String inside .format is always generated by Java, even if the logging level does not match and the String is not logged. I found out that org.slf4j.Logger supported a syntax similar to String.format that only computed the String if the logging level matches;
- JUnit tests should have the same visibility as their package. I was making the public.

3.5 Continuous integration pipeline

To assure that all my commits were generating code that was passing the tests and to avoid me running them manually every time I did a change, I used Jenkins to create an automation pipeline. Every 5 minutes it checked if my local repository had new commits and if so, it runned the tests to make sure I did not break anything. Below is a screenshot of Jenkins Blue interface with the results of the last tests.



4 References & resources

Project resources

- Video demo: will be available at /homework inside my GitHub repository, with name **demo.mp4**;

Reference materials

API IPMA: https://api.ipma.pt/
TQS and IES class materials (slides and practical guides)
Spring official documentation: https://spring.io/
StackOverflow for doubts and bug solving: https://stackoverflow.com/
Baeldung for tutorials on Spring Boot and Java testing tools: https://www.baeldung.com/