universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Pedro Miguel Alves Silva [93011], 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.
The following project was made by utilizing the Spring framework and displays Air Quality statistics when provided with a city, and optionally, a state and a country and then submitting that information, receiving in return not only it's AQI score but also the concentration of pollutants in the air, it possess both an API and a website.
To get this air pollution data the OpenWeatherMap Pollution API is used and in order to translate from the triple of parameters given (city, state, country) it's used the OpenWeatherMap Reverse Geocoding API.
The system returns by way of its API in JSON format, results for air pollution data and statistics for its cache.
In all operations the cache timeout is 30 sec.

1.2 Current limitations

A feature that could be implemented still is the use of another API as an alternative and, if necessary, an AirQuality class that can be used for both, as a different API might include different data.

Another feature that could also be implemented would be the ability to show both the forecast and history of a location when provided with a date instead of only being able to see data for today.
Finally, improvements could be the done to the web interface to allow users to introduce more easily a country instead of requiring the use of a country's code.

2 Product specification

2.1 Functional scope and supported interactions

The application allows users to see the air pollution data at the moment in a specified location given by a city, and optionally by a state and country.
The application possesses both an API and a single page website. The data returned by the API includes it's AQI rating, which is given according to OpenWeatherMap API as a score from 1-5, where score 1 indicates good air quality and score 5 indicates very hazardous air quality, the API also returns the concentrations of various pollutants in the air (CO, NO, NO2, O3, SO2, PM25, PM10, NH3).Taking into account this information should allow any user to understand the potential health risks of going to some more dangerous place. The API also returns data about the cache used, including total number of requests, number of successful requests to the cache, number of failures requests to the cache, and the size of the cache at that moment.
In the website it's possible for a user insert a city, state and country code and by clicking in the search button receive the same information the API gives presented in the webpage, including an indication of the main pollutant in that area.

2.2 System architecture

The system is composed of four components as can be seen in Diagram 1.
The AirQualityController uses the Thymeleaf template engine to make a web page by using what data was requested and returns it to the user. Both AirQualityAPIController and AirQualityController request data from the AirQualityService. AirQualityService first calls the OpenWeatherMap to reverse geocode the location given by the user into a pair of coordinates (latitude and longitude) followed by then requesting if data for said coordinates already exists in cache and if it has not been in there for more that TTL (30 seconds), if yes then it returns cached data, if not it follows up by requesting OpenWeatherMap again, in order to get the latest pollution data for the locality, after receiving it, saves it to cache and returns it to Controller.

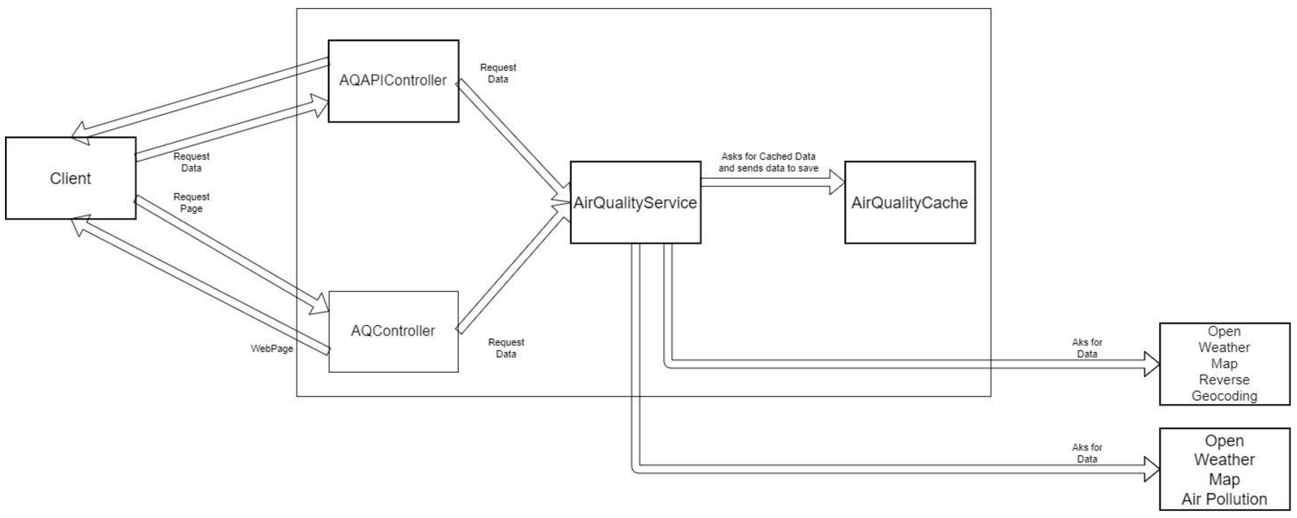


Diagram 1 – System Architecture

2.3 API for developers

The API contains two endpoints, one for each group (both use the root "/api"):

- Problem Domain – contains the information usually given to a website user plus the coordinates used for the query, when the request was made to the API (timestamp_request) and when did the API record those values (date).

  Endpoint: /now?city=<cityName>&state=<stateName>&country=<countryCode>

  Both state and country are optional.
- Cache usage – returns information about the cache including total number of requests, number of successful requests to the cache, number of failures requests to the cache, and the size of the cache at that moment.

  Endpoint: /statistics

3 Quality assurance

3.1 Overall strategy for testing

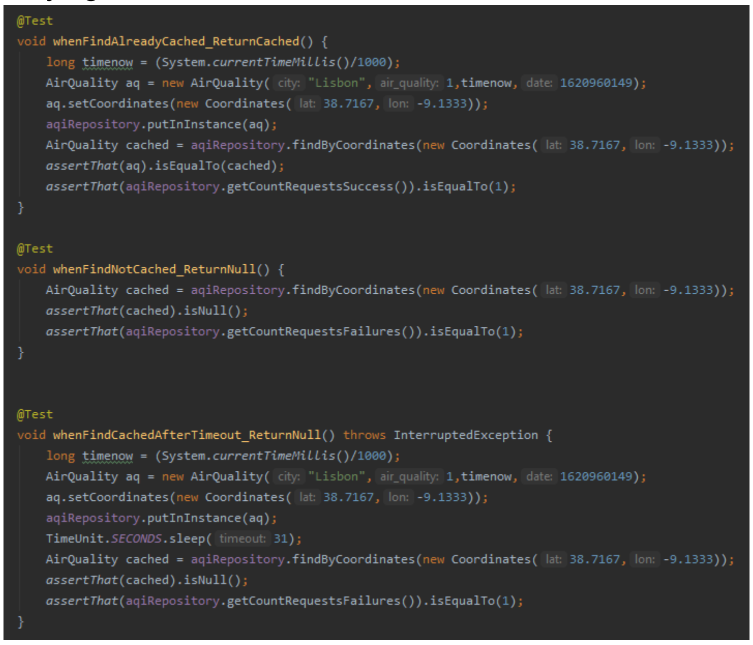
Each component in the system was though out beforehand and its methods decided, following that part of the functionality of some was made, while the rest was made after writing tests for them. Taking that into account test-driven development wasn't used during the whole project but was used in a big portion of the project.
For the testing, different tools like the Mockito framework used for both tests in the API and the service component, MockMvc was used in API tests and Selenium was used for interface testing by using SeleniumJupiter.

3.2 Unit and integration testing

Unit tests were made for verifying the operations made and results given by cache, including timeout.

The tests implemented are the following:

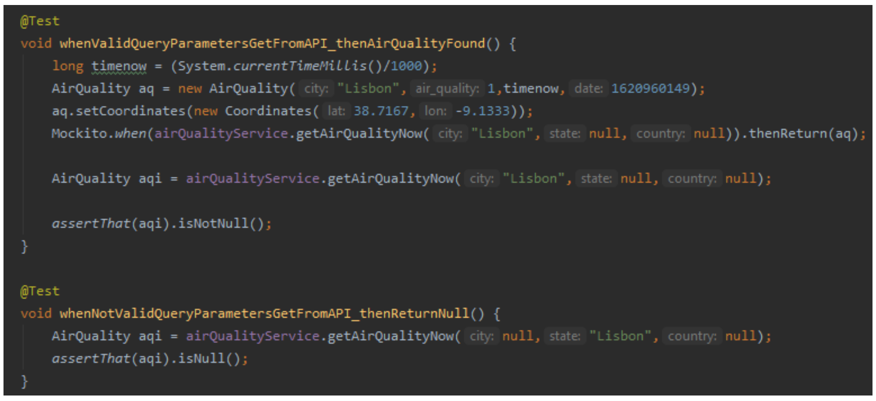
- Test if the cache is returning valid data after having saved it beforehand and being requested for it, the number of successful requests is also tested.
- Test if the cache is retuning null in case of requested of data not present in cache, number of failed requests is also tested.
- Test if TTL is working as intended, this is done by saving data, waiting TTL+1 seconds and then verifying the data is unavailable.



```java
@Test
void whenFindAlreadyCached_ReturnCached() {
    long timenow = (System.currentTimeMillis()/1000);
    AirQuality aq = new AirQuality( city: "Lisbon", air_quality: 1,timenow, date: 1620960149);
    aq.setCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    aqiRepository.putInInstance(aq);
    AirQuality cached = aqiRepository.findByCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    assertThat(aq).isEqualTo(cached);
    assertThat(aqiRepository.getCountRequestsSuccess()).isEqualTo(1);
}

@Test
void whenFindNotCached_ReturnNull() {
    AirQuality cached = aqiRepository.findByCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    assertThat(cached).isNull();
    assertThat(aqiRepository.getCountRequestsFailures()).isEqualTo(1);
}


@Test
void whenFindCachedAfterTimeout_ReturnNull() throws InterruptedException {
    long timenow = (System.currentTimeMillis()/1000);
    AirQuality aq = new AirQuality( city: "Lisbon", air_quality: 1,timenow, date: 1620960149);
    aq.setCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    aqiRepository.putInInstance(aq);
    TimeUnit.SECONDS.sleep( timeout: 31);
    AirQuality cached = aqiRepository.findByCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    assertThat(cached).isNull();
    assertThat(aqiRepository.getCountRequestsFailures()).isEqualTo(1);
}
```

Figure 1: AirQualityCacheTest

Tests were also made on the service class by using the framework Mockito in order to mock the behavior of the cache class and inject it into service class.
For the service class, there was a focus on testing if it got the correct data when passed a valid "query", as in set of (city, state, country) for both when needing to request the OpenWeatherAPI and when using the cache.
The opposite is also tested for when given an invalid "query". Lastly the ability of the cache of returning its statistics is also tested.

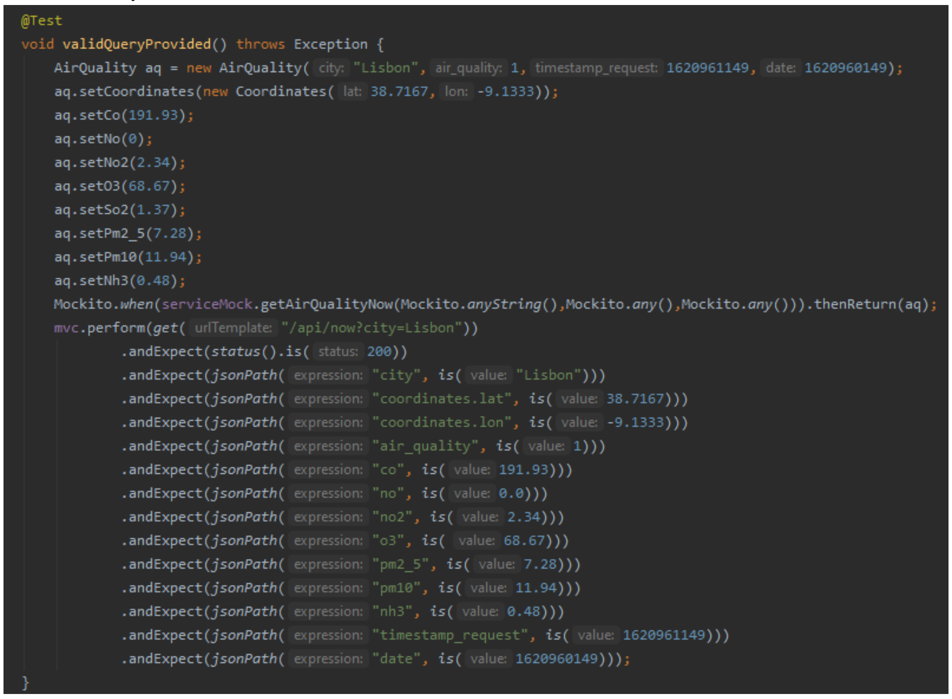


```java
@Test
void whenValidQueryParametersGetFromAPI_thenAirQualityFound() {
    long timenow = (System.currentTimeMillis()/1000);
    AirQuality aq = new AirQuality( city: "Lisbon", air_quality: 1,timenow, date: 1620960149);
    aq.setCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    Mockito.when(airQualityService.getAirQualityNow( city: "Lisbon", state: null, country: null)).thenReturn(aq);

    AirQuality aqi = airQualityService.getAirQualityNow( city: "Lisbon", state: null, country: null);

    assertThat(aqi).isNotNull();
}

@Test
void whenNotValidQueryParametersGetFromAPI_thenReturnNull() {
    AirQuality aqi = airQualityService.getAirQualityNow( city: null, state: "Lisbon", country: null);
    assertThat(aqi).isNull();
}
```

Figure 2: AirQualityServiceTest example of tests for API

Lastly, there were integration tests done in order to test both system controllers. To do this Spring's MockMvc is used alongside @MockBean to mock the service class.
The tests for the API controller are the following:

- Test if no city is given in the request, then no data is received by the user.
- Test if invalid values for parameters given in request, result is no data is received by the user.
- Test if a valid query is given in the request, then a JSON must be returned with all the parameters expected and valid values for each.
- Test if request is done for cache statistics, then a JSON must be returned with the parameters expected of it.



```java
@Test
void validQueryProvided() throws Exception {
    AirQuality aq = new AirQuality( city: "Lisbon", air_quality: 1, timestamp_request: 1620961149, date: 1620960149);
    aq.setCoordinates(new Coordinates( lat: 38.7167, lon: -9.1333));
    aq.setCo(191.93);
    aq.setNo(0);
    aq.setNo2(2.34);
    aq.setO3(68.67);
    aq.setSo2(1.37);
    aq.setPm2_5(7.28);
    aq.setPm10(11.94);
    aq.setNh3(0.48);
    Mockito.when(serviceMock.getAirQualityNow(Mockito.anyString(),Mockito.any(),Mockito.any())).thenReturn(aq);
    mvc.perform(get( urlTemplate: "/api/now?city=Lisbon"))
            .andExpect(status().is( status: 200))
            .andExpect(jsonPath( expression: "city", is( value: "Lisbon")))
            .andExpect(jsonPath( expression: "coordinates.lat", is( value: 38.7167)))
            .andExpect(jsonPath( expression: "coordinates.lon", is( value: -9.1333)))
            .andExpect(jsonPath( expression: "air_quality", is( value: 1)))
            .andExpect(jsonPath( expression: "co", is( value: 191.93)))
            .andExpect(jsonPath( expression: "no", is( value: 0.0)))
            .andExpect(jsonPath( expression: "no2", is( value: 2.34)))
            .andExpect(jsonPath( expression: "o3", is( value: 68.67)))
            .andExpect(jsonPath( expression: "pm2_5", is( value: 7.28)))
            .andExpect(jsonPath( expression: "pm10", is( value: 11.94)))
            .andExpect(jsonPath( expression: "nh3", is( value: 0.48)))
            .andExpect(jsonPath( expression: "timestamp_request", is( value: 1620961149)))
            .andExpect(jsonPath( expression: "date", is( value: 1620960149)));
}
```

Figure 3: AirQualityAPITest example test for API Controller

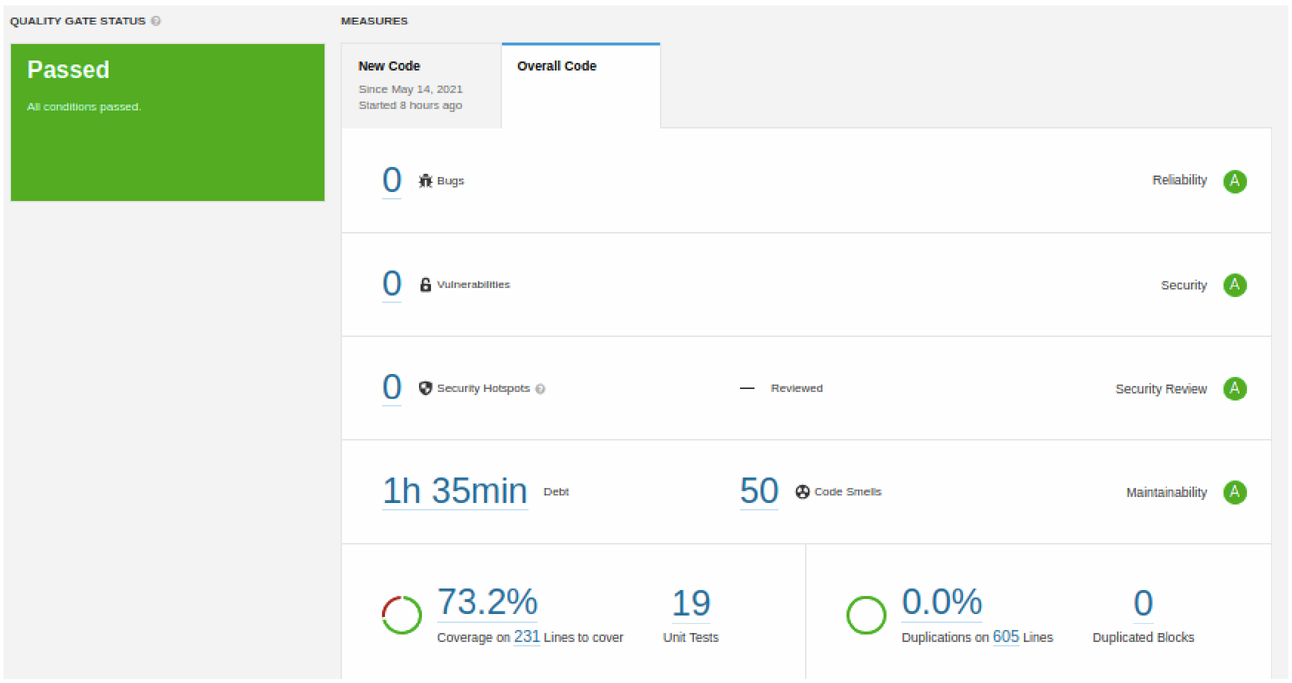
3.3 Functional testing

Functional tests for the web frontend were made by using the Selenium IDE and then exported and into Java code and adapted to match the order they should be run.
These tests run on the following order:

- Initially we see the state of the cache, currently empty, and verify it does return it's state.
- Then we see the index page and make a search for a city, asserting the information received is correct.
- After, we return to the cache and verify that it now has an element and has a failed request.
- Now, we make a new request for the same city, once again asserting the information received is correct.
- And finally, we make a final request for the cache state and verify that it now has a successful request

3.4 Static code analysis

SonarQube was used for static code analysis, as seen in Fig.4 we can see the dashboard of the project. While all there are still a significant of code smells present in the project all of the critical bugs and vulnerabilities were corrected, and most of the code smells seem to be minor things like names of variables and methods. SonarQube detected 27 minor problems and 27 info problems.


QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code — Since May 14, 2021 — Started 8 hours ago

Overall Code

0 Bugs — Reliability A

0 Vulnerabilities — Security A

0 Security Hotspots — — Reviewed — Security Review A

1h 35min Debt — 50 Code Smells — Maintainability A

73.2% Coverage on 231 Lines to cover — 19 Unit Tests

0.0% Duplications on 605 Lines — 0 Duplicated Blocks

Figure 4 – SonarQube Dashboard

While an attempt was made, the goal of reaching 80% of code coverage proved to be very difficult as most of the lines uncovered are either in one the domain classes or are small lines where a log happens, because of that the threshold to pass all the quality gates was changed from 80% of line coverage to 70%.
Between the code smells treated some of the ones I found interesting was the repeated use of the same string literal which is wasteful but easy to miss while working and the need to check if logger has info mode activated before using it as I was assumed that wasn't necessary.

4 References & resources

Project resources

- Video demo:is available in the project repository (https://github.com/pmasilva20/TQS_repository/blob/master/HW/hw1/frontend.mp4)

Reference materials

- Selenium-Jupiter: https://github.com/bonigarcia/selenium-jupiter
- Jackson Json Parser: https://www.baeldung.com/jackson-object-mapper-tutorial
- Logging: https://www.baeldung.com/spring-boot-logging
- Thymeleaf: https://www.baeldung.com/thymeleaf-in-spring-mvc
- OpenWeatherAPI: https://openweathermap.org/api/air-pollution