HW1: Mid-term assignment report

Pedro Miguel Loureiro Amaral [93283], v2021-05-14

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. The software product is named “Air Quality TQS” and consists of a Spring Boot application containing a Rest API and an in-memory cache system and a webpage that uses the referred API. This API obtains its data from the IQAir API allowing for a quick access of air quality statistics from all over the world seen as the user can select the country, state, and city that he wishes to know information about. In addition to this feature the user can also see some statistics about the use of the API.

1.2 Current limitations

Some features that could be added but were not include the use of more than one API to fetch data so that even if one goes down it continues to work. Also, in terms of tests the UI can only be tested with a copy of the project being ran which could be improved to be possible without.

2 Product specification

2.1 Functional scope and supported interactions

This product consists of two main features:

1. The user can obtain information about a given city by specifying its country, state, and name. The information obtained consists of its humidity, American air quality index, Chinese air quality index, main pollutant for the American AQI and main pollutant for the Chinese AQI.
2. The user can also obtain statistics about the use of the API consisting of the number of requests it has received, the number of requests that were answered with the in-memory cache (hits) and the number of requests that were answered with the external API instead (misses).

2.2 System architecture

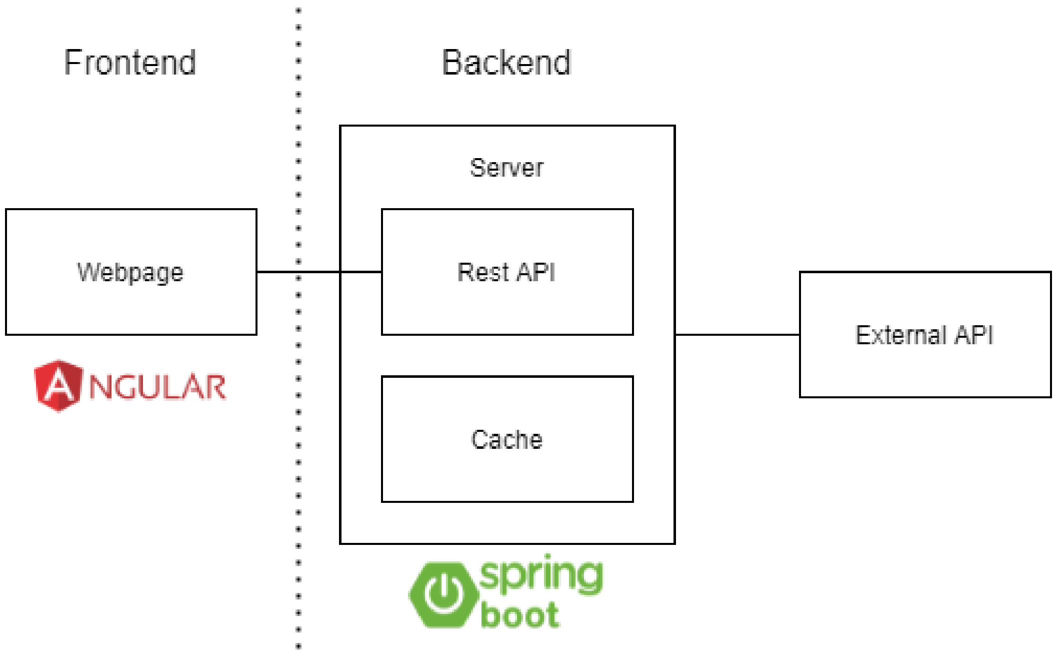


Figure 1 - Architecture Diagram

This product's frontend consists of a webpage implemented with the Angular framework which requests data from the backend. The backend consists in an application server developed in spring boot and contains a Rest API and a custom implementation of an in-memory cache system. When the cache does not contain the requested data, it makes a request to the [external API](#).

2.3 API for developers

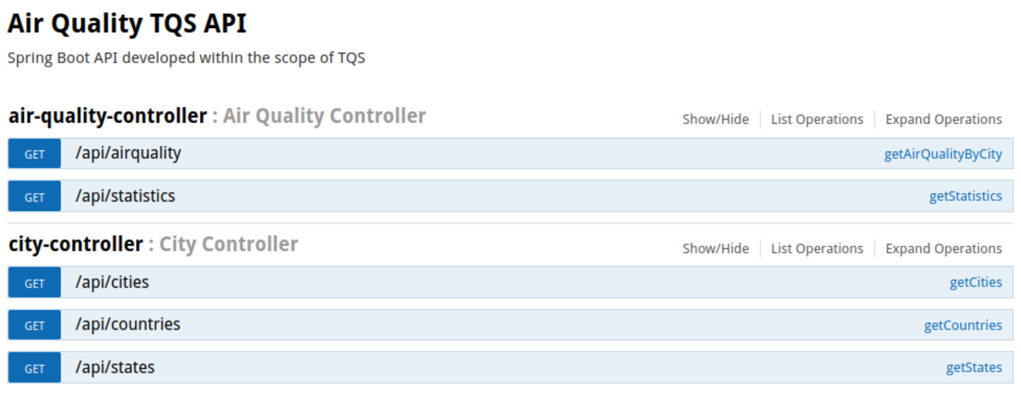
This project contains the following 4 API endpoints about the problem domain:

Endpoint	Arguments	Description
/api/countries		Obtain the list of countries where we can get air quality data.
/api/states	country	Obtain the list of states from a given country where we can get air quality data.
/api/cities	country, state	Obtain the list of cities from a given state from a given country where we can get air quality data.
/api/airquality	country, state, city	Obtain air quality data from a given city from a given state from a given country. This data consists of the humidity, the American air quality index, the Chinese air quality index, the main pollutant for the American AQI and the main pollutant for the Chinese AQI.

It also contains the following API endpoint about the cache usage:

Endpoint	Arguments	Description
/api/statistics		Obtain statistics about the use of the API consisting of the number of requests it has received, the number of requests that were answered with the in-memory cache (hits) and the number of requests that were answered with the external API instead (misses).

Details about the available endpoints can also be found on the API documentation implemented with swagger.


Air Quality TQS API

Spring Boot API developed within the scope of TQS

air-quality-controller : Air Quality Controller — Show/Hide | List Operations | Expand Operations

GET /api/airquality getAirQualityByCity

GET /api/statistics getStatistics

city-controller : City Controller — Show/Hide | List Operations | Expand Operations

GET /api/cities getCities

GET /api/countries getCountries

GET /api/states getStates

Figure 2 - API Documentation with Swagger

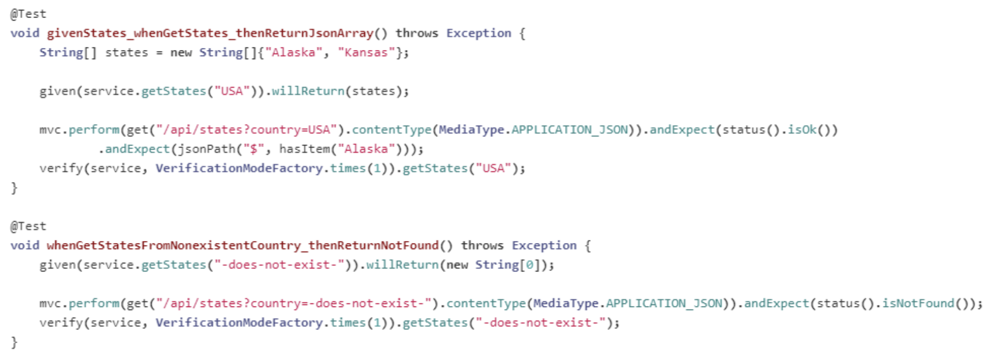
3 Quality assurance

3.1 Overall strategy for testing

To develop the API, a TDD (Test-driven development) strategy was adopted. First, a simple structure of the API was coded followed by some tests using JUnit5. In some unit tests the Mockito framework was used to simulate the needed dependencies and to implement the controller tests Spring MockMVC was used. After that, a better implementation of each module was coded to satisfy all requirements.
To develop the webpage, a BDD (Behavior-driven Development) strategy was adopted. First Cucumber was used to define the scenarios that were needed. Then the webpage was developed having those scenarios in mind. Finally, those scenarios were recorded with Selenium IDE and, after some code refactoring, the test scenarios were implemented with a mixture of Cucumber and Selenium.

3.2 Unit and integration testing

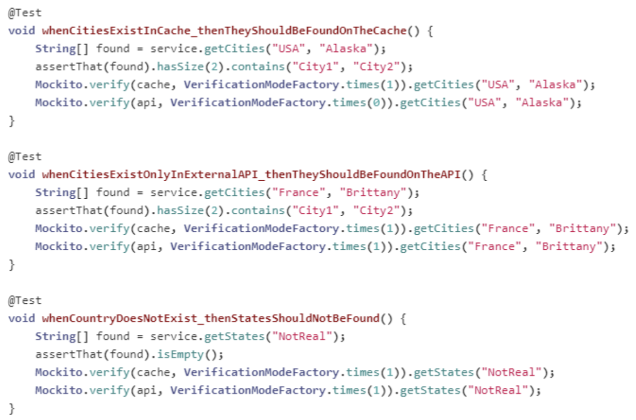
First the unit tests for the controllers were implemented with the help of a mock of the services using Mockito. In these tests it was verified if each controller returned the output of service if possible, if not a “404 Not Found” response should be returned.



```java
@Test
void givenStates_whenGetStates_thenReturnJsonArray() throws Exception {
    String[] states = new String[]{"Alaska", "Kansas"};

    given(service.getStates("USA")).willReturn(states);

    mvc.perform(get("/api/states?country=USA").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
            .andExpect(jsonPath("$", hasItem("Alaska")));
    verify(service, VerificationModeFactory.times(1)).getStates("USA");
}

@Test
void whenGetStatesFromNonexistentCountry_thenReturnNotFound() throws Exception {
    given(service.getStates("-does-not-exist-")).willReturn(new String[0]);

    mvc.perform(get("/api/states?country=-does-not-exist-").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isNotFound());
    verify(service, VerificationModeFactory.times(1)).getStates("-does-not-exist-");
}
```

Figure 3 - Some tests from the class CityController_UnitTest

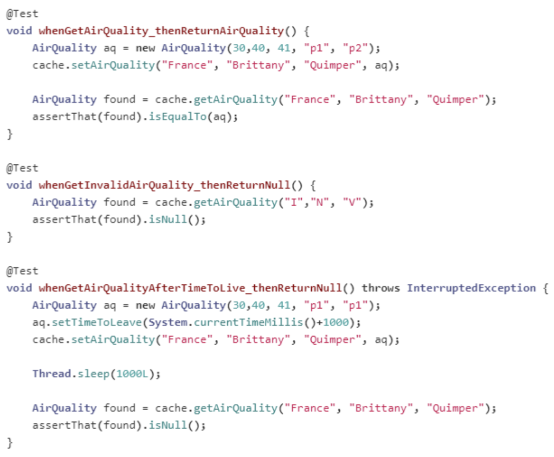
Secondly, the unit tests for the services were implemented mocking both the cache and the external API. These tests verify if when the data is both on the cache and the external API then only the cache should be used, and if the data is not on the cache and is on the external API then the data from the external API should be returned after trying to get it from the cache. Finally, if the data does not exist anywhere the services must return either null (if an air quality object is requested) or an empty array (if a list countries, states or cities is requested).



```java
@Test
void whenCitiesExistInCache_thenTheyShouldBeFoundOnTheCache() {
    String[] found = service.getCities("USA", "Alaska");
    assertThat(found).hasSize(2).contains("City1", "City2");
    Mockito.verify(cache, VerificationModeFactory.times(1)).getCities("USA", "Alaska");
    Mockito.verify(api, VerificationModeFactory.times(0)).getCities("USA", "Alaska");
}

@Test
void whenCitiesExistOnlyInExternalAPI_thenTheyShouldBeFoundOnTheAPI() {
    String[] found = service.getCities("France", "Brittany");
    assertThat(found).hasSize(2).contains("City1", "City2");
    Mockito.verify(cache, VerificationModeFactory.times(1)).getCities("France", "Brittany");
    Mockito.verify(api, VerificationModeFactory.times(1)).getCities("France", "Brittany");
}

@Test
void whenCountryDoesNotExist_thenStatesShouldNotBeFound() {
    String[] found = service.getStates("NotReal");
    assertThat(found).isEmpty();
    Mockito.verify(cache, VerificationModeFactory.times(1)).getStates("NotReal");
    Mockito.verify(api, VerificationModeFactory.times(1)).getStates("NotReal");
}
```

Figure 4 - Some tests from the class CityService_UnitTest

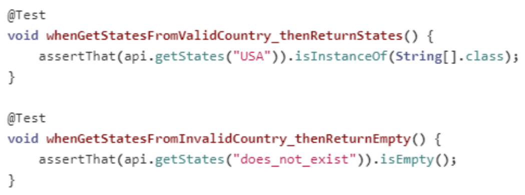
Thirdly, the unit tests for the cache were implemented. In these tests, it is verified if after storing data on the cache, it returns that data when requested if the time-to-live was not surpassed and null or empty list if it was. Also, it should return null or empty list if the cache does not contain the information needed.



```java
@Test
void whenGetAirQuality_thenReturnAirQuality() {
    AirQuality aq = new AirQuality(30,40, 41, "p1", "p2");
    cache.setAirQuality("France", "Brittany", "Quimper", aq);

    AirQuality found = cache.getAirQuality("France", "Brittany", "Quimper");
    assertThat(found).isEqualTo(aq);
}

@Test
void whenGetInvalidAirQuality_thenReturnNull() {
    AirQuality found = cache.getAirQuality("I","N", "V");
    assertThat(found).isNull();
}

@Test
void whenGetAirQualityAfterTimeToLive_thenReturnNull() throws InterruptedException {
    AirQuality aq = new AirQuality(30,40, 41, "p1", "p1");
    aq.setTimeToLeave(System.currentTimeMillis()+1000);
    cache.setAirQuality("France", "Brittany", "Quimper", aq);

    Thread.sleep(1000L);

    AirQuality found = cache.getAirQuality("France", "Brittany", "Quimper");
    assertThat(found).isNull();
}
```

Figure 5 - Some tests from the class Cache_UnitTest

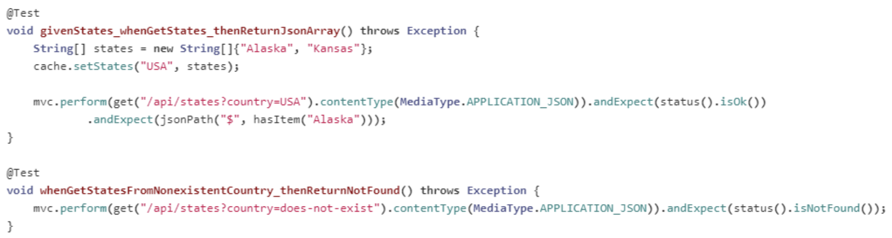
Next, the unit tests for the class which takes care of the communication with the external API were implemented. These tests verify if it returns valid data with valid input and returns null or an empty array otherwise.



```java
@Test
void whenGetStatesFromValidCountry_thenReturnStates() {
    assertThat(api.getStates("USA")).isInstanceOf(String[].class);
}

@Test
void whenGetStatesFromInvalidCountry_thenReturnEmpty() {
    assertThat(api.getStates("does_not_exist")).isEmpty();
}
```

Figure 6 - Some tests from the class ExternalAPI_UnitTest

Finally, the integration tests were implemented which verify if data available on the cache is returned correctly when making an HTTP request and a “404 Not Found” response if the data does not exist anywhere.

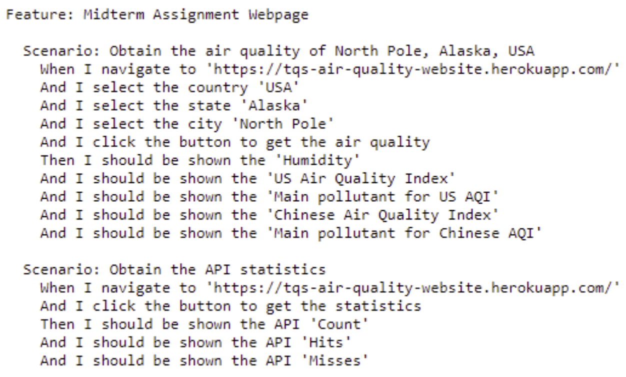


```java
@Test
void givenStates_whenGetStates_thenReturnJsonArray() throws Exception {
    String[] states = new String[]{"Alaska", "Kansas"};
    cache.setStates("USA", states);

    mvc.perform(get("/api/states?country=USA").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
            .andExpect(jsonPath("$", hasItem("Alaska")));
}

@Test
void whenGetStatesFromNonexistentCountry_thenReturnNotFound() throws Exception {
    mvc.perform(get("/api/states?country=does-not-exist").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isNotFound());
}
```

Figure 7 - Some tests from the class AirQualityController_IntegrationTest

3.3 Functional testing

Functional tests were implemented to test the use scenarios of the webpage. Therefore, we have a scenario in which the user tries obtains the air quality in a specific location and another where the user obtains the statistics of the API. So that these tests can be run they access the online deployed version of the project since I could not find a way to run them locally except for having another instance of the spring boot application running simultaneously. They were implemented with Cucumber, Selenium, and a Firefox Driver.



```gherkin
Feature: Midterm Assignment Webpage

  Scenario: Obtain the air quality of North Pole, Alaska, USA
    When I navigate to 'https://tqs-air-quality-website.herokuapp.com/'
    And I select the country 'USA'
    And I select the state 'Alaska'
    And I select the city 'North Pole'
    And I click the button to get the air quality
    Then I should be shown the 'Humidity'
    And I should be shown the 'US Air Quality Index'
    And I should be shown the 'Main pollutant for US AQI'
    And I should be shown the 'Chinese Air Quality Index'
    And I should be shown the 'Main pollutant for Chinese AQI'

  Scenario: Obtain the API statistics
    When I navigate to 'https://tqs-air-quality-website.herokuapp.com/'
    And I click the button to get the statistics
    Then I should be shown the API 'Count'
    And I should be shown the API 'Hits'
    And I should be shown the API 'Misses'
```

Figure 8 - Tests in the resource website.feature

3.4 Static code analysis

To perform static code analysis the tool Sonarcloud was used together with Gitlab CI (which will be seen in more detail in the next section). In the following image we can see the current project dashboard after all vulnerabilities, security hotspots and code smells were either solved or ignored. Also, the coverage percentage was obtained with the help of the Jacoco plugin.

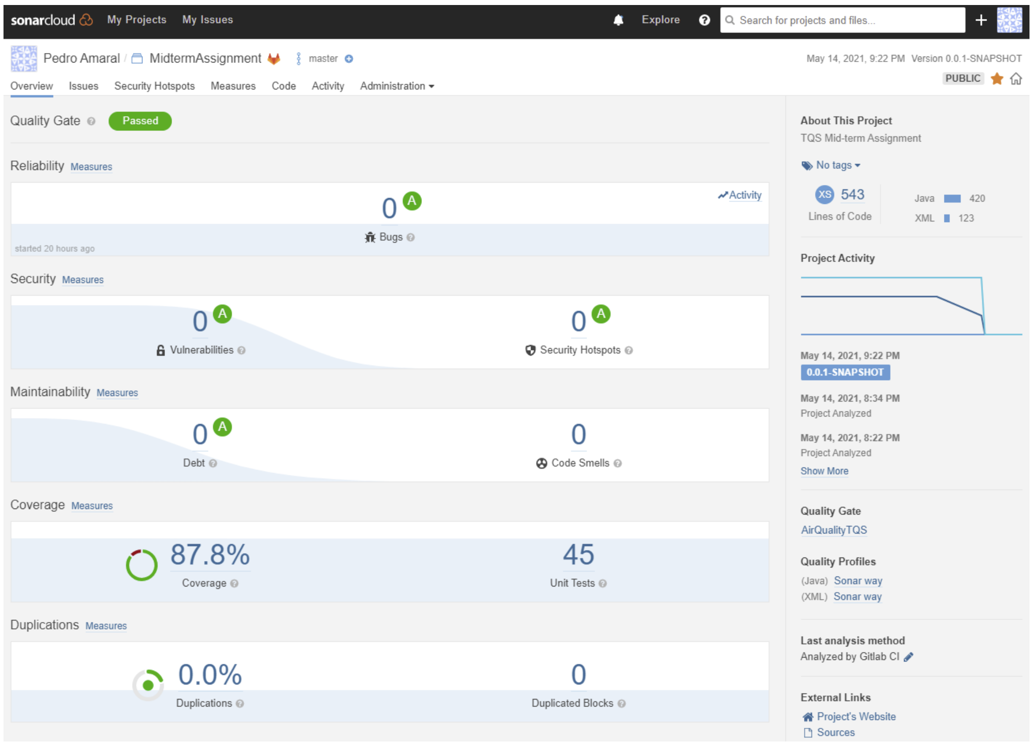
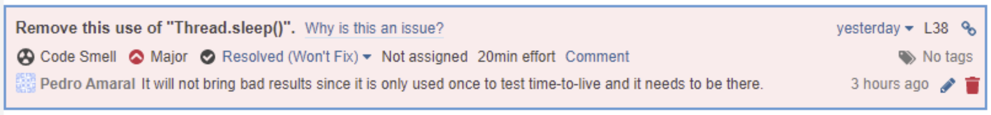


Figure 9 - Sonarcloud Dashboard

Initially I had around a hundred code smells across all levels of severity most of them being of lower severity. These results indicated that there are good practices that developers do not comply with because it is simply faster to code in a worse way that also works. In terms of the code smells detected most of them required only small fixes but made the code cleaner and smaller. For example, I learned to use other methods from the class HashMap that I was not used to but simplify the code, in particular replacing "*if (!map.containsKey(x)) map.put(x, y);*" with "*map.putIfAbsent(x, y)*". However, there were also code smells that did not make sense to change in this context.



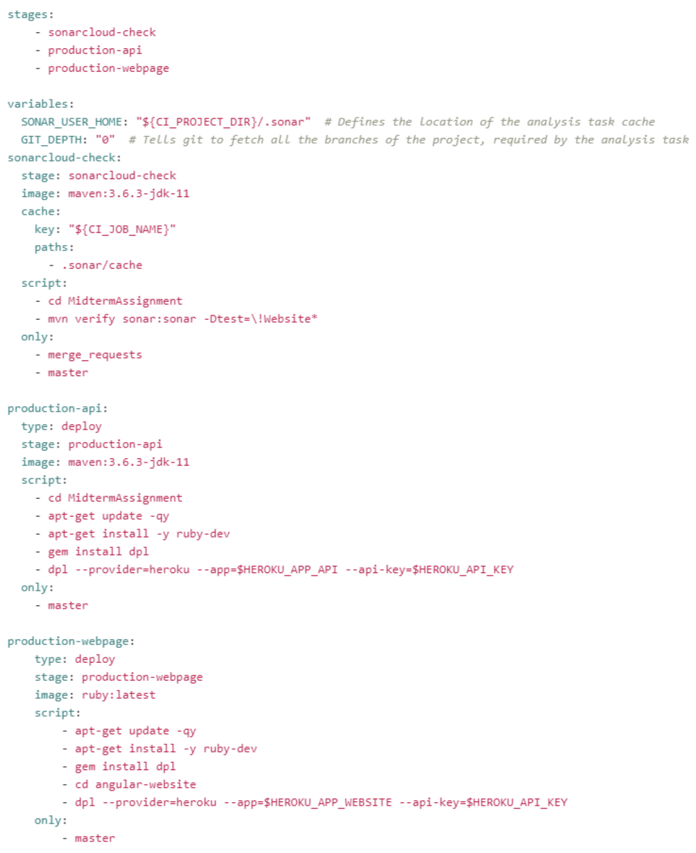
Remove this use of "Thread.sleep()". Why is this an issue? yesterday L38
Code Smell Major Resolved (Won't Fix) Not assigned 20min effort Comment No tags
Pedro Amaral It will not bring bad results since it is only used once to test time-to-live and it needs to be there. 3 hours ago

Figure 10 - Ignored Code Smell

For example, this code is needed to test the time-to-live of the cache.
To conclude, I would like to say that obtaining a coverage percentage of 100% is not required since some of the code also includes auto-generated getters and setters and configuration files such as the ones needed for Swagger API documentation.

3.5 Continuous integration pipeline [optional]

In this project, a CI pipeline was implemented with Gitlab CI. It follows the following script:



```yaml
stages:
    - sonarcloud-check
    - production-api
    - production-webpage

variables:
  SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"  # Defines the location of the analysis task cache
  GIT_DEPTH: "0"  # Tells git to fetch all the branches of the project, required by the analysis task
sonarcloud-check:
  stage: sonarcloud-check
  image: maven:3.6.3-jdk-11
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - cd MidtermAssignment
    - mvn verify sonar:sonar -Dtest=\!Website*
  only:
    - merge_requests
    - master

production-api:
  type: deploy
  stage: production-api
  image: maven:3.6.3-jdk-11
  script:
    - cd MidtermAssignment
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=$HEROKU_APP_API --api-key=$HEROKU_API_KEY
  only:
    - master

production-webpage:
    type: deploy
    stage: production-webpage
    image: ruby:latest
    script:
        - apt-get update -qy
        - apt-get install -y ruby-dev
        - gem install dpl
        - cd angular-website
        - dpl --provider=heroku --app=$HEROKU_APP_WEBSITE --api-key=$HEROKU_API_KEY
    only:
        - master
```

Figure 11 - CI Pipeline Script

Every time a commit is pushed to the repository, this pipeline runs the tests in the spring boot project except for the Cucumber scenarios and makes static code analysis of the API sending the results to the project's Sonarcloud dashboard. Then it deploys the Spring Boot API in Heroku followed by the Angular Webpage which is also deployed in Heroku. Although in Gitlab CI it is possible to run 2 jobs in the same phase and therefore at the same time, the free version of Heroku did not allow 2 deployments to be made at the same time so I had to separate these 2 jobs in 2 different phases.

4 References & resources

Project resources

- Video demo
- Ready to use application
- QA dashboard
- Git repository

Reference materials

- https://www.iqair.com/
- https://sonarcloud.io/
- https://www.javainuse.com/spring/boot_swagger
- https://docs.gitlab.com/ee/ci/
- https://medium.com/swlh/how-do-i-deploy-my-code-to-heroku-using-gitlab-ci-cd-6a232b6be2e4
- https://itnext.io/how-to-deploy-angular-application-to-heroku-1d56e09c5147
- https://betterprogramming.pub/how-to-deploy-your-angular-9-app-to-heroku-in-minutes-51d171c2f0d
- https://devcenter.heroku.com/articles/java-support