HW1: Mid-term assignment report

Daniel Martins Ferreira [102442], 11-04-2023

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 report presents the midterm individual project "Air Quality Information" required for TQS, which is a web application that allows users to view current and past air quality metrics by entering a city name. The application is built using the Spring Boot framework with Maven for backend services and React for the frontend user interface.

1.2 Current limitations

- Limited to specific air quality metrics, as provided by the OpenWeather API.
- The application relies on external APIs (OpenWeather and MapQuest) for data, which may impose restrictions on usage and introduce potential latency issues.
- The application currently supports only city names for location input and does not provide alternative ways of specifying a location, such as postal codes or addresses.
- The frontend interface is not fully tested with functional tests for all scenarios.

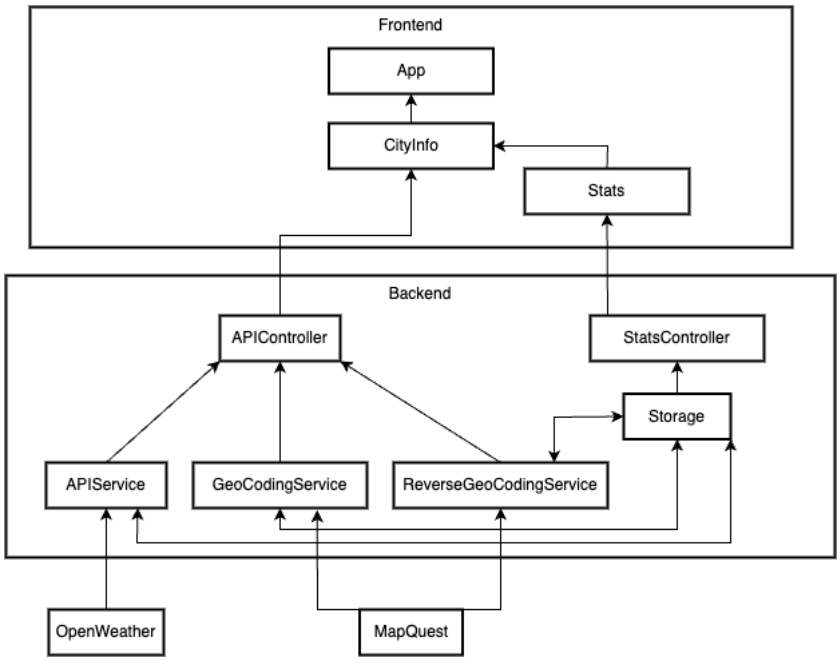
2 Product specification

2.1 Functional scope and supported interactions

The Air Quality Information application supports the following interactions:

1. Users can enter a city name to retrieve current air quality metrics.
2. Users can view historical air quality metrics for the entered city.
3. The application provides the coordinates for the given city.
4. Users can see where the data from a given request is being pulled from (API/Cache)

2.2 System architecture



The software architecture of the Air Quality Monitor application is organized into two main components: the backend, built using Spring Boot and Maven, and the frontend, developed using the React framework. The backend also incorporates an in-house caching mechanism to improve performance and reduce the load on external APIs.

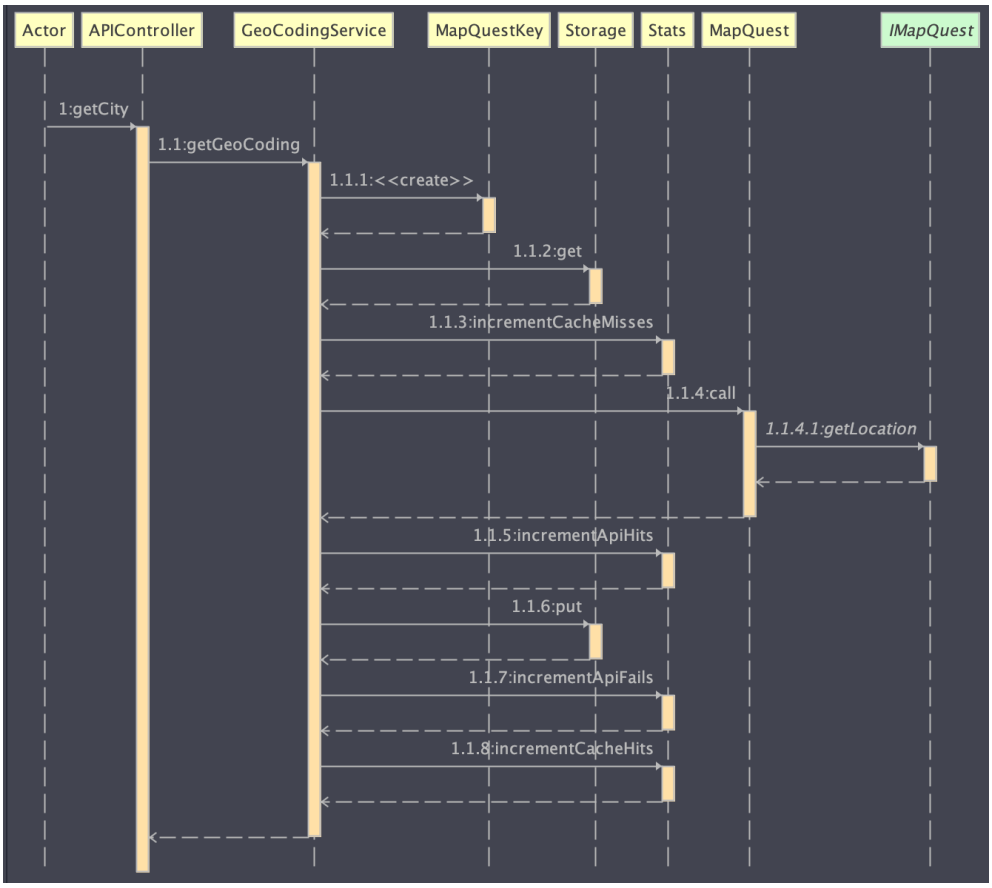
1. Backend: The backend, built using Spring Boot and Maven, is responsible for managing API requests and processing data from external APIs (OpenWeather and MapQuest).
   It exposes RESTful endpoints that the frontend consumes to display air quality information.
   To enhance the application's response time and reduce the number of calls to external APIs, the backend includes an in-house caching mechanism. This cache stores recently accessed air quality data and city information, allowing the application to serve cached data for repeated requests, instead of making additional calls to external APIs.
2. Frontend: The frontend, developed using the React framework, provides a user-friendly interface for interacting with the application.
   Users can input a city name, and the frontend sends requests to the backend using the a fetch to retrieve the corresponding air quality data. The frontend handles the presentation

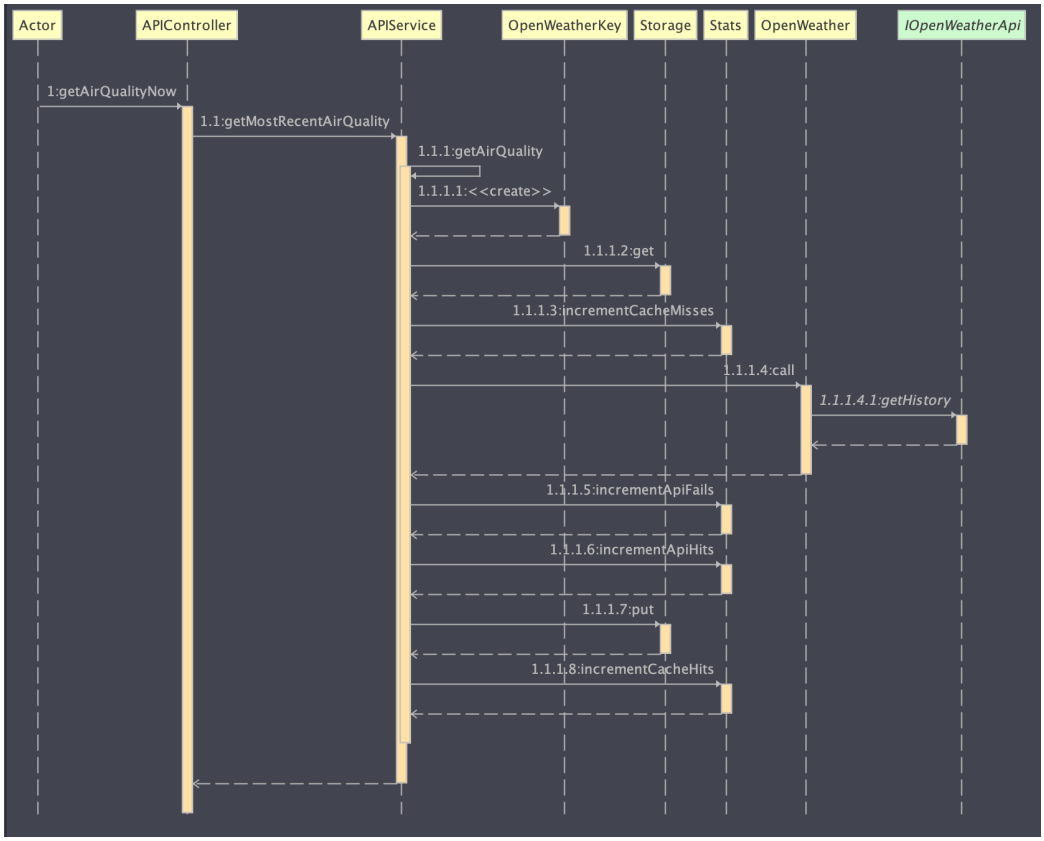
of the data fetched from the backend by feeding the MapQuest API with the given city string, retrieving the corresponding coordinates, feeding this to the OpenWeather API and displaying the desired metrics, ensuring a seamless and visually appealing user experience. To achieve a modern and responsive design, the application utilizes the TailwindCSS framework for styling.

Specific technologies/frameworks used in the application:

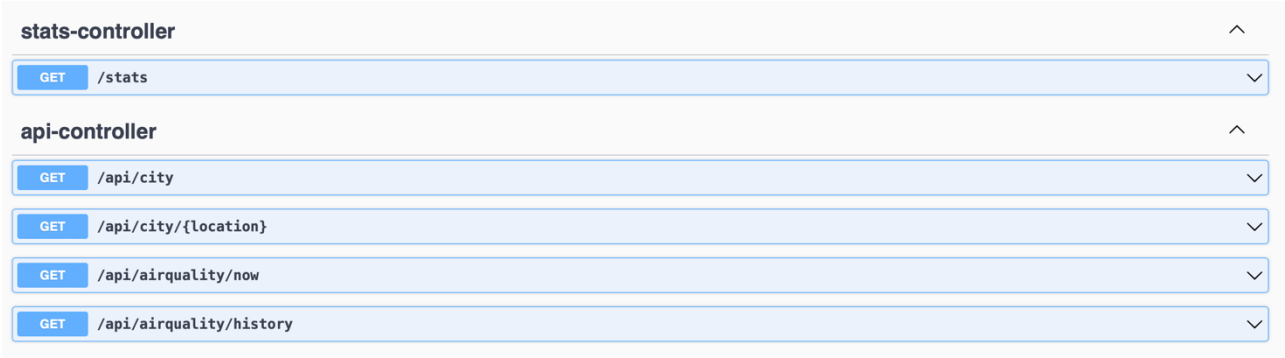
Backend: Spring Boot, Maven, SLF4J for logging, Spring Web, Jackson for JSON processing, and a custom in-house caching implementation based on a HashMap data
Frontend: React and TailwindCSS for styling.

Below, two sequence diagrams for the extraction of data from the two external APIs previously mentioned:





2.3 API for developers



In this project, there are two controllers: **APIController** and **StatsController**. Each controller has different endpoints that serve specific purposes. Here follows the explanation of each endpoint, how to use it, the respective payload and content.

1. APIController:
   a. **/api/airquality/now** (GET)
      Explanation: This endpoint returns the current air quality for a given set of latitude and longitude coordinates.
      How to use: Make a GET request to **/api/airquality/now** with **lat** and **lon** as request parameters.
      Payload: Provide latitude (lat) and longitude (lon) as request parameters.
      Content:


```
{
    "aqi": 0,
    "co": 0,
    "no": 0,
    "no2": 0,
    "o3": 0,
    "so2": 0,
    "pm2_5": 0,
    "pm10": 0,
    "nh3": 0,
    "dt": 0,
    "logger": {
        "name": "string",
        "infoEnabled": true,
        "debugEnabled": true,
        "errorEnabled": true,
        "traceEnabled": true,
        "warnEnabled": true
    }
}
```


   b. **/api/airquality/history** (GET)
      Explanation: This endpoint returns historical air quality data for a given set of latitude and longitude coordinates.
      How to use: Make a GET request to **/api/airquality/history** with **lat** and **lon** as request parameters.
      Payload: Provide latitude (lat) and longitude (lon) as request parameters.
      Content:

```
{
        "latitude": 0,
        "longitude": 0,
        "logger": {
            "name": "string",
            "infoEnabled": true,
            "debugEnabled": true,
            "errorEnabled": true,
            "traceEnabled": true,
            "warnEnabled": true
        },
        "list": [
            {
                "aqi": 0,
                "co": 0,
                "no": 0,
                "no2": 0,
                "o3": 0,
                "so2": 0,
                "pm2_5": 0,
                "pm10": 0,
                "nh3": 0,
                "dt": 0,
                "logger": {
                    "name": "string",
                    "infoEnabled": true,
                    "debugEnabled": true,
                    "errorEnabled": true,
                    "traceEnabled": true,
                    "warnEnabled": true
                }
            }
        ]
    }
```


c. **/api/city** (GET)

Explanation: This endpoint returns the city information for a given set of latitude and longitude coordinates using the MapQuest API.
How to use: Make a GET request to **/api/city** with **lat** and **lon** as request parameters.
Payload: Provide latitude (lat) and longitude (lon) as request parameters.
Content:

```
{
    "state": "string",
    "city": "string",
    "road": "string",
    "zip": "string",
    "logger": {
      "name": "string",
      "infoEnabled": true,
      "debugEnabled": true,
      "errorEnabled": true,
      "traceEnabled": true,
      "warnEnabled": true
    }
  }
```

d. **/api/city/{location}** (GET)

Explanation: This endpoint returns the city information for a given location name using the Reverse MapQuest API.
How to use: Make a GET request to **/api/city/{location}** where **{location}** is the name of the location you want to find.
Payload: Replace **{location}** with the desired location name in the endpoint path.
Content:

```
{
    "lat": 0,
    "lon": 0
}
```

2. StatsController:

a. **/stats** (GET)

Explanation: This endpoint returns the statistics collected by the application.
How to use: Make a GET request to **/stats**.
Payload: No payload is required for this endpoint.
Content:

```
{
    "cacheMisses": 0,
    "cacheHits": 0,
    "apiFails": 0,
    "apiHits": 0
}
```

3 Quality assurance

3.1 Overall strategy for testing

The testing strategy adopted for the Air Quality Monitor application was to combine various testing tools and methodologies. This approach aimed to ensure thorough test coverage and enable the detection of any potential issues or inconsistencies in the application. The testing process included unit and integration testing with Mockito and MockMvc, functional testing with Cucumber and Selenium WebDriver, and code quality analysis with SonarQube and IntelliJ analytics.

3.2 Unit and integration testing

Unit and integration tests were implemented to test individual components and their interactions within the backend. Mockito was employed for testing services and the cache, ensuring that each service functioned as expected and that the caching mechanism worked correctly. MockMvc was used for controller unit testing, allowing the validation of request handling and response generation in the backend. The implementation strategy involved writing test cases for various scenarios, including expected behavior, edge cases, and potential error handling.

3.3 Functional testing

Functional testing focused on the user-facing aspects of the application. Cucumber was used for implementing user stories and scenarios, while Selenium WebDriver was employed for interacting with the frontend and simulating user interactions. Functional tests aimed to ensure that the frontend accurately displayed air quality data and responded correctly to user input. Some of the test cases included: Entering a valid city name and retrieving air quality data.

3.4 Code quality analysis

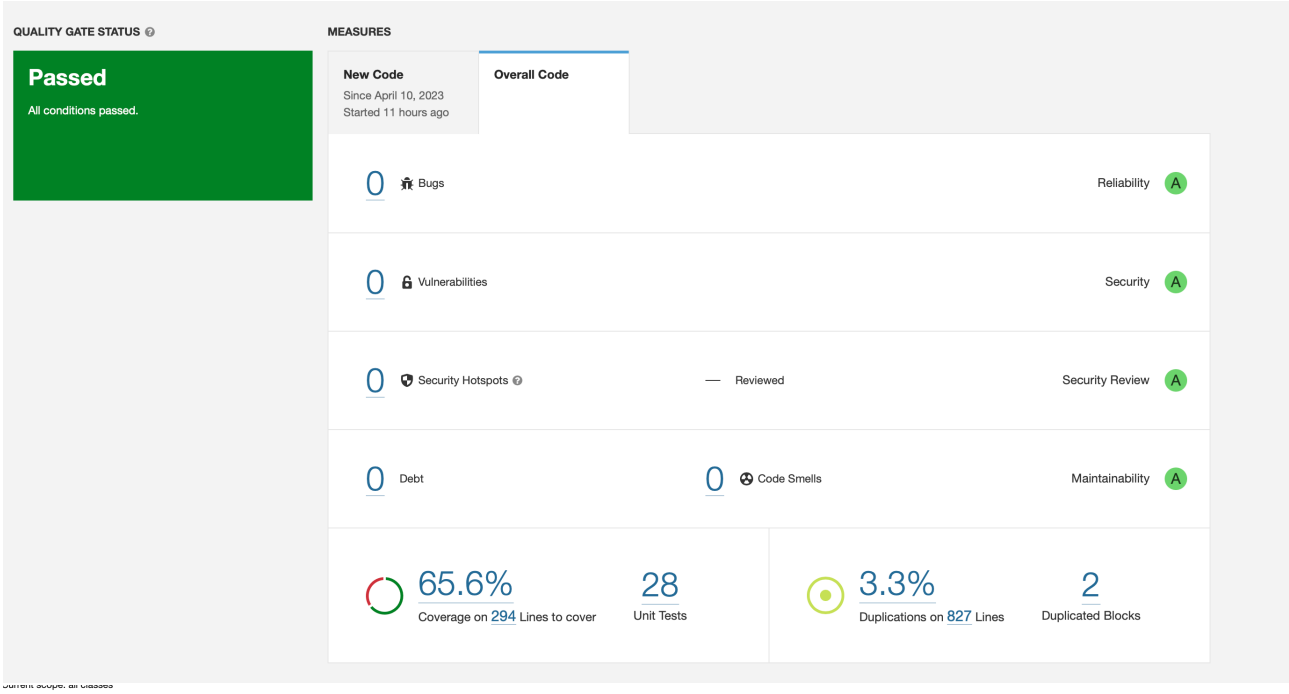
SonarQube and IntelliJ analytics were used for static code analysis to identify potential code issues, such as code smells, bugs, or vulnerabilities. These tools helped maintain high code quality and identify areas for improvement. Insights from these tools were used to refactor the codebase, ensuring that it adhered to best practices and remained maintainable.

One interesting case was when SonarQube advised to not use the specific HashMap type, and to instead use the more generic Map type in this method:

```
Daniel
@JsonProperty("results")
public void parseResults(List<Object> results) {
    Map<String, Object> result = (HashMap<String, Object>) results.get(0);
    Map<String, Object> locations = (HashMap<String, Object>) ((List<Object>) result.get("locations")).get(0);
    Map<String, Double> latlng = (HashMap<String, Double>) locations.get("latLng");
    lat = latlng.get("lat");
    lon = latlng.get("lng");
}
```

This allows for a more flexible use of the method and contributes to code reusability.

Below, the dashboards for both static code analysis tools:



Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (16/16)	90% (72/80)	89.4% (245/274)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
tqs.homework	100% (1/1)	50% (1/2)	50% (1/2)
tqs.homework.api.mapquest	100% (4/4)	84.2% (16/19)	82.1% (46/56)
tqs.homework.api.openweather	100% (4/4)	87.5% (28/32)	85.1% (57/67)
tqs.homework.cache	100% (1/1)	100% (4/4)	100% (15/15)
tqs.homework.controller	100% (2/2)	100% (7/7)	76.5% (26/34)
tqs.homework.services	100% (3/3)	100% (7/7)	100% (70/70)
tqs.homework.statistics	100% (1/1)	100% (9/9)	100% (30/30)

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/dferrero17/TQS_102442

Reference materials

1. OpenWeather. (n.d.). OpenWeatherMap API. Retrieved from https://openweathermap.org/api
2. MapQuest. (n.d.). MapQuest API. Retrieved from https://developer.mapquest.com/
3. Pivotal Software. (n.d.). Spring Boot. Retrieved from https://spring.io/projects/spring-boot
4. Apache Maven Project. (n.d.). Apache Maven. Retrieved from https://maven.apache.org/
5. React. (n.d.). React – A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/

6. Tailwind Labs. (n.d.). Tailwind CSS - A utility-first CSS framework for rapid UI development. Retrieved from https://tailwindcss.com/
7. Mockito. (n.d.). Mockito Framework. Retrieved from https://site.mockito.org/
8. Spring Framework. (n.d.). Spring MockMvc. Retrieved from https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework
9. Cucumber. (n.d.). Cucumber – Behaviour-Driven Development. Retrieved from https://cucumber.io/
10. Selenium. (n.d.). Selenium WebDriver. Retrieved from https://www.selenium.dev/documentation/en/webdriver/
11. SonarQube. (n.d.). SonarQube – Continuous Code Quality. Retrieved from https://www.sonarqube.org/
12. JetBrains. (n.d.). IntelliJ IDEA – The Java IDE for Professional Developers. Retrieved from https://www.jetbrains.com/idea/