HW1: Mid-term assignment report

Pedro Miguel Almeida Santos [93221], v2021-05-14

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy.

This project's theme is centered around providing details of air quality for certain regions. Different aspects were considered as useful, like particles in suspension or the presence (and quantity) of certain gases.

The project has several components:

1. A minimalist web page through which users can select locations and access their air quality details.
2. The integration of an external API from which the data we'll be using is fetched.
3. An in-memory cache which stores the latest results fetched from the external API, which should also include a time-to-live policy regarding data persistence. Useful statistics regarding the cache (hits, misses, requests) should also be stored (in-memory) so they can later be accessed.
4. A REST API which can be invoked by external clients, this API should be the bridge between clients and the external API, by providing air quality data. It should also provide cache statistics data. This API should also include logging for inspection/ debugging purposes.

1.2 Current limitations

Current limitations include the inability to search for a broad range of parameters (currently only supports searching by address and a time interval) and the inclusion of only one external API, meaning that if it fails there are no safeguards, therefore the project will become unusable.

The external Geocoding API used to translate addresses to coordinates is using a key with a limited number of request per day and per minute (values high enough to allow thorough app testing), so if those limits are reached, the app will become unavailable, the key may also expire in the future but it's possible to obtain a new one for free through the Google Maps Platform.

2 Product specification

2.1 Functional scope and supported interactions

The product can be access in two ways:

- Through the web app that was developed, from which the user can interrogate the backend through an address and/ or a time interval, the results will be shown through a numbered table with all available details, appearing a maximum of 25 rows per page (each row maps to the air quality for a certain hour). Although it was intended at first, the cache statistics cannot be accessed through the web app, this functionality was left undone due to time limitations.

  The web app includes some error handling providing users with warnings when no data is fetched or when parameter values are invalid.
- Through the REST API, clients can interrogate the backend through 4 endpoints, each one regarding a different use case:
  1. A client who wants to fetch the current air quality of a certain region;
  2. A client who wants to fetch the forecast air quality (data from 5 days ago until 5 days later);
  3. A client who wants to fetch historical air quality, having the ability to define a region, a start date and an end date;
  4. A client who wishes to view data regarding the cache usage, this data can be relative to one of the 3 endpoints above or regarding all 3 of them.

2.2 System architecture

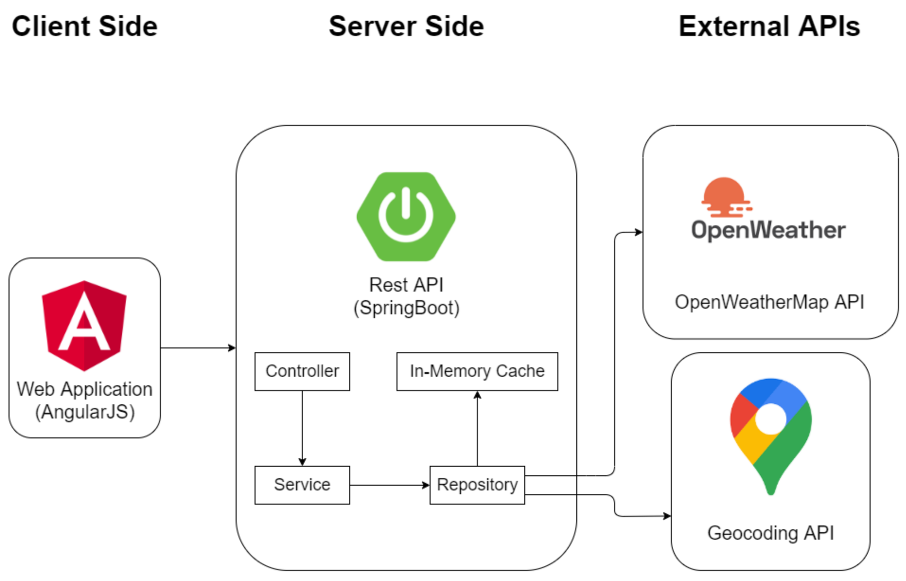


Figure 1 - Architecture Diagram

The system is composed of 3 modules, the web application used to display data in a more presentable way, the Rest API which handles the requests, and the external APIs from which data is fetched.

Starting with the web application, it was developed in angular, a JavaScript framework, and display response data in a more readable way. The client interacts directly with the web app, he can then use its search method to make requests to the API, these requests are made to the several endpoints available based on the user's input.

The Rest API was developed in SpringBoot, it contains a controller, a service, two repositories and an in-memory cache. When a request is made to an endpoint, the appropriate method in the controller is triggered, which then makes a call to the service in order to obtain the appropriate data. The service is then in charge of obtaining the requested data by utilizing both repositories. The location repository is used to translate the address into a location by sending a request to a geocoding api, which returns a latitude and a longitude based on an address. With the latitude and longitude, it's then possible to obtain the air pollution analysis, this is done by utilizing the air pollution repository, which gets the wanted information either by accessing the internal cache (the cache won't be used if the data has expired based on it's TTL), or by requesting data from the OpenWeatherMap API.

Once the data has successfully been obtained, it's returned to the user and presented on the web page dynamically.

2.3 API for developers

To more easily showcase the available endpoints, swagger was used to generate API documentation, the results can be seen in the pictures bellow.

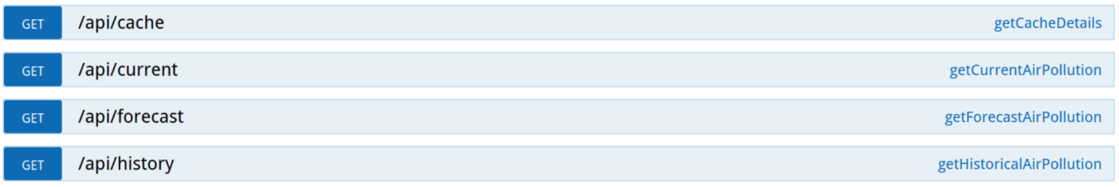
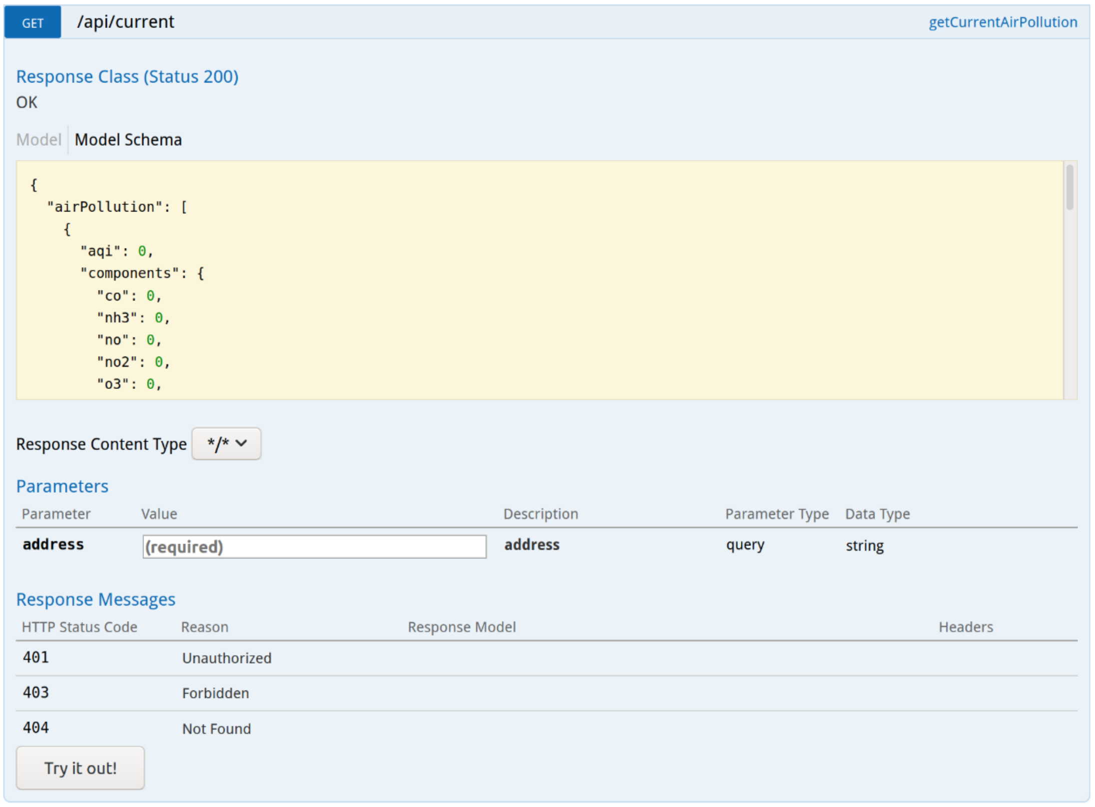

GET /api/cache getCacheDetails
GET /api/current getCurrentAirPollution
GET /api/forecast getForecastAirPollution
GET /api/history getHistoricalAirPollution

Figure 2 - API Endpoints

From the endpoints seen in figure 2, users can:
/api/cache => Get cache statistics (from one cache type if specified, or all of them)
/api/current => Get current air pollution for a location
/api/forecast => Get hourly air pollution for a location from 5 days earlier until 5 days later
/api/history => Get hourly air pollution for a location within a given range


GET /api/current getCurrentAirPollution

Response Class (Status 200)
OK

```
{
  "airPollution": [
    {
      "aqi": 0,
      "components": {
        "co": 0,
        "nh3": 0,
        "no": 0,
        "no2": 0,
        "o3": 0,
```

Response Content Type */*

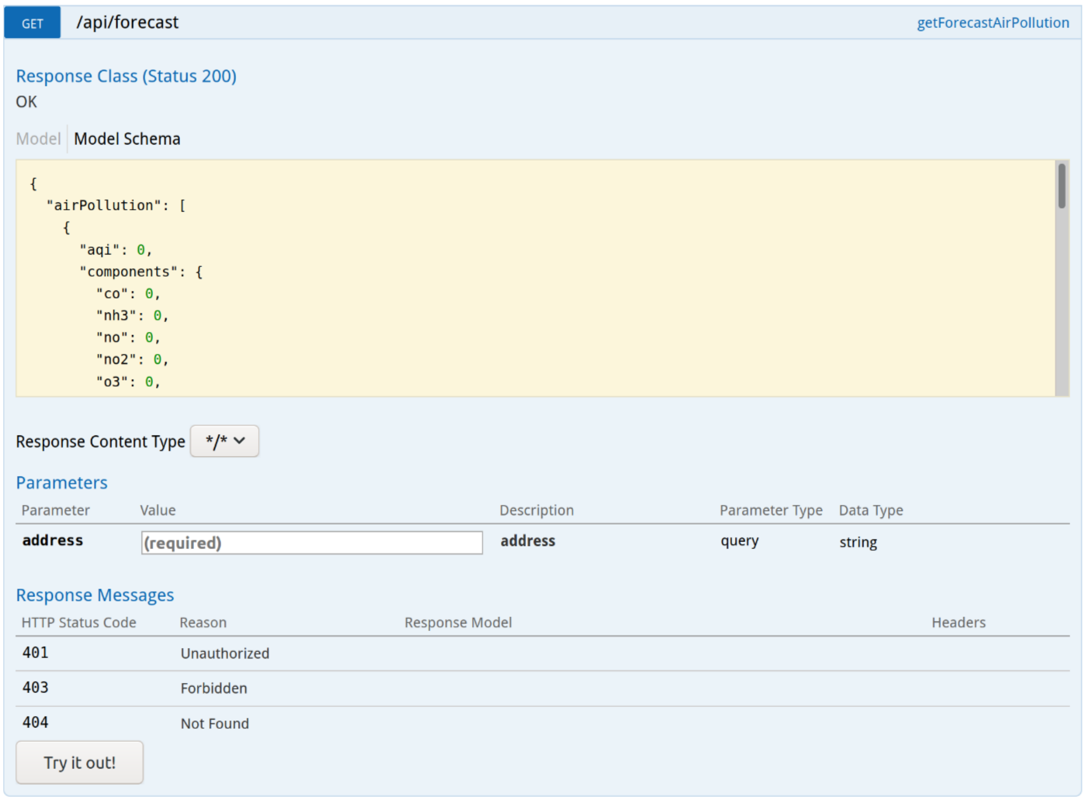
Parameters

Parameter	Value	Description	Parameter Type	Data Type
address	(required)	**address**	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 3 - /api/current Endpoint



GET /api/forecast getForecastAirPollution

Response Class (Status 200)
OK

Model | Model Schema

```
{
  "airPollution": [
    {
      "aqi": 0,
      "components": {
        "co": 0,
        "nh3": 0,
        "no": 0,
        "no2": 0,
        "o3": 0,
```

Response Content Type */*

Parameters

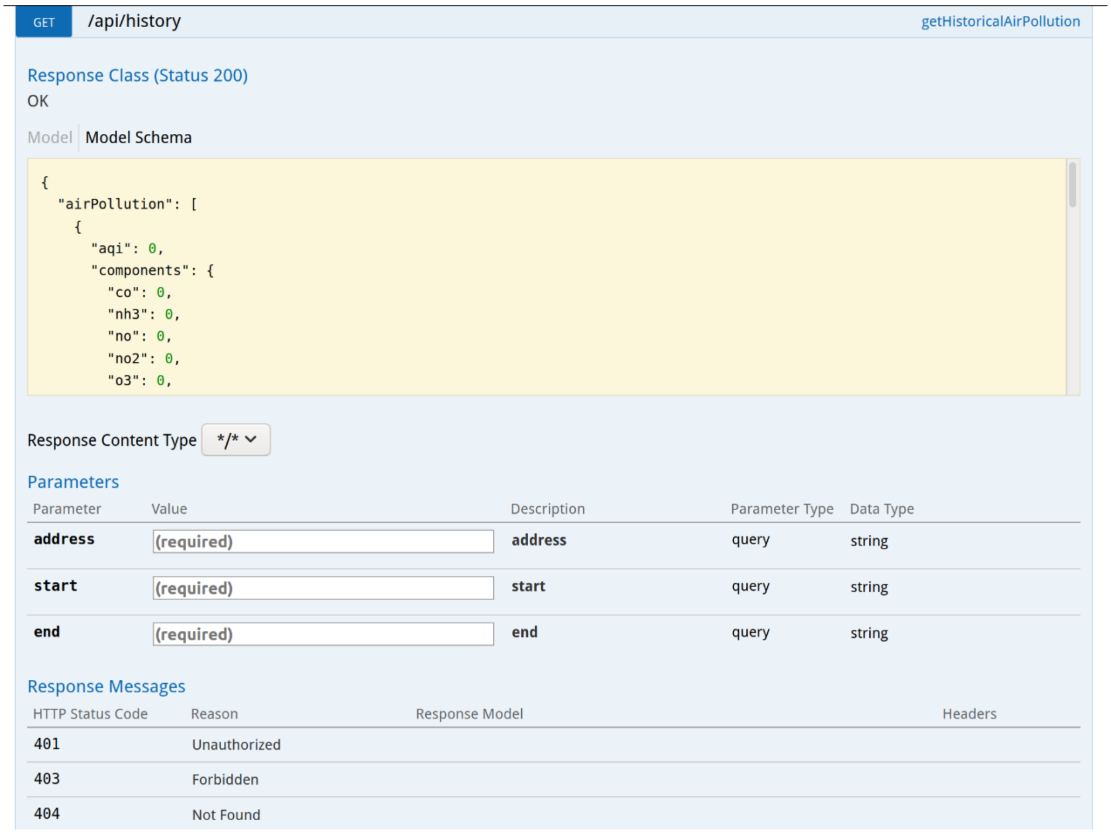
Parameter	Value	Description	Parameter Type	Data Type
address	(required)	**address**	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 4 - /api/forecast Endpoint



GET /api/history getHistoricalAirPollution

Response Class (Status 200)
OK

Model | Model Schema

```
{
  "airPollution": [
    {
      "aqi": 0,
      "components": {
        "co": 0,
        "nh3": 0,
        "no": 0,
        "no2": 0,
        "o3": 0,
```

Response Content Type */*

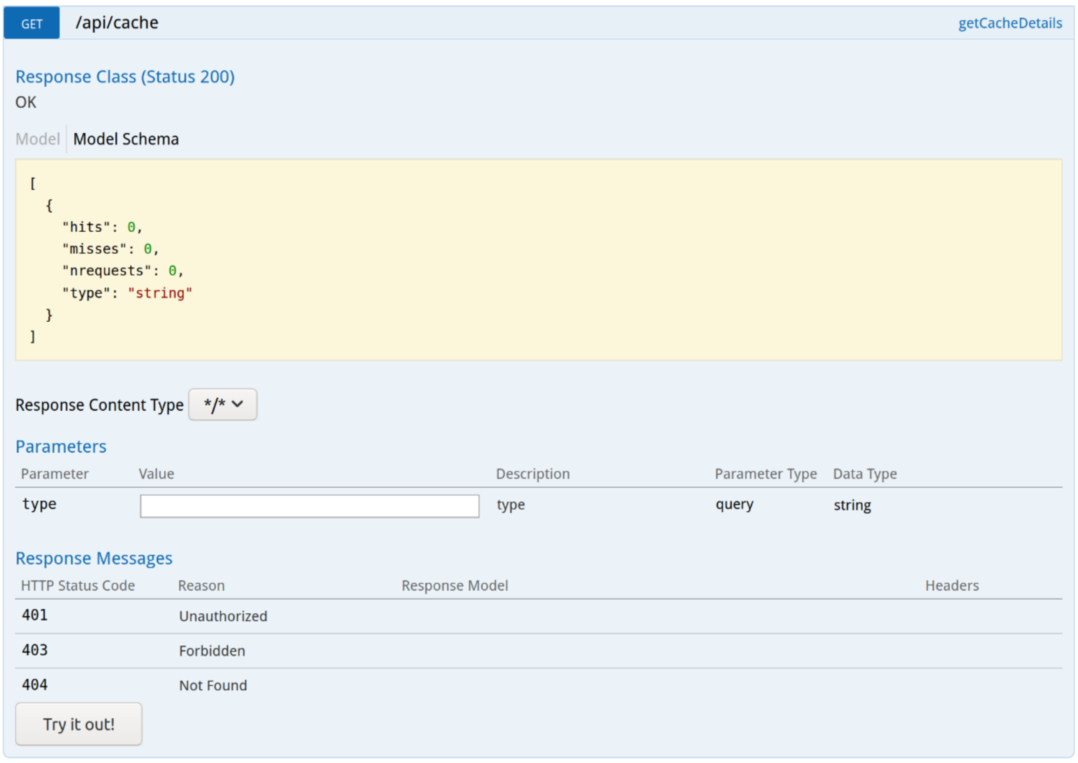
Parameters

Parameter	Value	Description	Parameter Type	Data Type
address	(required)	**address**	query	string
start	(required)	**start**	query	string
end	(required)	**end**	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 5 - /api/history Endpoint


GET /api/cache getCacheDetails

Response Class (Status 200)
OK

Model | Model Schema

```
[
  {
    "hits": 0,
    "misses": 0,
    "nrequests": 0,
    "type": "string"
  }
]
```

Response Content Type */*

Parameters

Parameter	Value	Description	Parameter Type	Data Type
type		type	query	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Figure 6 - /api/cache Endpoint

3 Quality assurance

3.1 Overall strategy for testing

Although TDD was recommended, it wasn’t the strategy I used, therefore most tests were created after the functionality had been developed.

As the project guidelines stated, I created unit tests for some individual components, like the cache, cache details and a converter class used to convert between LocalDateTime and Unix Epoch. I also implemented service level tests with dependency isolation using Mockito, Mocking the usage of the external API where it was needed, repositories, the service and the controller were all tested. I also included integration tests to make sure all components were behaving correctly when used together.

The web app was tested using BDD with the help of Cucumber and Selenium, a general scenario was developed and tested.

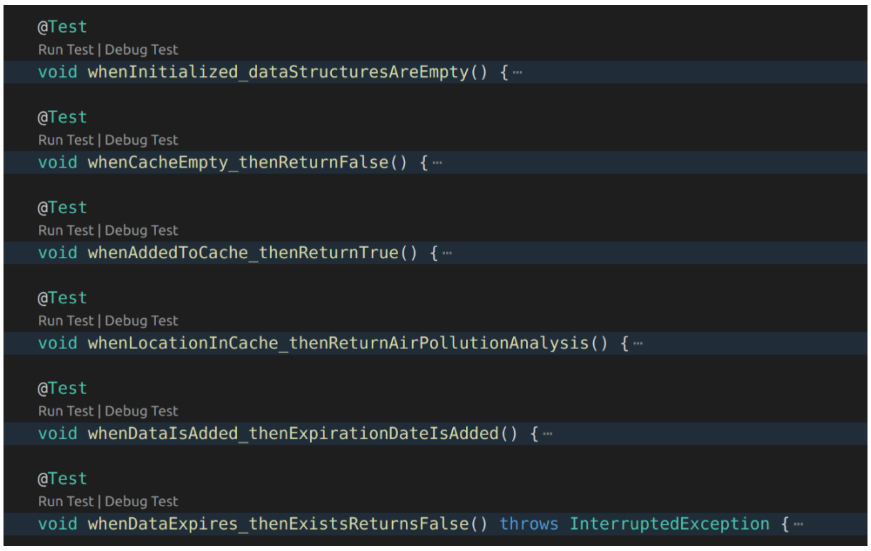
The GitHub repository was also used to add CI with the help of Sonarcloud, making sure that every push was tested and, if successful, analyzed by sonarcloud to detect potential ways to improve the overall quality. It was very useful in regard to viewing code coverage (JaCoCo was used in earlier stages of test development) and exposing code smells as well as critical bugs.

3.2 Unit and integration testing

3.2.1 Cache

Starting with the cache, I wrote 6 unit tests to make sure all functionalities worked as intended. The tests covered the following situations:

1. When a cache is initialized, make sure it is empty;
2. When a cache is empty, it should return false when queried for a location;
3. When an analysis (associated with a location) is added to the cache, and then the existence of the location is queried, it should return true;
4. When a location is in the cache (associated with an analysis) and it's searched for, then the analysis should be returned;
5. When new data is added to the cache, make sure its expiration date is also added;
6. When data expires, and then it's queried using a location, make sure it returns as though it doesn't exist in the cache.



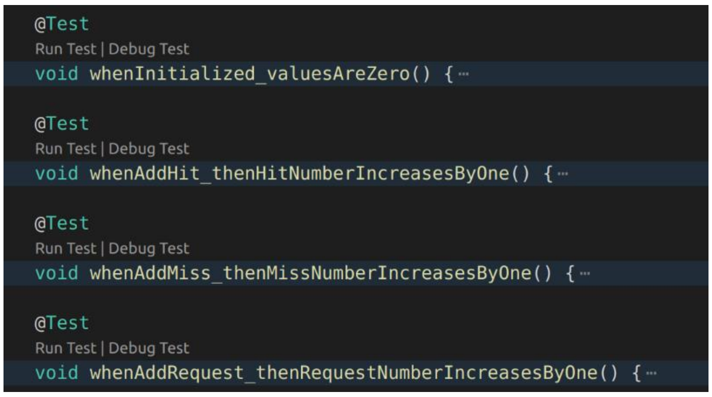
```
@Test
Run Test | Debug Test
void whenInitialized_dataStructuresAreEmpty() {⋯

@Test
Run Test | Debug Test
void whenCacheEmpty_thenReturnFalse() {⋯

@Test
Run Test | Debug Test
void whenAddedToCache_thenReturnTrue() {⋯

@Test
Run Test | Debug Test
void whenLocationInCache_thenReturnAirPollutionAnalysis() {⋯

@Test
Run Test | Debug Test
void whenDataIsAdded_thenExpirationDateIsAdded() {⋯

@Test
Run Test | Debug Test
void whenDataExpires_thenExistsReturnsFalse() throws InterruptedException {⋯
```

Figure 7 - Cache Tests

3.2.2 Cache Details

In order to test the statistics part of the cache, some simple tests were written to make sure the hits, misses and requests counters were working correctly.

1. When initialized, the cache statistics should all be 0;
2. When a hit is added, the value of hits should increase by 1;
3. When a miss is added, the value of misses should increase by 1;
4. When a request is added, the value of requests should increase by 1;



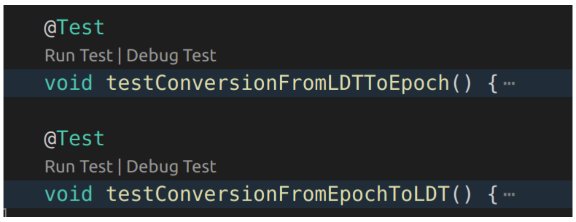
```
@Test
Run Test | Debug Test
void whenInitialized_valuesAreZero() {⋯

@Test
Run Test | Debug Test
void whenAddHit_thenHitNumberIncreasesByOne() {⋯

@Test
Run Test | Debug Test
void whenAddMiss_thenMissNumberIncreasesByOne() {⋯

@Test
Run Test | Debug Test
void whenAddRequest_thenRequestNumberIncreasesByOne() {⋯
```

Figure 8 - Cache Details Tests

3.2.3 Converter

To test the converter 2 tests were created to make sure the dates were being converted correctly.

1. Verify the conversion from LocalDateTime to Unix Epoch;
2. Verify the conversion from Unix Epoch to LocalDateTime.



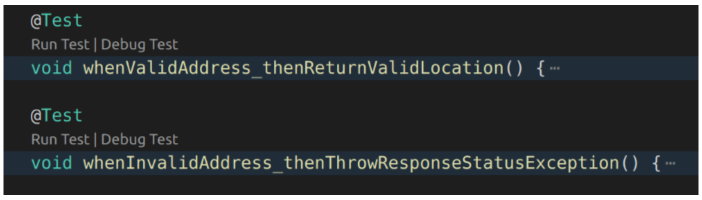
```
@Test
Run Test | Debug Test
void testConversionFromLDTToEpoch() {⋯

@Test
Run Test | Debug Test
void testConversionFromEpochToLDT() {⋯
```

Figure 9 - Converter Tests

3.2.4 Location Repository

The location repository is a simple class with only one usable method, therefore testing was simple with only 2 tests.

1. Verify that, when given a valid address, the method getLocation() returns a valid location with the same address that was searched for;
2. Verify that, when given an invalid address, the method throws an exception.



```
@Test
Run Test | Debug Test
void whenValidAddress_thenReturnValidLocation() {⋯

@Test
Run Test | Debug Test
void whenInvalidAddress_thenThrowResponseStatusException() {⋯
```

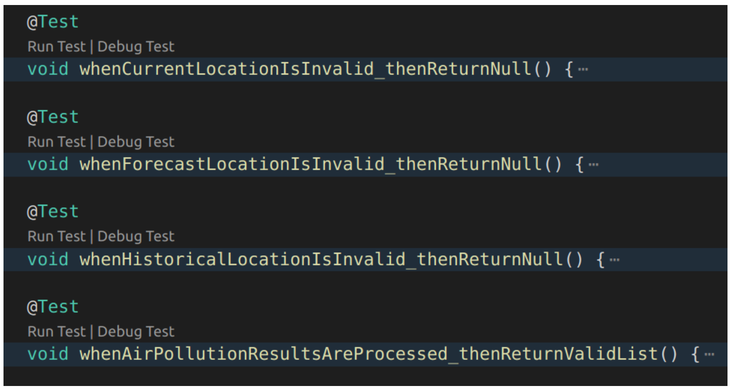
Figure 10 - Location Repository Tests

3.2.5 Air Pollution Repository

4 tests were used, 3 of them to test the behavior when given invalid locations and the last one to test the method used to convert from the json response with the Air Pollution results to

a list of Air Pollution objects. The data received wasn't validated since there was no reason to validate information originating from the external api.

1. When given an invalid location, the current method should return null;
2. When given an invalid location, the forecast method should return null;
3. When given an invalid location, the history method should return null;
4. When a valid json response is given to the processing method, it should return a valid and correct list of Air Pollution objects.



```
@Test
Run Test | Debug Test
void whenCurrentLocationIsInvalid_thenReturnNull() {⋯

@Test
Run Test | Debug Test
void whenForecastLocationIsInvalid_thenReturnNull() {⋯

@Test
Run Test | Debug Test
void whenHistoricalLocationIsInvalid_thenReturnNull() {⋯

@Test
Run Test | Debug Test
void whenAirPollutionResultsAreProcessed_thenReturnValidList() {⋯
```

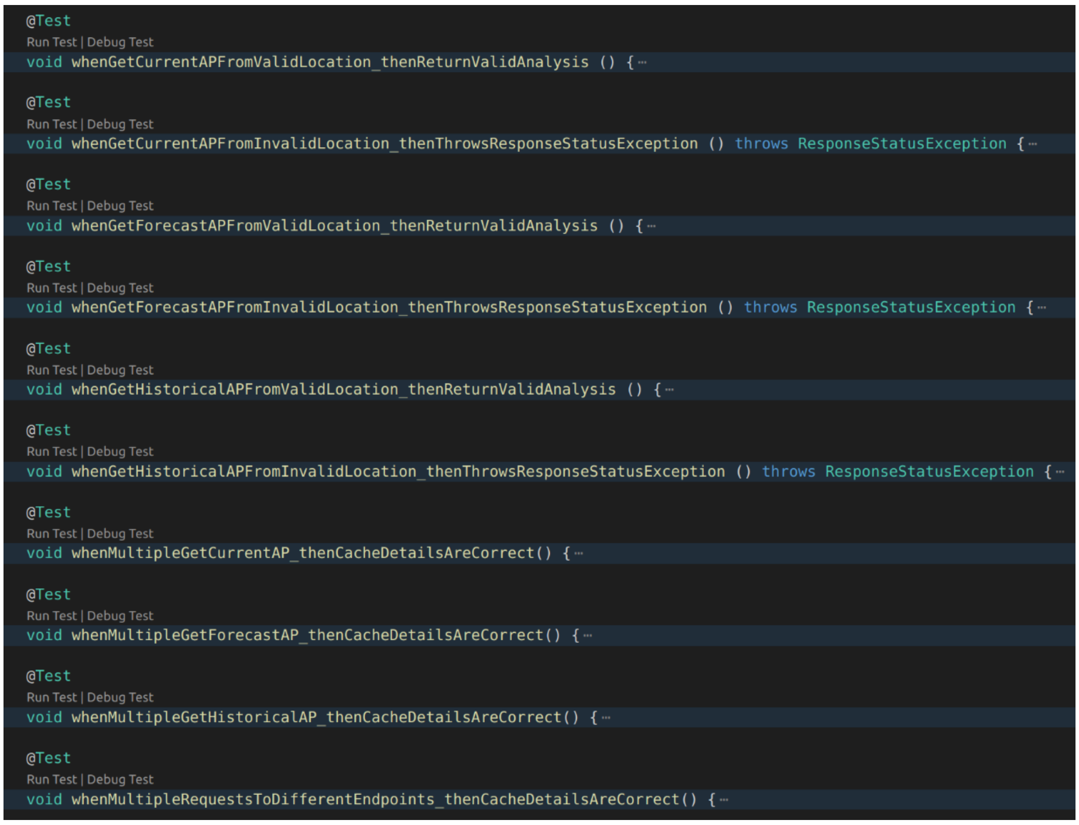
Figure 11 - Air Pollution Repository Tests

3.2.6 Air Pollution Service

The service module of the API is a fundamental part for it to work, due to its purpose of connecting the repositories to the endpoints, therefore it was thoroughly tested with 10 tests. Mockito was used extensively to mock the behavior of both repositories, making sure that the tests were as isolated as possible. Throughout the following tests, verifications of how many times some methods were called were also made.

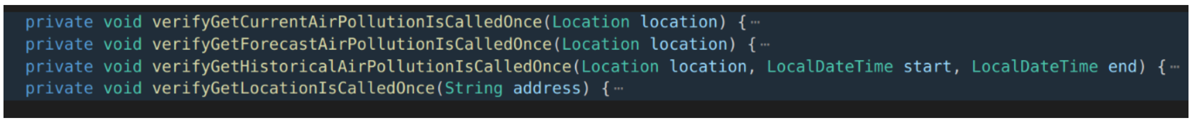
1. When requesting current air pollution with a valid address, make sure the analysis returned is also valid;
2. When requesting current air pollution with an invalid address, verify that an exception is thrown;
3. When requesting forecast air pollution with a valid address, make sure the analysis returned is also valid;
4. When requesting forecast air pollution with an invalid address, verify that an exception is thrown;
5. When requesting historical air pollution with a valid address, make sure the analysis returned is also valid;
6. When requesting historical air pollution with an invalid address, verify that an exception is thrown;
7. When several current air pollution requests are made, make sure the cache details are updated correctly;
8. When several forecast air pollution requests are made, make sure the cache details are updated correctly;
9. When several historical air pollution requests are made, make sure the cache details are updated correctly;

10. When requests are made to distinct endpoints make sure all cache details remain correct.



```
@Test
Run Test | Debug Test
void whenGetCurrentAPFromValidLocation_thenReturnValidAnalysis () {⋯

@Test
Run Test | Debug Test
void whenGetCurrentAPFromInvalidLocation_thenThrowsResponseStatusException () throws ResponseStatusException {⋯

@Test
Run Test | Debug Test
void whenGetForecastAPFromValidLocation_thenReturnValidAnalysis () {⋯

@Test
Run Test | Debug Test
void whenGetForecastAPFromInvalidLocation_thenThrowsResponseStatusException () throws ResponseStatusException {⋯

@Test
Run Test | Debug Test
void whenGetHistoricalAPFromValidLocation_thenReturnValidAnalysis () {⋯

@Test
Run Test | Debug Test
void whenGetHistoricalAPFromInvalidLocation_thenThrowsResponseStatusException () throws ResponseStatusException {⋯

@Test
Run Test | Debug Test
void whenMultipleGetCurrentAP_thenCacheDetailsAreCorrect() {⋯

@Test
Run Test | Debug Test
void whenMultipleGetForecastAP_thenCacheDetailsAreCorrect() {⋯

@Test
Run Test | Debug Test
void whenMultipleGetHistoricalAP_thenCacheDetailsAreCorrect() {⋯

@Test
Run Test | Debug Test
void whenMultipleRequestsToDifferentEndpoints_thenCacheDetailsAreCorrect() {⋯
```

Figure 12 - Service Tests



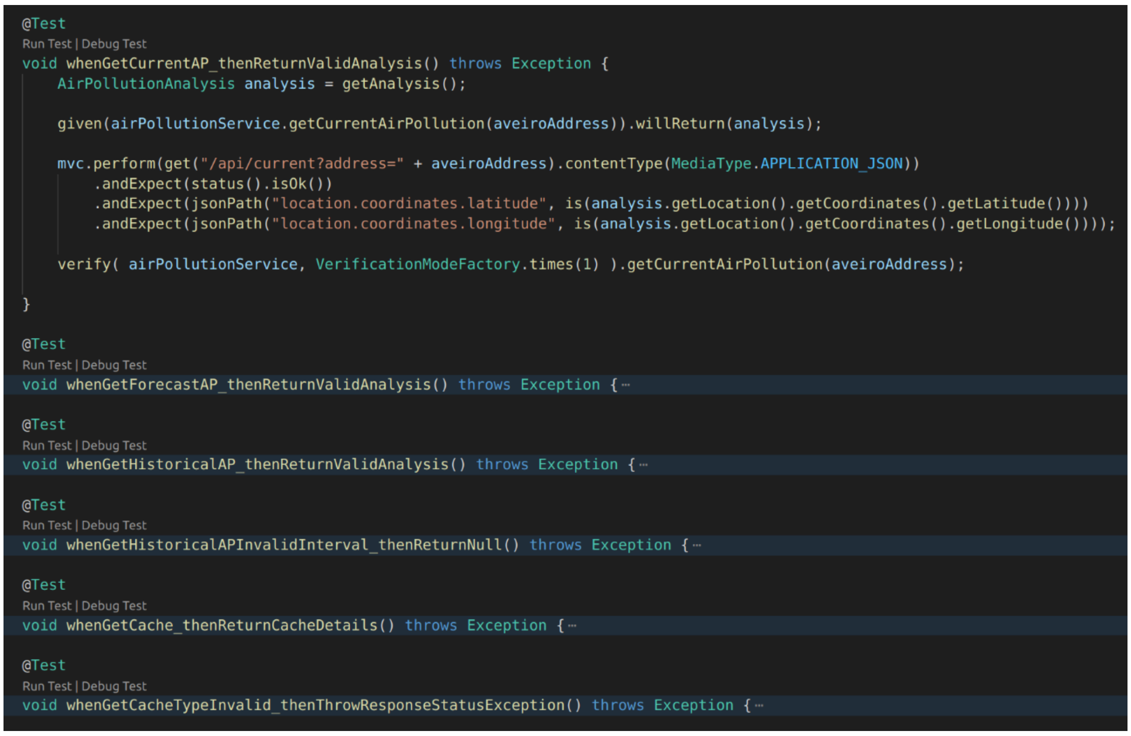
```
private void verifyGetCurrentAirPollutionIsCalledOnce(Location location) {⋯
private void verifyGetForecastAirPollutionIsCalledOnce(Location location) {⋯
private void verifyGetHistoricalAirPollutionIsCalledOnce(Location location, LocalDateTime start, LocalDateTime end) {⋯
private void verifyGetLocationIsCalledOnce(String address) {⋯
```

Figure 13 - Service Additional Verifications

3.2.7 Air Pollution Controller

In order to test the controller some unit tests where developed to make sure the requests were being correctly processed. 6 tests were created which included testing with both valid and invalid values. The unit tests used Mockito to mock the service instance.

1. When a valid request to the current air pollution endpoint is made, a valid analysis should be returned;
2. When a valid request to the forecast air pollution endpoint is made, a valid analysis should be returned;
3. When a valid request to the historical air pollution endpoint is made, a valid analysis should be returned;
4. When a request to the historical air pollution endpoint is made (for example, with a start date later than the end date) the result should be a bad request status code;
5. When a valid request is made to the cache details endpoint, the results should be correct and valid;
6. When a request is made to retrieve cache details of an invalid type, the result should be a bad request status.



```
@Test
Run Test | Debug Test
void whenGetCurrentAP_thenReturnValidAnalysis() throws Exception {
    AirPollutionAnalysis analysis = getAnalysis();

    given(airPollutionService.getCurrentAirPollution(aveiroAddress)).willReturn(analysis);

    mvc.perform(get("/api/current?address=" + aveiroAddress).contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("location.coordinates.latitude", is(analysis.getLocation().getCoordinates().getLatitude())))
        .andExpect(jsonPath("location.coordinates.longitude", is(analysis.getLocation().getCoordinates().getLongitude())));

    verify( airPollutionService, VerificationModeFactory.times(1) ).getCurrentAirPollution(aveiroAddress);

}

@Test
Run Test | Debug Test
void whenGetForecastAP_thenReturnValidAnalysis() throws Exception {⋯

@Test
Run Test | Debug Test
void whenGetHistoricalAP_thenReturnValidAnalysis() throws Exception {⋯

@Test
Run Test | Debug Test
void whenGetHistoricalAPInvalidInterval_thenReturnNull() throws Exception {⋯

@Test
Run Test | Debug Test
void whenGetCache_thenReturnCacheDetails() throws Exception {⋯

@Test
Run Test | Debug Test
void whenGetCacheTypeInvalid_thenThrowResponseStatusException() throws Exception {⋯
```

Figure 14 - Controller Unit Tests

The integration tests were the same as the ones stated above, without mocking the air pollution service, therefore testing the interaction between both components.

3.3 Functional testing

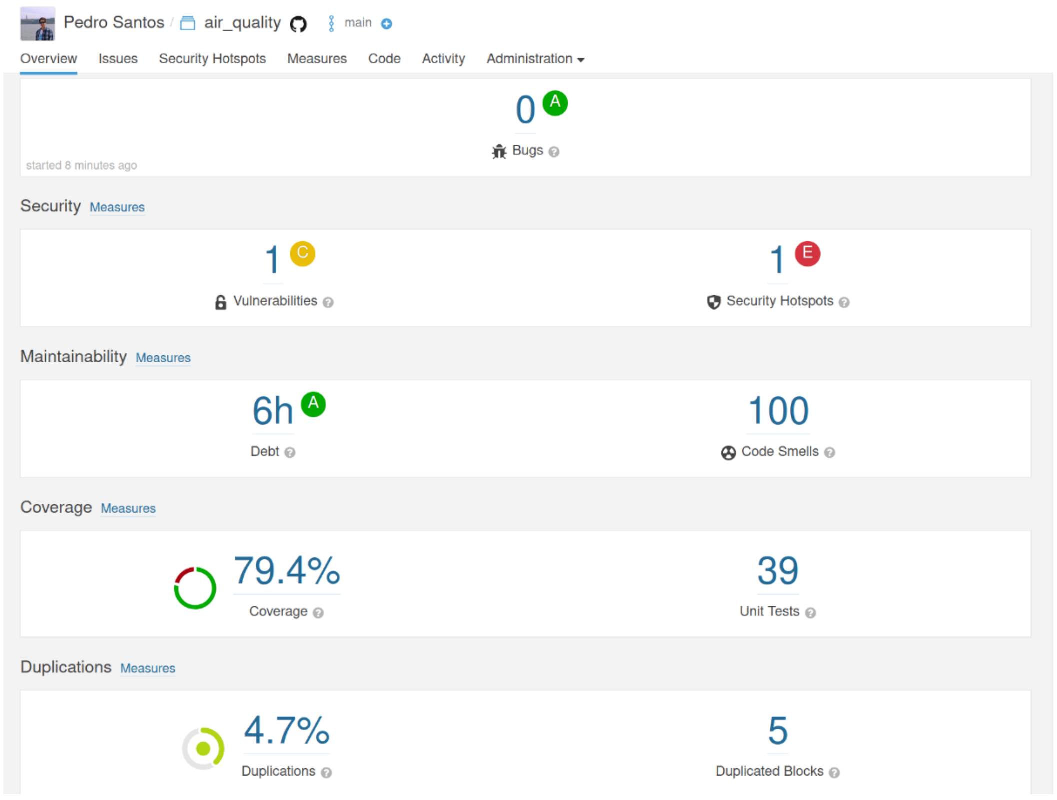
Functional tests were developed to test the UI. A scenario was built based on the simple but common example of a client searching for historical data on the website. The inputs, clicks, headers, and location present on the received information are all considered. To help with this process and make it more authentic, I used cucumber, which helped in translating the use case to tests. Selenium was also essential in creating the UI tests.

To develop these tests, I also used the Page Object pattern for developing tests easier to understand and maintain. 2 page classes were created to represent both pages, home and results, present on the web app.

3.4 Static code analysis

The code analysis tool I used was the Sonarcloud analysis, by using it I was able to identify multiple code smells that I wasn't aware were present in my code, as well as some security risks. It was also useful in detecting unused methods and untested blocks of code.

The first analysis can be seen in figure 15 (which didn't pass).


Pedro Santos / air_quality main

Overview Issues Security Hotspots Measures Code Activity Administration

0 A
Bugs

started 8 minutes ago

Security Measures

1 C
Vulnerabilities

1 E
Security Hotspots

Maintainability Measures

6h A
Debt

100
Code Smells

Coverage Measures

79.4%
Coverage

39
Unit Tests

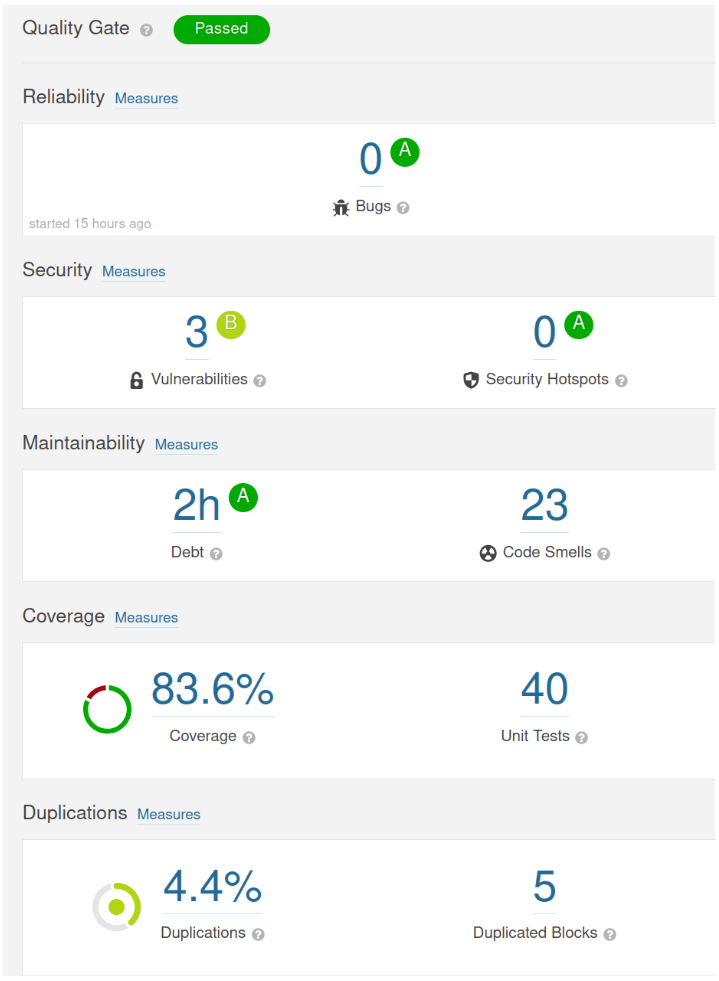
Duplications Measures

4.7%
Duplications

5
Duplicated Blocks

Figure 15 - Initial SonarCloud Analysis

The final analysis from sonarcloud was as follows:


Quality Gate Passed

Reliability Measures

0 A
Bugs

started 15 hours ago

Security Measures

3 B
Vulnerabilities

0 A
Security Hotspots

Maintainability Measures

2h A
Debt

23
Code Smells

Coverage Measures

83.6%
Coverage

40
Unit Tests

Duplications Measures

4.4%
Duplications

5
Duplicated Blocks

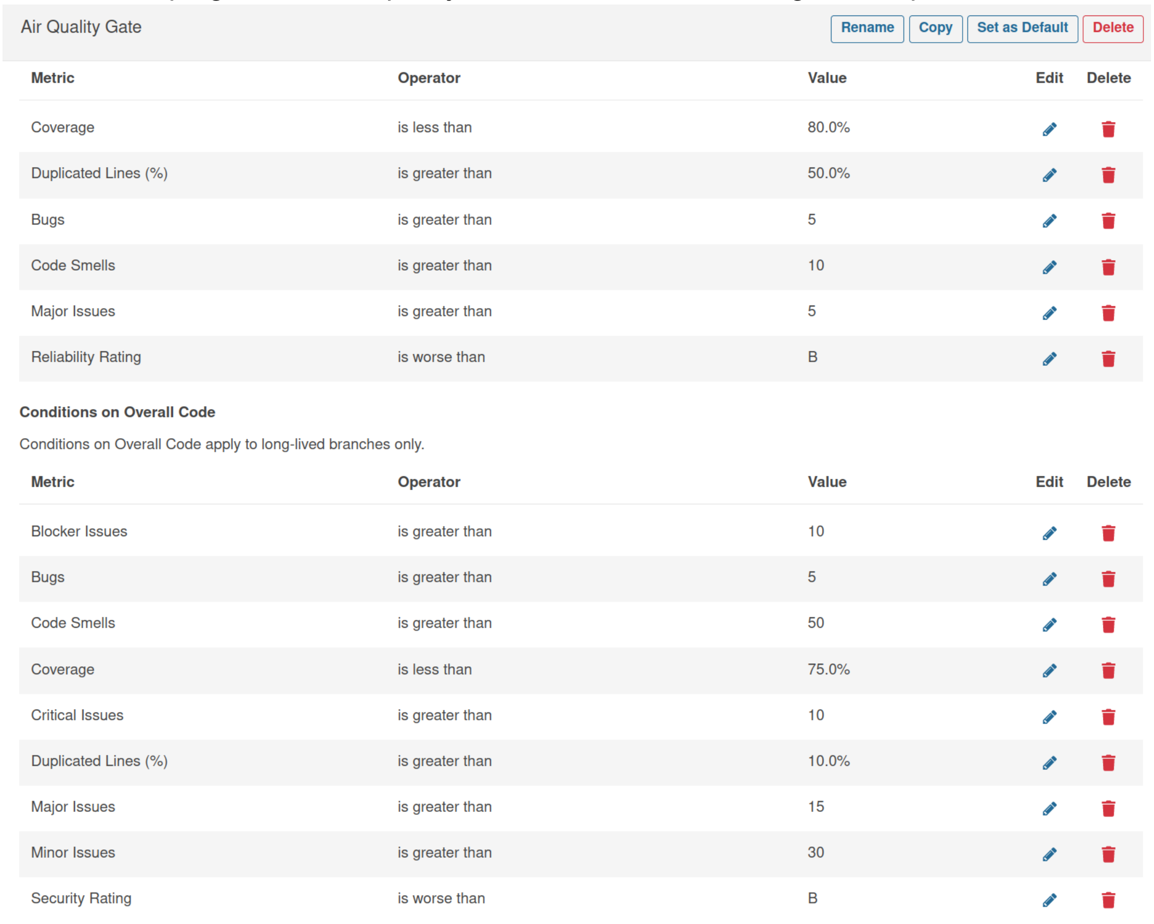
Figure 16 - Final SonarCloud Analysis

In this analysis we can see that code coverage is very well rated 82% coverage, the number of code smells and vulnerabilities is also relatively low, especially compared to the first analysis. When I first analyzed the project there were over 100 code smells and code coverage was about 10% lower.

By using the tips provided it was a lot simpler to both detected and correct the flaws in the code developed, one code smell that was very prevalent was about using the "var" keyword, which I probably had never used when coding in java. There were also a few vulnerabilities regarding the usage of direct user input in urls (which would be a terrible risk in a real world project, for example, when developing a web site).

This was all made easier due to the implementation of SonarCloud in the GitHub repository, which made every push be analyzed (in case of successful tests), giving instant feedback on the code developed. To help with this, I created a new Quality Gate to make sure that quality wasn't worsening over time.

On the overall code I created some rules to help guide the project and assure its quality, like keeping coverage above 75% and maintaining a security rating of at least B. I also added limits to code smells, bugs, etc, mostly to help with maintain clean code, since code smells can really begin to pile up even in smaller projects, having an analysis tool really helps prevent some causes of un-maintainable code. I also added a few metrics for new code to assist in keeping the overall quality consistent, without adding a lot of problems at once.


Air Quality Gate

Metric	Operator	Value	Edit	Delete
Coverage	is less than	80.0%		
Duplicated Lines (%)	is greater than	50.0%		
Bugs	is greater than	5		
Code Smells	is greater than	10		
Major Issues	is greater than	5		
Reliability Rating	is worse than	B		

Conditions on Overall Code

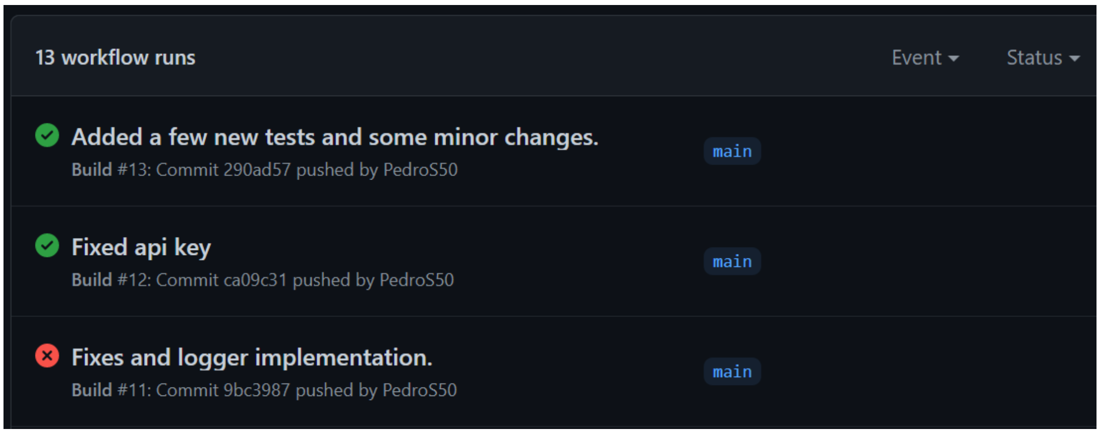
Conditions on Overall Code apply to long-lived branches only.

Metric	Operator	Value	Edit	Delete
Blocker Issues	is greater than	10		
Bugs	is greater than	5		
Code Smells	is greater than	50		
Coverage	is less than	75.0%		
Critical Issues	is greater than	10		
Duplicated Lines (%)	is greater than	10.0%		
Major Issues	is greater than	15		
Minor Issues	is greater than	30		
Security Rating	is worse than	B		

Figure 17 - Quality Gate

3.5 Continuous integration pipeline

As stated in the section above, I used the CI functionalities of GitHub to both run tests every time I made a push to the repository, as well as to run the SonarCloud analysis. This actually helped identify a mistake of having set a limit of requests too low on the Geocoding API Key which often made tests fail for no apparent reason.


13 workflow runs

Added a few new tests and some minor changes.
Build #13: Commit 290ad57 pushed by PedroS50 — main

Fixed api key
Build #12: Commit ca09c31 pushed by PedroS50 — main

Fixes and logger implementation.
Build #11: Commit 9bc3987 pushed by PedroS50 — main

Figure 18 - CI Pipeline Workflows

4 References & resources

Project resources

- Video demo: https://www.youtube.com/watch?v=BM0wgsvIC_8
- GitHub Repository: https://github.com/PedroS50/tqs_air_quality
- QA dashboard: https://sonarcloud.io/dashboard?id=PedroS50_tqs_air_quality

Reference materials

- https://openweathermap.org/
- https://developers.google.com/maps/documentation/geocoding/overview
- https://www.javainuse.com/spring/boot_swagger
- https://www.baeldung.com/rest-template
- https://restfulapi.net/http-status-codes/