deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Mariana Ladeiro [92964], v2021-05-13

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 main goal of the project was to develop a multi-layer web application in Spring Boot that would show the Air Quality of a certain place, with different types of automated tests. The assignment should meet some objectives, including a web-page, in order to interact with the user, an external API to get the data to show, as well as a REST API to be used by external users and also some type of cache system that reports the use of the API.

The web app developed, AirQuality, displays the air quality, including it's AQI level, CO and NO2 levels, and much more, of a certain city that the user searches for. Additionally, it shows the cache details (requests, hits, misses and cities searched). The same data displayed in the web pages is also provided by the REST API, to the use of developers, in JSON format.

1.2 Current limitations

The main limitations of the work developed fall on the fact that the API chosen, *weatherbit.io*, doesn't have information regarding air quality on a certain day or forecast data, meaning the application only displays current air quality information. Another known limitation is, if for some reason the API stops working or it's not available anymore the application will not display any type of information on air quality, since it only fetches from that one API. The last limitation I would like to mention is the fact that sometimes the API may return a different city that we expect, for example searching for London will not give us London, England, but a city in the US. This would be easy to solve though, asking the user not only the city but also the country. These are all things that I would like to implement later on.

2 Product specification

2.1 Functional scope and supported interactions

AirQuality offers the display of air quality information of a chosen city as well as updated cache statistics.

Regarding the web application, when accessing the main page, the user types a city in the search box and clicks "Search" and is shown all the information regarding the current air quality, from CO to PM10, to the AQI - air quality index - level, as well as a message informing the user the meaning of the level of the AQI. If the city doesn't exist or is not available in the API a message will be shown. The user also has access to the cache, which shows the number of requests made to the cache, the hits (in case a city searched is already in cache) and misses (in case a city searched is not in cache). If the user ends up in a page not recognized, the application displays a message allowing the user to return to the Home Page.

Moving now to the REST API, that developers or other types of users can use, it also provides information about the information mentioned above, only it returns that information in JSON format.

More information regarding both the web application and the API are discussed further in the report.

2.2 System architecture

I followed SpringBoot conventions and structured the code into packages, according to their functionalities. The following diagram shows an overall idea of how components connect to one another.

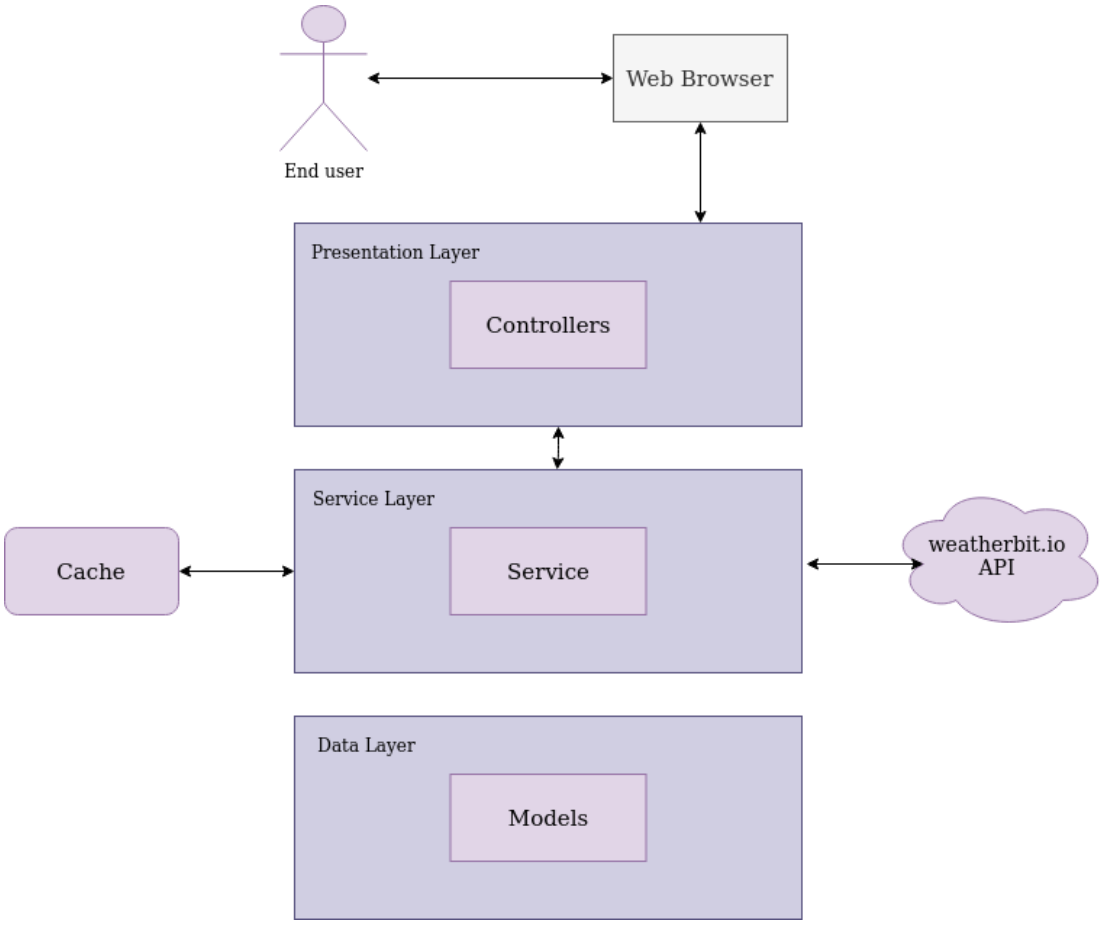


Figure 1 - Overall Architecture

I tried to make it as simple as possible while also meeting the project objectives.

The end user uses the webpage in order to get access to the application in which the presentation layer returns the frontend to the user. To develop the frontend that consists of three pages, I used Thymleaf. This layer contains the Controller which assigns a view to each possible URL, as well as a REST Controller that handles the API Json results. The business/service layer performs the intermediate between the frontend and where the data is being fetched from, the Service class. The Service has the responsibility of checking if a city is in the cache and getting the requested information, through the user's input, from the weatherbit.io API or just getting the cache saved information. In case a city is in the cache, meaning it was searched before and it has not expired, then the Service will get the information previously saved when that city was first searched and display it to the user. On the other hand, in case a city is not in the cache, hasn't been searched before or it has expired, then the Service will make a call to the external API and construct the CityData model, from the Data Layer, returning the fetched information.

Going into a bit of detail regarding the Cache, I developed a pretty simple not persistent internally in-memory cache, meaning if the application is stopped it's data is not saved. I defined the time to live policy as one minute (so the tests could be faster). I should also mention that the request field is updated every single time the cache is called, and this call doesn't have to come from a city searched, simply reloading the cache webpage will increase it's requests. To make sure that when removing a city from the citiesSearched list, no exception was thrown when trying to access the rest api (api/cache), I used an iterator.

```
for (Iterator<String> city = cacheMap.keySet().iterator(); city.hasNext();) {
    String s = city.next();
    if (!isInCache(s)){
        city.remove();
    }
}
```

Figure 2 - Iterator

There is also an ErrorController that returns the error html page in case a page doesn't exist for the web application. This was done to have a more user-friendly interface and also prevent whitelabels.

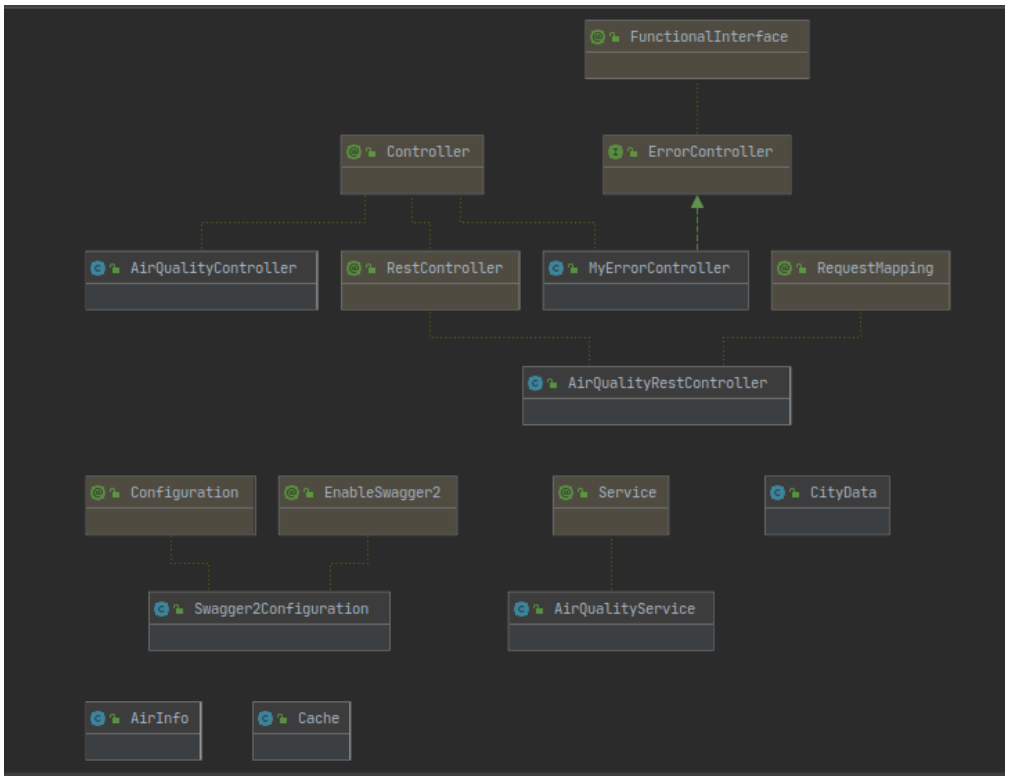


Figure 3 - IntelliJ generated Diagram

2.3 API for developers

Developers can obtain from the AirQuality api the following information:

- Air Quality Details of a City: AQI, CO, O3, SO2, NO2, PM10 and PM25
- Cache Usage Statistics: hits (city searched is in cache), misses (city searched in not in cache), cities searched and requests (everytime a request is made to the cache)

Note: in the event of an unknown city the air quality details fields return as null.

To access these endpoints, the developers only need to specify in the URL the data they want to access. The following swagger documentation shows more in depth the API details:

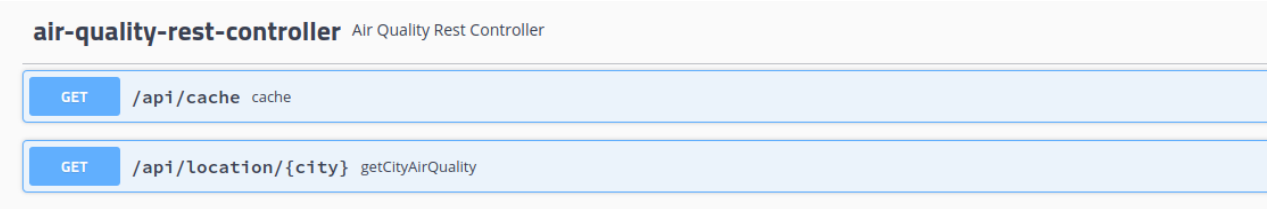


Figure 4 - Swagger API

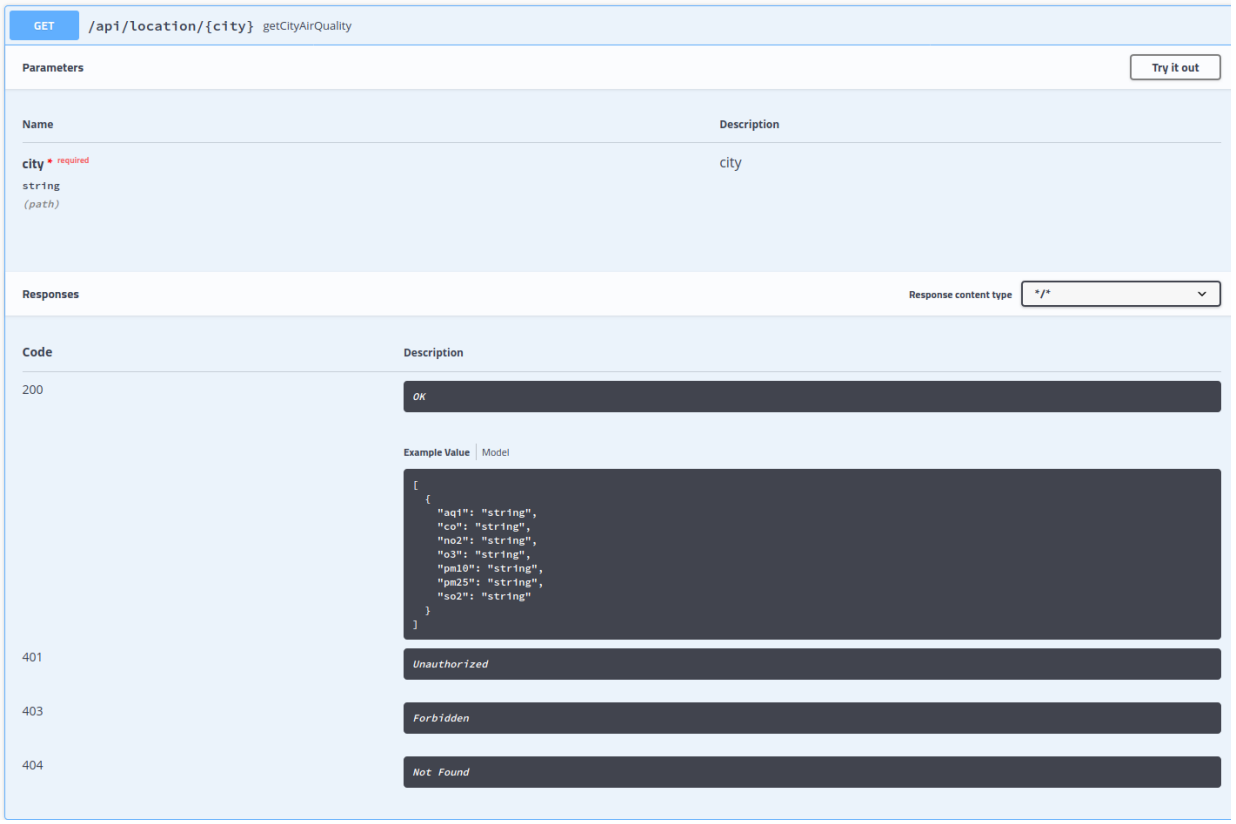


Figure 5 - Swagger API city information

3 Quality assurance

3.1 Overall strategy for testing

The overall test development strategy was based on starting to develop some code and adding tests according to the code snippets created. This helped me make sure my application was returning the information needed and the right one, as well as having a more tactical approach since bugs were discovered early on. I also took a TDD approach in other classes, so I wasn't writing tests with the already expected results in mind and not testing every possible scenario. This approach was very useful especially when using SonarQube that helps us see which code is not being covered yet.

The technologies used were based on the guidelines of the project as well as what was taught during the semester: JUnit, SpringBoot MockMvc, Mockito, Selenium and Cucumber, SonarQube and JaCoCo.

3.2 Unit and integration testing

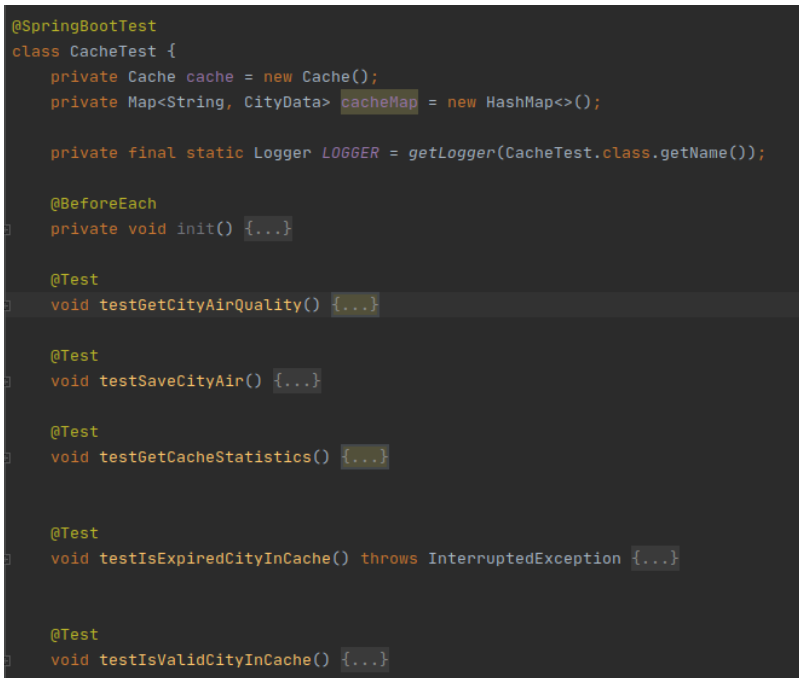
For unit and integration testing I used JUnit, SpringBoot MockMvc and Mockito. I tested functionalities implemented in the Cache, Controllers and Service. I also developed some tests for the models to be picked up by SonarQube.

Let's begin with the models. For both of them I tested the setters and getters to make sure everything was returning the wanted values and also to increase the coverage in SonarQube. These tests are not really necessary, since getters and setters are usually not a problem but I still decided to develop them.

```
class AirInfoTest {

    @Test
    void testSettersAndGetters() {
        AirInfo data = new AirInfo( pm10: "1",  pm25: "2",  aqi: "54",  co: "1",  o3: "5",  so2: "6.6",  no2: "0");
        data.setAqi("55");
        data.setPm10("2");
        data.setPm25("1");
        data.setCo("5");
        data.setSo2("5.8");
        data.setNo2("2.3");

        assertThat(data.getAqi()).isEqualTo("55");
        assertThat(data.getPm10()).isEqualTo("2");
        assertThat(data.getPm25()).isEqualTo("1");
        assertThat(data.getCo()).isEqualTo("5");
        assertThat(data.getSo2()).isEqualTo("5.8");
        assertThat(data.getNo2()).isEqualTo("2.3");

    }
}
```

Figure 6 - AirInfoTest

Moving on to the Cache tests, I developed some methods in order to test the Cache behavior: get the searched city air quality information; save the city information in the cache; get the cache statistics; test if an expired city (after 1 minute for the scope of the project) is still in cache, which should return false; and lastly, if a valid city (meaning is not expired yet) is in cache, which should return true.



```
@SpringBootTest
class CacheTest {
    private Cache cache = new Cache();
    private Map<String, CityData> cacheMap = new HashMap<>();

    private final static Logger LOGGER = getLogger(CacheTest.class.getName());

    @BeforeEach
    private void init() {...}

    @Test
    void testGetCityAirQuality() {...}

    @Test
    void testSaveCityAir() {...}

    @Test
    void testGetCacheStatistics() {...}


    @Test
    void testIsExpiredCityInCache() throws InterruptedException {...}


    @Test
    void testIsValidCityInCache() {...}
```

Figure 7 - CacheTest

For the testIsExpiredCityInCache unit test I had to use a method that would make the test wait 60 seconds before checking if the city was valid. I first used Thread.sleep but soon realized that this was a major code smell (by analysing it in SonarQube) so I searched for other options. The first that I found was Awaitability, but this doesn't allow you to just wait, without passing some type of until() method. So, I ended up using TimeUnit.*SECONDS*.sleep which ended up getting the job done.

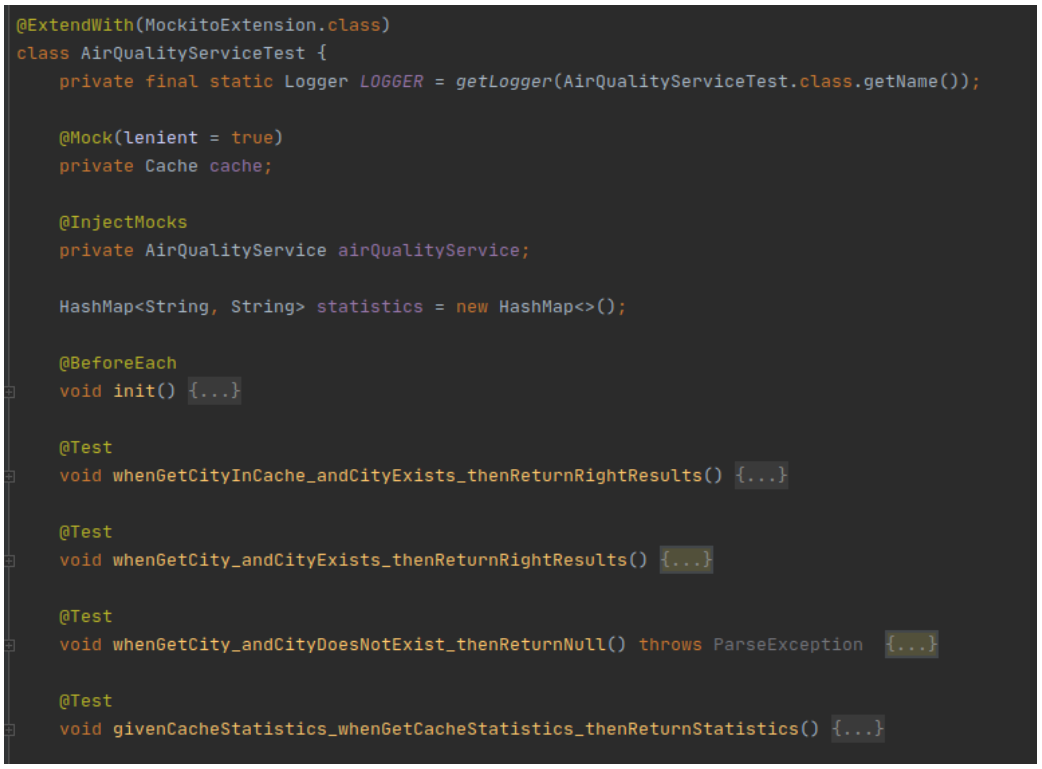
For the rest controller tests, I ran tests both on the server and on the client sides. Starting with the RestController I used SpringBoot WebMvc. For this I mocked the AirQualityService. The following tests were created: when getting a certain city air quality, the correct information should be returned in JSON format; when getting a city that is not recognized in the API, the returned JSON should be null in all of the parameters; when the cache statistics are called the correct statistics should be returned in JSON format; and finally, I also tested in case no cities were in cache if the array remained null.

```
@WebMvcTest(AirQualityRestController.class)
class AirQualityRestControllerTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private AirQualityService airQualityService;

    @Test
    void whenGetCityAirQuality_thenReturnCityAirQualityInfo() throws Exception{...}

    @Test
    void whenGetInvalidCityAirQuality_thenReturnNull() throws Exception {...}

    @Test
    void whenGetCacheStatistics_thenReturnStatisticsInJson() throws Exception {...}

    @Test
    void givenNoCitiesInCache_whenGetCacheStatistics_thenReturnStatisticsInJson() throws Exception {...}
```

Figure 8 - AirQualityRestControllerTest

I also ended up testing the other 2 controller classes (AirQualityController and ErrorController) to test and complement the functional testing with Selenium. For this I used SpringBoots @LocalServerPort annotation and tested if the right main pages were being displayed.

To conclude this section, the service tests were done with Mockito to mock the Cache behaviour. The situations covered are: when a city is searched and it exists the right air quality information should be returned; when a city is searched and it is not recognized/ doesn't exist in the API then results should come as null; when a city that is already in cache is searched the right information should be returned; and given the cache statistics when the statistics are wanted the right information is returned.



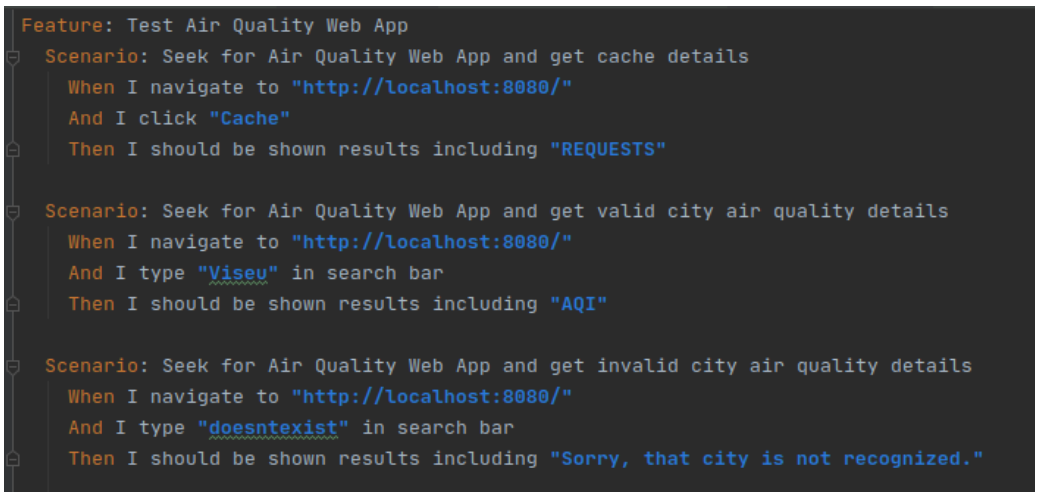
```java
@ExtendWith(MockitoExtension.class)
class AirQualityServiceTest {
    private final static Logger LOGGER = getLogger(AirQualityServiceTest.class.getName());

    @Mock(lenient = true)
    private Cache cache;

    @InjectMocks
    private AirQualityService airQualityService;

    HashMap<String, String> statistics = new HashMap<>();

    @BeforeEach
    void init() {...}

    @Test
    void whenGetCityInCache_andCityExists_thenReturnRightResults() {...}

    @Test
    void whenGetCity_andCityExists_thenReturnRightResults() {...}

    @Test
    void whenGetCity_andCityDoesNotExist_thenReturnNull() throws ParseException  {...}

    @Test
    void givenCacheStatistics_whenGetCacheStatistics_thenReturnStatistics() {...}
```

Figure 9 - AirQualityServiceTest

3.3 Functional testing

For the functional testing I used Selenium with a combination of Cucumber to make sure everything related to the web application interface returned the right information. For this, I created four scenarios in Cucumber: show the cache details in which the page displays a table with the contents of the cache; show the searched city air quality in which the page displays the data and a quick information about the AQI level returned; show a message to the user when a city searched is not recognized by the API and lastly show the error page when a invalid URL is triggered. In order for these tests to pass, I created four methods: when you navigate to a webpage, when something is clicked on, results that should be shown and also when you type in the search bar. An important side note is that in order for these tests to pass, the application needs to be running.

The following shows an example of scenarios as well as the Selenium tests implemented.



```
Feature: Test Air Quality Web App
  Scenario: Seek for Air Quality Web App and get cache details
    When I navigate to "http://localhost:8080/"
    And I click "Cache"
    Then I should be shown results including "REQUESTS"

  Scenario: Seek for Air Quality Web App and get valid city air quality details
    When I navigate to "http://localhost:8080/"
    And I type "Viseu" in search bar
    Then I should be shown results including "AQI"

  Scenario: Seek for Air Quality Web App and get invalid city air quality details
    When I navigate to "http://localhost:8080/"
    And I type "doesntexist" in search bar
    Then I should be shown results including "Sorry, that city is not recognized."
```

Figure 10 - Cucumber Scenarios

```
public class AirQualityWebTest {
    private WebDriver webDriver;

    @When("I navigate to {string}")
    public void iNavigateTo(String url) {...}

    @And("I click {string}")
    public void iClickCache(String cache) {...}

    @Then("I should be shown results including {string}")
    public void iShouldBeShownResultsIncluding(String result) {...}

    @And("I type {string} in search bar")
    public void iType(String city) {...}
```

Figure 11 - AirQualityWebTest

3.4 Static code analysis

SonarQube was used to run the static code analysis and it is definitely one of the most important tools to integrate when developing code. I used SonarQube early on the development of my project which really helped me to solve problems and code smells early on, and consequently decrease the debt, along the way. The constant analysis made me realize that code quality control is an inseparable part of software development.

The following image shows the last analysis made with SonarQube. As seen, the code passes all the conditions that are defined in the quality gate, by default. I did some tests to setters and getters in the model classes to make the coverage go up.

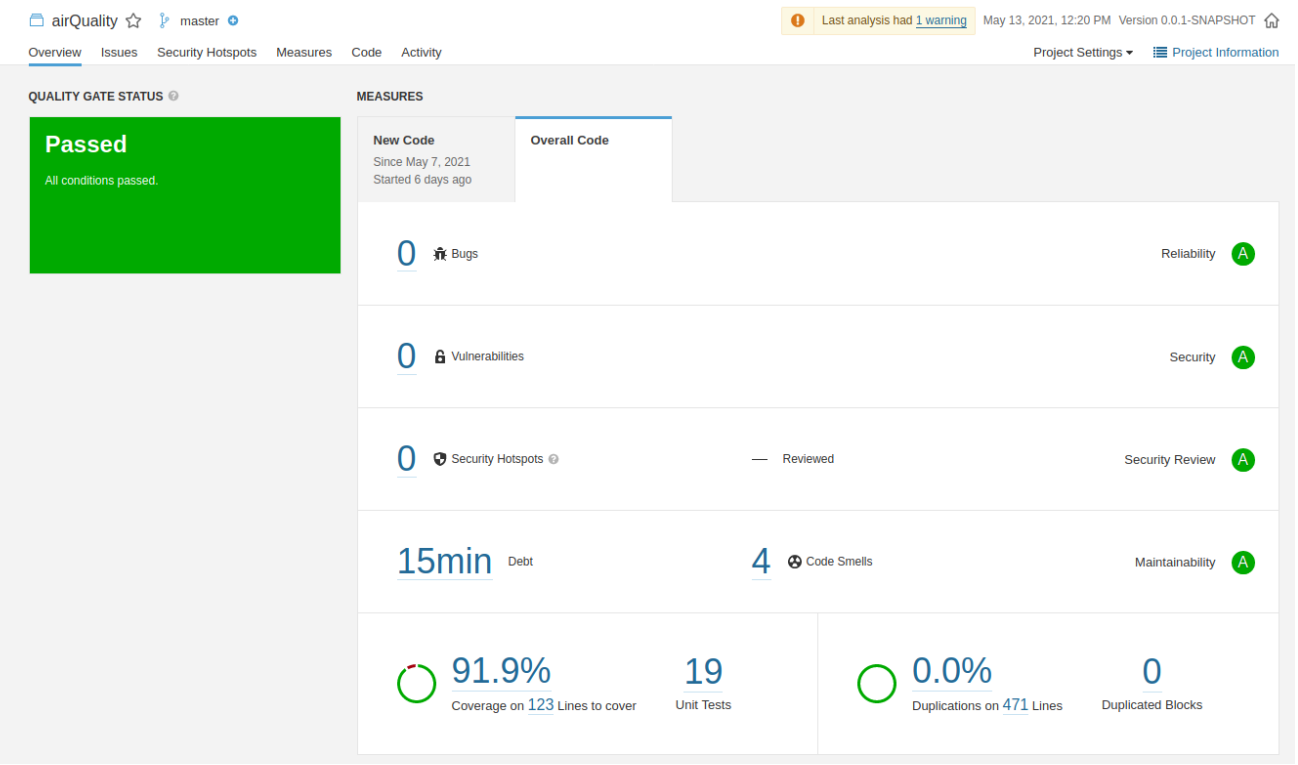


Figure 12 - SonarQube analysis

As far as code smells go, the 4 remaining consist of minor, info or blocker that I didn't give much of my attention to. For example, 2 of those 4 code smells are blockers in which "Add at least one assertion to this test case" was the code smell. I did remove all the code smells that were major or critical making the debt go to only 15 minutes. There were a few minor ones that I had no idea made a code smell. An example is the following in which according to SonarQube you should use the available methods that most fit your needs.

Figure 13 - Code Smell Example

I also tried integrating SonarCloud in my git repository but I had some trouble with dependencies and could not solve it.

4 References & resources

Project resources

- Video demo: https://github.com/marianabladeiro/TQS/tree/main/HW1_92964

Reference materials

API used: Weatherbit.io air quality current
Available at: https://www.weatherbit.io/api/airquality-current

Incorporate Swagger: Swagger 2 documentation for spring rest api at Baeldung
Available at: https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api

Selenium with Cucumber tests: Selenium with Cucumber (BDD Framework) at Guru99
Available at: https://www.guru99.com/using-cucumber-selenium.html

Unit testing: Unit Testing with JUnit 5 at Vogella
Available at: https://www.vogella.com/tutorials/JUnit/article.html

Materials on Elearning provided by professor
Available at: https://elearning.ua.pt/