HW1: Mid-term assignment report

José Lucas Sousa [93019], v2021-05-1414

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.

This is a small web app built with Spring Boot framework. It uses 2 external APIs in order to perform all the required tasks. Given an address, one of the APIs converts the address to coordinates and then the second API uses those coordinates to find air pollution data (current or forecast data).

Although what was asked for the forecast was 5 days, my API doesn't support 5 days of forecast and I only noticed it after I had implemented the rest of the app's features. So, I only could do the forecast for the next 3 days.

APIs:

- https://www.breezometer.com/products/air-pollution-api
- https://positionstack.com

1.2 Current limitations

- No historical data
- No API redundance

Since each service relies on a single API, if either one fails there's not a way to deliver the wanted results. Also, I wanted to have a map where I could pick with the mouse a location and get the data for that information, that's not a limitation but more like a feature on a wish list.

1.3 Functional scope and supported interactions

The main goal of this application is to allow users to access air pollution data in a location.

Main scenarios:

- User wants to know the current air quality in a location
- User wants to know the forecast air quality in a location

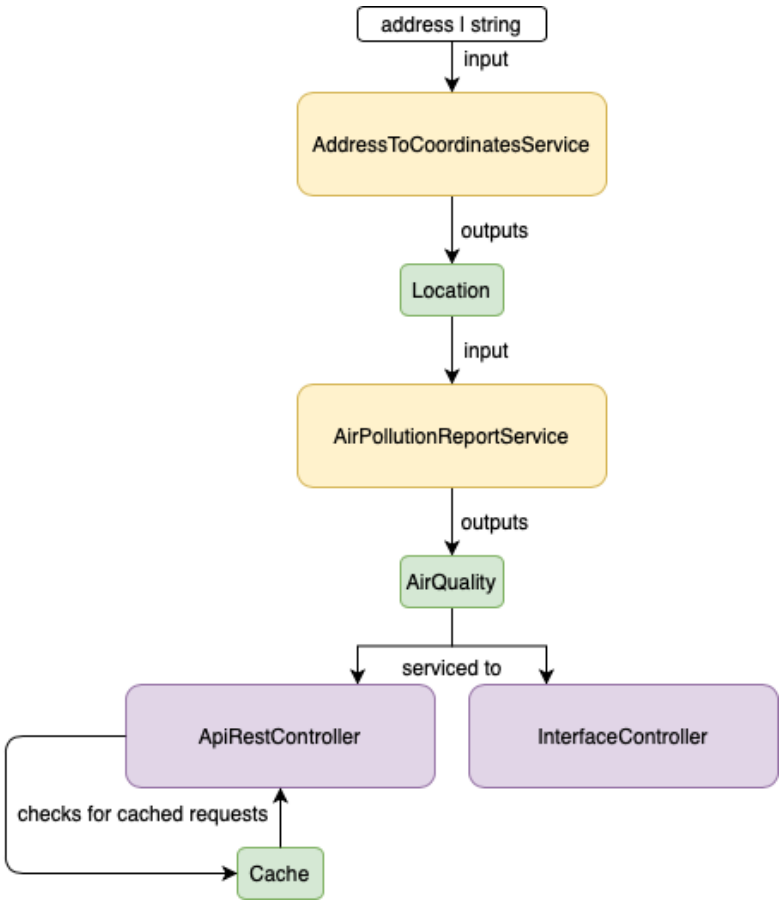
Other interaction I've added to the interface:

- Request cache data

The latter is not something the average user may want to see, but I found interesting having that data available on the interface.

1.4 System architecture

As stated before, this is a Spring Boot Application. Besides that, Thymeleaf was used as the template language used to present data client-side.
I used Lombok to simplify the entities' class code.



The user accesses the website and interacts with the served webpages via the InterfaceController. This allows the user to query the system and access some stats.

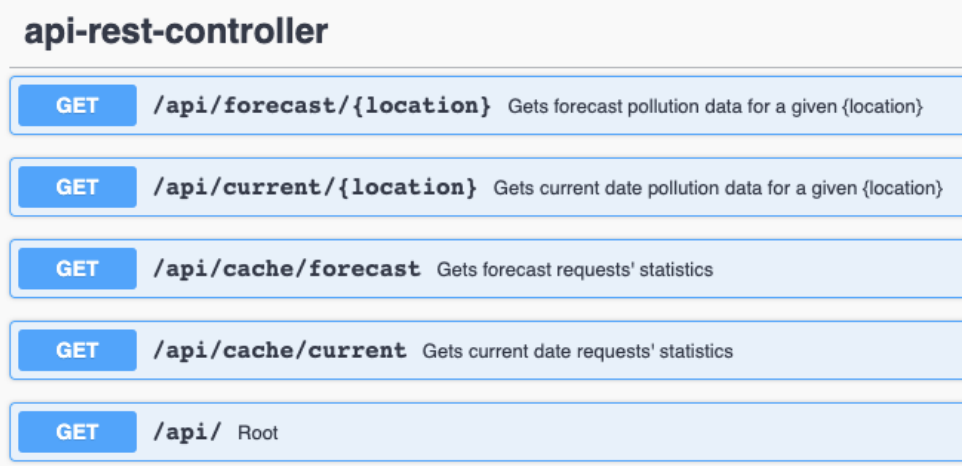
The service layer of the application has two services:

- AddressToCoordinatesService
- AirPollutionReportService

The AddressToCoordinatesService takes an address, converts it to a Location object and the AirPollutionReportService takes that Location and does the necessary processing to add the pollutants (resulting from the API query) to an AirQuality object. This AirQuality contains the Location object and the set of pollutants for that location.

1.5 API for developers

You can see the documentation in here.



2 Quality assurance

2.1 Overall strategy for testing

The overall strategy was to use Junit5, Mockito and Selenium. I didn't take a full-on a TDD approach, but I did implement the forecast feature using that method and found it really useful and ended up regretting not having used TDD since the beginning because it proved to be very efficient, albeit intimidating at first.

2.2 Unit and integration testing

When it comes to **unit tests**, I used them to test the cache and services. Unit tests:

AddressToCoordinatesService:

- whenValidLocation_thenReturnCoordinates()
    - Tests if given a valid location (string, not Location object) it provides the correct coordinates of that location.

AirPollutionReportService:

- whenValidLocation_thenReturnCurrentAirQuality()
    - Tests if given a valid location (this time yes, the Location object) ir provides the correct pollution data of that location.

Cache:

- whenAddHit_thenHitsShouldIncreaseOne()
    - Tests if when it gets a hit, the hit count increases by 1.
- whenAddMiss_thenMissesShouldIncreaseOne()
    - Tests if when it gets a miss, the miss count increases by 1.
- whenAddRequest_thenRequestCountShouldIncreaseOne()
    - Tests if when a request is registered, the request count increases by 1.
- whenCheckIfHasLocation_thenReturnTrueOrFalseAccordingly()
    - Tests the method that checks that a request is cached or no.

When it comes to **integration tests**, I used them to test the controllers. IT tests:

InterfaceController:

- testLocationCurrentData()
- testLocationCurrentDataWithParameter()
    - These two checks that given a request (in the current date) the resulting view is the correct one, only difference being that the second one receives a parameter.
- testLocationForecastData()
- testLocationForecastDataWithParameter()
    - These two are similar to the previous ones but instead of being for the current date, it's for the forecast section.

- testLocationCacheData()
    - This also checks if the cache data request renders the correct view.
- testError()
    - This one performs 2 requests that should be invalid and checks if the page redirects to the correct place.

ApiRestController:

- whenGetCurrentDateOfLocation_thenReturnAirQuality()
    - This test is not calling the rest template mock I don't know why, but the logic is there
- whenGetForecasteOfLocation_thenReturnAirQualityList()
    - This test is not calling the rest template mock I don't know why, but the logic is there
- whenGetCurrentCache_thenReturnAirQuality()
    - Mocks the request's response and compares with hard-coded answer
- whenGetForecastCache_thenReturnAirQuality()
    - Mocks the request's response and compares with hard-coded answer

Functional testing
These tests were made by using the selenium chromedriver from the practical classes.

- checkCurrentDataForValidLocation()
    - Goes to the current date pollution page, searches for "Aveiro".
    - Checks if the query result matches to the supposed one and if it found pollutants
- checkCurrentDataForInvalidLocation()
    - Goes to the current date pollution page, searches an invalid location
    - Checks if the query result matches to the supposed one
- checkForecastDataForValidLocation()
    - Goes to the current date pollution page, searches for "Aveiro".
    - Cheks if the query result matches to the supposed one and checks if the pollutions for the forecast days are there.
- checkForecastDataForInvalidLocation()
    - Goes to the current date pollution page, searches for an invalid location.
    - Checks if the query result matches to the supposed one
- checkCache()
    - Goest to the cache stats page
    - Checks if all the data is there

2.3 Static code analysis

I used Sonar for static code analysis. Not Sonar Qube like what was used in lab classes, but Sonar Cloud which was integrated with my CI pipeline. When it builds the code on github, if all checks pass, it updates Sonar Cloud with the newly updated stats of my project.

I find this very useful, because it allows us to review and improve our code. One of the things that surprised me the most was that we shouldn't print actual data (in production) – for example, a location, or coordinates, etc..

2.4 Continuous integration pipeline [optional]

I implemented a CI pipeline in github actions by following an online tutorial. It builds my code, runs the tests, updates Sonar Cloud and publishes the new version in DockerHub.

I had a few hiccups in running my functional tests in the github actions server. Apparently chrome is not pre-installed and I tried to install itbefore the build but I couldn't make it work.

3 References & resources

Project resources

- Video demo: https://www.youtube.com/watch?v=gHCnOYNMTIk
- Github: https://github.com/l-sousa/air-quality-tqss
- Ready to use application: docker run -p 8080:8080 dbotto/air-quality-tqs:latest
- QA dashboard: https://sonarcloud.io/dashboard?id=l-sousa_air-quality-tqs

Reference materials

Breezometer API
PositionStack API

4 Side Notes

Regarding the Breezometer API, it has a trial period of 2 weeks which is what I used in this app. So, in case that the key is expired when this work is being evaluated, I'd be appreciated if either I'd be given the opportunity to provide a new key or access this website and generate a key for yourself and replace the BM_API_KEY variable on service/AirPollutionReportService.java with your key.