HW1: Mid-term assignment report

Daniel Jorge Bernardo Ferreira [102885], v2023-04-09

1 Introduction

1.1 Overview of the work

This report is about the midterm individual project created by Daniel Ferreira for the TQS class. The project is a software application that allows users to check current and future air quality data for a specific location.

The application consists of a back-end/API developed using Spring Boot and a front-end built with React.

The purpose of this report is to provide an overview of the project, including its key features and the quality assurance strategy that was adopted. The application has been fully tested with both unit and integration tests.

1.2 Current limitations

The weather API used in the project only provides air quality data forecasts for the pro mode (or first-month free trial). This means that my application will only work until the free trial ends unless I upgrade to a paid plan.

There is currently no REST API documentation. I tried to use Swagger, but I found out that SpringFox is a dead project, and Swagger2 does not work in Spring Boot version 3.

2 Product specification

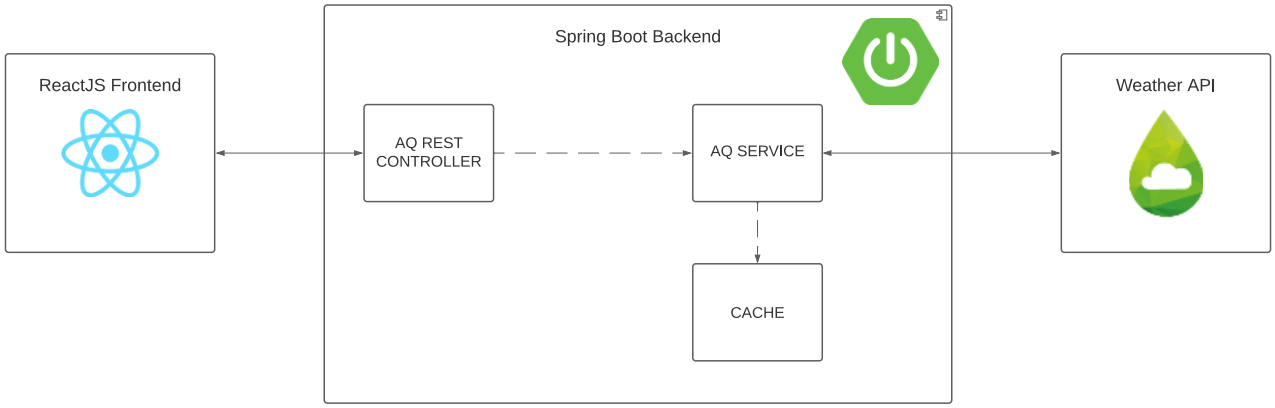
2.1 Functional scope and supported interactions

The main actors for this application are developers who want to integrate air quality data into their applications (or just me).

The main usage scenario for this application would be a developer who wants to retrieve air quality data for a specific location. The developer would make a request to my API with the location information (and additional parameters), and my API would retrieve the relevant air quality data from the weather API.

The application simplifies the process of retrieving air quality data from the weather API. It pre-processes the data and returns it to the developer in a format that can be easily integrated into their application.

2.2 System architecture



- The front-end was built using React.
- The back-end was built using Spring Boot.
- The project uses the Weather API com for obtaining environmental data.

2.3 API for developers

Developers can obtain two main types of services/resources from this project: environmental data and cache usage statistics.

Environmental data can be retrieved through the following endpoints:

- GET /current: This endpoint retrieves the current air quality data for a specified location. The location is specified as a query parameter location.
    - The response includes an AirQualityData object with various fields, including the air quality index for different pollutants such as CO, NO2, O3, SO2, PM10, and PM2.5.

- GET /forecast: This endpoint retrieves the forecasted air quality data for a specified location. The location is specified as a query parameter location, and the timeframe for the forecast can be specified using the optional query parameters date, current, days, and hours.
    - The date parameter takes priority over the days parameter and can be used to specify the exact date for which the forecast is desired.
    - The current parameter (default=false), when set to true, includes the current air quality data in the response.
    - The days parameter (default=1) specifies the number of days for which the forecast is required.
    - The hours parameter (default=false), when set to true, includes the air quality data for each hour of the forecast.
    - The response includes an AirQualityData object with various fields, including the air quality index for different pollutants. The format of the response may differ based on each specific case.

Cache usage statistics can be retrieved through the following endpoint:

- GET /cache: This endpoint retrieves statistics on the usage of the cache.
    - The response includes a CacheAnalyticsData object with fields such as date, request_count, cache_hits, cache_misses, and expired_count. These fields provide information on the number of requests made to the cache, the number of cache hits and misses, and the number of expired cache entries.

In case of any errors during the request, the exception handler component returns an error with the format {"error": errorMessage} and the appropriate HTTP status code.

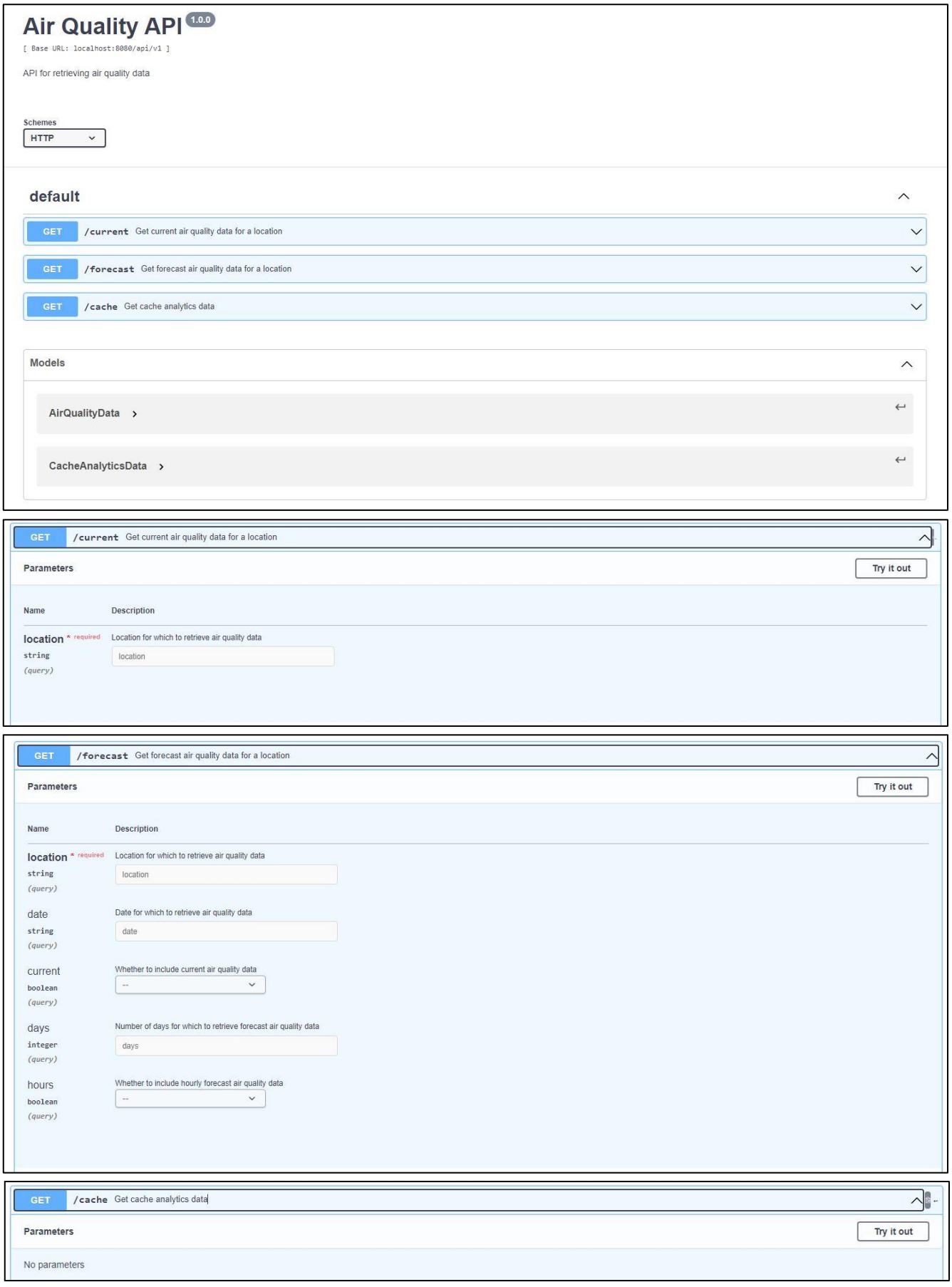


Fig. 1-4 – Images generated using Swagger web editor (Swagger is not compatible with Spring boot version 3 as of right now).

3 Quality assurance

3.1 Overall strategy for testing

I didn't use TDD because I felt that it would slow down my development process and I preferred to focus on writing tests after I had implemented the functionality. This is because I didn't know from the start what my app would look like, especially due to dependencies like the external weather API.

I used Cucumber in one integration test, but overall, I wouldn't say I used BDD. I chose to go at my own pace and decided myself what was the best approach for each case.

I used Rest-assured to test my API. I personally found it very useful ever since I first learned it in class, so I decided to use it again.

I used AssertJ as the default set of assertions, because it is the Spring Boot standard, and makes the tests easier to read. I also used Hamcrest matchers with Rest-assured.

3.2 Unit and integration testing

I used Lab 3.3 as the basis for my tests.

Purpose/scope	**Strategy**
A/ Verify cache behaviour [*A_AirQualityCache_UnitTest*]	Unit test mocking the data to cache, as the format of the data should be irrelevant. Used Awaitility DSL to set asynchronous expectations and test the expiration of data.
B/ Verify the business logic associated with the services implementation. [*B_AirQualityService_UnitTest*]	Unit test mocking the RestTemplate client object (that performs HTTP requests to the external API). Used Mockito to control the test and to set expectations and verifications. I used static json files with previous responses that I recorded to serve as the mocked response. I used, in one test, a static text file that helped me create the test data object that matches the response.
C/ Verify the boundary components (controllers); just the controller behavior. [*C_AirQualityController_WithMockServiceTest*]	Test in a simplified and light environment, simulating the behavior of an application server using @WebMvcTest mode. Used a reference to the server context with @MockMvc. To make the test more localized to the controller I mocked the dependencies on the service (@MockBean). Tested the whole response body, not just part of the responses. Used RestAssuredMockMvc (Spring MVC module's equivalent of RestAssured) with Hamcrest to test each endpoint.

D/ Verify the boundary components (controllers). Load the full Spring Boot application. No API client involved. [*D_AirQualityRestController_IT*]	Tested in the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal Spring Boot context. Used the entry point for server-side Spring MVC test support (MockMvc). I tested integral parts of the responses, as well as some random ones. Due to their dynamic nature and large number of fields, I couldn't test the entire response body as objects. Used RestAssuredMockMvc (Spring MVC module's equivalent of RestAssured) with Hamcrest to test each endpoint.
E/ Verify the boundary components (controllers). Load the full application. Test the REST API with explicit HTTP client. [*E_AirQualityRestControllerTemplate_IT*]	Tested in the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal Spring Boot context. Use a REST client to create realistic requests (RestAssured).

3.3 Functional testing

Purpose/scope	**Strategy**
F/ Verify behavior of the whole application. [*F_FunctionalWeb_ITUnitTest*]	Functional test using Selenium Web driver. Uses the Chrome driver to interact with the application and assertions to verify the data displayed on the UI is consistent with the expected results. Used POM (Page Object Model) to organize the test in a clean and logical way.
G/ Verify behavior of the whole application with cucumber. [*G_FunctionalWebCucumber_IT*]	Same as before but uses cucumber to write the test in natural language. It makes it easier for people with no knowledge of java or coding to understand the content of the test.

I initially tried using HTMLUnit (headless browser) to test my React web app but found that it struggled with the dynamic client-side rendering. This caused some timing issues and made it difficult to interact with certain elements. I switched to Chrome, and it worked as expected.

3.4 Code quality analysis

I used SonarQube through Docker for static code analysis. Initially, the tool detected 21 code smells, most of which were classified as major. However, I addressed each one, and after several rounds of refactoring and retesting, I was able to eliminate them all.

One code smell that was particularly challenging to fix was the fetchForecastAirQualityData method in the service, that had a cognitive complexity of 19. The code was confusing, even for me as the developer, and it required a significant amount of effort to refactor it. This experience taught me the importance of writing modular and clean code that is easy to understand and maintain.

In addition, I learned about Awaitility, a library that allows for asynchronous testing. Previously, I was using Thread.sleep, but I learned that it is not always reliable and should not be used for testing.

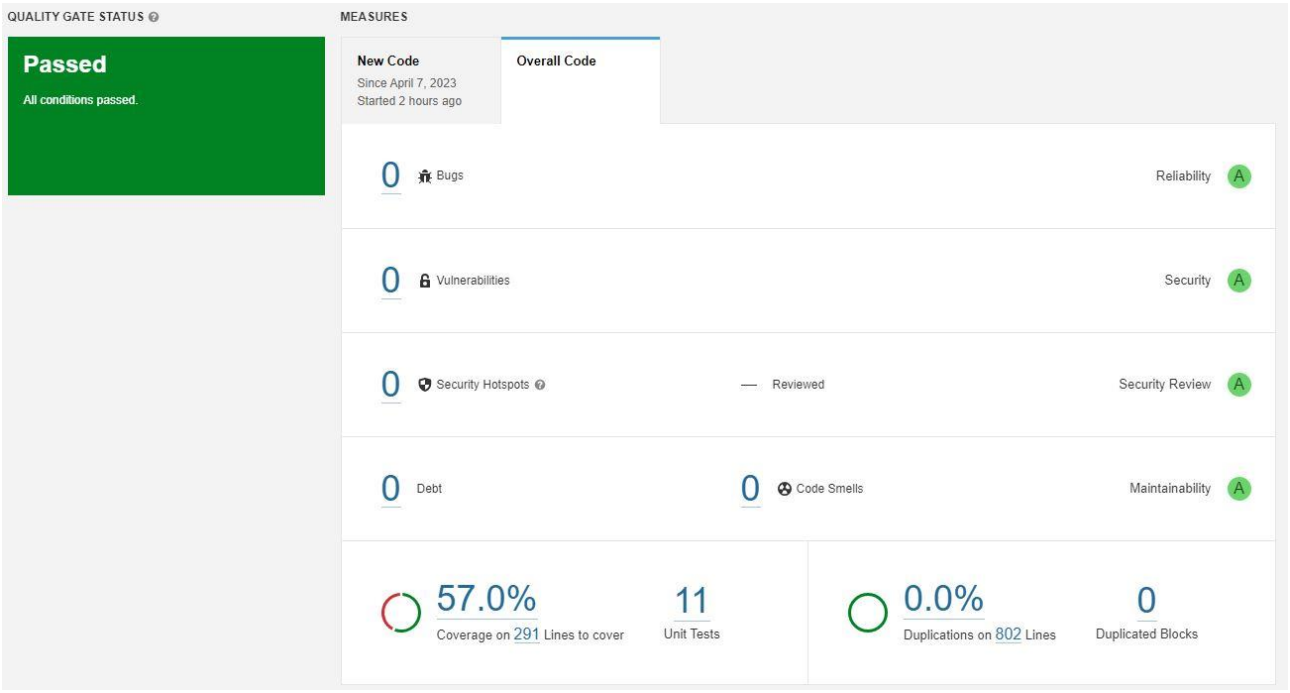


Fig. 5 – SonarQube dashboard final results

3.5 Continuous integration pipeline [optional]

I implemented a CI pipeline using GitHub Actions. The setup involved defining a job named "build" that runs on the latest version of Ubuntu and sets up JDK 18. The pipeline checks out the code from the repository, builds the project using Maven, and runs the unit tests. The pipeline is triggered by push or pull request events and ensures that the code is automatically built and tested every time a change is made to the repository.

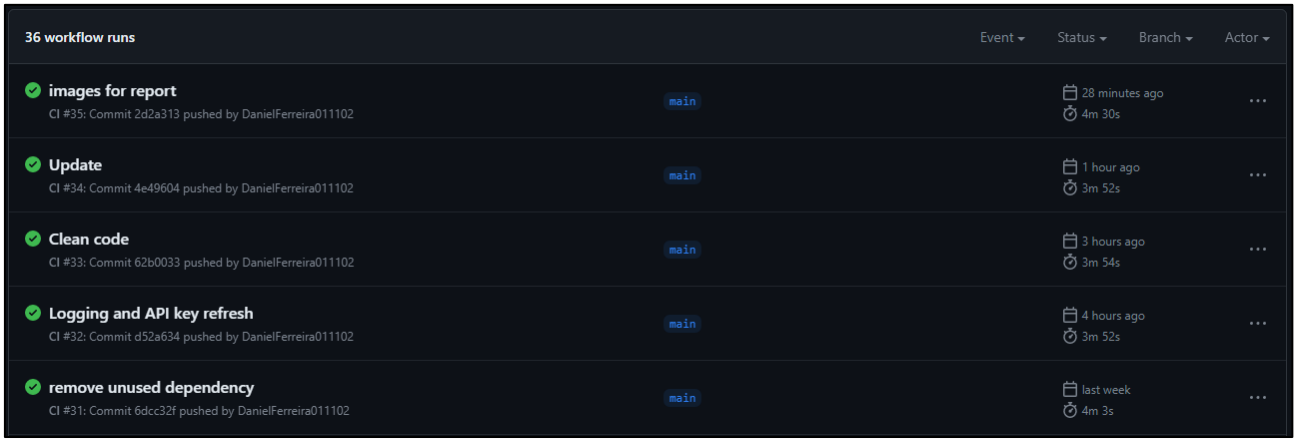


Fig. 6 – Preview of github actions workflows in my repository.

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/DanielFerreira011102/TQS_102885/tree/main/HW1
Video demo	https://www.youtube.com/watch?v=SnTnpJ2fMIs
CI/CD pipeline	https://github.com/DanielFerreira011102/TQS_102885/blob/main/.github/workflows

Reference materials

- Baeldung
- Weather API
- Stack Overflow
- TQS repository (Labs)
- Maven Github actions