HW1: Mid-term assignment report

José André Lopes Frias [89206], v2020-04-15

1 Introduction

1.1 Overview of the work

Introducing the application and its main aspects, it allows the search of air quality metrics and information for a place, given its coordinates, providing an API as well where that data can be retrieved. In the web interface, the user specifies the latitude and longitude of the place in the home page, appearing then the air quality information produced by the service, being that health recommendations for the general population, concentration of the pollutants and an index calculated by the remote BreezoMeter API, where the value of AQI is defined. This AQI value is responsible to tell the current quality of the air, telling if it is clean or polluted. The value of the AQI is calculated given the concentration of pollutants in the air. The higher the AQI value, more air pollution exists, being that AQI values below 100 are acceptable and therefore normally don't represent great dangers to public health. The AQI standards are defined by EPA, the Environmental Protection Agency, with the headquarters located in the United States.
The BreezoMeter Service collects and calculates air quality variables through air pollution monitoring stations, where sensors are disposed, obtaining the presence and concentration of the pollutants and with that provide the BreezoMeter index, where the AQI value is, being accessible by the provided API. The BreezoMeter API also makes available index calculated by the government, however despite that governments usually do a great job monitoring the air, the data many times is not real time, is not location based and is not possible to fetch data from the specific coordinates that we are looking, that is, the data is fetched through the nearest sensor and sometimes that information is not the best. Furthermore, government ways of calculating air quality are different from country to country. With this, the government index will not be presented in the application, given that only the BreezoMeter index will be available, where the results are standardized across the world.
When it comes to the tests, the project also has unit and functional testing as well static code analysis, contextualized in TQS course. Unit tests are present to test the mechanisms of the Cache, the service

layer and the controller mapping and responses. Functional testing is mainly for testing the web interface, where Selenium is used.

1.2 Limitations

The application has some limitations, considering the interaction with the BreezoMeter API and how it is presented, as well limitations defined and considered in the stages of the development.
Therefore, the limitations are that considering the web interface, it is only possible to search given the coordinates of a city, not allowing to search by the city name, which is not user friendly, since that the user first needs to search the coordinates of the city or place that wants to search. Another limitation, also in the web interface, is that is only possible to show the air quality data of the current day and not search by a specific day, limitation not present in the API of the application, where it is possible to search for the current date or specify a date and see the intended information. Also, the application only retrieves information from BreezoMeter API, being that if the endpoint of this remote API is not available, then the application is not useful unless the required data is still available in cache, where in that case can be fetched without using the remote API.
One of the planned features would be to have the ability to search by the city/place name, considering for that another remote API that when the users enters in the search field a city name, the product would be responsible to convert that name to its respective coordinates, since the service as it is only works with coordinates. To bridge the second limitation, the web interface would be able to specify a date for when the air quality is required.
For the third limitation, the application would have to have mechanisms to see if a certain endpoint is available, being that if it isn't, would contact another remote API. For that, it also would be necessary to model the response of this new API, as it happens for the BreezoMeter API.

2 Product specification

2.1 Functional scope and supported interactions

The intended users for this application are the people that simply see how the air quality is evolving, that is, people that have concerns for the environment that want to observe the evolution of the pollutant's concentration. Another purpose would be following the health recommendations provided by the service, taking health precautions given the data, affecting our decisions and behaviors, since air pollution is extremely bad for public health. Therefore, people would rely on the application to make informed decisions and evaluate, for example, when to go running given the air quality data that is being presented.
A different approach would be statistic or research purposes, allowing weather and air quality services to study the behavior of the air quality or simply showing the results in their systems, using for that the available API of the service to retrieve the data. However, it would not be usual, since the services are available in BreezoMeter and this way two API calls would be necessary, hindering the performance of the service.

2.2 System architecture

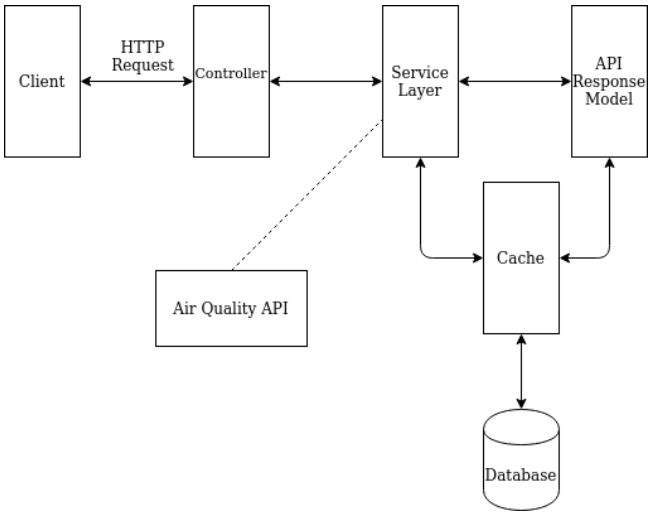

Figure 1: Architecture diagram of the application.

It was necessary to model the response of the BreezoMeter API, in order to create the necessary classes to save the corresponded data in objects, as well to save and query the repository. Cache is responsible to query the repository and save responses of the remote API. When at the controller is requested air quality data, the controller will ask to service layer to fetch the data. The service then verifies if the requested data is available in the cache, leading to one of two ways: if the requested data is available at the cache, then there is no need to contact the remote API to new data, it's only necessary to contact the cache and fetch the data, which in turn will contact the repository. With this way, the data is fetched faster. If the data is not available at the cache, whether that be because the object has already expired or never was requested from the API, then it is necessary to call the BreezoMeter API to show the data and save it locally, so that if it is requested again, it is already in cache. Cache wraps the API response data in a new object composed by the data and an expire time associated to that data, that is, the time where the data is no longer valid. With the help of a cleaning thread in cache, it is verified regularly if there is stale data in the repository, that is, if the expire time of a certain record is bigger than the moment at what the cleaning thread is running. If that happens, then the object is removed from the repository and if another request for the same data arrives it is necessary to fetch the data from the BreezoMeter API again.
The application uses Spring Boot Framework, on which it is presented a controller responsible for mapping the requests and contacting the service layer, as well to provide the web pages and the application API responses. In the web pages was used Thymeleaf with HTML to present the data. To retrieve the data, it is necessary to contact with the service layer, that implements the business logic and contact with the cache to get the data, in order to be used by the controller. The cache is the only responsible object to contact with the repository, where methods are mapped into native queries in the repository, an H2 memory database, that is, an in memory lightweight database that stores coordinates and the API response for that coordinates, associated with a Time To Live for each record.

2.3 API for developers

The services available by the application provided by the REST API, using Swagger, are presented below.

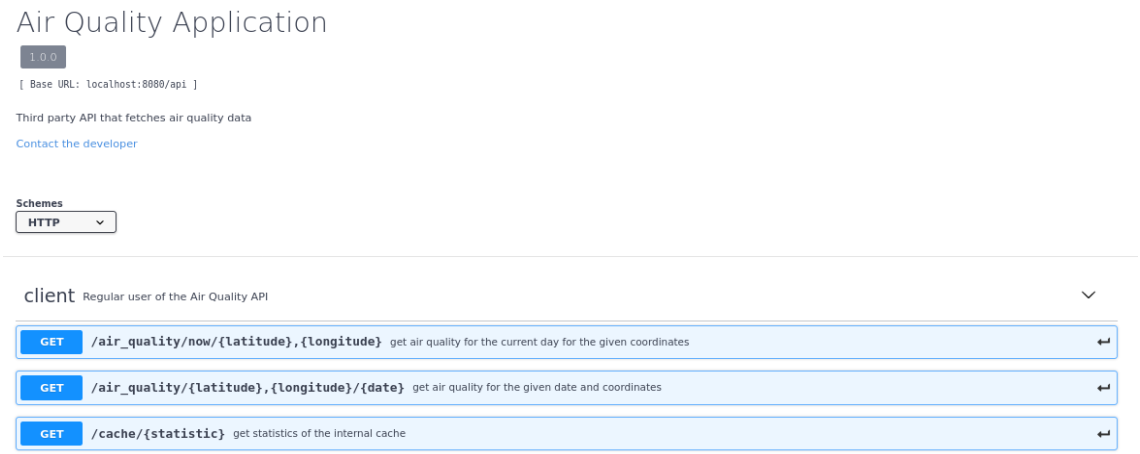

Figure 2: API endpoints of the application.

Example of results obtained when getting the air quality metrics for the current date given determined coordinates:

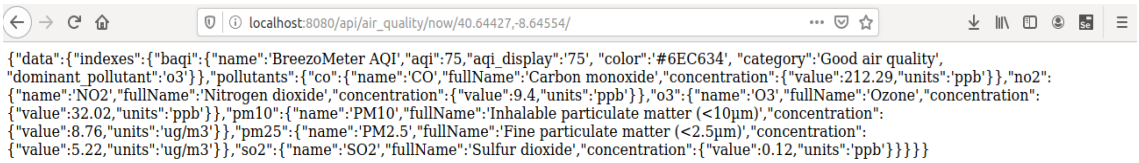

Figure 3: Example of the response of the application API when fetching air quality data for the current date.

Given a specified date, results do not show the pollutants information, which is related to how BreezoMeter API provides the data. This leads to an issue where if we define the date as being the current date, then at the web page, since it verifies if the data is in cache, the web page will also not present the pollutants information. Therefore, to fetch the data for the current date the best way is to use the mapping defined above.

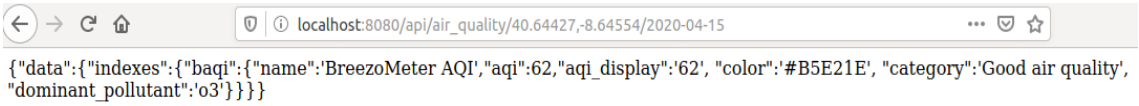

Figure 4: Response example of the application API when fetching data air quality data for a specific date.

3 Quality assurance

3.1 Overall strategy for testing

For testing, the overall strategy that was used was TDD, mainly in the Cache and Service layers, where the unit tests first were designed and only after the code that would make pass the tests was defined, without discarding the refactoring of the code produced, leading to a method were the tests were first written and corrected and only after move to next feature.

It was also used REST-Assured where the REST API provided by the application was tested and validated, watching after the responses provided by the API calls, seeing if they match the expectations.

3.2 Unit and integration testing

For the service layer, the tests that were done were primarily related to cache accesses and when and not to call the remote API, that is, testing if the data requested is in cache then there is no need to do a remote call. On the other hand, if the data is not in cache then testing to see if the remote API is called were needed. Therefore, to produce unit tests it was necessary to mock the behavior of the cache using Mockito. The next figure shows the mock of the cache, used in Service Layer Tests.

```
Mockito.when(cache.get(coordinate, currentDate)).thenReturn(coord_response);
Mockito.when(cache.containsCoord(new Coordinate( lat: -40.0, lon: -40.0),currentDate)).thenReturn(true);
Mockito.when(cache.containsCoord(new Coordinate( lat: -20.0, lon: -20.0),currentDate )).thenReturn(false);
Mockito.when(cache.get(coordinate, date: "2020-04-02")).thenReturn(coord_response);
Mockito.when(cache.containsCoord(coordinate, date: "2020-04-02")).thenReturn(true);
Mockito.when(cache.containsCoord(new Coordinate( lat: -20.0, lon: -40.0), date: "2020-04-02")).thenReturn(false);
Mockito.when(cache.add(coordinate, responseData)).thenReturn(coord_response);
Mockito.when(cache.containsCoord(coordinate, date: "2020-04-09")).thenReturn(false);
```

Figure 5: Mock of the cache behavior in the tests of the service layer.

With these specifications, concerning the way that cache should act, multiple unit tests were written concerning the service layer, being one example the next figure:

```
@Test
public void whenExistingCoordinates_thenResponseShouldBeFound() throws ParseException {
    double lat=-40.0;
    double lon=-40.0;
    Index index=new Index( name: "baqi", aqi: 70, aqiDisplay: "70", color: "#fff", category: "category1", dominantPollutant: "co");
    IndexList indexList= new IndexList(index);
    DateFormat formatter = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");
    Date date = formatter.parse( source: "2020-04-02 13:00:00");
    Data data=new Data(date, indexList);
    ResponseData responseData= new ResponseData( metadata: null, data, error: null);
    CoordResponse found = airQualityService.findByCoordinateAndDate(new Coordinate(lat, lon), date: "2020-04-02");
    assertThat(found.getR().getResponseData()).isEqualTo(responseData);
}
```

Figure 6: Example of a unit test inserted in the service layer testing.

This test verifies if when there are the coordinates that the user is looking in cache, then response should be found and should be equal to the data that was originally placed in the mock of the cache. On the other hand, when the coordinates are not in the cache then response of cache should be null, like is tested in the next screenshot, verifying also if the corresponded method called in cache by the service is in fact called.

```
@Test
public void whenInValidCoordinates_thenResponseShouldNotBeFound() {
    double lat=0.0;
    double lon=0.0;
    CoordResponse fromDb = airQualityService.findByCoordinateAndDate(new Coordinate(lat, lon), date: "2020-04-02");
    assertThat(fromDb).isNull();
    verifyCacheFindByCoordAndDateIsCalledOnce(lat, lon, date: "2020-04-02");
}
```

Figure 7: Example of a unit test in the service layer.

Other tests considered were covering of getting the data specifying not only the coordinates, like above, but also considering the intended date, to check if the repository has data with certain coordinates and a specified date. Overall, that tests are similar to the tests already described. Also, tests to retrieve the statistics of the cache were considered.

For the cache unit testing, the tests covered the interactions between the cache and the repository, verifying the number of calls in the functions and the cache statistics and methods, using a mock for the repository.

```
Mockito.when(airQualityRepository.findByCoordAndDate(coordinate.getLat(), coordinate.getLon(), year: 2020, month: 4, day: 2))
        .thenReturn(coord_response);
```

Figure 8: Mock of the repository

This way, when the data is available then data should be retrieved and be equal to the data mocked.

```
@Test
public void whenExistingCoordinatesAndDate_thenResponseShouldBeFound() {
    double lat=-40.0;
    double lon=-40.0;
    CoordResponse found = cache.get(new Coordinate(lat, lon), date: "2020-04-02");

    assertThat(found.getC().getLat()).isEqualTo(lat);
    assertThat(found.getC().getLon()).isEqualTo(lon);
}
```

Figure 9: Example of a unit test in cache.

When there is no data available to the specified coordinates or a date then response should be null.

```
@Test
public void whenNotExistingDate_thenResponseShouldNotBeFound() {
    double lat=-40.0;
    double lon=-40.0;
    CoordResponse fromDb = cache.get(new Coordinate(lat, lon), date: "2019-04-02");
    assertThat(fromDb).isNull();

    verifyFindByCoordAndDateIsCalledOnce(lat, lon, date: "2019-04-02");
}

@Test
public void whenNotExistingCoords_thenResponseShouldNotBeFound() {
    double lat=-20.0;
    double lon=-20.0;
    CoordResponse fromDb = cache.get(new Coordinate(lat, lon), date: "2020-04-02" );
    assertThat(fromDb).isNull();

    verifyFindByCoordAndDateIsCalledOnce(lat, lon, date: "2020-04-02");
}
```

Figure 10: Example of unit tests in cache.

At the Controller, the API was tested with integration testing, using Rest Assured to make the tests, firstly doing a more localized test in the controller, testing the API provided, seeing the data that is

returned by it. One of the examples is the following snapshot, where it is being tested a get request to fetch air quality for a given date and verify if the data that is returned is the data that is expected. In this example the service layer was a mock.

```
@Test
public void givenLatAndLonAndDate_whenData_thenReturnData() throws Exception {
    Coordinate coordinate= new Coordinate( lat: -40.0,  lon: -40.0);
    Index index=new Index( name: "baqi",  aqi: 70,  aqiDisplay: "70",  color: "#fff",  category: "category1",  dominantPollutant: "co");
    IndexList indexList= new IndexList(index);
    DateFormat formatter = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");
    Date date = formatter.parse( source: "2020-04-04 13:00:00");
    Data data=new Data(date,indexList);
    ResponseData responseData= new ResponseData( metadata: null, data,  error: null);
    CacheObject cacheObject= new CacheObject(responseData,  expireTime: 10000);
    CoordResponse coord_response= new CoordResponse(coordinate, cacheObject);
    given(service.getDataByDate(coordinate,  date: "2020-04-04")).willReturn(responseData);
    mvc.perform(get( urlTemplate: "/api/air_quality/-40.0,-40.0/2020-04-04").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
    .andExpect(jsonPath( expression: "$.data.indexes.baqi.name", is(index.getName())))
            .andExpect(jsonPath( expression: "$.data.indexes.baqi.aqi", is(index.getAqi())))
    .andExpect(jsonPath( expression: "$.data.indexes.baqi.aqi_display", is(index.getAqiDisplay())))
    .andExpect(jsonPath( expression: "$.data.indexes.baqi.color", is(index.getColor())))
            .andExpect(jsonPath( expression: "$.data.indexes.baqi.category", is(index.getCategory())))
            .andExpect(jsonPath( expression: "$.data.indexes.baqi.dominant_pollutant", is(index.getDominantPollutant())));

    verify(service, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).getDataByDate(coordinate,  date: "2020-04-04");
    reset(service);
}
```

Figure 11: Example of an integration test in controller.

Besides the test case considered above, additional integration testing was done in Controller, starting the full web context using @SpringBootTest and enabling Web Environment, deploying the API into the SpringBoot context and using Spring Boot MockMvc, like it is shown in the next figure.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, classes = AirQualityApplication.class)
@AutoConfigureMockMvc
@AutoConfigureTestDatabase
public class ControllerITest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private AirQualityRepository airQualityRepository;

    @AfterEach
    public void resetDb() {
        airQualityRepository.deleteAll();
    }
```

Figure 12: Controller configuration for Integration Test.

With this configuration, the tests were identical to the ones already defined where the service layer was a mock.

3.3 Functional testing

It was used Selenium for the functional testing at the web interface. Since the web interface is simple, the tests were directed to the presence of certain text, mainly the titles and subtitles, where then the rest of the information is shown. The page object model was considered but not implemented since

the interface does not have a great number of items. In the next snapshot it is being considered the test were the user digits valid coordinates, that will lead to the air quality data to be presented.

```
public void interface_test() {
  driver.get("http://localhost:8080/");
  driver.manage().window().setSize(new Dimension( width: 1299,  height: 713));
  driver.findElement(By.id("lat")).click();
  driver.findElement(By.id("lat")).clear();
  driver.findElement(By.id("lon")).clear();
  driver.findElement(By.id("lat")).sendKeys( ...charSequences: "40.64427");
  driver.findElement(By.id("lon")).click();
  driver.findElement(By.id("lon")).click();
  {
    WebElement element = driver.findElement(By.id("lon"));
    Actions builder = new Actions(driver);
    builder.doubleClick(element).perform();
  }
  driver.findElement(By.id("lon")).click();
  driver.findElement(By.id("lon")).click();
  driver.findElement(By.id("lon")).sendKeys( ...charSequences: "-8.64554");
  driver.findElement(By.cssSelector(".fa")).click();
  driver.findElement(By.cssSelector("h1")).click();
  assertThat(driver.findElement(By.cssSelector("h1")).getText(), is( value: "Air Quality for lat 40.64427 and lon -8.64554"));
  driver.findElement(By.cssSelector("li:nth-child(1)")).click();
  assertThat(driver.findElement(By.cssSelector("li:nth-child(1) > span")).getText(), is( value: "BreezoMeter AQI"));
  driver.findElement(By.cssSelector("h4:nth-child(3)")).click();
  assertThat(driver.findElement(By.cssSelector("h4:nth-child(3)")).getText(), containsString( substring: "Health Recommendations:"));
  driver.findElement(By.cssSelector("h3")).click();
  assertThat(driver.findElement(By.cssSelector("h3")).getText(), is( value: "Pollutants and concentrations"));
}
```

Figure 13: Functional testing in the web interface using Selenium.

3.4 Static code analysis

For static code analysis it was used Sonarqube. Initial code smell was due to static variables and the way that were being accessed, since at the beginning were not being used getters to retrieve the static variables in Cache class. Minor issues involved package names matching regular expressions and using prints instead of loggers, that without sonarqube they would not be addressed. The 9 code smells present in the dashboard below are due to package naming and the security hotspot present is inherent to Spring Boot Applications.

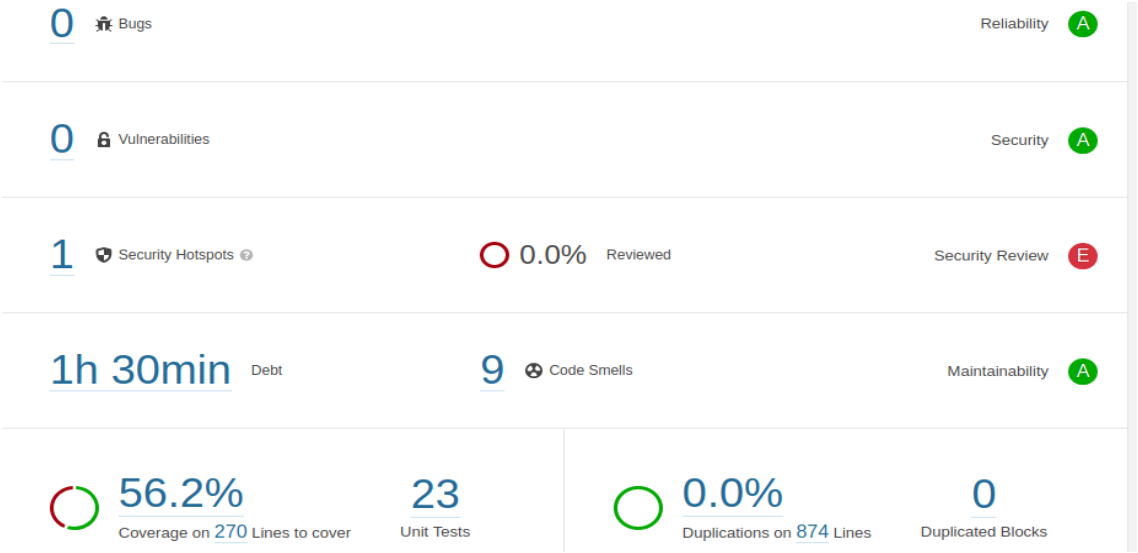

Figure 14: Sonarqube dashboard.

Concerning the coverage of the tests, it is 56.2%. For the service layer and cache, as well for the tests in the controller the percentage of coverage is above the average described by the sonarqube. Since tests not considered getters, setters, equals and toString methods of the model classes, the medium coverage is lower than the percentages of coverage in cache and service.

4 References & resources

Project resources

- Git repository: https://github.com/joselfrias/air_quality
- Video demo: file air_quality_demo.mp4 on git repository root