HW1: Mid-term assignment report

Alexandre Lopes [88969], v2020-04-13

1 Introduction

1.1 Overview of the work

This project assignment was proposed by Professor Ilídio Oliveira in the scope of the Software Quality and Tests course.

The goal of the project was to develop a multi-layer web application in Spring Boot with automated tests of different types (unit tests, service level tests, integration tests, and functional testing, mainly on the web interface).

This application should provide details on air quality for a certain region (provided by the user) showing metrics like particles in suspension or gases present. To achieve these objectives, the project should include different components: a web page, that allows the interaction with the user; integration with external sources, like a third-party API to fetch that air quality data; an in-memory cache to save the latest results from that API; and its own REST API that can be called by external clients and obtain air quality data as well.

The application I will present in this report, simply named “TQS Air Quality Meter”, met the goals described above, as it has the different components and tests mentioned. Using my application through its web page it’s possible to enter a location in a search bar and to see current air quality data, if available, for that location, such as the air quality index, the dominant and remaining pollutants, as well as their concentration. The same data is also provided by the REST API developed, with some extras like cache usage statistics (hits, misses and number of requests for each location and also globally) and timestamps that provide context about the time that data corresponds to.

1.2 Limitations

Due to the simplicity of the web page requested for this assignment, this project did not suffer from major limitations. However, the Breezometer API, which I used to fetch air quality information, also provides historical and forecast data about air quality details, which would be useful to show as well on the web interface. However, due to time constraints and the volume of work requested by the teachers of all courses of the semester when this application was developed, those features had a low priority and were not implemented.

Concerning the APIs, since the MapQuest API (which I used for geolocation) is free up to 15000 transactions per month, its invalidity might not be a problem due to the scope of this assignment. As for the Breezometer API, since it's limited to a trial, it goes from working for all countries supported to only one after the trial expires. If by the time you try this project, the key had already expired, you can create a free account in Breezometer site and generate your own. Then, you can copy that key and replace the one in the file "breezometerKey.txt" in "src/main/java/com/example/airquality". In this same directory, you will also find "mapquestKey.txt" to replace the key in it by your MapQuest key if you have problems with that key too.

2 Product specification

2.1 Functional scope and supported interactions

The developed application allows, through the web page, to search by a location name, writing it in the web page search bar, and check information about current air quality in that location, if such data is available. The intent is to provide the users that are concerned about the quality of the air they are breathing a simple and minimalist tool that shows the data they are looking for with no great complexity.

However, besides the straightforward web page it provides to the end-users, it also provides its own API with similar data that developers can use. More details about its usage and the data it provides are discussed further in.

2.2 System architecture

To better structure the development of this application, I decided to have the various entities of the project divided into packages, according to their concerns, following some Spring Boot conventions. The general architecture of the project is the following:

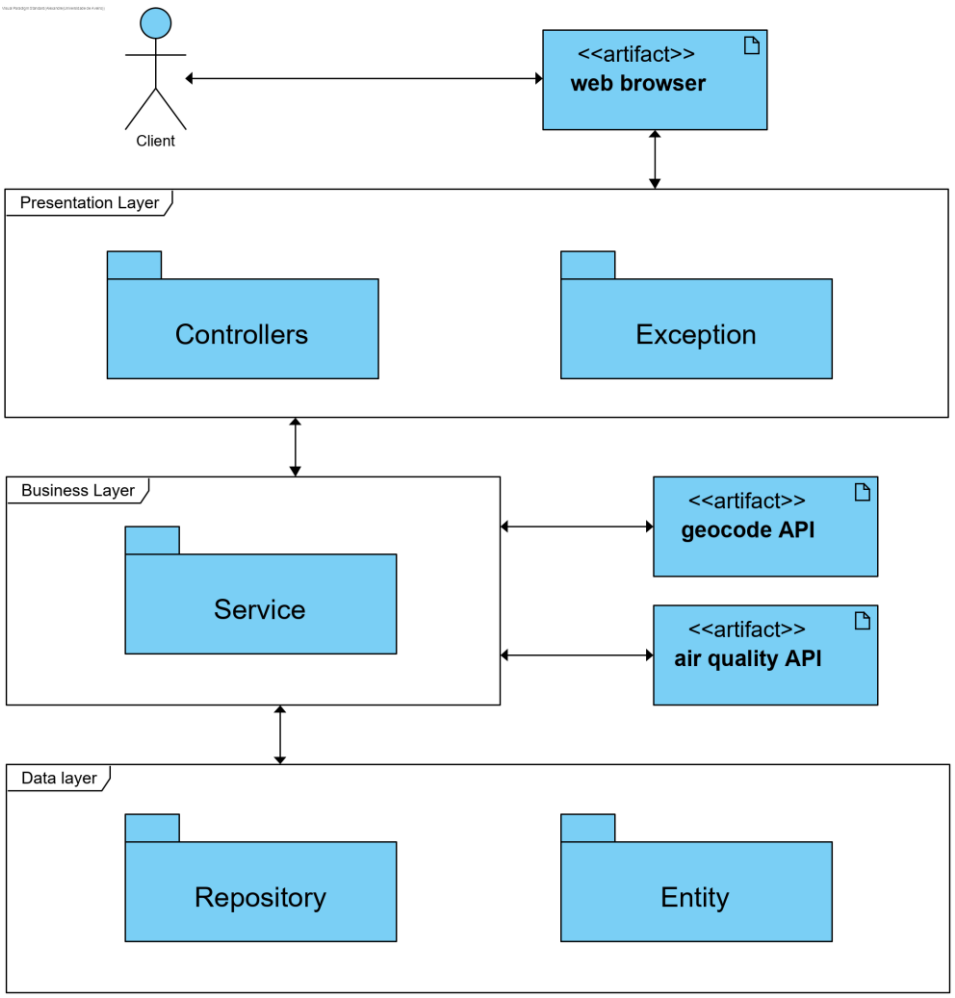


Figure 1 – General application architecture

The client, that is, the end-user, accesses the application through the webpage, whose layout is the result of the presentation layer components processing. This layer consists of a Rest (Web) Controller of the Spring Framework that maps each URL of the application to a certain layout. Besides that, this controller (Report Controller) also uses a model to get the right data to present to the user at each layout.

In the business layer, we have a Service of the Spring Framework that will be the intermediate between the layout and the data it needs. The ReportService will be responsible for taking the input of the user and retrieve the air quality report for the corresponding location. To do this, the service, through an HTTP client (ReportHttpClient), will make a request to the MapQuest API to geocode the location string provided by the user, saving its coordinates, address, and country. With this information, the service will then fetch the Breezometer API to get the actual data related to air quality and build the Report object with relevant data to the user.

As I said, the service would be the intermediate between the presentation layer and the data, but this data does not come exclusively from external sources directly each time a request is made. This is because the service also manages and accesses the data layer, which consists of an in-memory database, a repository (ReportRepository), to provide a cache mechanism in order to save the data from each new location requested to retrieve it later if the same location is requested again. The goal of this approach is to avoid fetching the external APIs when a location was already requested before, avoiding some latency and lowering of performance as well as eventual network issues. It's important to note that, since we are talking about air quality data, which measures change in time, even if the same location is requested again during the lifecycle of the application, the service will fetch the Breezometer API if this request occurs in a different hour or day of the cached results for that location. This happens because, usually, Breezometer's data is updated every hour and my objective is to provide to the user the most recent data available (I shall note that, for example, if you make

a request at 15:50 and another at 16:00, you might see the cached version from 15:50. This is because I gave a two-minute margin to Breezometer to update its data. So, you will only see data updated by Breezometer at 16:00 if your request to the API I developed is done at 16:02 or later). This mechanism also provides (through the REST API I developed) statistics about the cache usage, including hits and misses (number of requests that were redirected to cache or to Breezometer API, respectively) as well as the total number of requests made. These statistics are measured both for each location and globally.

To finish this topic, I would like to leave some more notes about the structure of this project.

First, since the air quality data to cache is saved in a repository, I created some entities to organize that same data. The main entity if, of course, the Report. This report will contain many other entities, represented by classes/objects, that contain data related data. For example, to save the data related to each pollutant, I created a class to encapsulate all the pollutant data within an object that a Report contains. The goal was to better structure the entities and the data, in order to have a more readable implementation. Besides that, the JSON object that is retrieved by the REST API also benefits from this structure, as each object that a Report contains is mapped to another JSON object, creating a more human-readable and structured document.

Secondly, and talking about the REST API, I also implemented another Controller for this matter. This controller, the ReportRestController, is a Rest Controller supported by Spring Boot Framework that defines the API endpoint and maps it to the location provided as a path variable in the URL (more on the API below).

The diagram directly below shows the global structure of the application, with the entities I mentioned bellow, and the following ones show the structure within each package for a clearer observation.

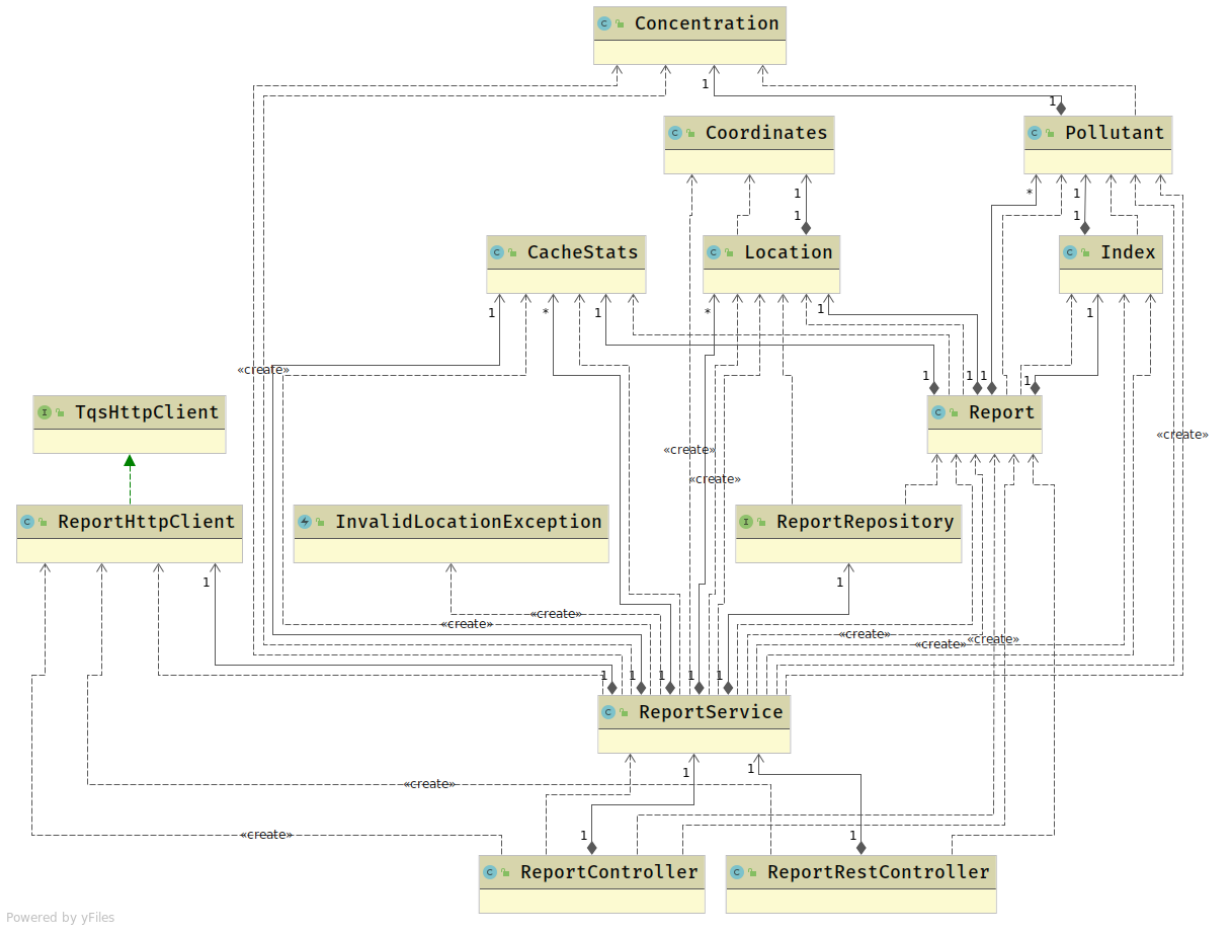


Figure 2 – Overall app diagram

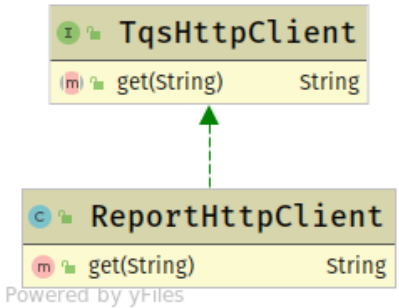
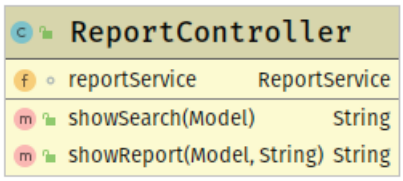


Figure 3 – Client package diagram



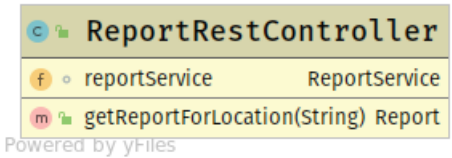


Figure 4 – Controller package diagram

Figure 5 – Repository package diagram

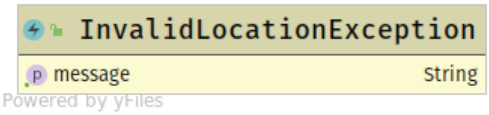


Figure 6 – Exception package diagram

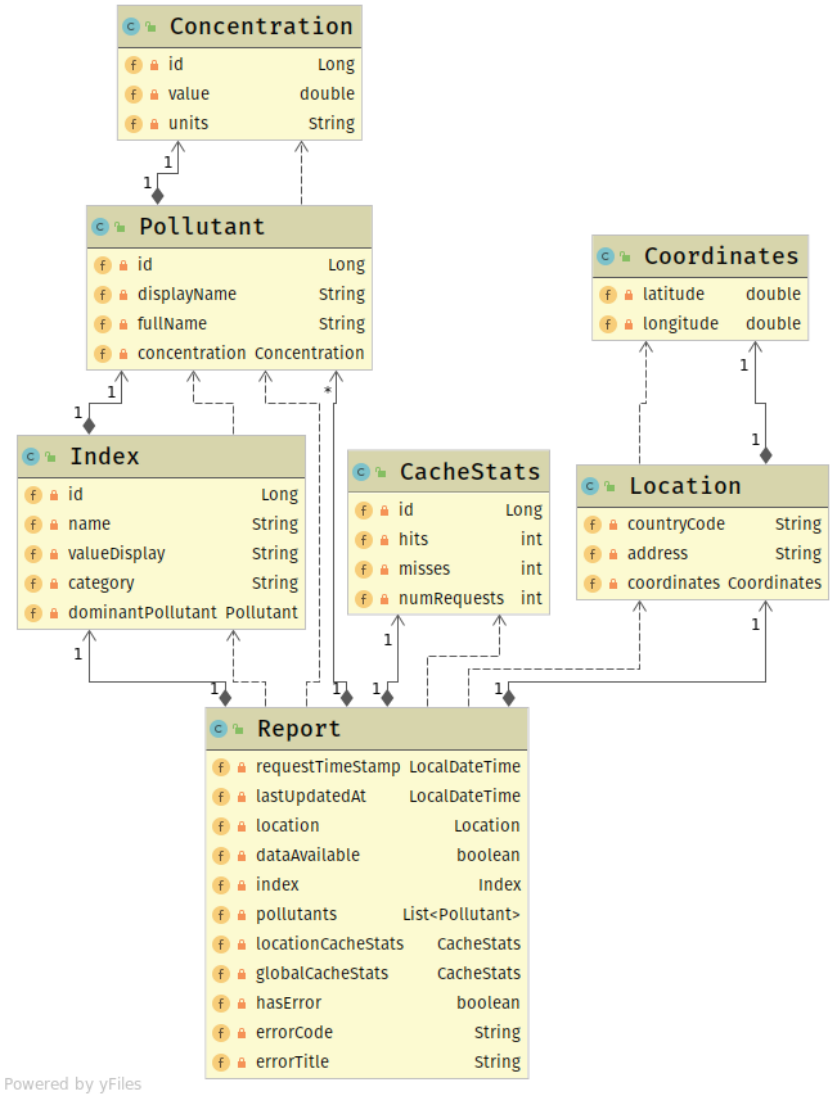


Figure 7 – Entity package diagram

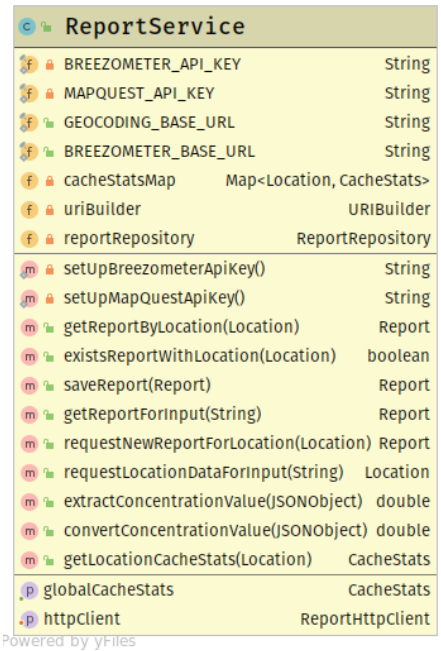


Figure 8 – Service package diagram

2.3 API for developers

As I said before, the developed application provides a REST API that can be used by developers to also get data about air quality for a location that can they use for their projects.

Given a location, this API provides data related to the air quality for that location, if such data is available. The data retrieved by this API is the same data as presented on the webpage (information about the location requested; air quality index metrics like its value and classification; as well as the present pollutants - their name and concentration), with some extras like cache usage statistics, some timestamps and error information that can be useful to developers.

The use of this API is very simple, as developers only have to fetch the only endpoint it provides. This endpoint requires that developers specify the location they want to access data about, if available. Details about this endpoint follow below.

2.3.1 Show Air Quality Report

Returns JSON data about the air quality for a single location name

URL
/api/location/:location

Method
GET

URL Params
location=[string] (required)

Data Params
None

Success Response

Code
200 OK

Content


```
{
    "requestTimeStamp": "2020-04-11T11:24:58",
    "lastUpdatedAt": "2020-04-11T11:00:00",
    "location": {
        "countryCode": "PT",
        "address": "Aveiro, Aveiro",
        "coordinates": {
            "latitude": 40.640496,
            "longitude": -8.653784
        }
    },
    "dataAvailable": true,
    "index": {
        "name": "BreezoMeter AQI",
        "valueDisplay": "70",
        "category": "Good air quality",
        "dominantPollutant": {
            "displayName": "O3",
            "fullName": "Ozone",
            "concentration": {
                "value": 38.39,
                "units": "ppb"
            }
        }
    },
    "pollutants": [
        {
            "displayName": "NO2",
            "fullName": "Nitrogen dioxide",
            "concentration": {
                "value": 4.65,
                "units": "ppb"
            }
        },
        {
            "displayName": "O3",
            "fullName": "Ozone",
            "concentration": {
                "value": 38.39,
                "units": "ppb"
            }
        },
        {
            "displayName": "PM2.5",
            "fullName": "Fine particulate matter (<2.5µm)",
            "concentration": {
                "value": 2.69,
                "units": "ug/m3"
            }
        },
        {
            "displayName": "SO2",
            "fullName": "Sulfur dioxide",
            "concentration": {
                "value": 0.17,
                "units": "ppb"
            }
        },
        {
            "displayName": "PM10",
            "fullName": "Inhalable particulate matter (<10µm)",
            "concentration": {
                "value": 2.6,
                "units": "ug/m3"
            }
        },
        {
            "displayName": "CO",
            "fullName": "Carbon monoxide",
            "concentration": {
                "value": 171.94,
                "units": "ppb"
            }
        }
    ],
    "locationCacheStats": {
        "id": null,
        "hits": 0,
        "misses": 1,
        "numRequests": 1
    },
    "globalCacheStats": {
        "id": null,
        "hits": 1,
        "misses": 2,
        "numRequests": 3
    },
    "errorCode": "NA",
    "errorTitle": "NA"
}
```

Response Data Fields
requestTimeStamp: string
ISO 8601 UTC timestamp indicating the time a request was made to the Breezometer API

lastUpdatedAt: string
ISO 8601 UTC timestamp indicating the time the data refers to

Location: object

- **countryCode:** string - requested location country code
- **address:** string - requested location address (“city”, “county”)
- **coordinates:** object
    - **latitude:** double - north-south position of a point in a range between -90 and 90
    - **longitude:** double - east-west position of a point in a range between -180 and 180

dataAvailable: boolean
True/false indicating whether the expected data is available

Index: object

- **name:** string - index name
- **valueDisplay:** string - textual representation of the index numeric score
- **category:** string - interpretation of the index numeric score
- **dominantPollutant:** object
    - **displayName:** string - pollutant name as acronym/chemical formula
    - **fullName:** string - pollutant full name
        - **concentration:** object
            - **value:** double - Value of pollutant concentration
            - **units:** string - Units for measuring this pollutant concentration

Pollutants: object
List of available pollutants at the requested location

- **displayName:** string - pollutant name as acronym/chemical formula
- **fullName:** string - pollutant full name
    - **concentration:** object
        - **value:** double - Value of pollutant concentration
        - **units:** string - Units for measuring this pollutant concentration

locationCacheStats: object
Statistics about the usage of the service cache for requested location

- **hits:** int - times the data for the requested location was retrieved from cache
- **misses:** int - times the data for the requested location was retrieved from external source
- **numRequestes:** int - total of requests made to this API for the requested location

globalCacheStats: object
Statistics about the usage of the service cache for all requested locations

- **hits:** int - times the data for a location was retrieved from cache
- **misses:** int - times the data for a location was retrieved from external source

- **numRequestes:** int - total of requests made to this API

errorCode: string

errorTitle: string
Explanation about the cause of the error

Error Response

Code
404 NOT FOUND

Content

```
{
    "timestamp": "2020-04-11T14:58:38.243+0000",
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/api/location/"
}
```

3 Quality assurance

3.1 Overall strategy for testing

To assure that the developed application would have quality in its implementation I implemented some tests after most of the implementation of the main app was done.

To implement the tests, I followed the suggestions on this project guidelines and wrote unit tests to test individual situations, service level tests to check if the business logic implemented in the service was functioning properly, integration tests to guarantee that all the app components communicate with each other and also functional tests on the web interface to ensure that it reacts as expected to user interaction. In the final submission, all tests were passing.

The technologies I chose to write these tests include JUnit, Mockito, Spring Boot MockMvc, Selenium WebDriver and, for static code analysis, SonarQube and JaCoCo plugin.

3.2 Unit and integration testing

For the unit testing, I wrote tests to situations related to the repository and service functionality, as well as some cache behavior.

For the repository tests, I considered the cases when a report for a location is requested from this in-memory database. This repository must return a report for a previously requested location and return null if we are asking the repository for data of a never requested location. The same applies when we ask for the existence of a report for that location: if the location is new, then the repository must retrieve false, and true otherwise.

```java
@DataJpaTest
public class ReportRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ReportRepository reportRepository;

    @Test
    public void whenFindByPreviouslyRequestedLocation_thenReturnReport() {...}

    @Test
    public void whenFindByNeverRequestedLocation_thenReturnNull() {...}

    @Test
    public void whenCheckExistenceWithPreviouslyRequestedLocation_thenReportShouldExist() {...}

    @Test
    public void whenCheckExistenceWithNeverRequestedLocation_thenReportShouldNotExist() {...}
```

Figure 9 – Repository tests

For the service unit testing I used Mockito to mock the repository behavior and, so, I defined expectations and verifications on its methods. The situations I covered in these tests are related to whether a Report for a Location exists or can be found on the cache (which is accessed by the service) depending on whether the Location has already been requested or not. Besides that, I also tested if the service could retrieve a report and geocode a location given a valid input and if it can't if the input is invalid. Finally, I also proceeded to test the cache statistics management by the services, testing if the number of hits, misses and requests change as expected while the application is running. As for the cache behavior itself, I just tested if its methods to add hits, misses and number of requests worked properly.

```java
@ExtendWith(MockitoExtension.class)
public class ReportServiceUnitTest {

    @Rule
    public MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);

    @Mock(lenient = true)
    private ReportRepository reportRepository;

    @InjectMocks
    private ReportService reportService;

    @BeforeEach
    public void setUp() {
        reportService.setHttpClient(new ReportHttpClient());
        Location aveiro = new Location(new Coordinates( latitude: 40.640496,  longitude: -8.653784),  countryCode: "PT", address: "Aveiro, Aveiro");

        Report aveiroReport = setUpAveiroReport();
        Report cachedAveiroReport = setUpCachedReportFrom(aveiroReport);


        Report antarcticaReport = setUpAntarcticaReport();

        // non requested location yet
        Location castanheiraDePera = new Location(new Coordinates( latitude: 40.00405,  longitude: -8.202775), countryCode: "PT", address: "Castanhei

        Mockito.when(reportRepository.existsById(aveiro)).thenReturn(false).thenReturn(true);
        Mockito.when(reportRepository.save(aveiroReport)).thenReturn(cachedAveiroReport);
        Mockito.when(reportRepository.findById(aveiro)).thenReturn(Optional.of(cachedAveiroReport));
        Mockito.when(reportRepository.findById(antarcticaReport.getLocation())).thenReturn(Optional.of(antarcticaReport));
        Mockito.when(reportRepository.findById(castanheiraDePera)).thenReturn(Optional.empty());
    }
```

Figure 10 – Service testing: expectations

Figure 11 – Service testing: tests

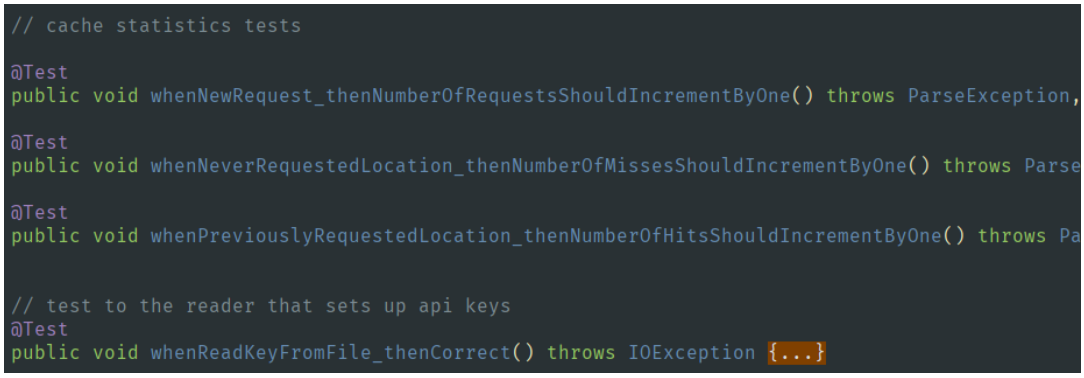


Figure 12 – Service testing: cache and reader tests

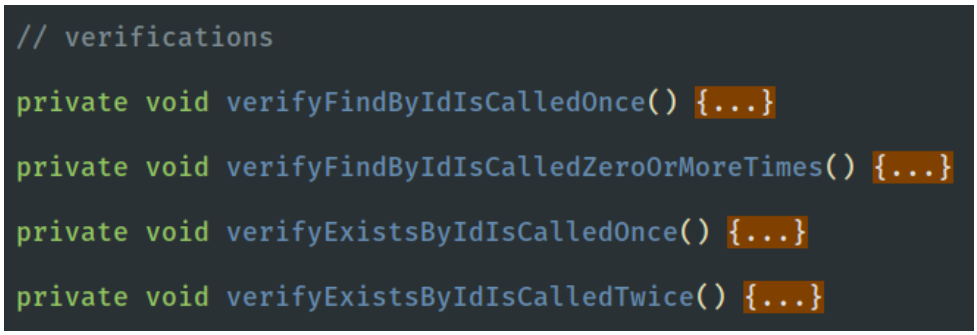


Figure 13 – Service testing: verifications

For the integration testing on the API, I run tests both on the client and server sides. From a point of view of a client, I tested if the server was returning a JSON object as expected.

```java
@WebMvcTest(ReportRestController.class)
public class ReportControllerIT {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private ReportService reportService;


    @Test
    public void givenLocationName_whenGetReportForThatLocation_thenReturnJson() throws Exception {...}
```

Figure 14 – Report Controller test (client side)

From the server perspective, I tested if the server was caching the report properly and its messages retrieved the expected HTTP status (200 OK when valid inputs/locations and 400 NOT FOUND otherwise). For these, I did a version with the Spring MockMvc and a REST Client to create realistic requests (TestRestTemplate).

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, classes = AirQualityWebAppApplication.class)
@AutoConfigureMockMvc
@AutoConfigureTestDatabase
public class ReportRestControllerIT {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private ReportRepository reportRepository;

    @AfterEach
    public void resetDb() { reportRepository.deleteAll(); }

    @Test
    public void whenValidLocationName_whenGetReportForThatLocation_thenCacheReport() throws Exception {...}

    @Test
    public void whenValidLocationName_whenGetReportForThatLocation_thenStatus200() throws Exception {...}

    @Test
    public void whenInvalidLocationName_whenGetReportForThatLocation_thenStatus400() throws Exception {...}
```

Figure 15 – Report Rest Controller testing with MockMvc

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureTestDatabase
public class ReportRestControllerTemplateIT {

    @LocalServerPort
    int randomServerPort;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private ReportRepository reportRepository;

    @AfterEach
    public void resetDb() { reportRepository.deleteAll(); }

    @Test
    public void whenValidLocationName_whenGetReportForThatLocation_thenCacheReport() {...}

    @Test
    public void whenValidLocationName_whenGetReportForThatLocation_thenStatus200() {...}

    @Test
    public void whenInvalidLocationName_whenGetReportForThatLocation_thenStatus400() {...}
```

Figure 16 – Report Rest Controller testing with TestRestTemplate

3.3 Functional testing

For functional testing, I used Selenium Web Driver (ChromeDriver) and adopted the Page Object Pattern to have more readable and understandable tests. Thus, I created an object to represent the interfaces the user can see while using the application.

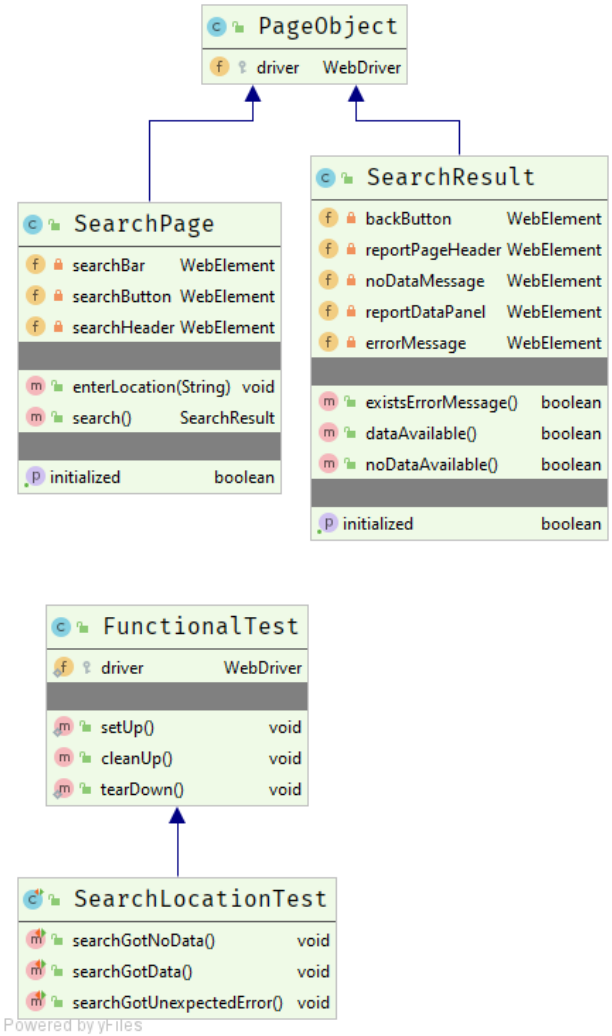


Figure 17 –Functional testing diagram with Page Object Pattern

To assure the flow of the application went right and the data that is presented complies with what the user requested, I covered three main cases: when the search for a location got data successfully (in which the page shows the air quality data for that location after the user enters a valid input and clicks search); when the search for a location got the location right, but there's no data for that place (in which the user enters a valid input, clicks search and the result page shows the location address and coordinates as well but shows a message informing there's no data for the location entered); and, finally, when the search results in an error (in which the user clicks search after entering, for example, an invalid input and the result page shows no address or coordinates at all and shows a message informing there was an error and a description about it, when available).

```
public class SearchLocationTest extends FunctionalTest {

    public SearchLocationTest() {...}

    @Test
    public void searchGotNoData(){...}

    @Test
    public void searchGotData(){...}

    @Test
    public void searchGotUnexpectedError(){...}

}
```

Figure 18 – Functional testing with Page Object Pattern: situations covered

It's important to note that, for functional tests to pass, the application must already be running, as the Selenium WebDriver accesses the browser to check if the web interface is presented as expected.

3.4 Static code analysis

Static code analysis consists of an analysis of the code without executing it. The goal of this analysis if to detect possible programming errors, bugs, suspicious constructions, etc. Thus, it's an important piece to guarantee the quality of our implementation.

To run a static code analysis on this project, I used SonarQube with conjunction of JaCoCo plugin for Maven to also have information about code coverage. The results follow below.

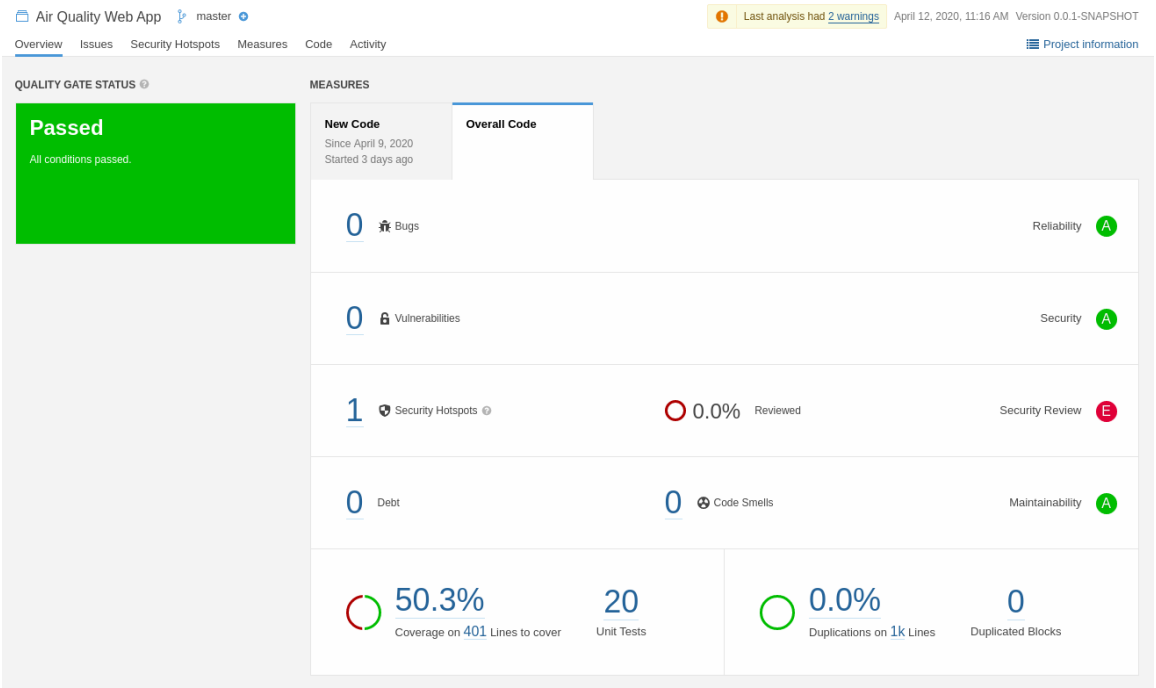


Figure 19 – Static code analysis results

For a project to get the green we see on the left it must pass all quality conditions defined in a quality gate. The quality gate I used for this project requires the following conditions:

Metric	Operator	Value
Coverage	is less than	50.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A

Figure 20 – Quality gate used for static code analysis

As you can see, my project has passed all conditions defined in this quality gate, having A in three last conditions, 0% of duplicated lines (maximum is 3%), and 50.3% of code coverage (the minimum is 50%). This value of code coverage seems low, but the reason I defined that way was due to the fact that SonarQube is reporting many uncovered conditions and lines related to autogenerated code (like getters, setters, equals, hashCode or toString), which, by common sense, is not worth testing.

The main issue we see in the project dashboard, however, is one major security hotspot that's related to the possible unsafe usage of command line arguments in the main class of the application. However, I ignored that issue as it's also autogenerated code by the Spring Boot framework.

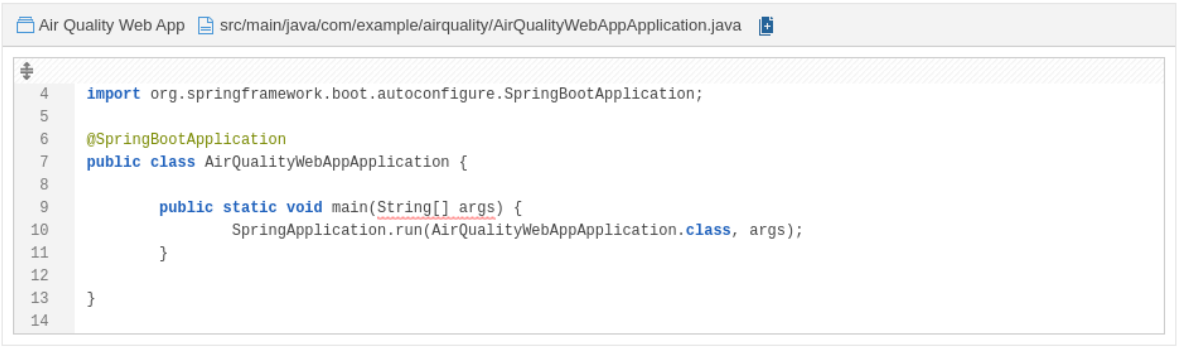


Figure 21 – Security hotspot detected during static code analysis

These results do not concern the first static code analysis I run. In fact, I had many code smells and bugs in the first analysis. This tool, thus, helped me take a better look at the coding I wrote by detecting, for example, encapsulate some extensive code in its own method for better readability or some redundant object instances and variables that I was not aware at the time of the writing of the code. It's indeed a helpful tool to improve the quality of our code and to incentive us, programmers, to look with different perspectives at our code.

4 References & resources

Project resources

- Git repository: https://github.com/alexandrejflopes/tqs-air-quality-app
- Video demo: https://drive.google.com/file/d/15m5EAIZ5kAJ_TS1NA8aYKiOym-02ayr-/view?usp=sharing

Reference materials

Spring Boot

- https://spring.io/projects/spring-boot
- https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-running-your-application

MapQuest API

- https://developer.mapquest.com/documentation/geocoding-api/address/get/

Breezometer API

- https://docs.breezometer.com/api-documentation/air-quality-api/v2/#current-conditions

Page Object Pattern

- https://www.pluralsight.com/guides/getting-started-with-page-object-pattern-for-your-selenium-tests

Mockito

- https://site.mockito.org/

Selenium

- https://www.selenium.dev/documentation/en/

SonarQube

- https://docs.sonarqube.org/latest/setup/get-started-2-minutes/

Other resources

- https://medium.com/backend-habit/generate-codecoverage-report-with-jacoco-and-sonarqube-ed15c4045885