HW1: Mid-term assignment report

Mariana Sousa Pinho Santos [93257], 2021-05-13

1 Introduction

1.1 Overview of the work

This report has the goal of presenting the individual homework proposed for the curricular unit TQS. This project included both developing a web application and adopt various quality assurance methods that will be discussed in this document.
The web-based application, to which I gave the name of Air Quality Forecast, can be used to get information about the air quality in different cities for the following few days. This information includes elements like Ozone (**o3**), particulate matter with a diameter of less than 2,5 micrometre (**pm25**) and 10 micrometre (**pm10**), and ultraviolet index (**uvi**). Usually it is possible to see the forecast for the 5 next days – although this is not constant and depends on the external API. For each of the different elements, 3 values are displayed: the average, the minimum, and the maximum value.
The external API chosen was from a project called ***World Air Quality Index Project***, and its home page can be accessed through the following link: www.aqicn.org/.

1.2 Current limitations

The project at this stage now has a lot of limitations. One of them is the fact that it relies completely on one single external API. If this one API fails, or isn't able to deliver the expected results, the system will not be able to provide with information.

Besides this major problem, there are a lot of features that could be implemented to improve this solution. For example, through the graphical interface the user should be able to chose the country from where the cities on the city list are. As of right now, the user only has the possibility to chose the cities from a single country (which is Finland by default), and this can only be changed by editing the source code. It should be said that even though this is impossible to do through the web interface, it is possible in the API. Another feature that would be useful is to allow the user to only request a specific type of data (for example, o3) and also a specific date, instead of showing the complete forecast.

2Product specification

2.1 Functional scope and supported interactions

As already stated, the main goal of the application is to allow the user to check the forecast of the elements already mentioned (O3, PM25, PM10 and UVI) for the next few days in a city that the user can chose. This information is valuable for the user so that he can be aware of the situation and health risks that he faces if he is/goes to those places. This way they can prepare and prevent these serious hazards.

The application is only one web-page. When the user goes to this page they can click a button and be shown a list of cities where there is info about these air quality factors in the remote service. At this moment, the list shows only the stations present in Finland. The user should then select one and a few tables should appear on the screen. Each table corresponds to a different day and contains all the values for that specific day. The average, minimum and maximum value of each element in the forecast is shown in any of these tables. The user can also change the city at any time in the same button without having to go back or reload the page.

2.2 System architecture

The project consists in a **Spring MVC** application. A diagram of the system architecture can be seen in Fig. 1.

The **Thymeleaf** template engine was used to generate the web page that is sent to the user when requested. The **Controller** – used to access the web pages – and the **RestController** rely on the **Service** `DataAccess` to provide them with information. Additional libraries were used to facilitate the writing and readability of code, like **Project Lambok**.

To further visualize the software structure, in Fig. 2 we can see a class diagram. This diagram contains all Java classes in the project, excluding the ones dedicated to testing.

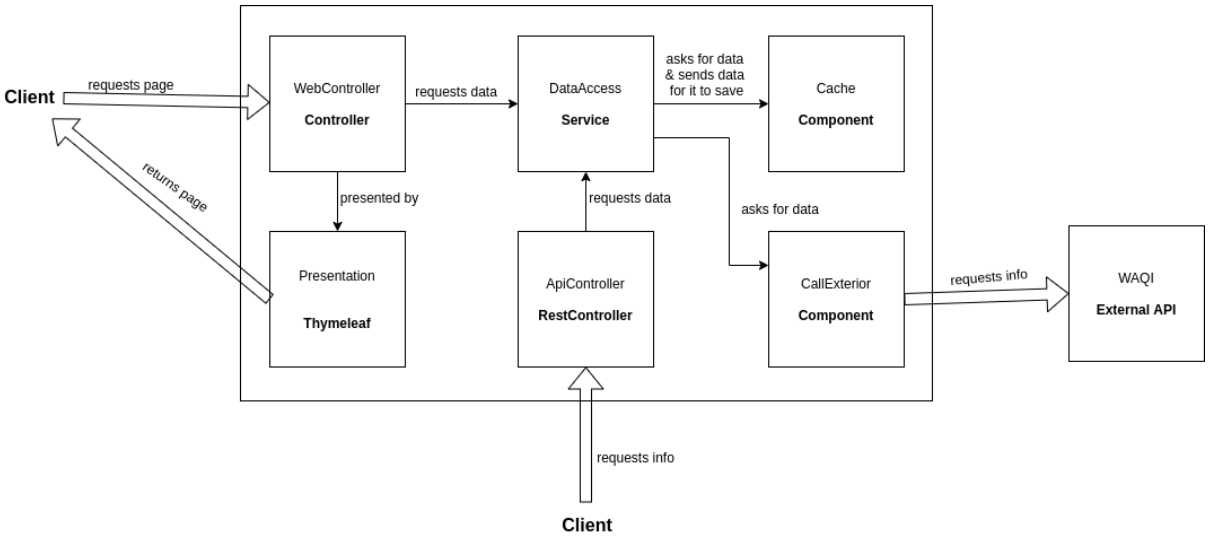


Figure 1: Architecture diagram

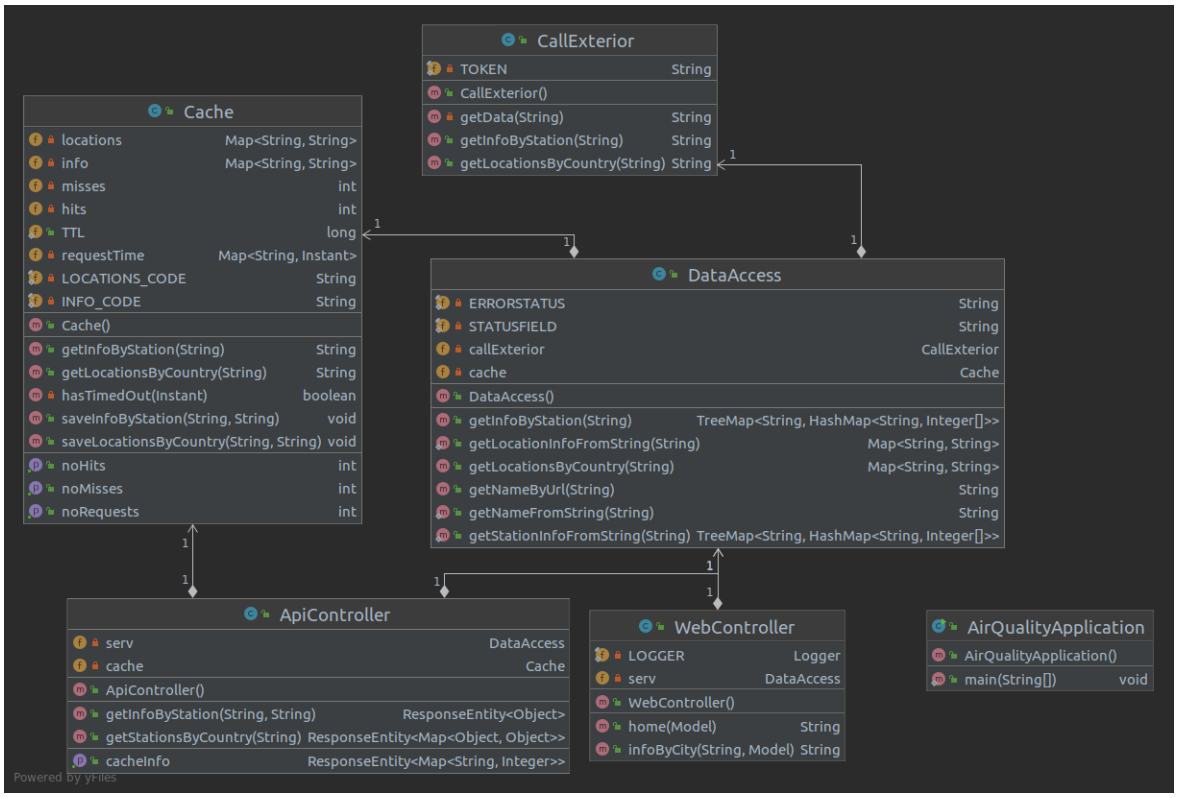


Figure 2: Class diagram

2.3 API for developers

The API developed is a very simple one, containing only 3 endpoints.
The following link contains an interactive documentation:
https://app.swaggerhub.com/apis-docs/marrrr/air-quality/1.0.0

There are 2 groups of endpoints:

- **the main functionality** – contains not only the same information as the one a user using the interface would get, but even allows to see this information for more places in the world than only places in one country
- **cache** – provides information about the cache state

In the first group, two endpoints are available:

In the second one, there is only one endpoint:

More information is provided in the already mentioned web-page.

2.4 Overall strategy for testing

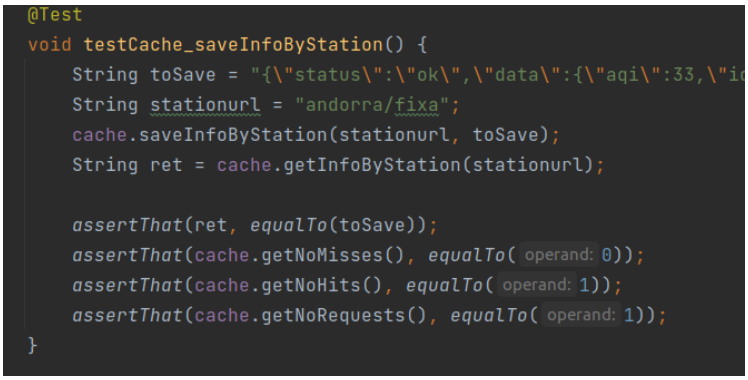
The testing of this project was based on the unit testing framework **JUnit5**, taking advantage of the **Hamcrest** framework for matching objects. The project wasn't completely developed using the test-driven development (TDD) process, even though at the end I started to understand that that would have been a good idea. Often when the tests were being composed I noticed that some things were not done correctly and had to change a lot of the application code.
A lot of different tools (besides the ones already mentioned) were used, like the **Mockito** framework for the service layer tests, **RestAssuredMockMvc** – a REST-assured API that lays on top of MockMvc, and also **Selenium** for the interface testing – taking advantage of the Selenium IDE as well.

2.5 Unit and integration testing

→ Unit Tests
All unit tests are implemented in the file **`UnitTests.java`**. They were made in order to verify the results of the **cache** methods and also some helper methods, found in the **`DataAccess`** class.

These tests include:

- Testing the save functionality of the cache (both for the locations and the forecast info); in these tests, the numbers of hits, misses, and total requests were also verified. An example of a test of this kind can be found on Fig. 3.



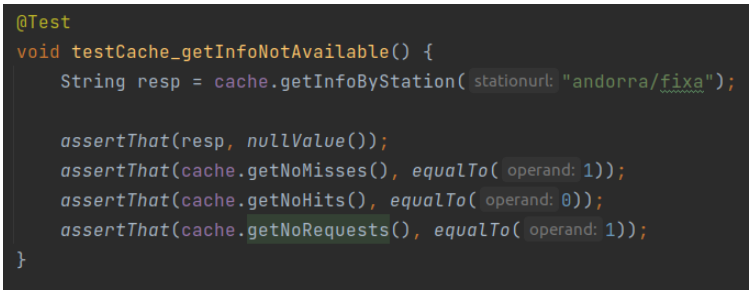
```
@Test
void testCache_saveInfoByStation() {
    String toSave = "{\"status\":\"ok\",\"data\":{\"aqi\":33,\"id
    String stationurl = "andorra/fixa";
    cache.saveInfoByStation(stationurl, toSave);
    String ret = cache.getInfoByStation(stationurl);

    assertThat(ret, equalTo(toSave));
    assertThat(cache.getNoMisses(), equalTo( operand: 0));
    assertThat(cache.getNoHits(), equalTo( operand: 1));
    assertThat(cache.getNoRequests(), equalTo( operand: 1));
}
```

Figure 3: testCache_saveInfoByStation test code example

The variable **`toSave`** in this case is similar to the answer that we would get if we queried the used remote API directly.

- Tests to verify the behaviour of the cache when it doesn't have the query results; 2 tests of this kind are being executed as well – one for getting the stations in a country, and the other for getting the forecast information for one specific station.



```
@Test
void testCache_getInfoNotAvailable() {
    String resp = cache.getInfoByStation( stationurl: "andorra/fixa");

    assertThat(resp, nullValue());
    assertThat(cache.getNoMisses(), equalTo( operand: 1));
    assertThat(cache.getNoHits(), equalTo( operand: 0));
    assertThat(cache.getNoRequests(), equalTo( operand: 1));
}
```

Figure 4: testCache_getInfoNotAvailable test code example

- Tests to verify if the Time To Leave (TTL) cache feature is working properly; this is also done on the two different type of data to be saved.

```
@Test
void testCache_testStationsTTL() {
    String andorra = "{\"status\":\"ok\",\"data\":[{\"uid\":8411,\"aqi\":\"2
    String country = "andorra";
    cache.saveLocationsByCountry(country, andorra);
    String ret = cache.getLocationsByCountry(country);
    // double checking that before the TTL it returned
    assertThat(ret, notNullValue());

    Cache.TTL = 1;
    // if the cache doesnt return null in Cache.TTL seconds, then it fails!
    Awaitility.await().atMost(Cache.TTL, TimeUnit.SECONDS).until(
            () -> cache.getLocationsByCountry(country) == null
    );
}
```

Figure 5: testCache_testStationsTTL test code example

These tests include inserting data in the cache and retrieving it right away in order to check that it is indeed there. Then the TTL value is changed from the default bigger value to a smaller one, so that the test can be executed in a short period of time. To test this, a tool called **`Awaitility`** was used. This tool is used to check if the value returned from the cache is `null` within the defined time period, which is equal to the TTL in this case (1 second). If the cache doesn't return a `null`, then the test fails.

- Three tests were created to test the helper methods in the DataAccess class. All of them aim to check if the method is able to extract the desired information from a `json` string identical to the one it receives in the normal functioning of the system. The methods verified with tests of this type were the `getLocationInfoFromString`, `getStationInfoFromString`, and `getNameFromString`. An example testing the first method mentioned can be seen on Fig. 6.

```
@Test
void testDataAccess_getLocationInfoFromString() {
    String andorra = "{\"status\":\"ok\",\"data\":[{\"uid\":8411,\"aqi\":\"26\",\"
    Map<String, String> expected = new TreeMap<>();
    expected.put("Escaldes Engordany, Andorra", "andorra/fixa");

    Map<String, String> ret = DataAccess.getLocationInfoFromString(andorra);
    assertThat(ret, equalTo(expected));
}
```

Figure 6: testDataAccess_getLocationInfoFromString test code example

In this case, the expected result is "manually" created, and then there is a simple check comparing the result and the expected result.

→ Service Layer Testing

These tests, implemented in the file **`ServiceLayerTest.java`**, aim to verify the results of the service class of the system, the `DataAccess` class. To accomplish this, the framework Mockito is used, allowing us to mock the behaviour of the cache and the component that contacts the external API. This way it is possible and easy to test the service without inconsistent and unknown results that depend on external factors.

Several test cases were covered here. There are tests in which the service can go get the information from the cache, tests where the information has to be requested from the class that contacts the external API, and tests where the request is invalid. All these situations are performed for the three types of information we can get: the list of cities/stations, the forecast information, and the name of a city/station given its identifier. Some examples are as follows:

- In the example on Fig. 7 the cache doesn't have information about the locations in that country. This is simulated by loading the expectation that it returns null when the method `getLocationsByCountry` is called. On the other hand, the component that would normally contact the external API is loaded with an example of the expected return. Then we call the method `getLocationsByCountry` of the service and

compare it to the expected result created manually previously. The last two verifications have to do with the number of times that the methods have been called on the mocked components.

```
@Test
void whenAllGood_Locations() {
    String json = "{\"status\":\"ok\",\"data\":[{\"uid\":8411,\"aqi\":\"26\",\"time\":{\

    Map<String, String> expected_res = new TreeMap<>();
    expected_res.put("Escaldes Engordany, Andorra", "andorra/fixa");

    when ( ext.getLocationsByCountry( contains("andorra")) ).thenReturn( json );
    when ( cache.getLocationsByCountry(anyString())).thenReturn(null);

    Map<String, String> result = serv.getLocationsByCountry("andorra");
    assertEquals(expected_res, result);

    verify(ext, times( wantedNumberOfInvocations: 1)).getLocationsByCountry(anyString());
    verify(cache, times( wantedNumberOfInvocations: 1)).getLocationsByCountry(anyString());
}
```

Figure 7: whenAllGood_Locations test code example

- This next example covers the situation where the cache has the information, and thus the remote API isn't contacted. In the example in the Fig. 8 we are testing the function that returns the name of the station given its identifier. The strategy used is very similar to the test last described. The mock object is loaded with expectations and then the results are compared.

```
@Test
void whenAllGood_getStationNameCache() {
    String info_json = "{\"status\":\"ok\",\"data\":{\"aqi\":33,\"idx\":8411,\"attributions\":[
    String stationUrl = "andorra/fixa";
    when ( cache.getInfoByStation(anyString())).thenReturn(info_json);

    assertThat(serv.getNameByUrl(stationUrl), equalTo( operand: "Escaldes Engordany, Andorra"));
}
```

Figure 8: whenAllGood_getStationNameCache test code example

- Three tests were also created to verify the behavior when the request was made using bad parameters, like a station identifier that does not exist.
  - The next figure (Fig. 9) is an example of a test where a bad station identifier is being used. This example in specific is to verify the `getNameByUrl` method. A very similar test was made to verify the `getStationInfo`, since both of them should return `null` if the input data is invalid. The `info_json` variable is a string corresponding to the response we get when we do the same request on the remote API.

```java
@Test
void whenBadReq_getStationName() {
    String info_json = "{\"status\":\"error\",\"data\":{\"Unknown station\"}";
    String stationUrl = "bad-location";
    when ( ext.getInfoByStation( contains( stationUrl )) ).thenReturn( info_json );
    when ( cache.getInfoByStation(anyString())).thenReturn(null);

    assertThat(serv.getNameByUrl(stationUrl), nullValue());
}
```

Figure 9: whenBadReq_getStationName test code example

- When a country that does not produce results is used as input to get a list of cities, the return should be an empty `TreeMap`, and that is what is being tested in the next example (Fig. 10).

```java
@Test
void whenBadReq_Locations() {
    String json = "{\"status\":\"ok\",\"data\":[]}";
    when (ext.getLocationsByCountry(anyString())).thenReturn(json);
    when (cache.getLocationsByCountry(anyString())).thenReturn(null);

    assertThat(serv.getLocationsByCountry("bad-loc"), equalTo(new TreeMap<>()));
}
```

Figure 10: whenBadReq_Location test code example

In total, there are 6 tests for the service layer.

→ Integration Tests

The integration tests created are in the file `IntegrationTests.java`. REST-assured, a Java Domain Specific Language (DSL), was used in integration with Spring's MockMvc – RestAssuredMockMvc. These tests aim to verify the responses of the system's REST API. They cover situations where the parameters are invalid, where additional options are being used, and also simple usage situations.

- In this first example (Fig. 11) we are testing if he returns the correct list of cities when passing the country Andorra. First, expectations are loaded to the DataAccess service. Then, using RestAssuredMockMvc, a call to the API is simulated and it is verified if the status code is the correct and if the result parameters correspond to the desired ones.
  When the country (or any other string) considered has no associated stations, then the stations field should have size 0, but the status code remains 200. This is also confirmed in another test.
- Tests were also created in order to verify the endpoint that gives the forecast given a station identifier. One of these tests accesses the endpoint without additional parameters, with a valid station; other passes an invalid station; and two test the usage of the `type` additional parameter: one with a valid type and the other with an invalid one. This last one can be seen in Fig. 12. We can see a function `configureBehaviourInfoAndorra` in this method. It Is simply a helper function that loads the service mock with expectations about the station in Andorra.

```
@Test
void givenCountry_thenReturnStations() {
    String country = "andorra";
    Mockito
            .when(dataAcc.getLocationsByCountry( anyString() ))
            .thenReturn(new HashMap<>() {{
                put("Escaldes Engordany, Andorra", "andorra/fixa");
            }});

    RestAssuredMockMvc
            .given() MockMvcRequestSpecification
            .when().get( path: "/api/stations?country="+country) MockMvcResponse
            .then() ValidatableMockMvcResponse
                .log().all()
                .statusCode(200)
                .body( path: "country", Matchers.equalTo(country))
                .body( path: "stations.size()", is( value: 1))
                .body( path: "stations[0].name", Matchers.equalTo( operand: "Escaldes Engordany, Andorra"))
                .body( path: "stations[0].uri", Matchers.equalTo( operand: "andorra/fixa"))
                ;
}
```

Figure 11: givenCountry_thenReturnStations test code example

```
@Test
void givenStationAndBadType_thenReturnHttpStatus400() {
    String stationuri = "andorra/fixa";
    String t = "bad-type";
    configureBehaviourInfoAndorra();
    RestAssuredMockMvc
            .given() MockMvcRequestSpecification
            .when().get( path: "/api/forecast/?station="+stationuri+ "&type=" + t) MockMvcResponse
            .then() ValidatableMockMvcResponse
            .log().all()
            .statusCode(400);
}
```

Figure 12: givenStationAndBadType_thenReturnHttpStatus400 test code example

- The last test of this kind has the goal of testing the API endpoint referred to the cache. Here, expectations are loaded into the cache mock, and then they are compared to the results gotten from the simulated API request.

```java
@Test
void test_getCacheStats() {
    Mockito
            .when(cache.getNoHits())
            .thenReturn(1);
    Mockito
            .when(cache.getNoMisses())
            .thenReturn(2);
    Mockito
            .when(cache.getNoRequests())
            .thenReturn(3);

    RestAssuredMockMvc
            .given() MockMvcRequestSpecification
            .when().get( path: "/api/cache") MockMvcResponse
            .then() ValidatableMockMvcResponse
            .log().all()
            .statusCode(200)
            .body( path: "hits", is( value: 1))
            .body( path: "misses", is( value: 2))
            .body( path: "requests", is( value: 3))
            ;
}
```

Figure 13: test_getCacheStats test code example

2.6 Functional testing

Functional testing on the web interface were made using the Selenium IDE Chrome browser extension. Then this was exported into Java code and placed in the project, adapting some parts to match the versions of the technologies used. An `implicit wait` of 10 seconds was used to allow the page to fully load before testing if an element was present. This code is on the file `WebInterfaceTest.java`.

2.7 Static code analysis

The tool used to analyze the code was Sonarqube. Fig. 14 is a screenshot of the project dashboard, corresponding to the last version of the system. The quality gate used has been modified so that the coverage on the new code can be only of 50% instead of the default 80%. Code coverage is being assessed through the use of the JaCoCo tool. Because this is a small project, the default threshold of 80% proved to be very hard to fulfill.

A big part of the code smells reported are related to comments that should be removed. However, some of these comments are only indications to help readability, so they should be kept.

Sonarqube reported no bugs, no vulnerabilities, and no security hazards. From the 12 code smells, only 2 of them are major, and both are related to the comments as already discussed above. 8 of them are considered Minor, and these include changing the way some arrays are

created. Even though I tried to change this, I could not find a way, since these arrays are used to populate `Maps`.

There appeared some code smells that I wouldn't have otherwise realized that were (potential) problems. For example, it was advised to use `try-with-resources` instead of the typical `try/catch/finally` classic way. This was a concept that I wasn't even aware that existed.

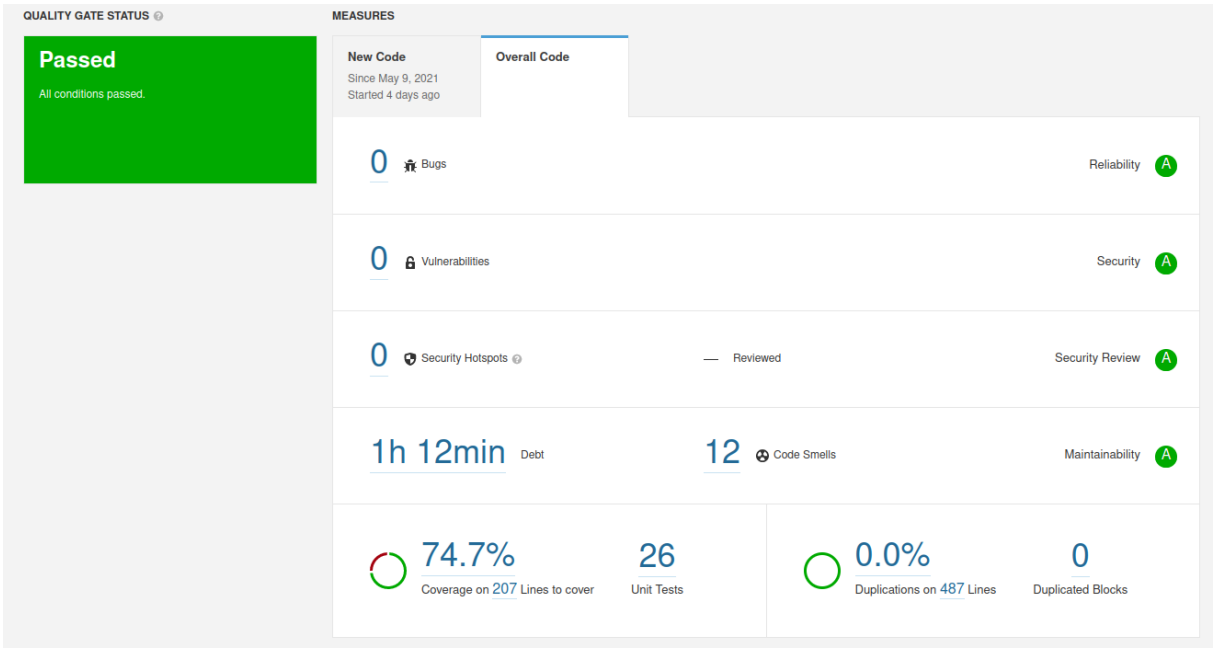


Figure 14: Sonarqube project dashboard

2.8 Continuous integration pipeline

A CI pipeline was implemented using Jenkins. The pipeline implemented is defined in the Jenkinsfile shown in Fig. 15. This pipeline was configured with an automated polling strategy that polls the git repository every 10 minutes.

```
pipeline {
    agent any
    tools {
        jdk 'jdk11'
        maven 'mvn'
    }
    stages {
        stage('install') {
            steps {
                dir('./HW/air-quality') {
                    sh 'mvn clean install'
                }
            }
            post {
                always {
                    junit '**/target/*-reports/TEST-*.xml'
                }
            }
        }
    }
}
```

Figure 15: Jenkinsfile

In Fig. 16 we can see the current Jenkins dashboard of the project.

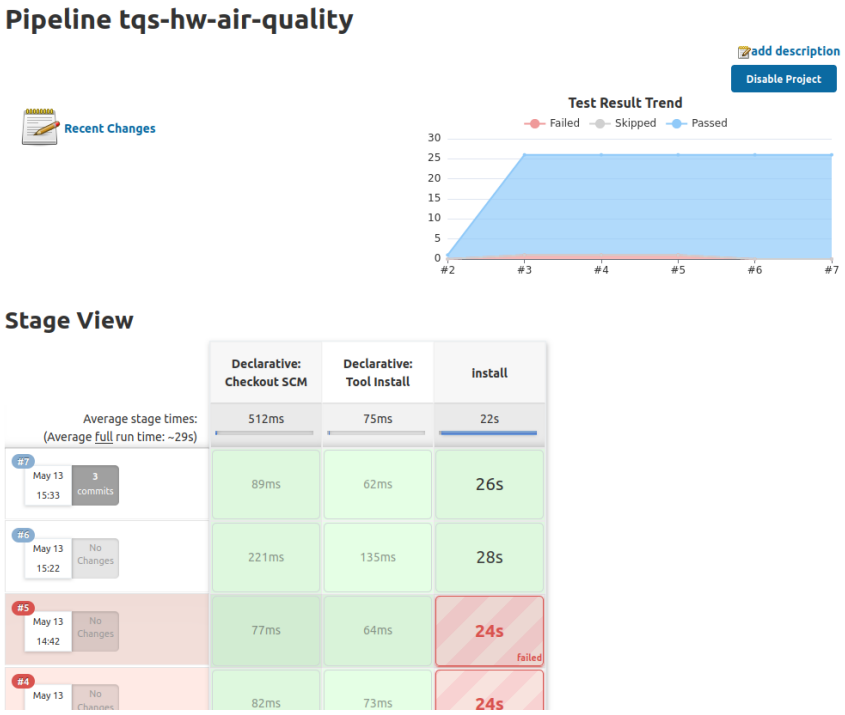


Figure 16: Screenshot of Jenkins project dashboard

3References & resources

Project resources

- **Video demo:** is available in the repository (https://github.com/marianasps/TQS-UA/blob/main/HW/demo.mp4)

Reference materials

https://www.baeldung.com/hamcrest-collections-arrays
https://www.baeldung.com/spring-mock-mvc-rest-assured
https://www.baeldung.com/spring-response-entity
https://www.baeldung.com/rest-assured-response
https://www.baeldung.com/rest-assured-tutorial
https://www.baeldung.com/log4j2-appenders-layouts-filters
https://www.baeldung.com/spring-boot-logging