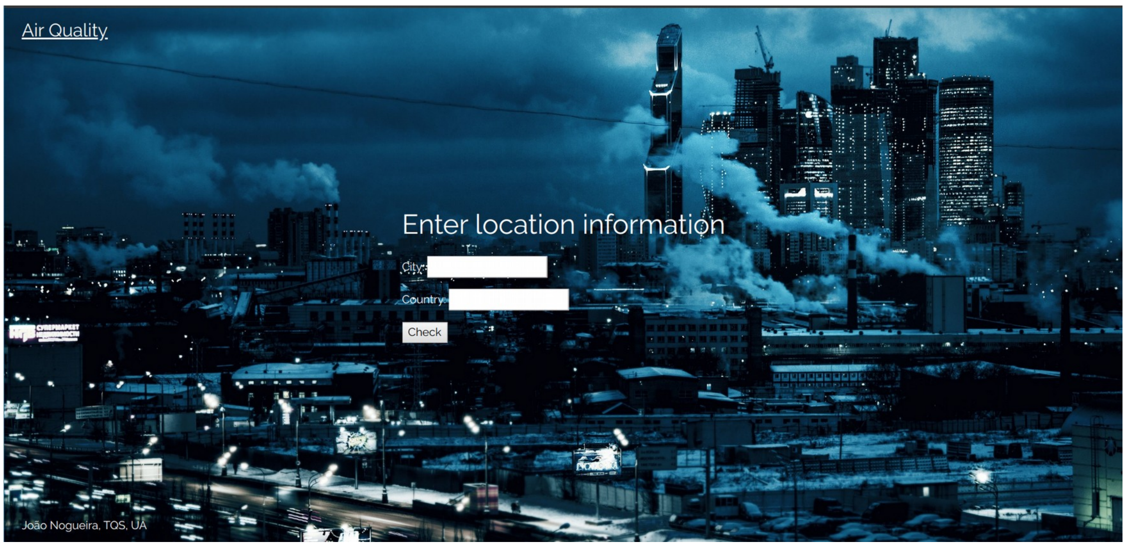
Technical Report
Air Quality Multi-Layer Web Application



João Nogueira
89262

https://github.com/Joao-Nogueira-gh/airQualitySpringBoot

Teste e Qualidade de Software, Universidade de Aveiro

1. Introduction

This document’s goal is to explain the strategy used while developing the ‘AirQuality’ application, taking note on the utilized sources, designed tests and code quality tools and results.

2. Application Context and Sources

The application is based on the subject of air quality, aiming to provide values of pollutant gases and general air quality (**A**ir **Q**uality **I**ndex) of a given city anywhere in the world.

The source API utilized is developed by Weatherbit.io. The website provides several API’s related to air quality, including hourly weather forecasts, historical weather data, and data on the energy industry (total sun hours, wind direction, precipitation, etc).

Most of those are paid, excluding the Air Quality API. From their website:

“This Air Quality API returns a 3 day / hourly forecast of air quality conditions for any location in the world. It returns air quality data on the 6 major surface pollutants - PM 2.5, PM 10, CO, SO2, NO2, and O3. Additionally, this API returns an air quality index score (AQI).”

The API itself is rather ‘limited’ regarding the queries that can be made. It can be queried by city_id, postal code, latitude/longitude and city/country names. Given the options, the selected method to use on the website was to query by city and country.

In order to be able to utilize the API one has to register in the website and request a key that is provided after a short time. That key is required in order to make any request to the Air Quality API.

3. Application Technologies and Development

3.1 Technologies

As requested, the application was developed using the Spring Boot framework. Other used technologies/modules include Thymeleaf for the HTML data-binding, the H2 memory database to host the cached data, Junit for the multi-layer testing, Selenium for the web interface testing, and SonarQube/Jacoco for the code quality metrics.

3.2 Class Structure

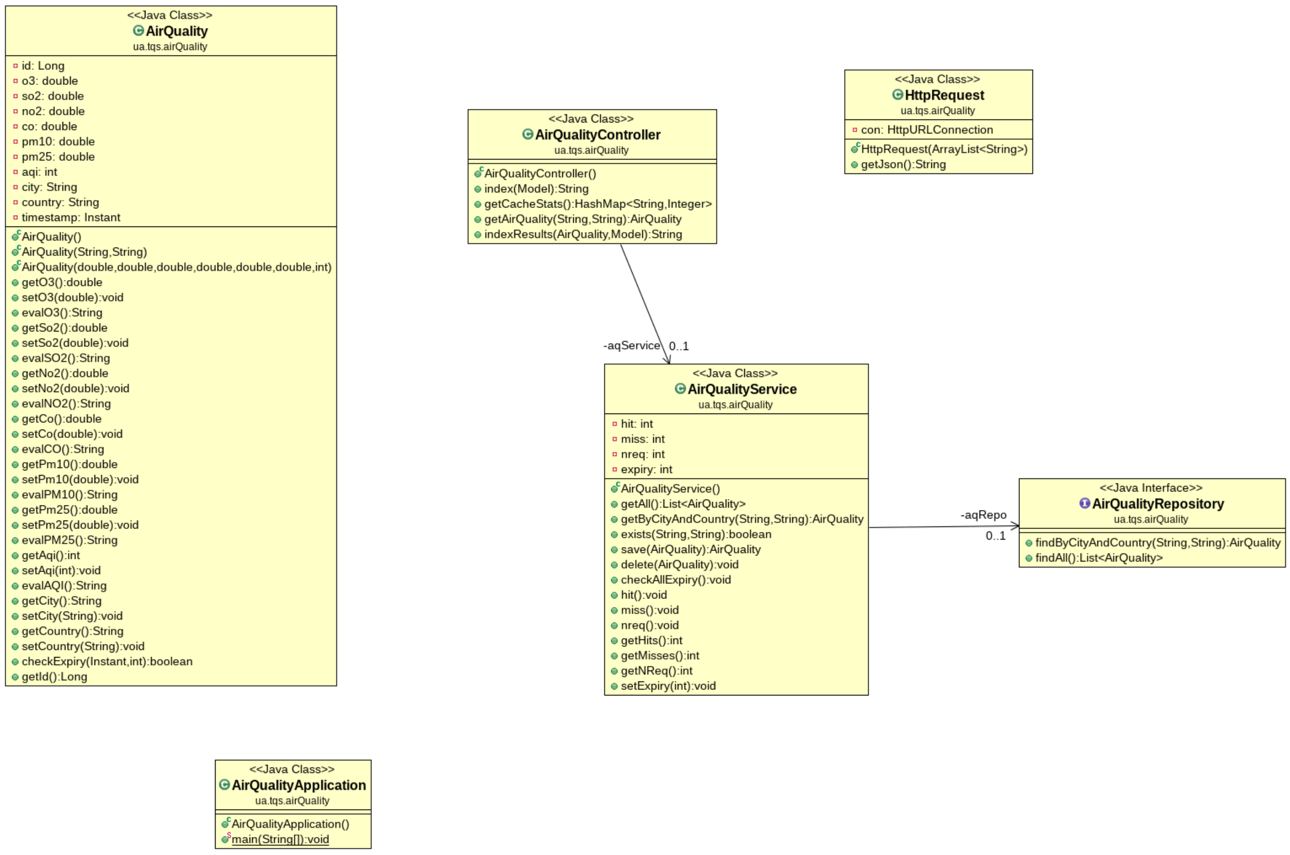


Illustration 1: Class Diagram

Above is present the class diagram depicting the modules created in order to fully develop the application.

The main AirQuality class hosts objects containing the data for each user request (location, pollutant gases values and timestamp for caching purposes).

The helper class HttpRequest handles the connection from the website to the external API.

The AirQualityRepository, an interface implementing a normal Spring Boot JPA Repository, with useful methods for saving, deleting and querying objects in the H2 memory database.

The AirQualityService providing useful methods on the repository, serving as a bridge between it and the controller.

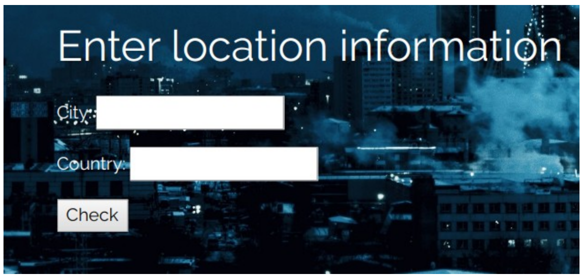
And finally the AirQualityController, coordinating everything happening on the web-page and internal API.

3.3 Application flow

All of these interactions are shown in the provided video 'airquality.mp4'.

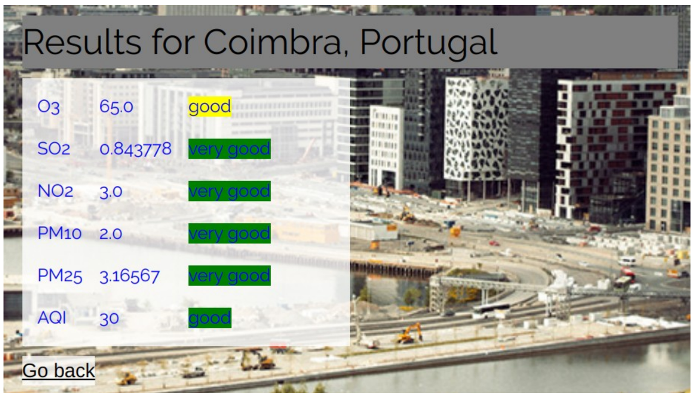
<u>Website:</u>

The user reaches the main website page. He is asked to select a city and country in order to obtain its air quality information.



If the provided data isn't valid a simple error page is returned and the user is given the option to go back to the main page.

If the data is valid then the results page is shown. The values of the 5 pollutant gases are shown in a table. By analyzing those data values range, a label/color pair is presented next to it, for example very good=green. According to the AQI Index the background image also changes between a polluted/non-polluted city.

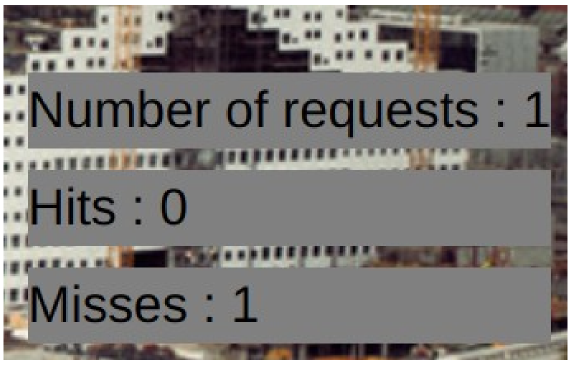


Results for Coimbra, Portugal

O3	65.0	good
SO2	0.843778	very good
NO2	3.0	very good
PM10	2.0	very good
PM25	3.16567	very good
AQI	30	good

Go back

On the side of the page statistics on the implemented cache are shown. The number of total requests, cache hits and cache misses.


Number of requests : 1
Hits : 0
Misses : 1

Internal API:

The internal API provides similar services to the web-page, giving the gases values when querying by location (http://localhost:8080/home/api?city=Paris&country=France), or providing the cache stats (http://localhost:8080/home/api/stats).

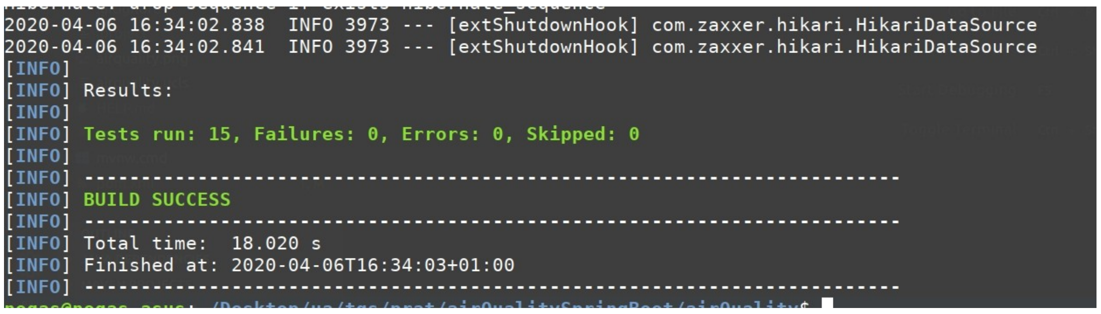
3.4 Cache

In order to implement a basic caching service, all the requests, related to a given location are stored in the H2 memory database with a timestamp of the created instant. Subsequent requests for locations already in the database are returned directly instead of querying the external API (cache hit).

After a predefined time (TTL) which is set to 20 seconds for testing/demonstration purposes, the cached objects are ready to be deleted. However, this TTL is not checked in real-time, instead, it is checked only whenever a request is made to the application. This means the data can persist in the memory database for quite some time if the application services are not being used. Therefore, it doesn't seem to be an extreme problem to the application functionalities, given it was also asked to be implemented '*in a basic way*'.

4. Testing

The application includes testing on the various layers, the Controller, the Repository and the Service modules, examining their basic functions. The tests were developed and ran successfully.



```
2020-04-06 16:34:02.838  INFO 3973 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource
2020-04-06 16:34:02.841  INFO 3973 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  18.020 s
[INFO] Finished at: 2020-04-06T16:34:03+01:00
[INFO] ------------------------------------------------------------------------
```

AirQualityRepositoryTest:

1.findByCityAndCountry_return() → test the function that searches the repository for AirQuality objects by the city/country pair.

2.ifDoenstExist_returnNull() → test the same function but this time query for a non existing object.

3.TestFindAll() → return several objects in the repository.

AirQualityServiceTest:

1.returnExistentAirQuality() → test service layer function to retrieve AirQuality objects, similar to the first repository test.

2.returnNullIfNotExistant() → test service layer function to check for non existent AirQuality objects, similar to the second repository test.

3,4.testExistFunctionTrue/False() → testing the service layer function that checks for the existence of objects in the repository.

5.testSeveralObjects() → return several objects through the service layer, similar to the third repository test.
6.testObjectDeletion() → test repository object deletion through the service layer.

AirQualityControllerAPITest:

1.verifyJsonGetWhenExists() → checks the GET method on the internal API to acquire information about a valid location.

2.verifyJsonGetWhenDoesntExist() → checks the GET method on the internal API to acquire information about an invalid location.

3.testCachingPresent() → tests for cache presence on an object that is requested twice.

4.testCachingNotPresent() → tests for cache presence (absence) on an object that is requested once.

5.testExpiry() → test cache expiry, forcing the service not to find the object in the repository instead of waiting for the passage of time during the test.

AirQualityWebTestingSelenium:

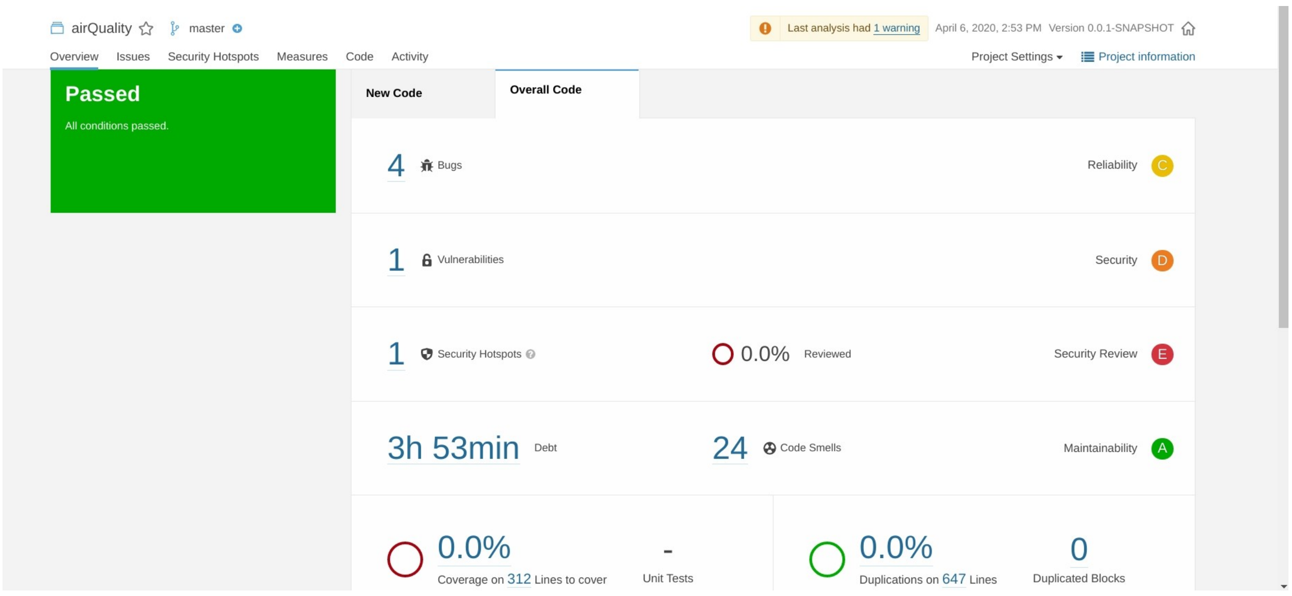
A simple test on the web interface was performed using the Selenium Web Driver. The project file is present in the 'airquality.side' file.
The test verifies the core web-page functionalities and caching by asserting the values displayed in the cache statistics.
The test was then exported to Java and adapted in order to run together with the remaining tests.

5. Code Quality Metrics

The code quality metrics and coverage were obtained using the SonarQube platform together with the Jacoco plugin.

The SonarQube analysis was done using a docker container. It was run an initial time, then a second one after the reported errors were fixed.
A detailed discussion is now done on those analyses.

First SonarQube Analysis:



Last analysis had 1 warning April 6, 2020, 2:53 PM Version 0.0.1-SNAPSHOT

Passed
All conditions passed.

4 Bugs — Reliability C

1 Vulnerabilities — Security D

1 Security Hotspots — 0.0% Reviewed — Security Review E

3h 53min Debt — 24 Code Smells — Maintainability A

0.0% Coverage on 312 Lines to cover — - Unit Tests

0.0% Duplications on 647 Lines — 0 Duplicated Blocks

The first analysis reported quite a few errors.

The four reported ‘bugs’ were all similar to each other, referring to the incorrect usage of the comparison operator ‘==’ between Strings, instead of the .equals() method. These were quickly fixed.

The reported vulnerability was urging to ‘Replace this persistent entity with a simple POJO or DTO object’, referring to a AirQuality object model used in the web-page controller in order to show all the data in the html.
After some search it didn’t seem to be a major problem in such a small Spring Boot application and required quite some refactoring, including a whole new class. The problem was noted but wasn’t fixed and its severity was thus tuned down.

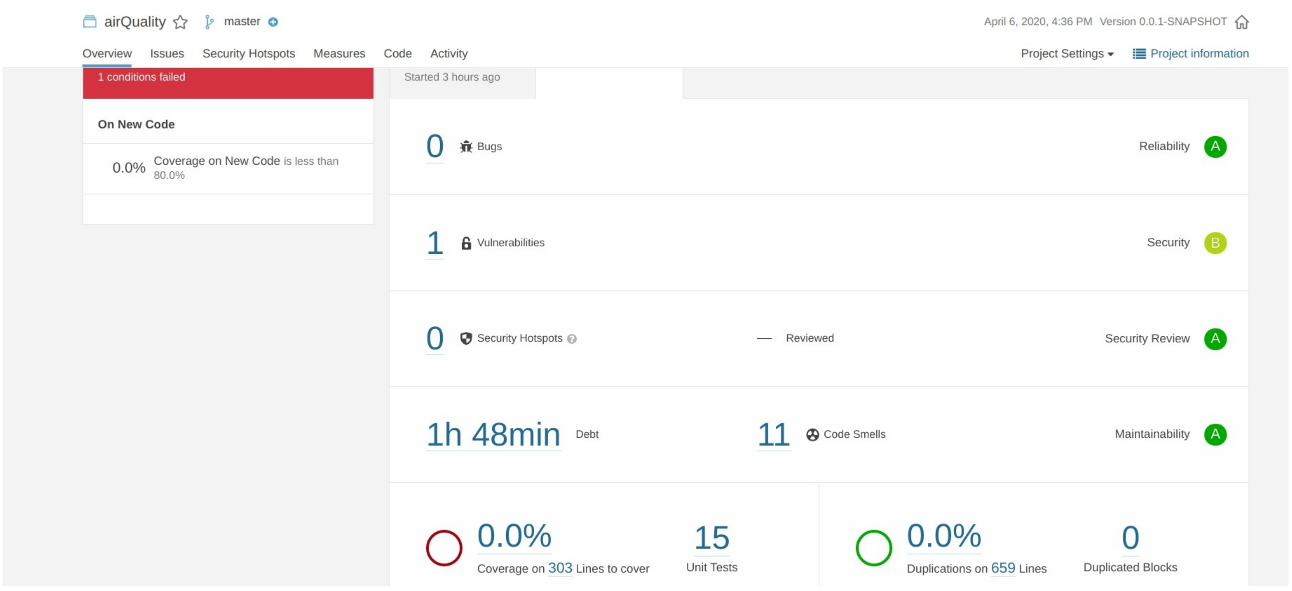
The mentioned Security Hotspot warned the developer to ‘Make sure that command line arguments are used safely here.’, referring to the auto-generated Spring Boot ‘AirQualityApplication.java’ class. Since no command line arguments were ever used with the program, that parameter was simply removed from the function call.

Code Smells (some of these were repeated throughout the code):

- ‘Incorrect package naming convention’, the project package where the main classes exist is called ‘ua.tqs.airQuality’, the capitalized Q triggers that warning. Not being a tremendous problem this one was not fixed.
- ‘Define a constant instead of duplicating this literal x times’, this one was properly handled in the situations where it made sense to do so (6 or more times repeated).
- ‘Replace println by a logger’, this served to warn about some forgotten prints for debugging purposes that were then deleted.
- ‘return (condition) ? true : false can be replaced by one line’, this was indeed correct and replaced by the shorter and more correct version, ‘return (condition)’.
- ‘The return type of this method should be an interface such as "Map" rather than the implementation "HashMap"’, this situation happened two times, in the return of a HashMap object instead of a Map one, like in the example, and using ArrayList as a function argument instead of just List. Referring to the interface is most often a better programming habit.
- ‘Refactor this method to reduce its Cognitive Complexity from 17 to the 15 allowed.’, these were not handled since there wasn’t found a way to reduce the method’s complexity without changing the code functionality.
- ‘Add at least one assertion to this test case.’, the auto-generated Spring Boot test class file has no actual tests, this was ignored.
- ‘Replace the synchronized class "StringBuffer" by an unsynchronized one such as "StringBuilder".’, the String Buffer class used in the helper class HttpRequest was thus replaced by String Builder as advised.

- ‘Replace this if-then-else statement by a single return statement.’, similar situation to the ‘return condition’ code smell, this one was also fixed, from:
```
if (aqRepo.findByCityAndCountry(city,country) != null) {
            return true;
        }
        return false;
```
  to:
```
return aqRepo.findByCityAndCountry(city,country) != null;
```

<u>Second SonarQube Analysis:</u>


airQuality master
April 6, 2020, 4:36 PM Version 0.0.1-SNAPSHOT
Overview Issues Security Hotspots Measures Code Activity
Project Settings Project information

1 conditions failed
Started 3 hours ago

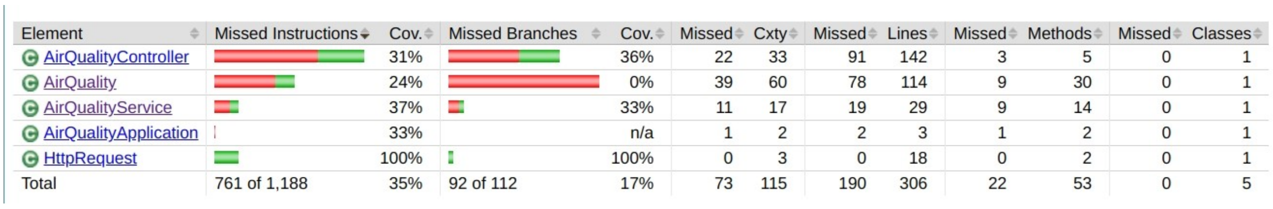
On New Code
0.0% Coverage on New Code is less than 80.0%

0 Bugs — Reliability A
1 Vulnerabilities — Security B
0 Security Hotspots — Reviewed — Security Review A
1h 48min Debt — 11 Code Smells — Maintainability A
0.0% Coverage on 303 Lines to cover — 15 Unit Tests
0.0% Duplications on 659 Lines — 0 Duplicated Blocks

After all those situations were handled this was the final resulting analysis, with bugs, code smells and technical debt significantly reduced, leaving only the previously mentioned minor vulnerability, and some code smells like package naming and repeated literals.

However, even after some troubleshooting the analysis kept on report 0% code coverage. This was however analyzed using the Jacoco plugin and the results are discussed below.

Jacoco Code Coverage Analysis:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AirQualityController		31%		36%	22	33	91	142	3	5	0	1
AirQuality		24%		0%	39	60	78	114	9	30	0	1
AirQualityService		37%		33%	11	17	19	29	9	14	0	1
AirQualityApplication		33%		n/a	1	2	2	3	1	2	0	1
HttpRequest		100%		100%	0	3	0	18	0	2	0	1
Total	761 of 1,188	35%	92 of 112	17%	73	115	190	306	22	53	0	5

At first glance these seem to be pretty bad results but open further evaluation there seem to be valid explanations on why this happened.

Regarding the 3 main application modules (since the helper class has 100% code coverage and the auto-generated application starter class is meaningless) :

- AirQuality entity class: all the reported test coverage failures are on helper functions to determine CSS/HTML values (previously mentioned 'good/green color' pair) and an unused constructor.
-
- AirQualityService class: reported failures refer to getters, setters, small methods that increment an internal integer variable by one, and the save() method that even though was used in the tests set up phase (@beforeEach) was not directly tested.
-
- AirQualityController class: reported failures are due to the web-page part of the class, which was tested with the Selenium Web Driver, and Jacoco is obviously not taking that fact into consideration.

It is thus considered that the results are definitely not as bad as they look, and all the main project functionalities were successfully tested.

6. Conclusion

The project goals were fulfilled, building a simple multi-layer web application that reports air quality on a given city around the world, making use of an external API, an internal one and a simple manually implemented caching mechanism.

Furthermore, all modules were successfully tested, as well as the web-page, and code quality metrics and coverage were analyzed and their main problems were fixed.