HW1: Mid-term assignment report

Pedro Dinis Bastos Tavares [93103], v2021-05-14

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

2 Product specification **1**
2.1 Functional scope and supported interactions 1
2.2 System architecture 2
2.3 API for developers 2

3 Quality assurance **2**
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

4 References & resources **3**

1 Introduction

1.1 Overview of the work

With this report I aim to explain the developed project for the TQS midterm individual project. To do so, I will go over the project features and how I went by the quality assurance process, by saying what types of tools were used, how they were used and what they were used for.

1.2 Current limitations

Even though it is possible with the chosen API and even though I started developing a pollution forecast for the next day, due to a lack of time I was not able to cover that feature. The cache is not fully implemented because, only through reading the SpringBoot logs can you see if it checked the cache or not, another issue with the cache is that it only covers one of the two services.

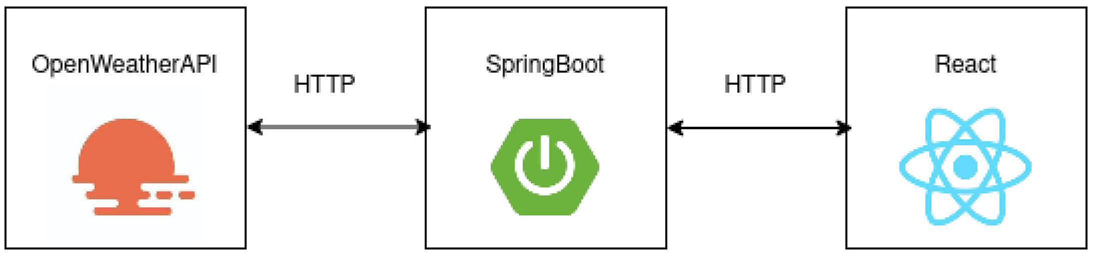
2 Product specification

2.1 Functional scope and supported interactions

The application can be used to search for a location, either by city or city and country code (Portugal => PT). With this search we get a page that displays to the user a set of particles present in the atmosphere and the air quality index from 1 to 5, one being Good and 5 being Very Poor, all this in the place under evaluation by the request. The data, even though not perceptible to the frontend user, if the location has already been searched recently, will return data from cache and print out the amount of hits if hit and misses if missed.

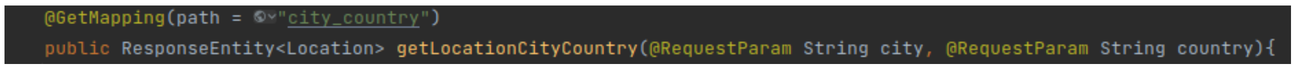
2.2 System architecture

When it comes to the architecture there were used three main tools, react, SpringBoot and the OpenWeather API. React was used to do the frontend side of things, more specifically the user interface and the requests, made through a service also in react, to the business logic section of the project developed in SpringBoot. This business logic section, as mentioned, was developed by using SpringBoot and in collaboration with java. The business logic section was the bridge from the APIs used and the frontend in React. Here, frontend requests are received and processed to be able to call the API or cache and it also processes responses from the external APIs from OpenWeather. The APIs used were a Geocoding api, to convert cities and country names to coordinates and also the Air Pollution API that converted the coordinates from the Geocoding API to actual information on the air pollution, both of these from OpenWeatherMap API service.



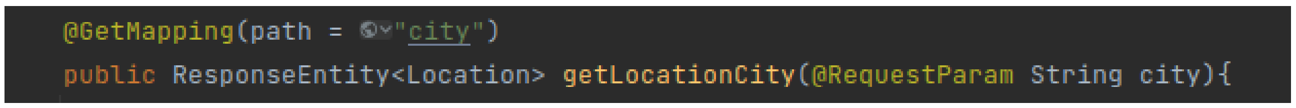
2.3 API for developers

By using the developed API a developer has access to a geocoding API that by receiving a city name or a city name and country code returns the coordinates associated with the parameters passed by the developer. For these, the endpoints are:
/api/location/city_country?city={city_name}&country={country_code}



```
@GetMapping(path = "city_country")
public ResponseEntity<Location> getLocationCityCountry(@RequestParam String city, @RequestParam String country){
```

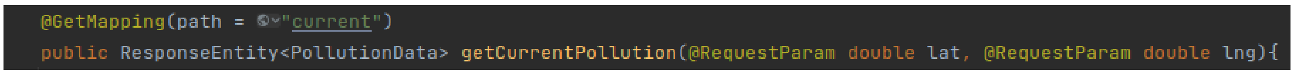
/api/location/city?city={city_name}



```
@GetMapping(path = "city")
public ResponseEntity<Location> getLocationCity(@RequestParam String city){
```

To complement the Geocoding api there is a second api, the Air Pollution API that takes a latitude and longitude and retrieves the pollution metrics relative to those same coordinates. This second API can either make a request to the OpenWeatherMap API or can access the cache directly, if the cache has that value in store.

/api/pollution/current?lat={latitude}&lng={longitude}



```
@GetMapping(path = "current")
public ResponseEntity<PollutionData> getCurrentPollution(@RequestParam double lat, @RequestParam double lng){
```

3 Quality assurance

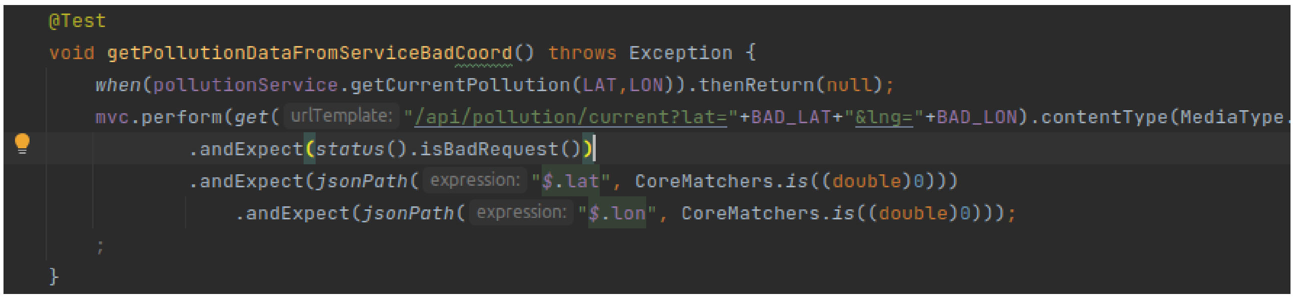
3.1 Overall strategy for testing

Test driven development was used mostly in the beginning of the project when creating the repositories for the SpringBoot section of the project, and from there on, the project was built and at the end, some controller related matters were handled taking in consideration some unit tests made previously.

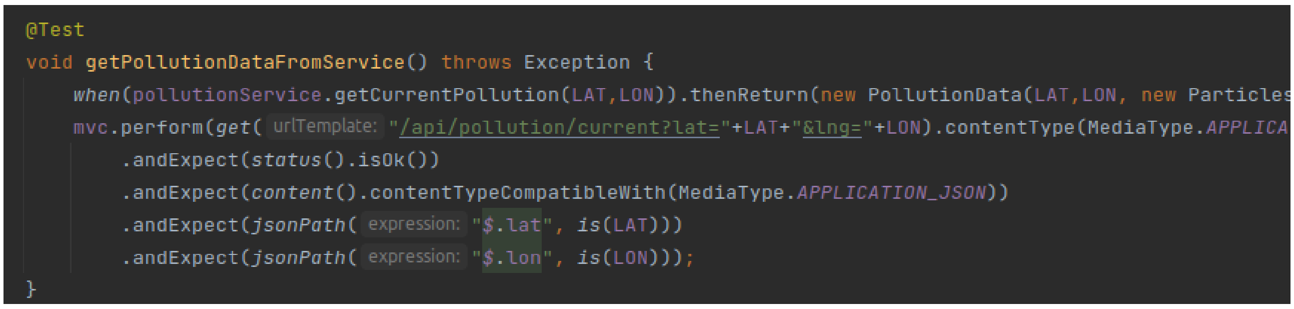
3.2 Unit and integration testing

For the unit and integration testing I tested the controllers, services and repositories both for the geocoding API and the Air Pollution API.

For the controllers, unit tests were made in order to predict what would happen if the requests were valid, invalid or in some cases one argument valid and the other invalid.
Examples:

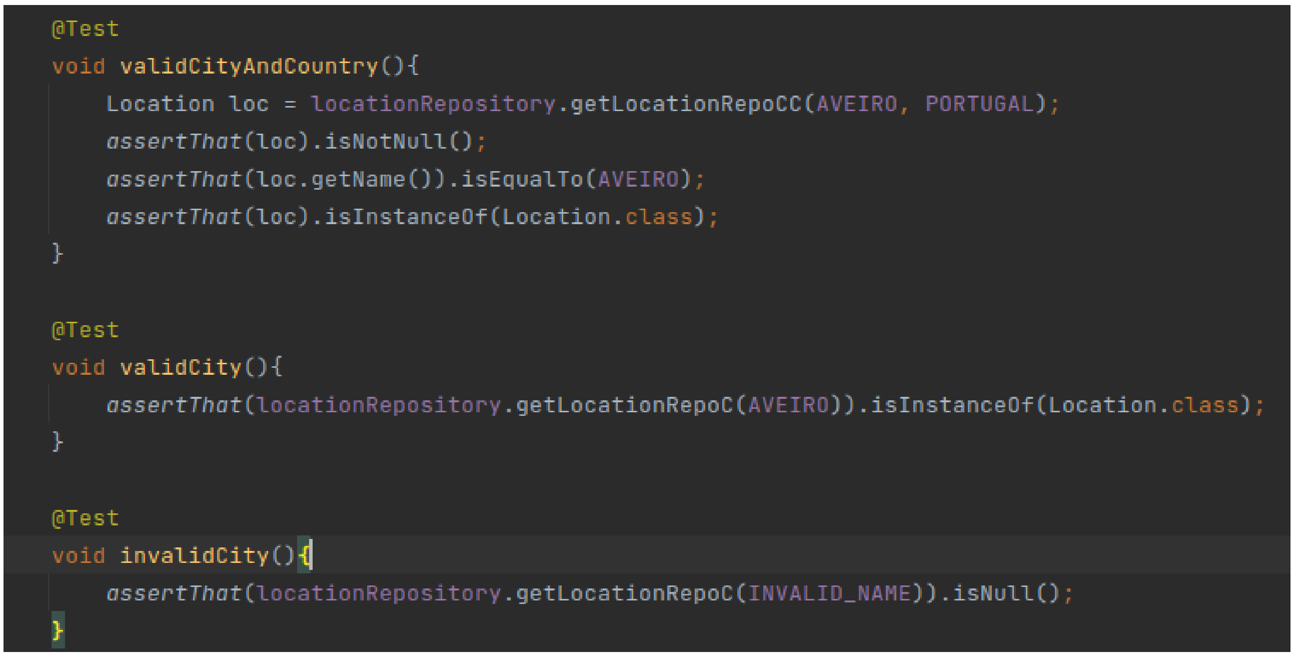


```
@Test
void getPollutionDataFromServiceBadCoord() throws Exception {
    when(pollutionService.getCurrentPollution(LAT,LON)).thenReturn(null);
    mvc.perform(get( urlTemplate: "/api/pollution/current?lat="+BAD_LAT+"&lng="+BAD_LON).contentType(MediaType.
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath( expression: "$.lat", CoreMatchers.is((double)0)))
                .andExpect(jsonPath( expression: "$.lon", CoreMatchers.is((double)0)));
        ;
}
```



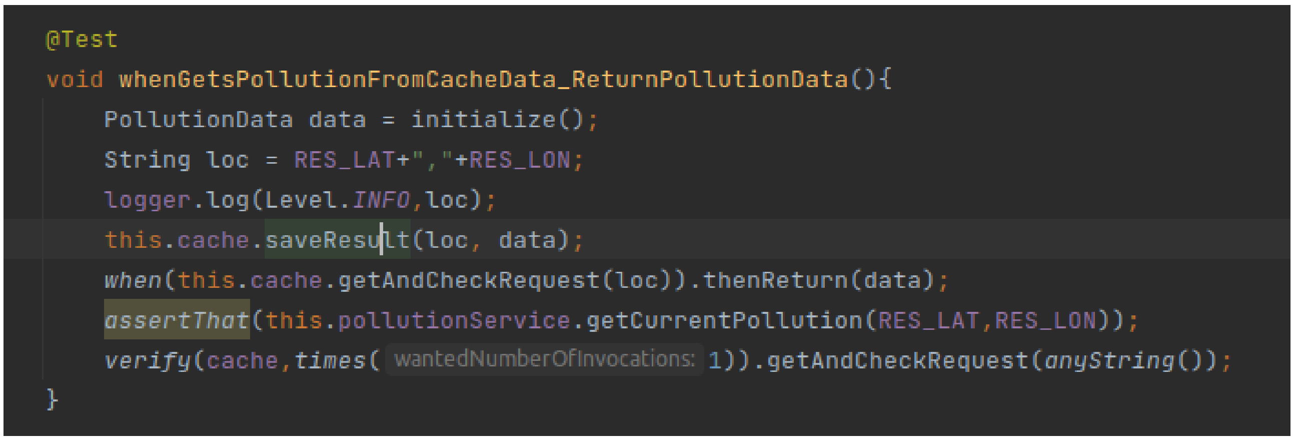
```
@Test
void getPollutionDataFromService() throws Exception {
    when(pollutionService.getCurrentPollution(LAT,LON)).thenReturn(new PollutionData(LAT,LON, new Particles
    mvc.perform(get( urlTemplate: "/api/pollution/current?lat="+LAT+"&lng="+LON).contentType(MediaType.APPLICA
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath( expression: "$.lat", is(LAT)))
        .andExpect(jsonPath( expression: "$.lon", is(LON)));
}
```

The creation of the tests for the repositories had the same points in mind in order to check how they would behave when confronted with valid and invalid information.
Examples:



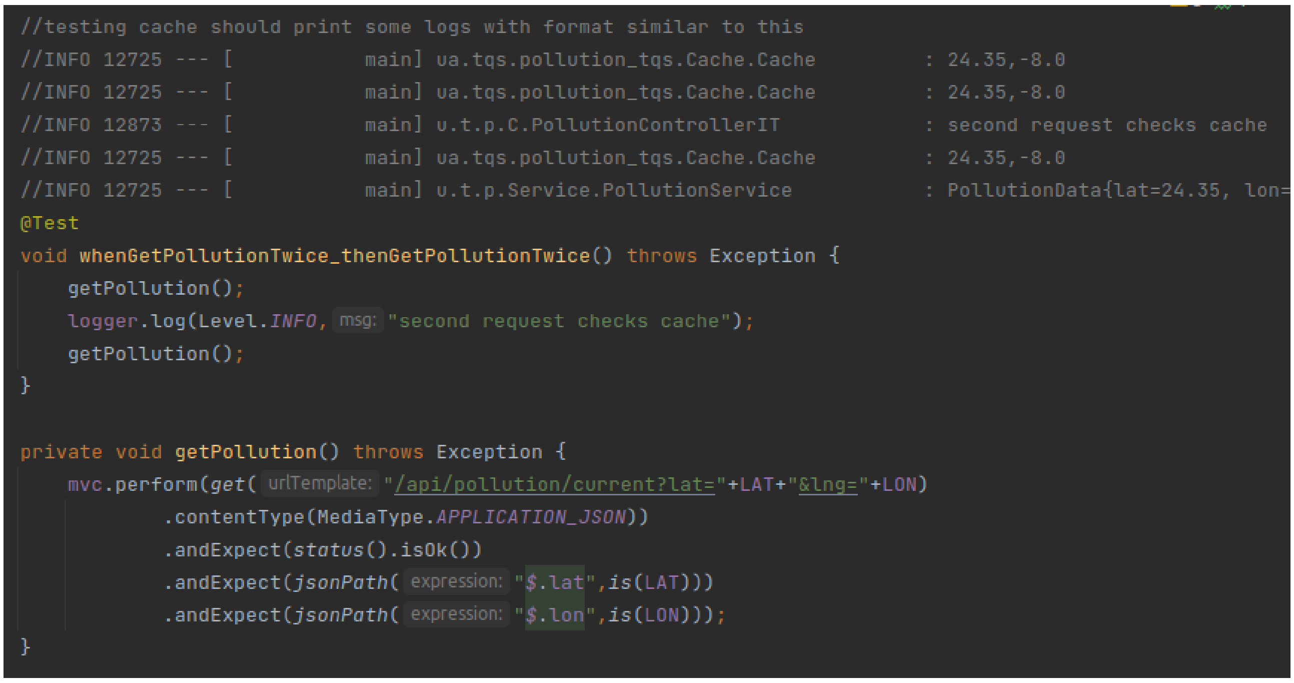
```
@Test
void validCityAndCountry(){
    Location loc = locationRepository.getLocationRepoCC(AVEIRO, PORTUGAL);
    assertThat(loc).isNotNull();
    assertThat(loc.getName()).isEqualTo(AVEIRO);
    assertThat(loc).isInstanceOf(Location.class);
}

@Test
void validCity(){
    assertThat(locationRepository.getLocationRepoC(AVEIRO)).isInstanceOf(Location.class);
}

@Test
void invalidCity(){
    assertThat(locationRepository.getLocationRepoC(INVALID_NAME)).isNull();
}
```

When it comes to the services the thought process was the same as before, but this time, I also checked whether or not the data was retrieved from cache or not when it comes to the pollution related service, so the following test was made:



```
@Test
void whenGetsPollutionFromCacheData_ReturnPollutionData(){
    PollutionData data = initialize();
    String loc = RES_LAT+","+RES_LON;
    logger.log(Level.INFO,loc);
    this.cache.saveResult(loc, data);
    when(this.cache.getAndCheckRequest(loc)).thenReturn(data);
    assertThat(this.pollutionService.getCurrentPollution(RES_LAT,RES_LON));
    verify(cache,times( wantedNumberOfInvocations: 1)).getAndCheckRequest(anyString());
}
```

Having a look at integration tests, each API (geocoding and pollution), had their own integration test, these were made using the SpringBootTest annotation and tried to cover as many cases as possible, when it comes to possible mistakes made from class to class. Here was also checked whether or not the cache was being used when the service receives information from the pollution controller.

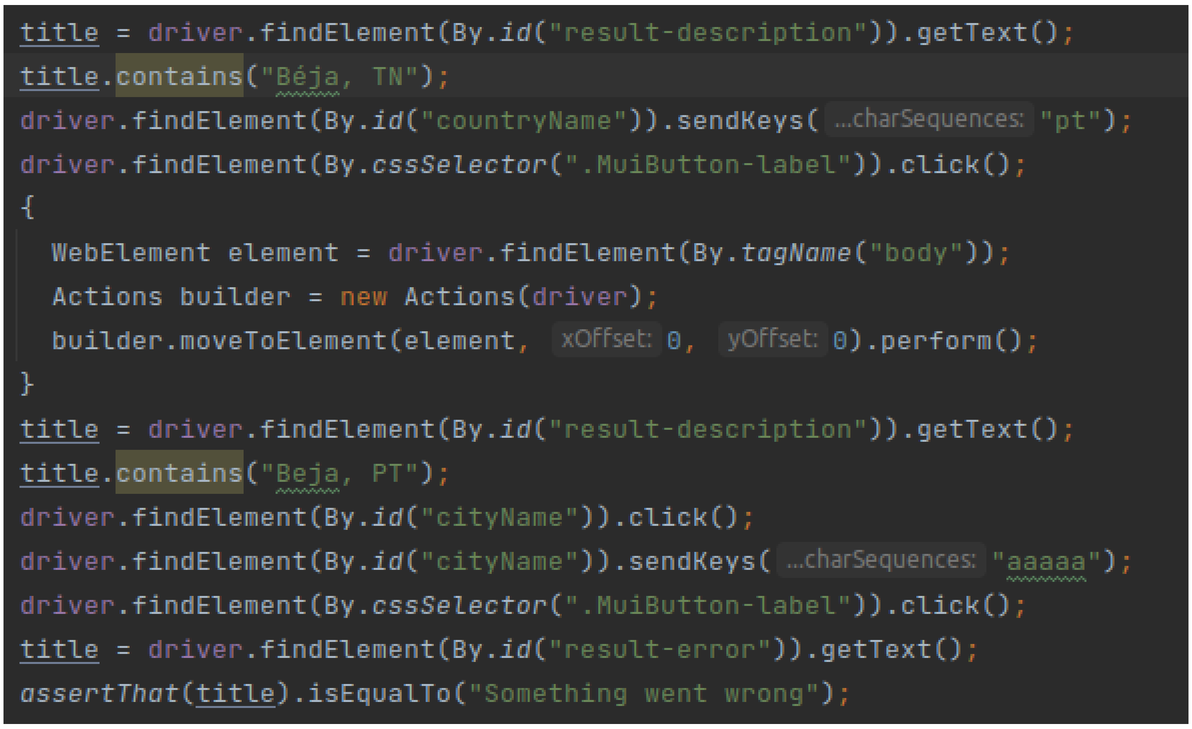


```
//testing cache should print some logs with format similar to this
//INFO 12725 --- [           main] ua.tqs.pollution_tqs.Cache.Cache         : 24.35,-8.0
//INFO 12725 --- [           main] ua.tqs.pollution_tqs.Cache.Cache         : 24.35,-8.0
//INFO 12873 --- [           main] u.t.p.C.PollutionControllerIT            : second request checks cache
//INFO 12725 --- [           main] ua.tqs.pollution_tqs.Cache.Cache         : 24.35,-8.0
//INFO 12725 --- [           main] u.t.p.Service.PollutionService           : PollutionData{lat=24.35, lon=
@Test
void whenGetPollutionTwice_thenGetPollutionTwice() throws Exception {
    getPollution();
    logger.log(Level.INFO, msg: "second request checks cache");
    getPollution();
}

private void getPollution() throws Exception {
    mvc.perform(get( urlTemplate: "/api/pollution/current?lat="+LAT+"&lng="+LON)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.lat",is(LAT)))
            .andExpect(jsonPath( expression: "$.lon",is(LON)));
}
```

The getPollution method is called when the attributes are valid and the method above calls the function twice, and since the information is the same in both it should check the cache in the second request and get a similar result to the comments above.

3.3 Functional testing

For functional testing, aided by selenium, I created a test which consisted of, searching by city, searching by city and finally searching for an invalid city and checking if the texts matched with the ones expected.

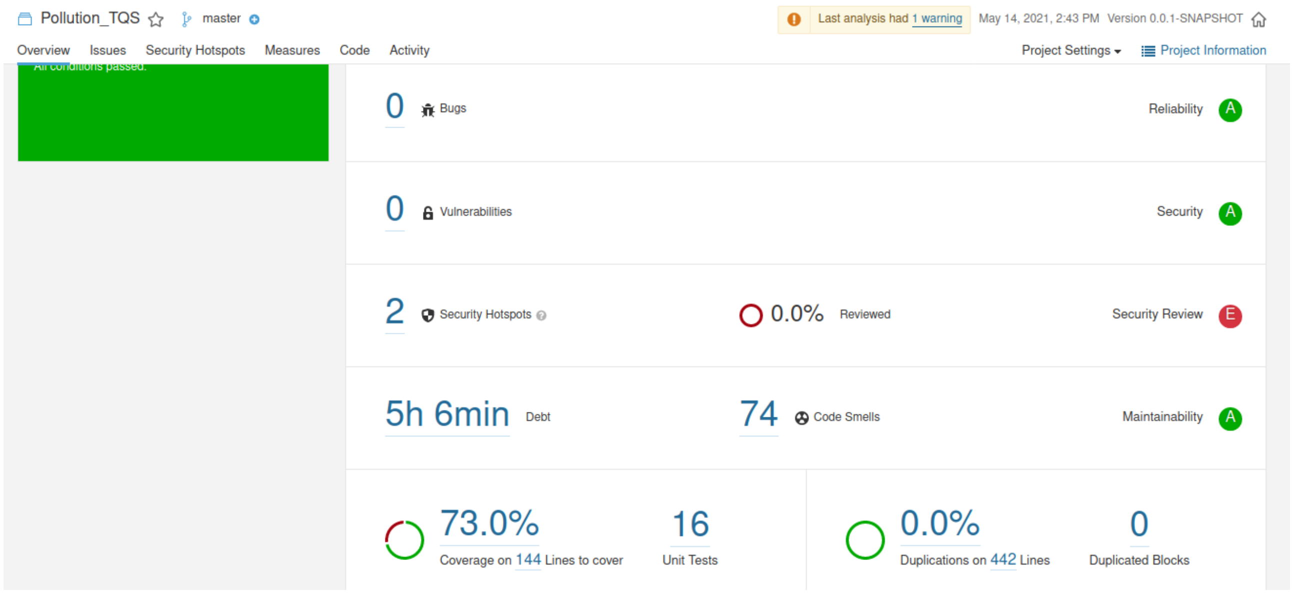


```
title = driver.findElement(By.id("result-description")).getText();
title.contains("Béja, TN");
driver.findElement(By.id("countryName")).sendKeys( ...charSequences: "pt");
driver.findElement(By.cssSelector(".MuiButton-label")).click();
{
  WebElement element = driver.findElement(By.tagName("body"));
  Actions builder = new Actions(driver);
  builder.moveToElement(element,  xOffset: 0,  yOffset: 0).perform();
}
title = driver.findElement(By.id("result-description")).getText();
title.contains("Beja, PT");
driver.findElement(By.id("cityName")).click();
driver.findElement(By.id("cityName")).sendKeys( ...charSequences: "aaaaa");
driver.findElement(By.cssSelector(".MuiButton-label")).click();
title = driver.findElement(By.id("result-error")).getText();
assertThat(title).isEqualTo("Something went wrong");
```

3.4 Static code analysis

In order to analyze the code statically SonarQube side-by-side with JaCoCo to check the unit test coverage of all unit tests.

Initially the code presented a considerable amount of code smells for its size, so these had to be fixed. The unit test coverage was 73%, which is not the best, yet these tests do not cover models nor the cache file.



Pollution_TQS master | Last analysis had 1 warning | May 14, 2021, 2:43 PM Version 0.0.1-SNAPSHOT

Overview Issues Security Hotspots Measures Code Activity | Project Settings | Project Information

All conditions passed.

0 Bugs — Reliability A

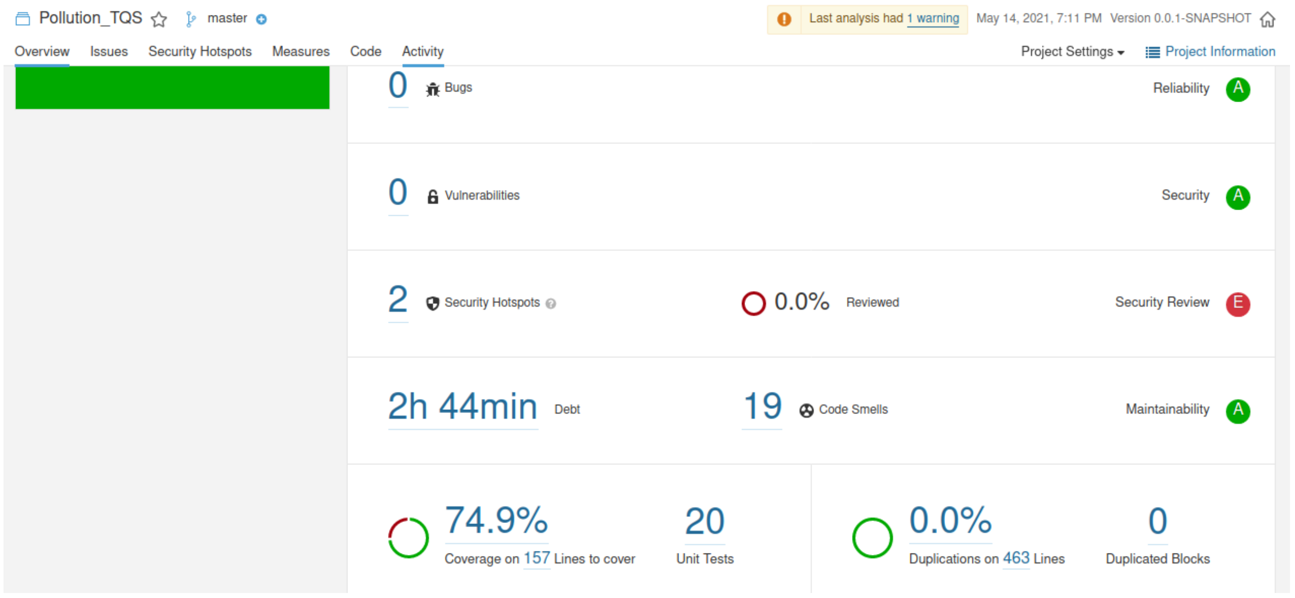
0 Vulnerabilities — Security A

2 Security Hotspots — 0.0% Reviewed — Security Review E

5h 6min Debt — 74 Code Smells — Maintainability A

73.0% Coverage on 144 Lines to cover — 16 Unit Tests — 0.0% Duplications on 442 Lines — 0 Duplicated Blocks

The final result, after removing code smells, improving some tests and fixing some bugs resulted in a lot less code smells, 19 to be exact, 11 of those are due to the package name not being of the desired format.



Pollution_TQS master | Last analysis had 1 warning | May 14, 2021, 7:11 PM Version 0.0.1-SNAPSHOT

Overview Issues Security Hotspots Measures Code Activity | Project Settings | Project Information

0 Bugs — Reliability A

0 Vulnerabilities — Security A

2 Security Hotspots — 0.0% Reviewed — Security Review E

2h 44min Debt — 19 Code Smells — Maintainability A

74.9% Coverage on 157 Lines to cover — 20 Unit Tests — 0.0% Duplications on 463 Lines — 0 Duplicated Blocks

Finally, both in the initial analysis and the final, there were 2 secure hotspots because of allowing CORS in the controllers.

4 References & resources

Reference materials

https://www.baeldung.com/spring-boot-testresttemplate
https://www.baeldung.com/rest-template
https://bonigarcia.github.io/selenium-jupiter/