HW1: Mid-term assignment report

Marcel de Araújo Santos Souza [101043], v2023-04-02

1 Introduction 1
1.1 Overview of the work 1
1.2 Current limitations 1

2 Product specification 2
2.1 Functional scope and supported interactions 2
2.2 System architecture 2
2.3 API for developers 3

3 Quality assurance 3
3.1 Overall strategy for testing 3
3.2 Unit and integration testing 3
3.3 Functional testing 6
3.4 Code quality analysis 7
3.5 Continuous integration pipeline [optional] 8

4 References & resources 8

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 app that I propose in this Homework is called AirQuality (not very creative). The purpose of the project is to present air quality data for different cities around the world. It is also possible to ask for air quality predictions for the next 5 days, thanks to the use of 2 different APIs that ensure this function.

1.2 Current limitations

- The API just returns 1 city, so, the name needs to be exactly the name of the city. If the user wants information from a city that coexists in 2 countries, needs to write the country code also. It would be better to implement a list for all the cities that are related to the text that the user sent to the API.
- The API is too slow for some cities. Of course, this is a problem with the third-part API provider, not with the implementation. If the user asks for Miami data is very fast, although if it asks about Aveiro or Braga, delays a lot sometimes.

2 Product specification

2.1 Functional scope and supported interactions

The actors who will use the application are individuals or organizations interested in obtaining data about air quality. They could be researchers, environmentalists, policy makers, or even regular citizens who are concerned about the air they breathe.

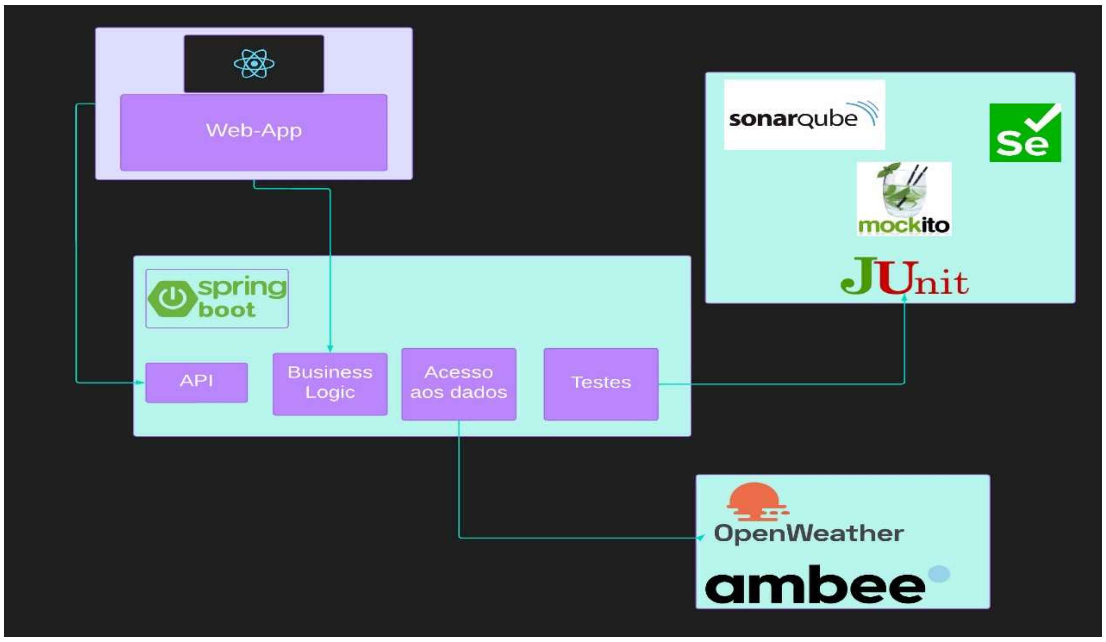
The main usage scenario involves using the API to retrieve air quality data for a specific location. Users will provide the API with the location they are interested in and the time range they want the data for (today and the next 5 days). The API will then use its data sources to retrieve and aggregate the relevant air quality data, and return it to the user in a structured format.

For example, a researcher studying the impact of air pollution on health might use the API to obtain air quality data for a particular city over the next 5 days. The researcher could then use this data to analyze trends and correlations with health outcomes. Similarly, a concerned citizen might use the API to check the air quality in their neighborhood and take appropriate precautions if necessary.

2.2 System architecture

To perform this project, I used many different technologies like:

- Spring Boot: The API itself was created and performed in Java Spring Boot;
- ReactJS: The front-end part was developed using ReactJS and Axios(requests);
- Junit and Mockito: For testing;
- Selenium Web driver: For front-end testing and all-together tests;
- OpenWeatherApp and AmbeeData: Third-Part APIs
- GitHub Actions: For CI.
- SonarQube: For static code analysis;



2.3 API for developers

The API uses two external APIs, the Ambee Air Quality API and the OpenWeatherMap Air Pollution API, to fetch the data.

The API has 3 endpoints:

/airquality/{city} - Retrieves the latest air quality data for a given city using the Ambee Air Quality API. If the city is not found, it returns a 404 Not Found status code.

/airquality/predictions/{city} - Retrieves air quality predictions for the next 5 days for a given city using the OpenWeatherMap Air Pollution API.

/airquality/stats – General statistics about the usage of API (API calls) and the cache (cache hits and misses)

The API also has a reset endpoint to reset all the counters and the cache itself.

3 Quality assurance

3.1 Overall strategy for testing

In general, the strategy was TDD. Normally, when I decided to create some functionality, the test is created first. This was very important in the case of my API, because as I am recreating a JSON from 2 different APIs in Java, it was very important to use Mock sometimes.

3.2 Unit and integration testing

I test the API's functionalities such as the cache and the statistics. It also includes an integration test to verify the API's functionality for the endpoint /airquality/stats.

The test class includes the following methods:

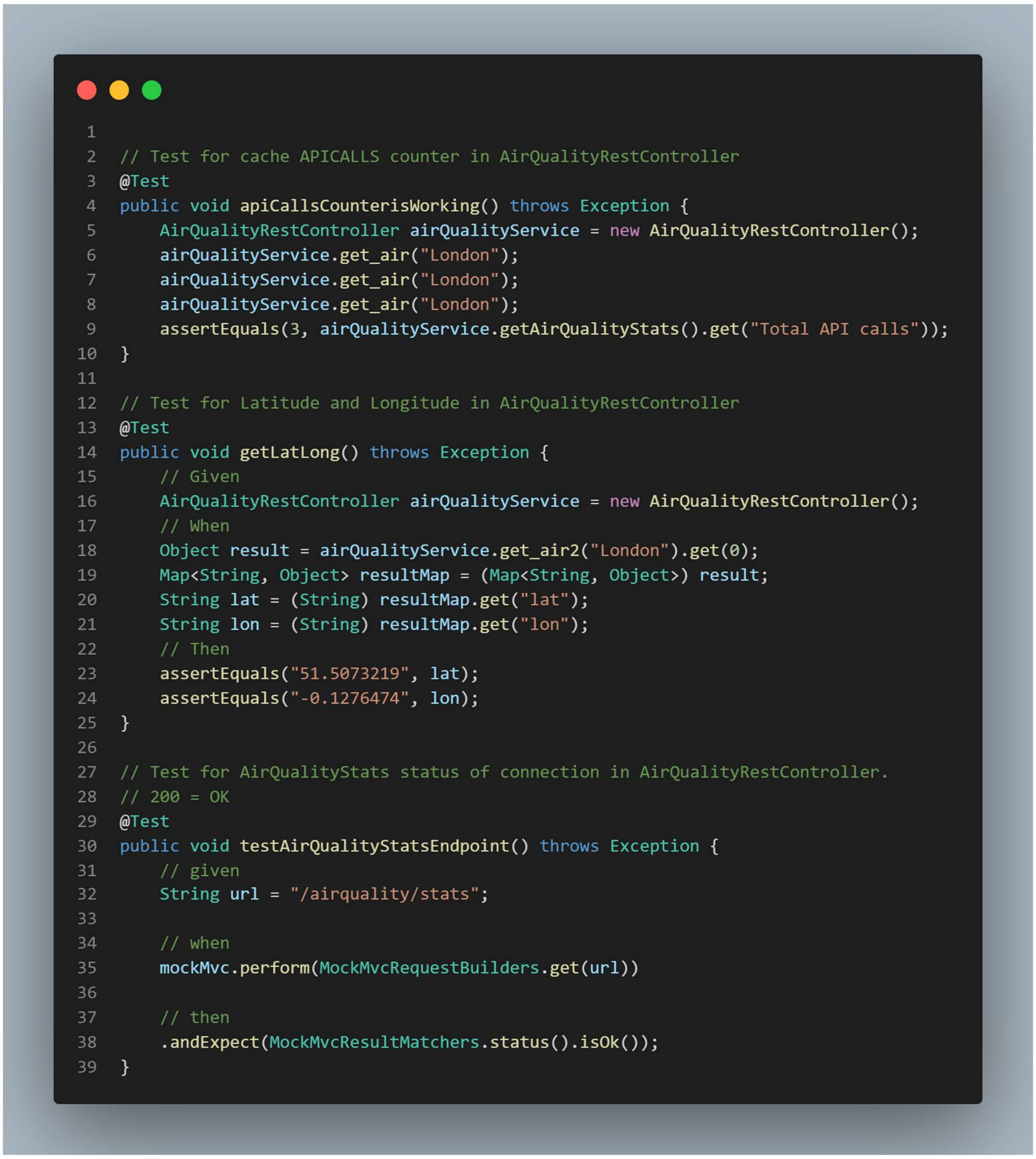
- contextLoads(): an empty test method that checks if the Spring Boot application context is properly loaded.
- apiCallsCounterisWorking(): a test method that verifies if the cache's API calls counter is working by calling the /airquality endpoint three times and checking if the counter increments by three.
- getLatLong(): a test method that verifies if the /airquality endpoint returns the correct latitude and longitude for a given location.
- testAirQualityStatsEndpoint(): an integration test that checks if the /airquality/stats endpoint returns a status of 200 (OK).
- testCleanCache(): a test method that checks if the cache is empty after calling the /airquality/clean endpoint.

- cacheCalled(): a test method that checks if the cache is working by calling the /airquality endpoint three times and verifying if the cache hits, cache misses, and total API calls counters are correct.
- testGetAirQualityStats(): an integration test that verifies if the /airquality/stats endpoint returns a JSON object containing numbers for each of the three keys: Cache hits, Cache misses, and Total API calls.

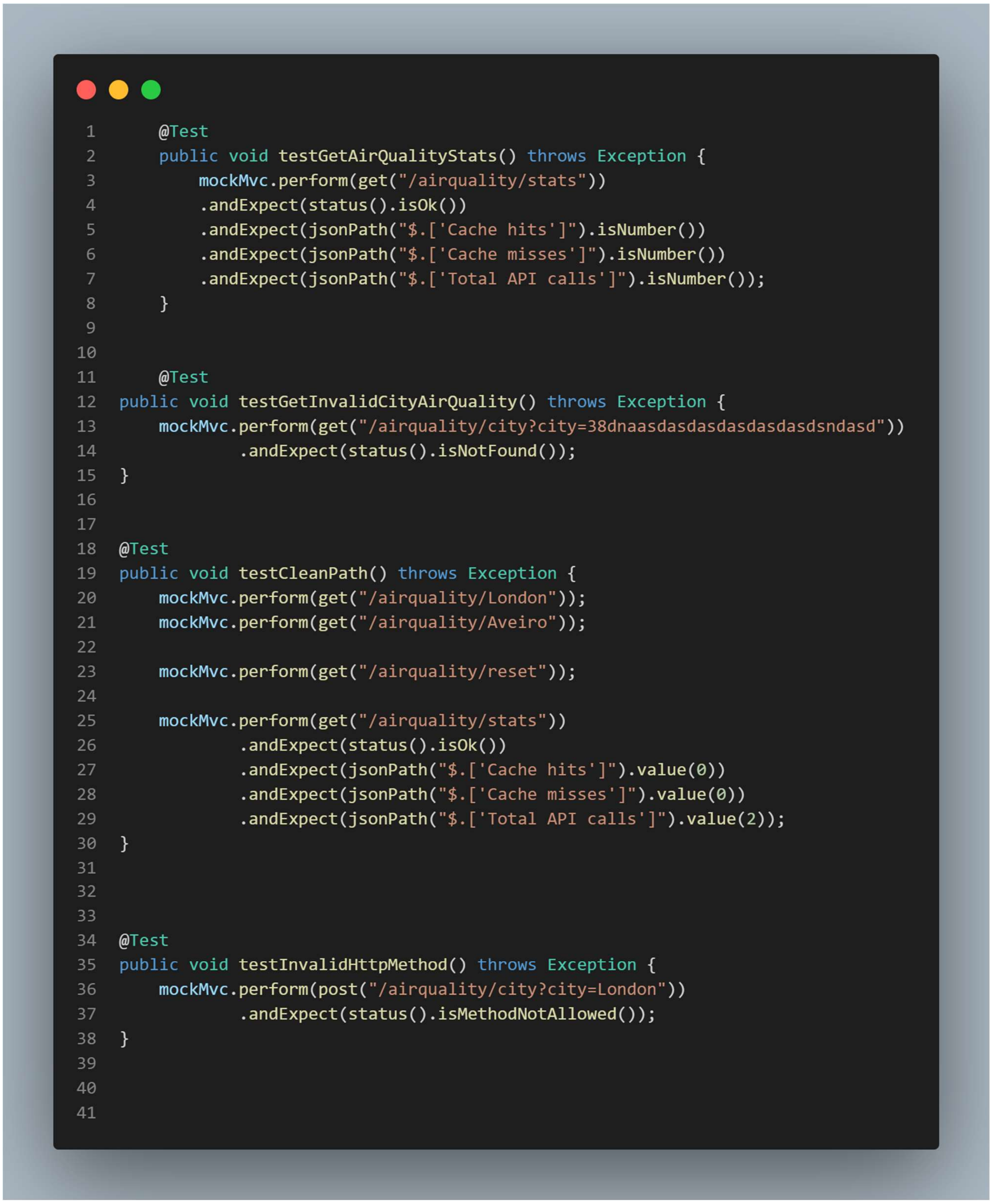
There is a commented out test method named cacheClearedAfter15Minutes() that would have tested if the cache is cleared after 15 minutes, but it is not currently being used.

The test class imports the following packages:

- org.junit.jupiter.api.Assertions for assertions in the test methods.
- org.springframework.test.web.servlet.MockMvc for creating mock HTTP requests to test controllers.
- org.springframework.test.web.servlet.result.MockMvcResultMatchers for matching expected results in mock HTTP requests.
- org.springframework.test.web.servlet.request.MockMvcRequestBuilders for building mock HTTP requests.
- org.springframework.test.context.web.WebAppConfiguration for configuring a web application context for testing.
- org.mockito.Mockito for mocking objects for testing.
- org.springframework.beans.factory.annotation.Autowired for dependency injection of objects.
- org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc for configuring a mock MVC object for testing.
- org.springframework.boot.test.context.SpringBootTest for creating a test Spring Boot application context.
- java.util.Map for working with map objects in the test methods.



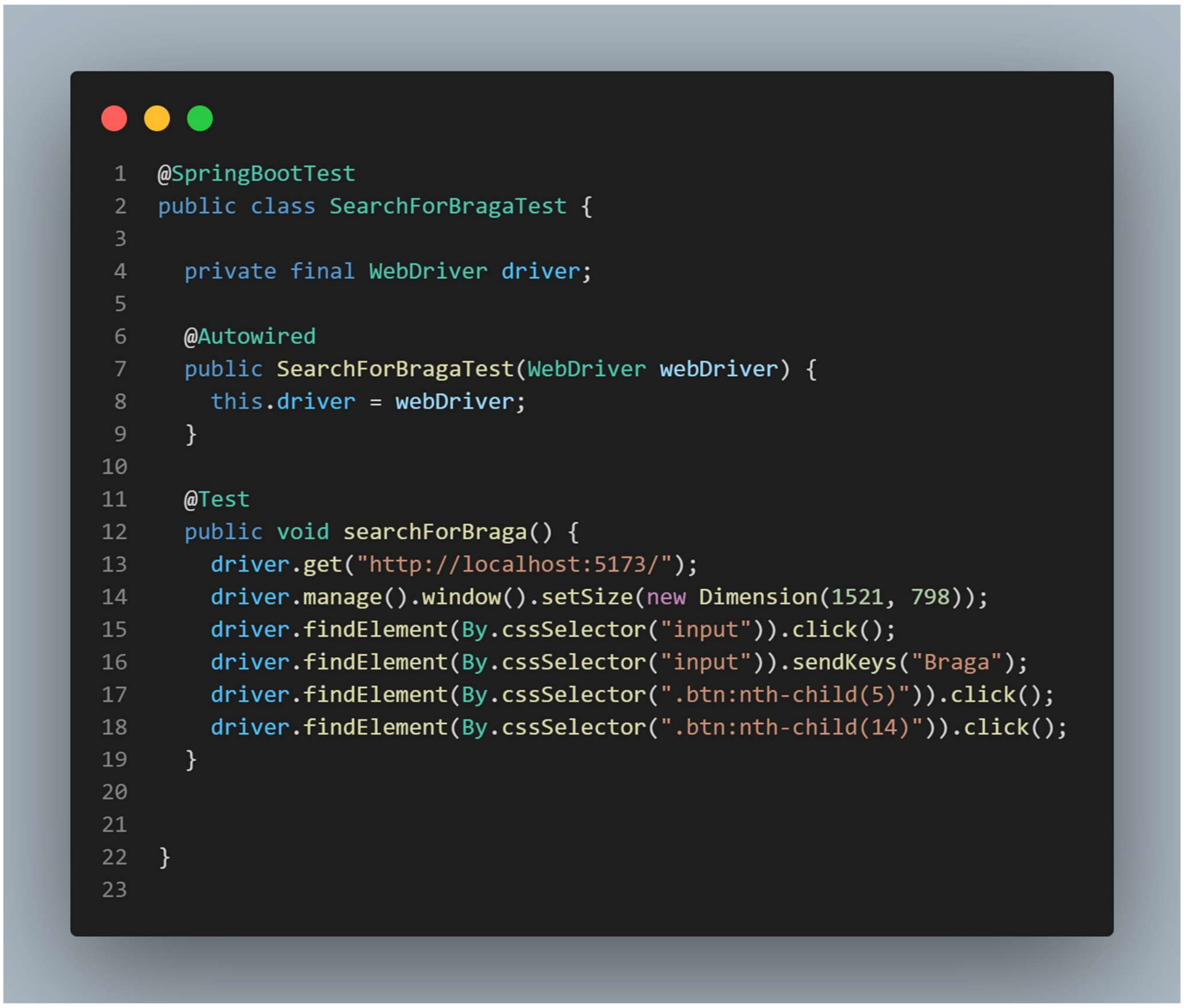
```java

// Test for cache APICALLS counter in AirQualityRestController
@Test
public void apiCallsCounterisWorking() throws Exception {
    AirQualityRestController airQualityService = new AirQualityRestController();
    airQualityService.get_air("London");
    airQualityService.get_air("London");
    airQualityService.get_air("London");
    assertEquals(3, airQualityService.getAirQualityStats().get("Total API calls"));
}

// Test for Latitude and Longitude in AirQualityRestController
@Test
public void getLatLong() throws Exception {
    // Given
    AirQualityRestController airQualityService = new AirQualityRestController();
    // When
    Object result = airQualityService.get_air2("London").get(0);
    Map<String, Object> resultMap = (Map<String, Object>) result;
    String lat = (String) resultMap.get("lat");
    String lon = (String) resultMap.get("lon");
    // Then
    assertEquals("51.5073219", lat);
    assertEquals("-0.1276474", lon);
}

// Test for AirQualityStats status of connection in AirQualityRestController.
// 200 = OK
@Test
public void testAirQualityStatsEndpoint() throws Exception {
    // given
    String url = "/airquality/stats";

    // when
    mockMvc.perform(MockMvcRequestBuilders.get(url))

    // then
    .andExpect(MockMvcResultMatchers.status().isOk());
}
```



```java
    @Test
    public void testGetAirQualityStats() throws Exception {
        mockMvc.perform(get("/airquality/stats"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.['Cache hits']").isNumber())
        .andExpect(jsonPath("$.['Cache misses']").isNumber())
        .andExpect(jsonPath("$.['Total API calls']").isNumber());
    }


    @Test
public void testGetInvalidCityAirQuality() throws Exception {
    mockMvc.perform(get("/airquality/city?city=38dnaasdasdasdasdasdasdsndasd"))
            .andExpect(status().isNotFound());
}


@Test
public void testCleanPath() throws Exception {
    mockMvc.perform(get("/airquality/London"));
    mockMvc.perform(get("/airquality/Aveiro"));

    mockMvc.perform(get("/airquality/reset"));

    mockMvc.perform(get("/airquality/stats"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.['Cache hits']").value(0))
            .andExpect(jsonPath("$.['Cache misses']").value(0))
            .andExpect(jsonPath("$.['Total API calls']").value(2));
}



@Test
public void testInvalidHttpMethod() throws Exception {
    mockMvc.perform(post("/airquality/city?city=London"))
            .andExpect(status().isMethodNotAllowed());
}
```

3.3 Functional testing

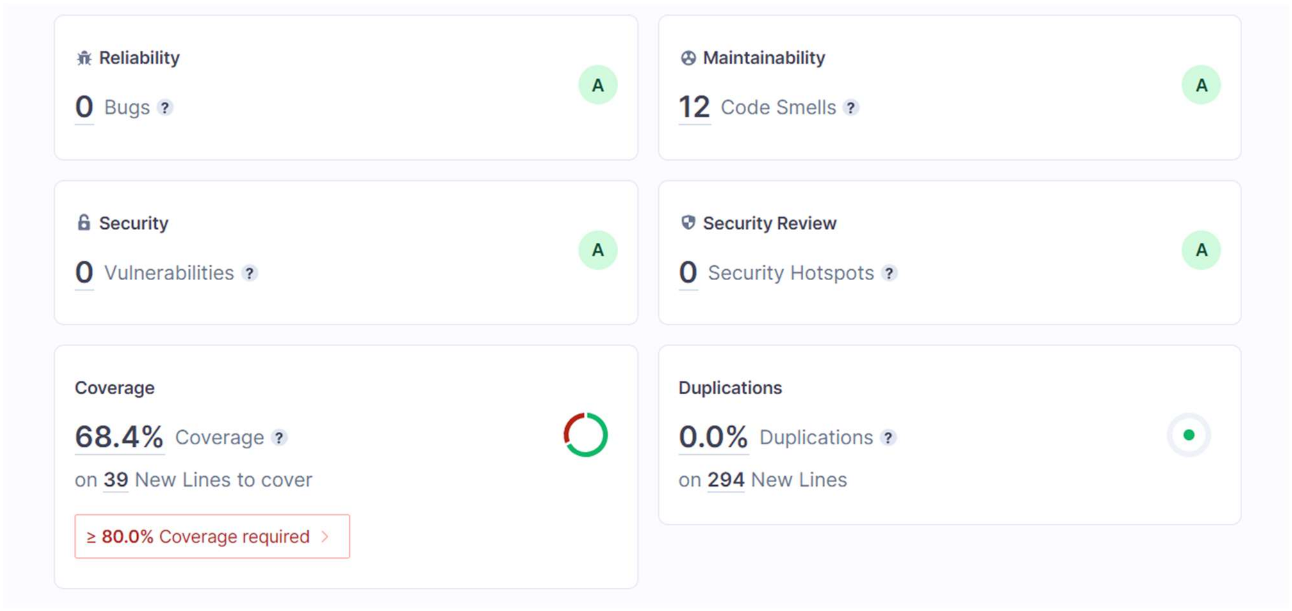
- The only possible: Search for a city and see the results. As this is a big project, the front-end is very simple, so, using the WebDriver for ChromeOS I could run the test of searching for Braga in my app. Above is the code:



```java
@SpringBootTest
public class SearchForBragaTest {

  private final WebDriver driver;

  @Autowired
  public SearchForBragaTest(WebDriver webDriver) {
    this.driver = webDriver;
  }

  @Test
  public void searchForBraga() {
    driver.get("http://localhost:5173/");
    driver.manage().window().setSize(new Dimension(1521, 798));
    driver.findElement(By.cssSelector("input")).click();
    driver.findElement(By.cssSelector("input")).sendKeys("Braga");
    driver.findElement(By.cssSelector(".btn:nth-child(5)")).click();
    driver.findElement(By.cssSelector(".btn:nth-child(14)")).click();
  }


}

```

3.4 Code quality analysis

Using the SonarQube I have created some reports together with Jacoco for Covering Reports. Without any doubts, the most interesting thing that I did learn is the important of these automatic tools for grading a code based on its “code smells” and the quantity of lines that are being tested. Initially, my project used to have 26 code smells, but after reading the recommendations from the Sonar Qube, it decreased a lot. For example, the code smell of “public” methods in Junit Test cases. By the SonarQube documentation, it is better to use the “protected” case.


Reliability
0 Bugs
A

Maintainability
12 Code Smells
A

Security
0 Vulnerabilities
A

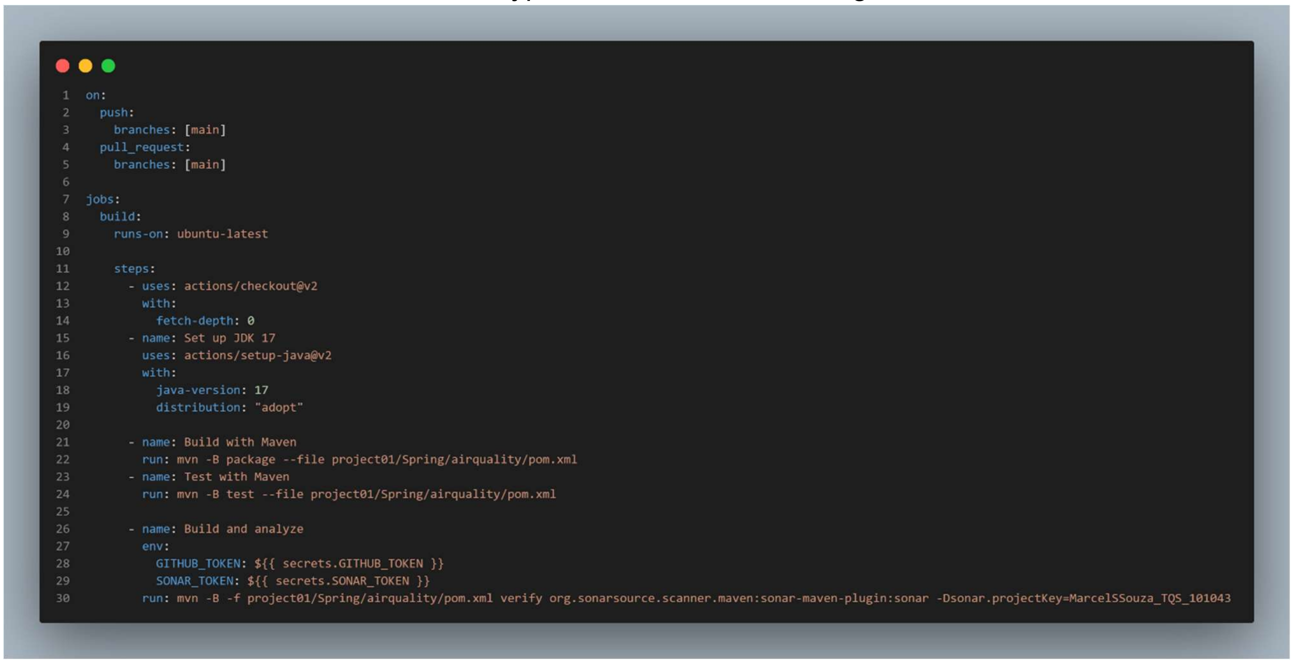
Security Review
0 Security Hotspots
A

Coverage
68.4% Coverage
on 39 New Lines to cover
≥ 80.0% Coverage required

Duplications
0.0% Duplications
on 294 New Lines

3.5 Continuous integration pipeline

To implement the CI pipeline, I decided to use the Github Actions option. It was very easy to set up and to implement. Some tests yet do not work properly (like the webdriver test), so I commit the code commented in these type of tests. Above is the .github/workflow/main file:


```yaml
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Set up JDK 17
        uses: actions/setup-java@v2
        with:
          java-version: 17
          distribution: "adopt"

      - name: Build with Maven
        run: mvn -B package --file project01/Spring/airquality/pom.xml
      - name: Test with Maven
        run: mvn -B test --file project01/Spring/airquality/pom.xml

      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B -f project01/Spring/airquality/pom.xml verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=MarcelSSouza_TQS_101043
```

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/MarcelSSouza/TQS_101043
Video demo	https://youtu.be/oj-lWg0-MrU
QA dashboard (online)	https://sonarcloud.io/project/overview?id=MarcelSSouza_TQS_101043

CI/CD pipeline	Actions Github
Deployment ready to use	-

Reference materials

- https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven
- https://openweathermap.org/api/air-pollution#current
- https://www.getambee.com/
- Youtube Tutorials in General
- https://chromedriver.chromium.org/