Air Rest API

Pedro Miguel Rocha Oliveira, 89156

2020-04-15

Contents

1 Introduction

1.1 Overview of the Work

This project was introduced in the course of TQS: Test and Quality of Software; where we had to implement a simple REST API that would make its own API calls to external sources, had a simple frontend to receive the inputs from the client, and a structure to simulate a cache. The main goal of the service is to allow the client to see the air statistics on an area by introducing its geographical coordinates.

This part of the project is only done as means to practice the focus of the course: testing the software functions, creating unit tests for each module, including Mock tests, integration tests for the REST classes, and finally functional tests.

As taught in class, the service was developed using the Spring framework and thoroughly tested with JUnit. While the process itself will be explained later in this report, JUnit is a tool that allows for unit testing of the classes implemented, where each class and its methods are being tested as single individual components.

Some of the classes from SpringBoot have dependencies that need to be met, namely Controller and the Server, so the JUnit library was not enough to fully test these classes. To meet the dependencies, the Mockito library was used for to allow for the mocks of the classes, i.e. it was possible to simulate the behaviour of the dependency for the test. With this tool, by controlling what the mock class returns, I was able to force the tested class into certain behaviours to test if it was doing what was expected.

However, it must be noted that just because all the classes are working in a vacuum, they may not be working as a whole, this is where Integration Testing is introduced. An integration test will now use the actual implementations of the dependencies, not simple mocks, and, by doing so, I was able to check if any information was being passed wrong between the classes that may not have been obvious at first with simple unitary testing.

Finally, the last type of tests were functional tests: by using the Selenium Drivers, I was able to test the frontend of the app, to check if all values were being returned as expected and appeared where they should. This is a simple type of testing, as it's just interacting with the Frontend components of the project, but the Selenium itself does have some important limitations regarding how it can check the HTML of the webpage, e.g. interacting with modals seems to be particularly hard for the drivers.

This project also calls for some type of code revision, to make sure that, while it may pass the tests I've created, the code remains well written, without security issues or bugs that I've not accounted for. Using the SonarQube tool, I was able to quickly detect the code smells - lines of code that may not initially cause any problems, but indicates a deeper problem.

Furthermore, version control was done through Github and a Continuous Integration Pipeline was developed to work with the GitLab services.

1.2 Limitations

The main focus of the project was to allow the clients to input the geographical coordinates of a location of their choice, and get the Air data of that area: including pollutants concentration and overall air quality. While this was achieved, there were a couple of features that were planned but could not be finished.

For instance, giving the user a friendlier way of choosing the location. Picking geographical coordinates can be a bit awkward, as it forces the client to go out of their way to check the data, instead of being able to look up the city or the park, they have to go check the particular coordinates. A future feature could be implemented to bypass this constriction by having some amount of more common locations by name, internally connecting these names to the correct coordinates. This would allow for a much easier look up on the user side.

Another constriction of this service is that it does not allow for the lookup of data on different time periods. As it stands, a user, by inputting the coordinates, get the current data on the area; it does not, however, let them search past data or future predictions. This would be an interesting feature to implement, since a lot of people would prefer to know how the air is predicted to be rather than how it currently is; or would like the option to check the statistics on the last days.

One final important thing to note is that was I not capable of covering all the functions with appropriate tests, especially in terms of exceptions or other wrongdoings.

2 Product Specification

2.1 Function Scope and Supported Interactions

The service in question will allow the clients two main functions, based on GET Requests to the main controller: air information based on the coordinates and what the user wants to see (he may filter based on the pollutants he wants to check); and the cache information, that is how many hits and misses or overall requests have been done since the server has been active or since the last time it has been reset (an operation also available to the user).

The first use case serves as the main goal of the project: allow the person to be able to get the quality of the air in a certain location by inputting the geographical coordinates. This is done by sending a get request to the main server, who will process the request and either forward the request to an external API or, if it is there, return what is saved in cache.

The second use case is a more specific one to our developers: the number of hits and misses our cache has had, a miss being defined as a request we had to forward to an external API, while a hit is a request which could be found on our cache.

Finally, there is a specific function for our clients in case they want a quick way of cleaning the cache. It will send the request to our server, who will then do a full wipe on the requests, starting the hits and miss count from 0.

2.2 System Architecture

The basic architecture of the system can be described as in the following diagram.

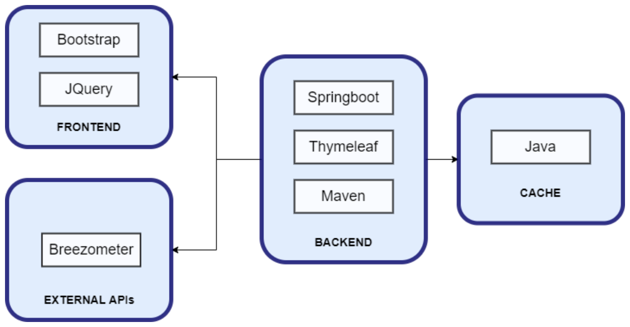


Figure 1: System Architecture for the Service

The main backend was done via the Spring Framework. The Spring framework provides a new comprehensive programming and configuration model for Java-based applications for any development environment. It's an open source tool, with a large and active community to provide constant feedback for the diverse range of use cases the framework provides.

Maven is a build automation tool for the java projects. It provides a good, high-level way to use outside dependencies, easing the way to compile, test, and package the app into a .JAR file.

Finally, to connect the backend to the frontend, I used the Thymeleaf engine, so that the server would automatically start on the index HTML page without need for further configurations.

As for the backend, name cache and handling of the requests, I used the simple Java language. The cache itself was based purely on a HashMap that would connect the coordinates coming from the user to the proper responses.

As for the external APIs, in the case the coordinates were not found in cache, the request would be forward to the Breezometer API. This will get us the most current air quality information, including the Breezometer AQI (Air Quality Index) and the information about the main pollutants, the information we give to our clients.

Finally for the frontend, I used Bootstrap and JQuery since it was simple to use, well documented and I already had prior knowledge about the technologies from other courses.

Moving on to the class structure, as seen in the diagram below, it follows a relatively simple and straightforward model.

Following the MVC architecture, AirRestController serves as the application's controller class, that will communicate between the View and the Model, implemented here as the AirService. AirService will make the external API requests to Breezometer when the need arrives, and save the results to the cache, implemented here as AirRepository; of course these external requests won't be necessary if the results already exist in cache and aren't too old. AirRepository is a class that contains a HashMap connecting the locations the user inputs, saved as AirCoords, and the responses from the Breezometer API, saved in Air-

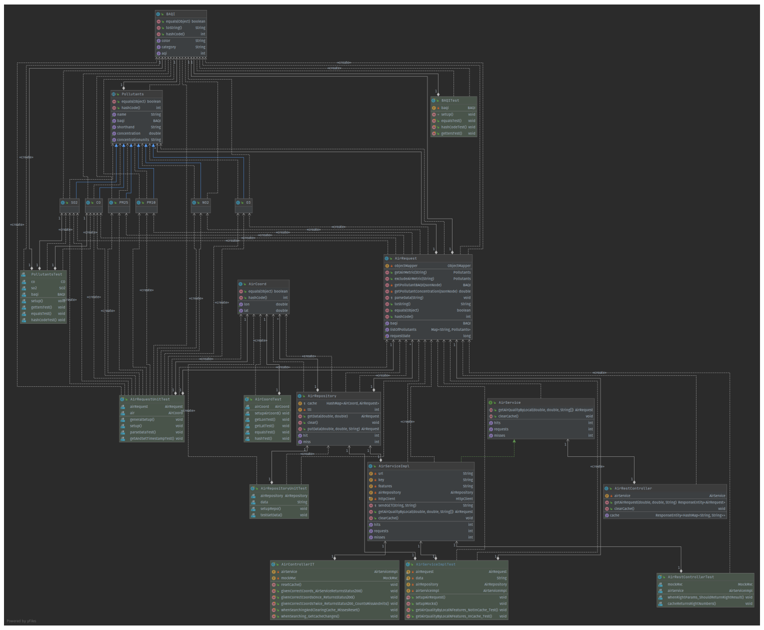
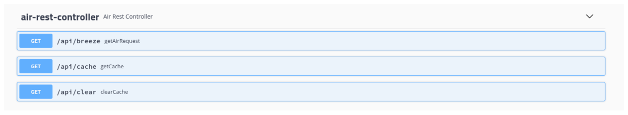


Figure 2: System Architecture for the Service

Request. Finally, AirRequest will contain all information we want to give our user: the Breezometer AQI (saved here as an object of the class BAQI) and all the different pollutants: CO, SO2, PM25, PM10, O3 and NO2.

2.3 API for Developpers

These are the endpoints currently available for the REST API.



air-rest-controller Air Rest Controller

GET `/api/breeze` getAirRequest

GET `/api/cache` getCache

GET `/api/clear` clearCache

Figure 3: System Architecture for the Service

3 Quality Assurance

3.1 Overall Strategy for Testing

For a project that prioritizes testing above all, I decided to go with a TDD - Test Driven Development. TDD is a development cycle where the requirements for a certain class or service will turn into specific Unit Tests. Code will then be developed to pass these tests. In case I needed to add a feature or function to a class, I first needed to create the appropriate test, and only then think about the implementation.

It was certainly different from what I usually do for my projects; it forced me to think about all functions I wanted my class to have, and, more specifically, what the concrete results I want each of these functions to return, and what the general workflow for each class should be.

While challenging at first, I quickly came around to the idea, realizing it was a very good and effective way of writing code; it forced my thought process to be more organized and properly structured. It also made sure that once I had passed the tests, I knew the class was ready to be used for the other entities.

3.2 Unit Testing

When the teacher introduced the project, he divided types of unit testing into three types of tests. In this report, I will talk of each of these categories by also introducing an example from my project.

3.2.1 Unit Testing

Unit testing in this case refers to the simpler classes, so I'll talk about the request and response management, and the tests that had to be done for the AirRequest class.

AirRequest receives as its argument for constructor the JSON result of a get request from the external API. The class has the goal of parsing this JSON String and get the right information from that string. Needless to say, this is a very integral part of project to work, as failing might put how the client receives everything in jeopardy.

So, there's only one important test: after construction of the object, all the results must be in accordance to the JSON. The test itself is then simple, it has a generic setup of:

```
@BeforeEach
  public void setup(){
      String data = (JSON request) ...;
      this.airRequest = new AirRequest(data);
  }
```

Now we have to check that all elements were properly initialized, by having assertions like the following:

```
BAQI baqi = new BAQI(63, "#AFDF21", "Good air quality");
assertEquals(baqi, this.airRequest.getBaqi());

CO co = new CO(
    new BAQI(99, "#009E3A", "Excellent air quality"),
    207.1
);
assertEquals(co, this.airRequest.getAirMetric("co"));
```

In other words, creating the object within the test with the data from the result we're using on the Set Up, and make sure that the object automatically created an object with the same parameters.

3.2.2 Service Level Testing

Service Level Testing is a type of testing that needs mocks for the other REST components, so I'll talk about the unit tests for the AirServiceImpl class.

To test the AirServiceImpl with Mocks, we had to use the classes and annotations from the Spring framework.

```
@Mock
private AirRepository airRepository;

@InjectMocks
private AirServiceImpl airServiceImpl;
```

Then I had to simulate the behaviour of airRepository to force a certain behaviour from my airService. For instance, in the following code:

```
when(airRepository.getData(48.857456, 2.354611)).thenReturn(null);

String[] features = new String[3];
features[0] = "co";
features[1] = "so2";
features[2] = "pm25";

AirRequest airResult = airServiceImpl
                .getAirQualityByLocal(48.857456, 2.354611, features);

AirRequest airExpected = new AirRequest(data);
airExpected.excludeAirMetric("no2");
airExpected.excludeAirMetric("o3");
airExpected.excludeAirMetric("pm10");

assertEquals(airExpected, airResult);
verify(airRepository, times(1)).putData(anyDouble(), anyDouble(), anyString());
verify(airRepository, times(1)).getData(anyDouble(), anyDouble());
```

I force the airRepository mock to not have any result for those specific coordinates. This means that, when checking for cache, my airService will not find any request in the repository, forcing it to check the Breezometer API.

In this test, I'm also checking if the feature filtering works as expected, by making sure that the result we got from service has had their features filtered from what was sent through the argument.

The final thing to check, and another advantage from using mocks, is checking if all the call functions line up. When the airService doesn't find the request in the cache, it must send out a request to the external API, and write the data in the cache. Therefore both the get function and put function must have been called once and only once.

3.2.3 Integration Tests on API level

Finally integration tests are the ones that require the use of the actual classes, as opposed to mocks. It's generally only done for the controller. This implies the use of some of the Spring annotations.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT,
                classes = AirApplication.class)
@AutoConfigureMockMvc
public class AirControllerIT {
    @Autowired
    private AirServiceImpl airService;

    @Autowired
    private MockMvc mockMvc;
```

And the most important test is to check that by inputting the coordinates and features, the result is a properly structured.

```
mockMvc.perform(get("/api/breeze")
        .param("lat", ""+lat)
        .param("lon", ""+lon)
        .param("features", "co,so2,no2,o3")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.*", hasSize(3)))
        .andExpect(jsonPath("$.listOfPollutants", hasKey("co")))
        .andExpect(jsonPath("$.listOfPollutants", hasKey("so2")))
        .andExpect(jsonPath("$.listOfPollutants", hasKey("o3")))
        .andExpect(jsonPath("$.listOfPollutants", hasKey("no2")))
        .andExpect(jsonPath("$.listOfPollutants", not(hasKey("pm10"))))
        .andExpect(jsonPath("$.listOfPollutants", not(hasKey("pm25"))))
    ;
```

3.3 Functional Testing

Functional testing was done with the Selenium Tool. It allowed for the quick testing of the frontend, by making sure the right values were in the right place when simulating the behaviour of a usual client.

In this project there were three main scenarios that had to be tested:

- When two coordinates are inserted, the right values must be shown, and the total number of requests must be 1, the number of misses must be 1 and the number of hits is also 1;
- When two coordinates are inserted, filtering by different things should not be considered two misses, which means if the only thing that's been changed were these filters, there shouldn't be two requests to the external API
- Clearing the cache will reset number of requests, hits and misses

3.4 Static Code Analysis

As said before, the code analysis was done with the help of SonarQube tool. After every push, I was able to check the quality of my code, this entails the amount of code smells, security vulnerabilities and other bugs that may have been found by the tool.

Code smells are any blocks of code that may not necessarily cause bugs or inconsistencies in the code in an obvious way, but may lead to bigger, unnoticeable, errors, or are generally considered ways of bad programming. Security vulnerabilities are small ways that the code has that may lead attackers to exploit the service in one way or another. Finally, bugs are other errors that were not covered by tests, and may lead to unexpected and wrong results.

Fortunately, the tool was very easy to check which lines were troublesome, and provided a way to easily fix them. So, while it did start as bothersome seeing so many code smells on what seemed to be very inconsequential lines, the tool also properly explained why it was a code smell with a succinct and reasonable reason.

Another thing to note is that SonarQube will also give a report of the current test coverage. While the initial configuration did take some extra work, including installing some dependencies that were not immediately obvious, the general output seemed to working, reaching a test coverage of above 90%. It must be noted though that some of "uncovered" lines seemed to be unjustifiably so, as they had the proper test that should cover them.

3.5 Continuous Integration Pipeline

Since I tried to focus on a TDD process, in the beginning I set up a CI pipeline, connecting my GitHub to a GitLab account. This pipeline has three different stages: build, where I compile the app to check if there's no basic errors; testing, where I run my tests to check if everything's passing; and code analysis, connected to my SonarQube account, to be automatically uploaded and verified with the SonarQube standards.

Having done that right in the beginning helped me catch any errors that I may have missed during development.

4 References

Project Resources
TQS Project Air Repository and Video
Reference Materials
SonarQube Documentation
Spring Documentation