

# HW1: Mid-term assignment report

*Rui Filipe Monteiro Fernandes, Nmec: 92952*

## Contents

<b>HW1: Mid-term assignment report.....</b>	<b>1</b>
<b>1   Introduction.....</b>	<b>1</b>
1.1   Overview of the work.....	1
1.2   Current limitations.....	1
<b>2   Product specification.....</b>	<b>1</b>
2.1   Functional scope and supported interactions.....	1
2.2   System architecture.....	2
2.3   API for developers.....	2
<b>3   Quality assurance.....</b>	<b>2</b>
3.1   Overall strategy for testing.....	2
3.2   Unit and integration testing.....	2
3.3   Functional testing.....	2
3.4   Static code analysis.....	2
<b>4   References &amp; resources.....</b>	<b>2</b>

## 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.

The Web Application developed, provides a simple interface to visualize air quality information of a desired location, furthermore a REST API is included in the application for more search options like by coordinates or current position.

### 1.2   Current limitations

The Web Application interface does not provide all search methods that the REST API has to offer, it also could show more detailed information in terms of the data.

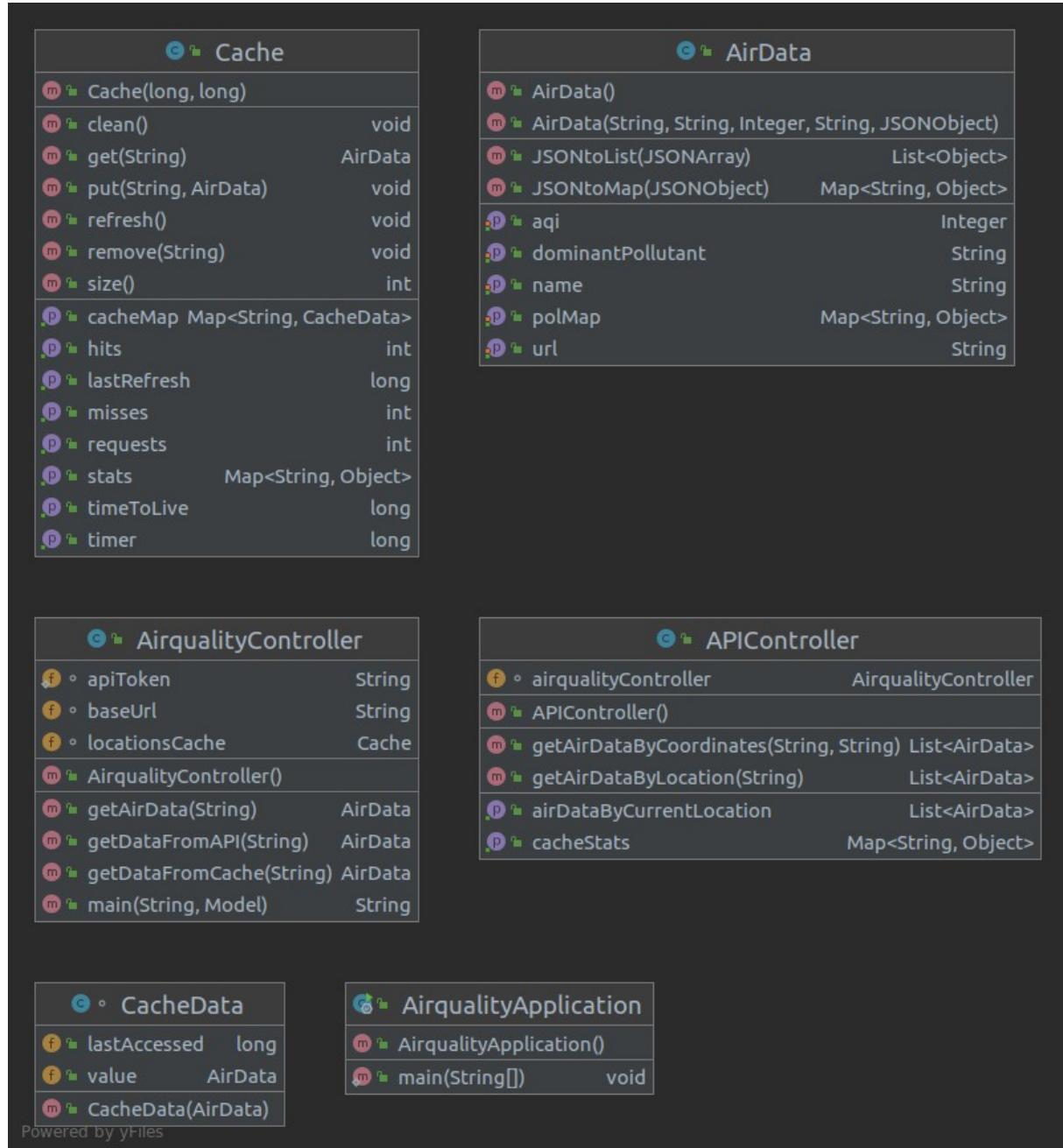
As for quality assurance, the testing could be much more expansive, and the functional tests with selenium are not fully integrated with JUnit5.

## 2 Product specification

### 2.1 Functional scope and supported interactions

The Application only has one main page, where air quality information immediately loads for the city of Lisbon, and from there on the user can check the suggested cities by clicking on the buttons with their names on the left, or for a more free search, use the search bar to write their desired location.

### 2.2 System architecture



- **AirData** → Class that models the air quality data for a certain location, it includes only basic information and is modeled to the Web application's REST API format.
- **Cache** → Structure used to simulate a cache, all stored data has a designated time to live, and the cache itself has a timer at which it will refresh to see if any data has exceeded its time to live and removes it. Includes some statistics such as number of requests, and how many of them were hits or misses.
- **CacheData** → Simple class that is used by the cache to store a single data fragment, this is to include a last accessed statistic.
- **AirQualityController** → Controller that fetches data from both the cache or the external API, it also is used to display the main web page.
- **APIController** → Controller that handles and provides the internal API, works in tandem with the AirQualityController and the endpoints it provides are explained in the following section.

### 2.3 API for developers

- **/api/location/{location}** → Returns the air quality information for the given location, this includes basic information like the name of the monitoring station for that location, its website url, the air quality index, the main pollutant and all the individual ratios of pollutants.
- **/api/coordinates?lat=x1&lng=x2** → Returns the air quality information for the given latitude and longitude.
- **/api/here** → Returns the air quality information for the station nearest to the user's location, based on the IP address information.
- **/api/cache** → Returns statistics of the internal cache usage, such as hits and misses, as well as the data still stored in the cache.

## 3 Quality assurance

### 3.1 Overall strategy for testing

Simple unit testing for the Cache class, service level tests for the AirQualityController with dependency isolation resorting to Mockito. Integration tests for the internal api features, resorting to MockMVC, and functional tests for the web page with Selenium.

### 3.2 Unit and integration testing

Unit testing for the Cache class to ensure its methods all work as intended, testing was done on time to live, basic functionality, as well as correct statistic retrieval.

The integration tests simply simulated calls to the internal API's endpoints and ensured the retrieved data had the correct format.

### **3.3 Functional testing**

Functional test use cases were extremely simple, choosing a city then checking if the retrieved data corresponds.

Selenium was used for functional testing, by simulating the use cases with the selenium IDE, inserting asserts and exporting to java. Some obstacles were found, since the search bar was on a side bar menu, selenium was having difficulties finding the page element and so the related use cases were dropped. These could have been to input an invalid location and check if the web page correctly displayed an error.

### **3.4 Static code analysis**

Static code analysis was done with Sonarlint, an extension for IntelliJ that provides many analysis rules. It was used mainly for minor code smells like renaming variables and removing redundancy.

## **4 References & resources**

### **Project resources**

- Video demo included in repository.

### **Reference materials**

- Practical and Theoretical class materials.
- SpringBoot Documentation
- Baeldung
- Stack Overflow