HW1: Mid-term assignment report

João Miguel Almeida Matos [103341], v2023-04-09

1 Introduction

1.1 Overview of the work

This report outlines the main stages involved in creating a web application for measuring air quality metrics. Aligning with the objectives of this course, a Test Driven Development approach was used, placing emphasis on writing the test cases before implementing any functionality.

1.2 Current limitations

Due to all data being dependent on external sources (public APIs), some data may be missing or inaccurate. Also, some cities work only using their international names.

Some utility classes don't have unit tests, lowering the final code coverage to around 75%.

2 Product specification

2.1 Functional scope and supported interactions

The web client is very simple and concise. It has an input field, where the user can write the name of the desired location; they can then choose between data for the current day (Today), future data (Forecast), or past data (History).

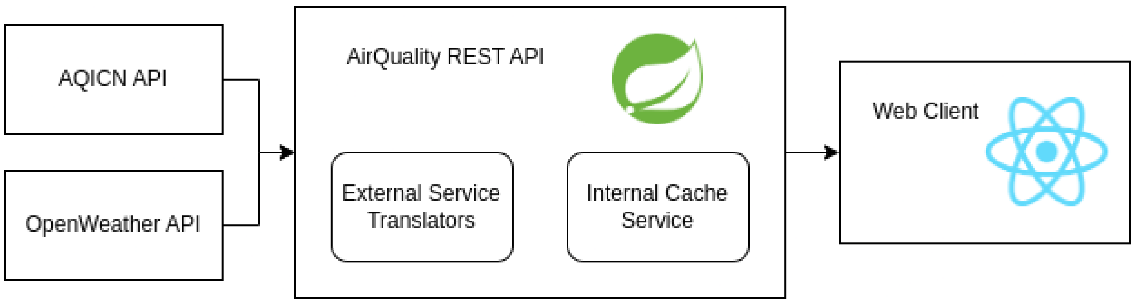
If a valid location is provided (a country, or an existing city, for example), a table with air quality metrics will show up, providing the desired data.

2.2 System architecture

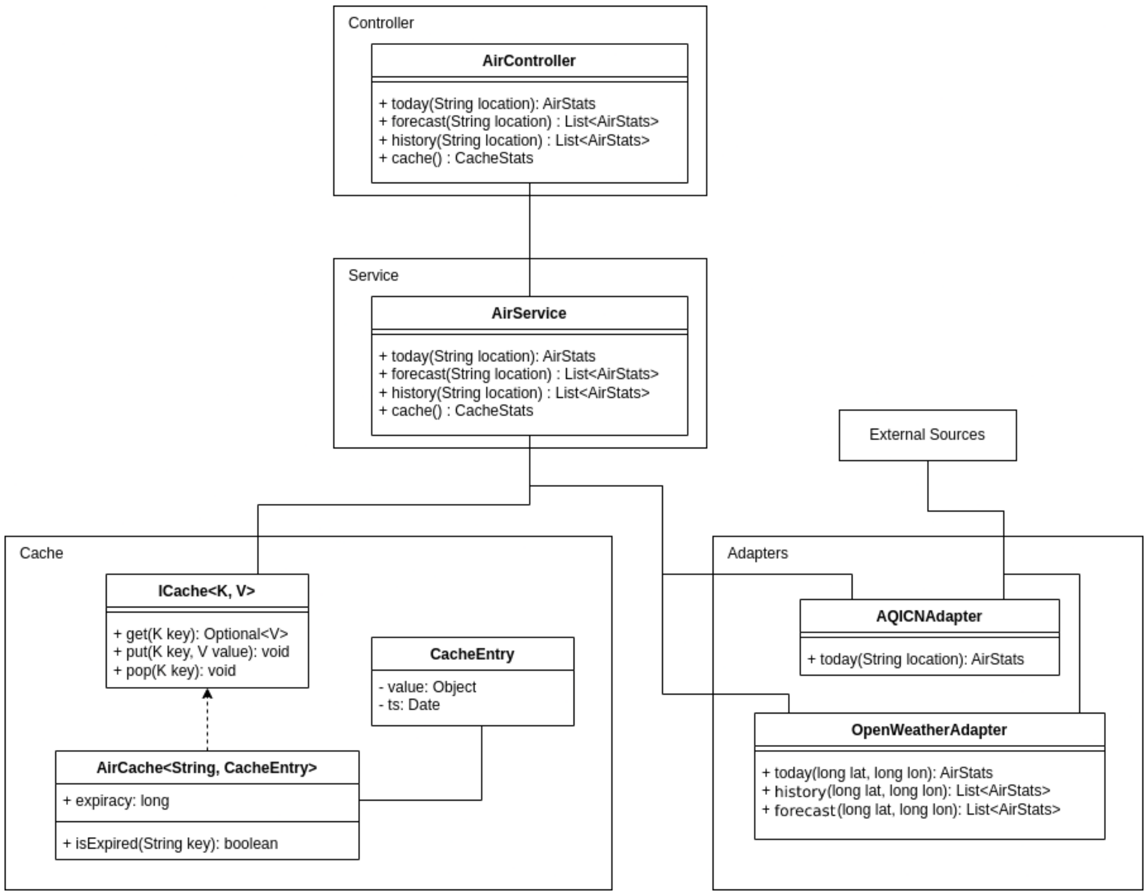
The application makes use of 2 different APIs to provide data to its users. The first one is called AQICN, used to query location by name and to get data for the current day. The second one is called OpenWeather, and is used to complement any missing data AQICN might return, and to fetch forecast and history data.

The frontend/web client was built using ReactJS, TailwindCSS and DaisyUI.

A docker-compose file is provided in the root folder of the repository, that exposes the API in port 8080 and the web client in port 5173.

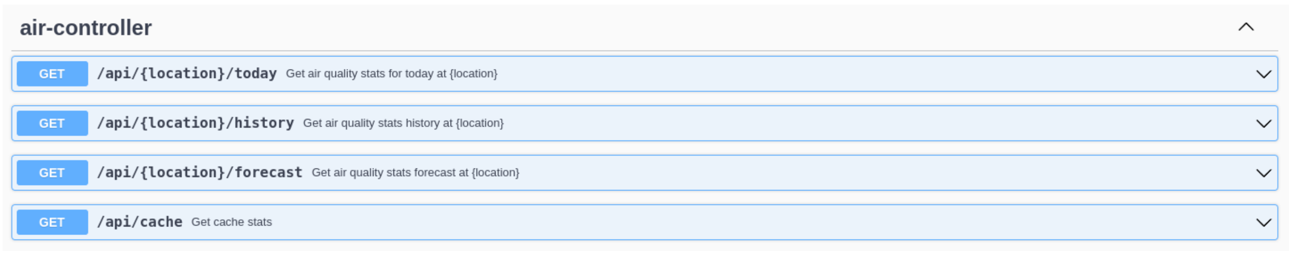


As for the architecture of the REST API itself, it follows a standard Controller - Service - Datasource approach. The controller picks up the request and calls a Service method that handles the logic of cache managing and data fetching. Two adapters were implemented, one for each external API, that translates their responses into objects from my application.


Controller
AirController
+ today(String location): AirStats
+ forecast(String location) : List<AirStats>
+ history(String location) : List<AirStats>
+ cache() : CacheStats
Service
AirService
+ today(String location): AirStats
+ forecast(String location) : List<AirStats>
+ history(String location) : List<AirStats>
+ cache() : CacheStats
External Sources
Cache
ICache<K, V>
+ get(K key): Optional<V>
+ put(K key, V value): void
+ pop(K key): void
CacheEntry
- value: Object
- ts: Date
AirCache<String, CacheEntry>
+ expiracy: long
+ isExpired(String key): boolean
Adapters
AQICNAdapter
+ today(String location): AirStats
OpenWeatherAdapter
+ today(long lat, long lon): AirStats
+ history(long lat, long lon): List<AirStats>
+ forecast(long lat, long lon): List<AirStats>

2.3 API for developers

The REST API documentation was generated using Swagger for Maven. It can be accessed at /swagger-ui.html resource of the API (localhost:8080/swagger-ui.html).


air-controller

GET `/api/{location}/today` Get air quality stats for today at {location}

GET `/api/{location}/history` Get air quality stats history at {location}

GET `/api/{location}/forecast` Get air quality stats forecast at {location}

GET `/api/cache` Get cache stats

3 Quality assurance

3.1 Overall strategy for testing

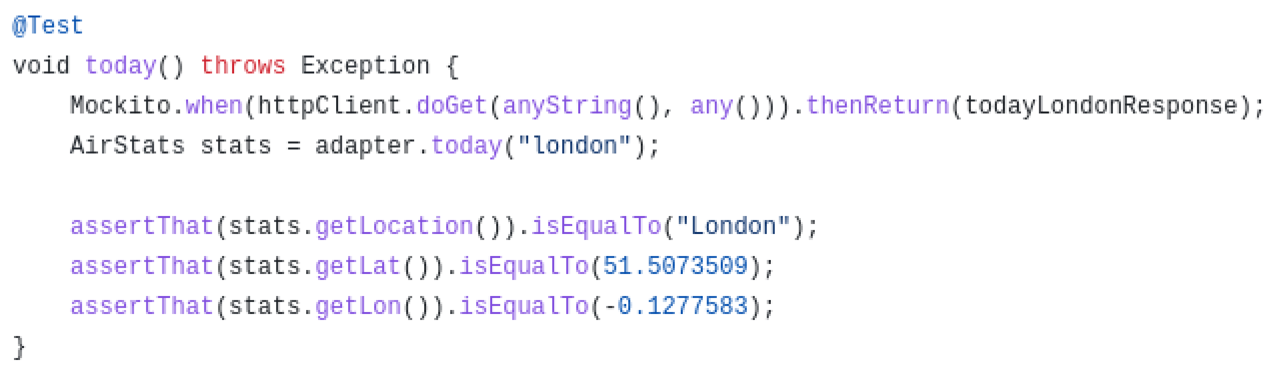
As mentioned previously, a TDD approach was used to test the application. I started by building the skeleton of the classes I was implementing, returning null values and making sure the application compiled. Then, the tests were written, and only then the classes were actually implemented. This allowed me to test my application while developing it, in order to find errors and bugs much earlier in development.

The API components were tested/developed in order, starting with the Controller, then the Service, the Adapters, and finally the Cache.

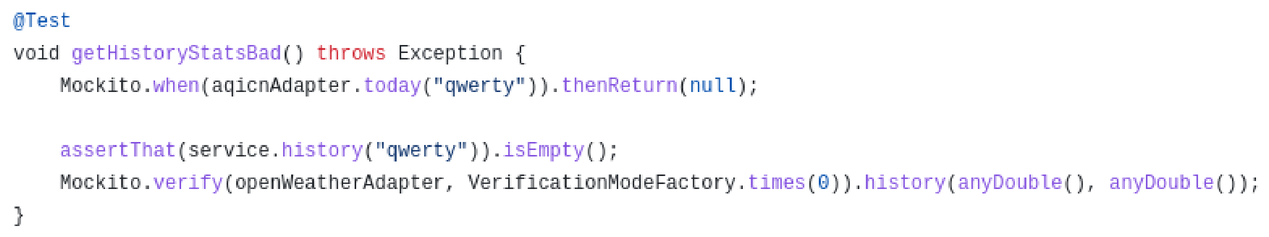
3.2 Unit and integration testing

Each API component had their own unit tests written before being implemented. Most of these tests use Mockito to mock dependencies and isolate the test subject, like the service tests or the adapter tests. MockMvc was also used for the controller tests, in order to mock the requests being made to the controller.

Here are some examples of dependency or request mocking using the specified libraries. The first one shows a AQICNAdapter test, where we mock the response from the external API and test the AirStats object building functionality; the second one tests the AirService class, by mocking a response to a bad location and expecting a 404 code when an invalid location is provided.



```
@Test
void today() throws Exception {
    Mockito.when(httpClient.doGet(anyString(), any())).thenReturn(todayLondonResponse);
    AirStats stats = adapter.today("london");

    assertThat(stats.getLocation()).isEqualTo("London");
    assertThat(stats.getLat()).isEqualTo(51.5073509);
    assertThat(stats.getLon()).isEqualTo(-0.1277583);
}
```



```
@Test
void getHistoryStatsBad() throws Exception {
    Mockito.when(aqicnAdapter.today("qwerty")).thenReturn(null);

    assertThat(service.history("qwerty")).isEmpty();
    Mockito.verify(openWeatherAdapter, VerificationModeFactory.times(0)).history(anyDouble(), anyDouble());
}
```

After developing the components, we tested their interactions with each other using integration testing, making sure the application behaves correctly as a whole. For this, no mocking was used, and instead real calls were made to the respective components, expecting them to reply correctly. For this purpose, the REST Assured library was used, to make actual requests to the controller.

The following snippet demonstrates one of the integration tests run - asking for the metrics for today in London, and expecting a valid response with London as the location.

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class AirQualityIT {

    @LocalServerPort
    private Integer port;

    @Test
    void whenGetAirQualityByCity_thenStatus200() {
        String uri = "http://localhost:" + port + "/api/London/today";
        given()
        .when()
            .get(uri)
        .then()
            .statusCode(200)
            .body("location", equalTo("London"));
    }
```

3.3 Functional testing

The web client testing was written using Selenium, and the Page Object pattern. The tests written give the app a location as input, through the text input present, and expect the app to return some data, which is then verified.

Here are some illustrative code snippets from the functional testing class:

```java
@FindBy(css = ".input")
private WebElement input;

@FindBy(id = "today")
private WebElement today;

@FindBy(id = "forecast")
private WebElement forecast;

@FindBy(id = "history")
private WebElement history;

@FindBy(className = "card-title")
private WebElement cardTitle;
```

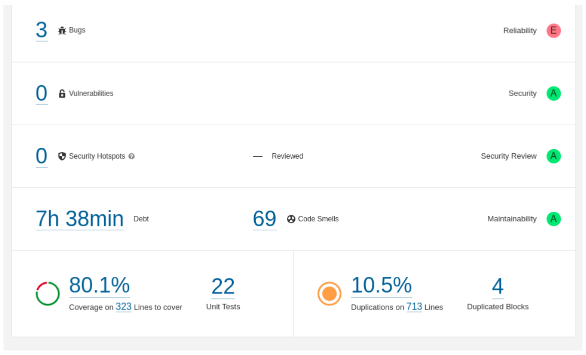
```java
@Test
// @Disabled("Frontend")
void today() throws InterruptedException {
    driver.get("http://localhost:5173/");

    driver.manage().window().setSize(new Dimension(1550, 1001));
    PageFactory.initElements(driver, this);

    input.click();
    input.clear();
    input.sendKeys("Oslo");

    today.click();
    Thread.sleep(3000);
    assertThat(cardTitle.getText()).isEqualTo("Spikersuppa, Oslo, Norway");
}
```

3.4 Code quality analysis

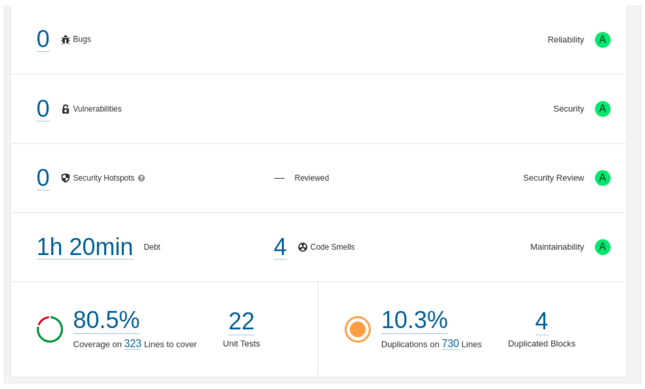
To analyze code quality, I used SonarQube. This tool was used at two points in the development lifecycle of the application. The first one was before integration testing. At this point, I had all classes implemented, along with their respective unit tests. Before writing the integration tests, I set up SonarQube and fixed the bugs and most code smells that existed at this point. At this point I had some considerable code debt, and didn't want to keep implementing more features and tests without cleaning up the code behind me.

I managed to reduce the project debt from around 7h30mins to 1h20mins, and fix the three bugs that I had not spotted.

The following screenshots show the SonarQube dashboard at this point, before and after cleaning up the code.



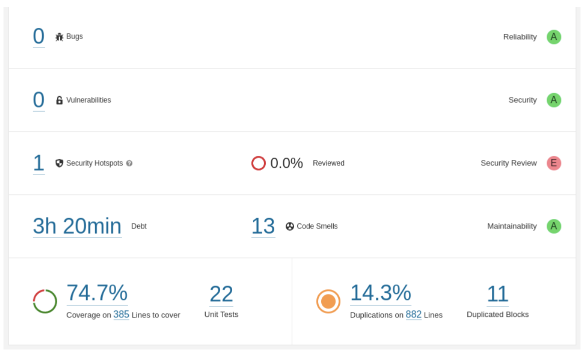
3 Bugs — Reliability E
0 Vulnerabilities — Security A
0 Security Hotspots — Reviewed — Security Review A
7h 38min Debt — 69 Code Smells — Maintainability A
80.1% Coverage on 323 Lines to cover — 22 Unit Tests
10.5% Duplications on 713 Lines — 4 Duplicated Blocks



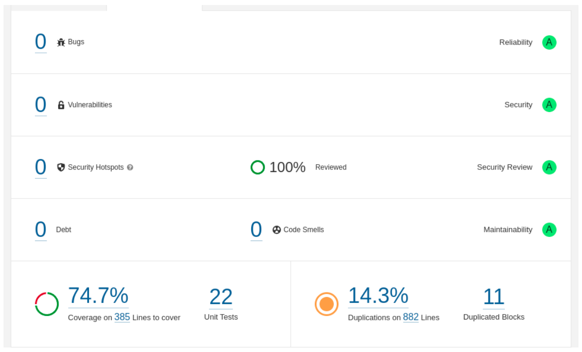
0 Bugs — Reliability A
0 Vulnerabilities — Security A
0 Security Hotspots — Reviewed — Security Review A
1h 20min Debt — 4 Code Smells — Maintainability A
80.5% Coverage on 323 Lines to cover — 22 Unit Tests
10.3% Duplications on 730 Lines — 4 Duplicated Blocks

I also analyzed the code with SonarQube before the submission. At this point, pretty much all of the code was already written, so I figured it would be a good time to fix up the project. A security issue had come up, because I had to annotate the controller with the @CrossOrigin decorator, in order to be able to access its endpoints through the web client.

This was duly reviewed, as were the new code smells that came up, as shown in the



0 Bugs — Reliability A
0 Vulnerabilities — Security A
1 Security Hotspots — 0.0% Reviewed — Security Review E
3h 20min Debt — 13 Code Smells — Maintainability A
74.7% Coverage on 385 Lines to cover — 22 Unit Tests
14.3% Duplications on 882 Lines — 11 Duplicated Blocks



0 Bugs — Reliability A
0 Vulnerabilities — Security A
0 Security Hotspots — 100% Reviewed — Security Review A
0 Debt — 0 Code Smells — Maintainability A
74.7% Coverage on 385 Lines to cover — 22 Unit Tests
14.3% Duplications on 882 Lines — 11 Duplicated Blocks

following screenshots.

3.5 Continuous integration pipeline [optional]

A simple CI Pipeline was implemented using GitHub Actions. I created a workflow that runs all unit and integration tests when a push is made to the application folder (in this case, the hw1 folder in the repository).

When tests are successful, the application is also packaged, setting up the production deployment. At this point, a CD pipeline is also trivial to implement, but was not done due to the fact that the project is not currently deployed.

Here is the GitHub Actions workflow responsible for continuously testing and building the application:

```yaml
name: HW1 CI Pipeline

on:
  push:
    branches:
      - main
    paths:
      - hw1/**


jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-java@v1
        with:
          java-version: 11
      - run: cd hw1/airquality && mvn test

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-java@v1
        with:
          java-version: 11
      - run: cd hw1/airquality && mvn package
```

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/mankings/TQS_103341/tree/main/hw1
Video demo	https://github.com/mankings/TQS_103341/blob/main/hw1/103341.mkv
CI Pipeline	https://github.com/mankings/TQS_103341/actions/workflows/ci-hw1.yml

Reference materials

External APIs Docs:

- https://openweathermap.org/api
- https://aqicn.org/json-api/doc/

GitHub Actions:

- https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven