HW1 Report

Ricardo Pinto (103078)

- ## Overview of the work

  This TQS assignment is an air quality web app. My implementation allows the user to select one of some cities and a day (today, tomorrow or the day after tomorrow). Then, after a button click, the air quality is presented. I have also implemented a REST API with an endpoint to get the air quality for a city and date and another endpoint to get the current cache stats.

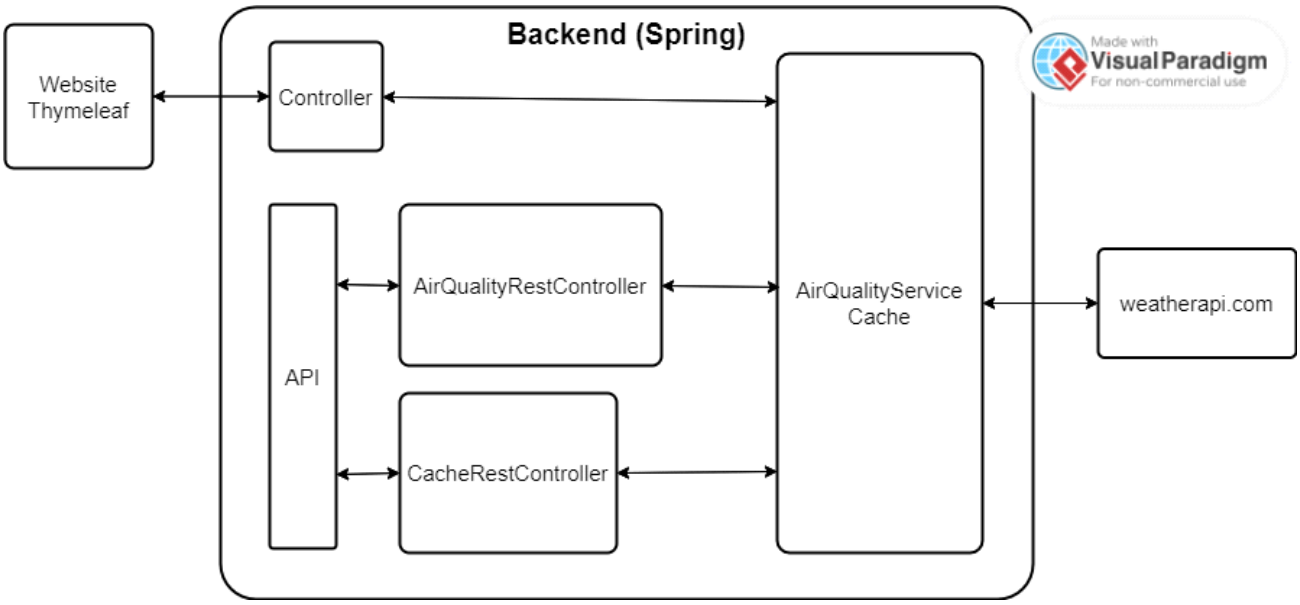
- ## Current limitations

  The air quality API I use to get the data from only allows free users to forecast up to 3 days from the current today. This is the main limitation of my implementation.

- ## Functional scope and supported interactions

  The main usage of my app is for any person to check the air quality in a city in a given day. The secondary usage is to get that data through my API. My self-made cache allows recent requests to be processed extremely fast (both in the frontend and in the API).

- System architecture



The cache is, in its essence, an hashmap where the keys are the city and date and the value is its air quality.

- API for developers

The first endpoint requires 2 url parameters “city” and “date” and returns the air quality there; the json response fields are “city”, “date”, “co”, “no2”, “o3”, “so2”, “pm2_5” and “pm10”.

If “city” request parameter is invalid, status code 500 (internal server error) is returned, and if “date” request parameter is invalid, status code 400 (bad request) is returned.

localhost:8080/api/v1/air-quality?city=London&date=2023-04-12
localhost:8080/api/v1/air-quality?city=New York&date=2023-04-13

The second endpoint returns the current cache stats in 4 json fields: “requests”, “hits”, “misses”, “timeToLiveSeconds”.

localhost:8080/api/v1/cache

- ## Overall strategy for testing

  My test development strategy was Test Driven Development. I wrote the skeleton of my classes first, then the tests, then implemented the classes. The only exception was the selenium tests, as they require a functional web app, so I wrote those later. TDD helped immensely. Later, some tests turned out slightly illogical, so I improved them.

- ## Unit and integration testing

  I have 4 unit tests in total, these are just standard junit tests.

  3 to test the 3 functions in my Utils class: strToDate(LocalDate date), dateToStr(String strDate) and round(double value, int numDecimalPlaces).

  The other unit test tests my cache implementation: asserts an item's TTL is refreshed when hit, asserts an item is gone from the cache after TTL, and asserts its stats (requests, misses and hits) are correct in the end.

  Since I have 2 API endpoints, I have 2 integration tests, annotated with @SpringBootTest(webEnvironment = WebEnvironment.MOCK) since they need the full app loaded and a mocked web environment. They also use a MockMvc to perform the GET requests.

  The first tests my cache API endpoint.

  The second integration test tests a valid air quality request and validates the response, tests a request with invalid city, and tests a request with invalid date.

- ## Functional testing

  I have 2 tests that test my service (AirQualityService.java). These are annotated with @ExtendWith(MockitoExtension.class) since they don't need the full app loaded and can run in a lighter environment such as Mockito.

  The service variable in this test class is

  ```
  @InjectMocks
  @Spy
  private AirQualityService airQualityService;
  ```

@InjectMocks allows injections, while @Spy allows me to call the methods of the service while still tracking every possible interaction similar to a normal service.

I have a .json file with the expected json for Paris in a certain day in order to mock the external air quality API response.

```
Mockito.when(airQualityService.getJson(new URL(parisUrlWithParameters))).thenReturn(expectedJson);
```

The first test asks the service for the air quality in Paris in a certain day and asserts that it is the same as in expected json.

The second test asserts that the service returns ‘null’ if the request is invalid (in this case the city is “Error city”).

I also made 2 selenium tests, which I implemented in java (SeleniumTests.java). These use @SpringBootTest as they need the full app, a random port and a WebDriver.

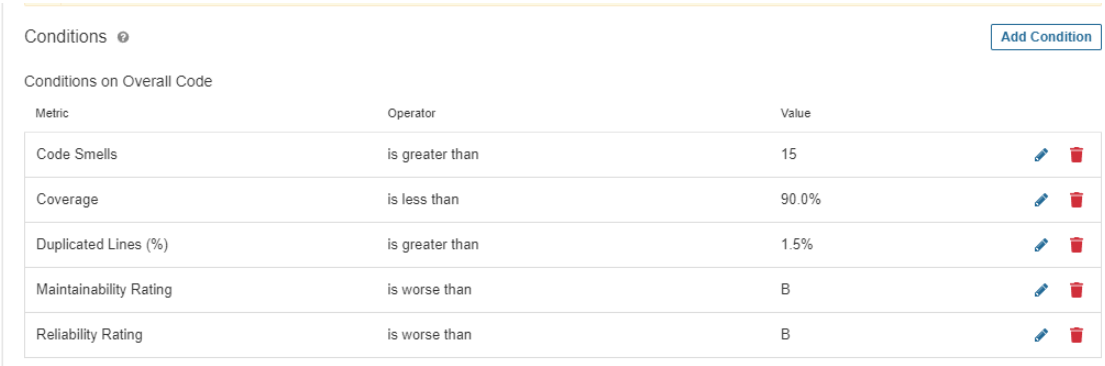
The first test asserts the day selection options in the frontend are "Today {dateToday}", "Tomorrow {dateTomorrow}" and "2 days from now {dateIn2days}".

The second test chooses Madrid and tomorrow's date, clicks the submit button and confirms the air quality display page is correct and as expected.

- ## Code quality analysis

I analyzed my app locally with Sonarqube's docker image.

I defined this ambitious quality game for this app.



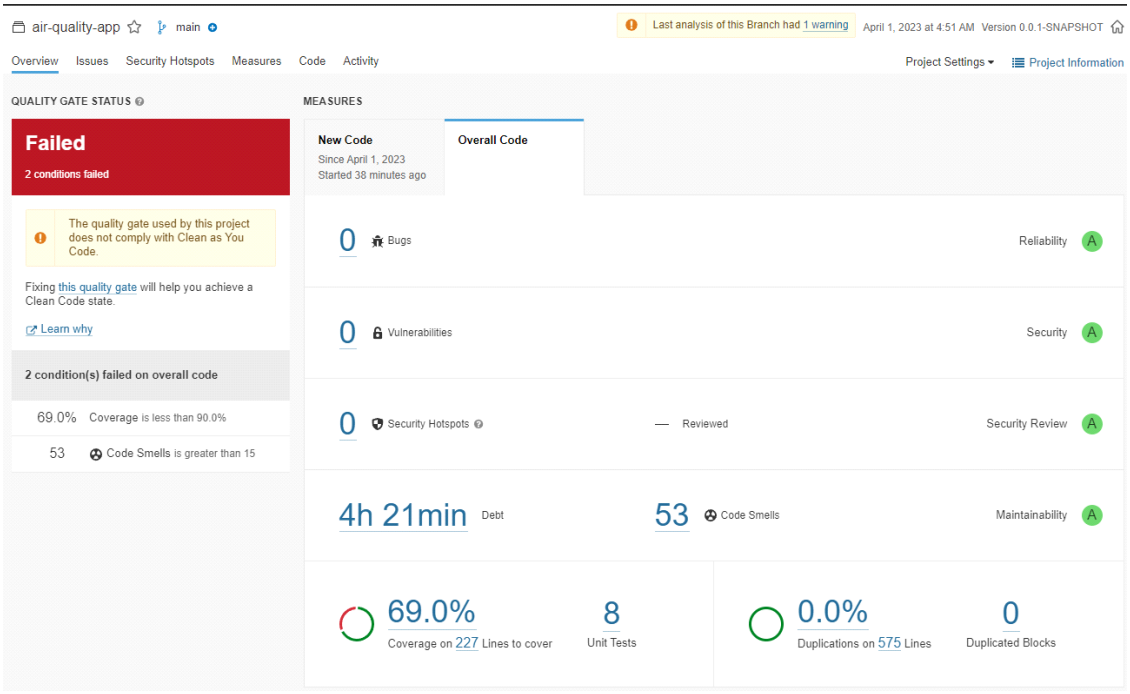
I decided to go with high coverage (90%), low % duplciated lines (1.5%), and a maximum of 15 code smells because of TQS's nature and objectives. Security is not very relevant in this context so I discarded it.

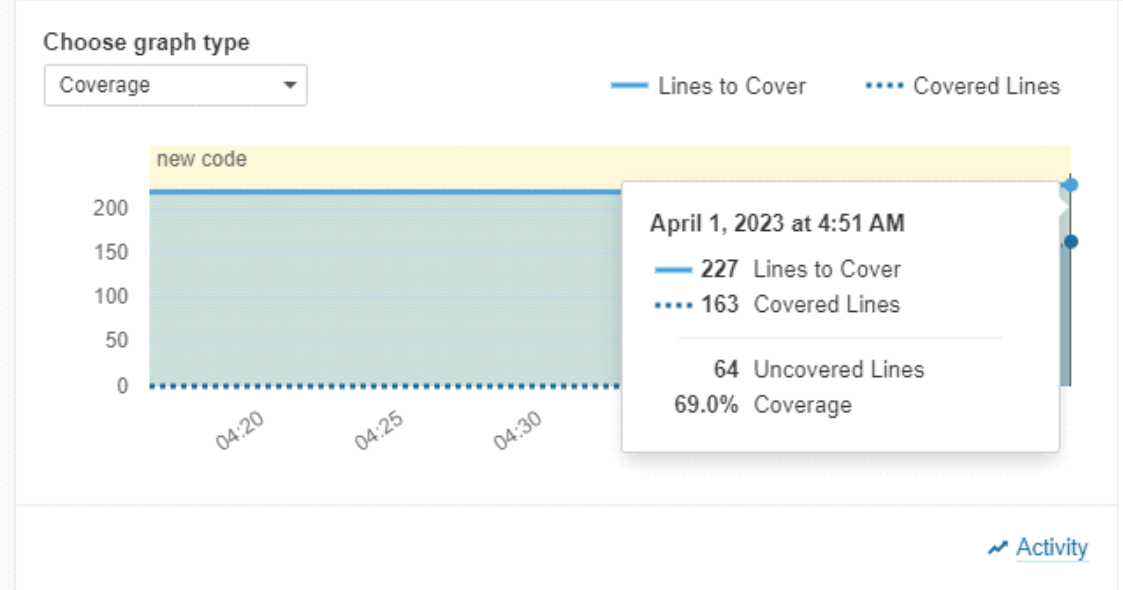
Sonarqube analysis shows 1 bug. It's in the cache implementation (Cache.java).

Fixing it...

```
// Start a background thread to periodically remove expired entries from the
// cache
Thread cleanupThread = new Thread(() -> {
    while (true) {
        try {
            Thread.sleep(1 / 10);
            cleanup();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // restore the interrupted status
        }
    }
});
```

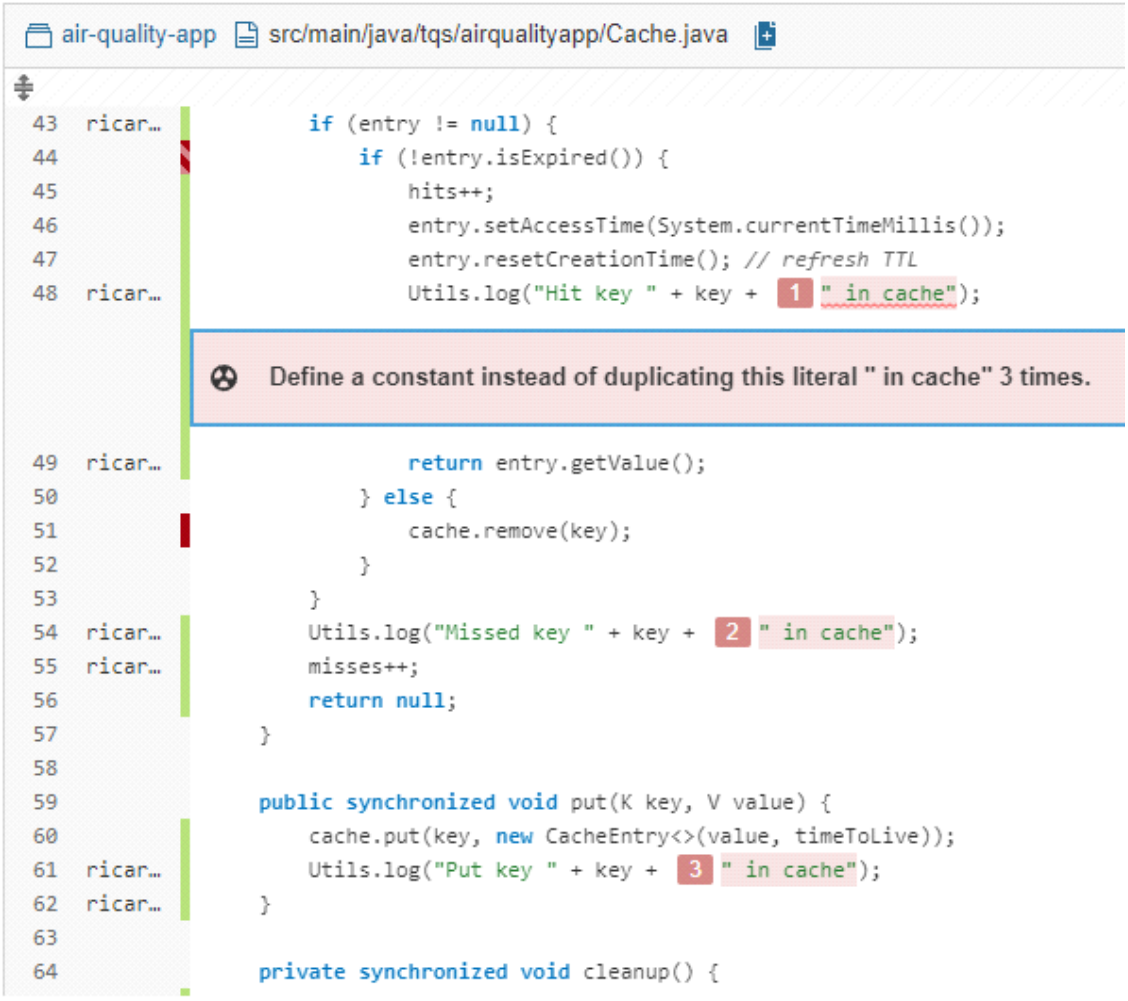
Now there's no bug or vulnerabilities, but my quality gate fails because coverage is only 69% and too many code smells (53).





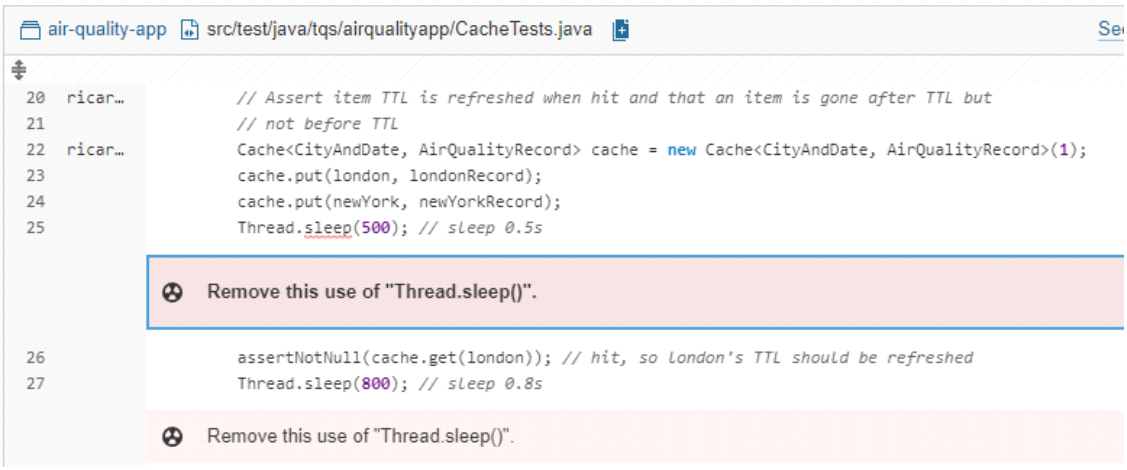
Some of the code smells I do not agree with, however some are things I overlooked while coding and I agree with Sonarqube.

For example, Sonarqube says the following is a major code smell.



I do not think it's worth separating part of a log string into a constant because at any time I may decide to have different strings in the 3 method calls. Even if this is a code smell, it makes no sense to classify it as severe as 'major'.

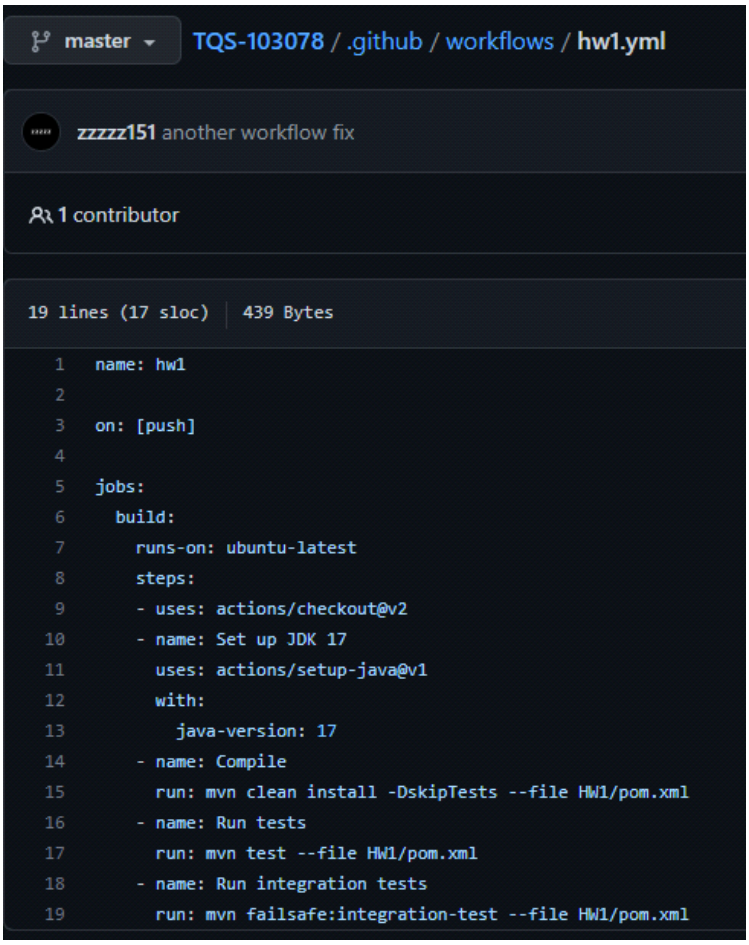
One that really opened my eyes was the following code smell.



I learned that it is much better to use the async Awaitility library rather than Thread.sleep() in tests.

- Continuous integration pipeline [optional]

The following github workflow yml file in my repository runs the HW1 tests and integration tests whenever I push to the repo.



- References & resources

Resource:	**URL/location:**
Git repository	https://github.com/zzzzz151/TQS-103078
Video demo	https://github.com/zzzzz151/TQS-103078/blob/master/HW1/demo.webm or https://www.veed.io/view/16705e49-7964-4cd2-8deb-b345f82c862f?panel=share
QA dashboard (online)	
CI/CD pipeline	https://github.com/zzzzz151/TQS-103078/actions
Deployment ready to use	localhost

Weather and air quality API used: https://www.weatherapi.com/