HW1: Mid-term assignment report

David Gomes Morais [93147], v2021-05-12

1 Introduction

1.1 Overview of the work

This report presentes the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy. The developed application was AirQuality whose main goal is to provide the client with valuable air quality information of a specific region. In addition to that, it provides and uses its own API that makes requests of the data to an external, already existing API.

1.2 Current limitations

In retrospect, the main limitations of the application as of yet are the absence of air quality forecast of a specific city as well as, in terms of data consistency, once that a user sees the search results and until it load the city info of one of those results, the city could have been already deleted in the external API.

2 Product specification

2.1 Functional scope and supported interactions

As mentioned in the previous section, both a web application and an API were developed in the course of this midterm assignment, this means that a user can both use the web page and the API to search and access air quality data, as well as some stats in regards to the built-in applications's cache.

This being said, the main scenarios work when the actor interacts either with the webpage or with the API (being that the latter is more geared towards developers) and are as follows:

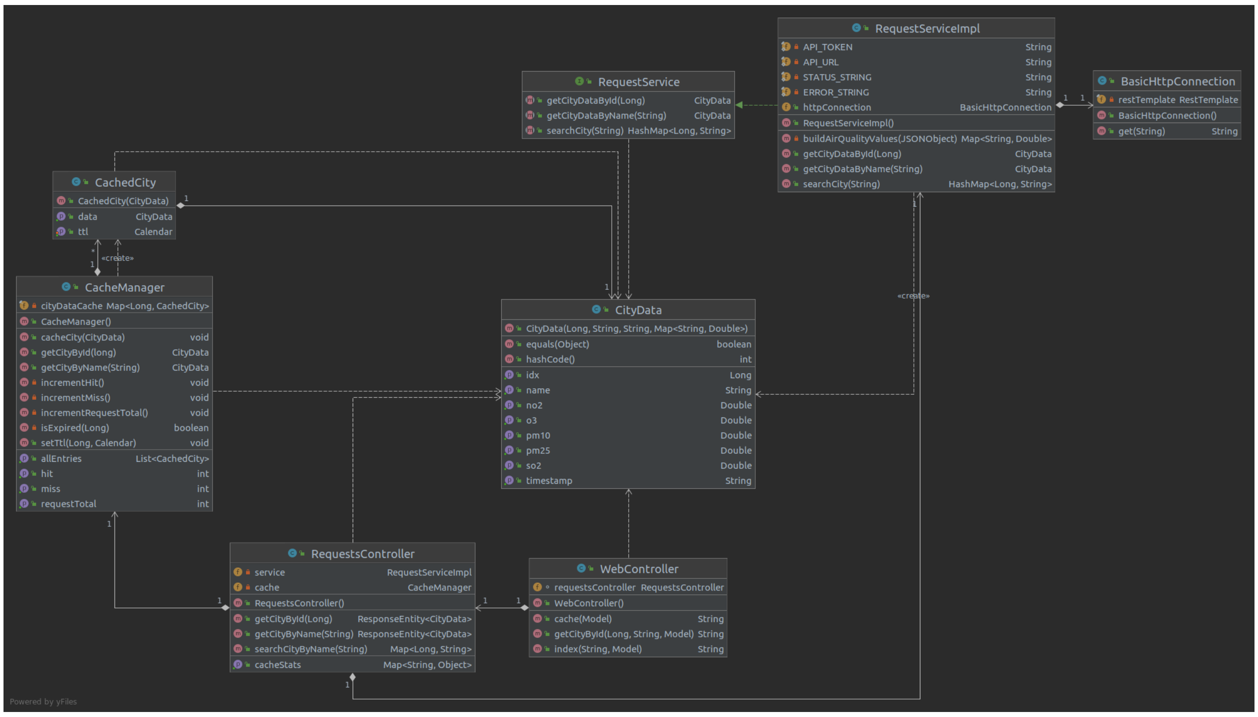
- The user can search for a city by its name, and the system provides a list of cities that match the search query, as well as the associated city id;
- The user can get information in regards of the air quality of a specific city/weather station by its id, and the system will return all the available data;
- The user can get information about the application cache, and the system will return the total number of requests, the number of cache hits, the number of cache misses as well as all the city data currently being stored in said cache.

In addiction, there is also one more scenario specific to the API:

- The user can get information about the air quality of a city by its name, and the system will return all the available data.

It is worth mentioning that this scenario was not included in the web application, once it became redundant because in the UI the user has to first search for the city and only upon selecting it in the search result can then see the information about that city.

2.2 System architecture



RequestServiceImpl
API_TOKEN String
API_URL String
STATUS_STRING String
ERROR_STRING String
httpConnection BasicHttpConnection
RequestServiceImpl()
buildAirQualityValues(JSONObject) Map<String, Double>
getCityDataById(Long) CityData
getCityDataByName(String) CityData
searchCity(String) HashMap<Long, String>
BasicHttpConnection
restTemplate RestTemplate
BasicHttpConnection()
get(String) String
RequestService
getCityDataById(Long) CityData
getCityDataByName(String) CityData
searchCity(String) HashMap<Long, String>
CachedCity
CachedCity(CityData)
data CityData
ttl Calendar
«create»
CacheManager
cityDataCache Map<Long, CachedCity>
CacheManager()
cacheCity(CityData) void
getCityById(long) CityData
getCityByName(String) CityData
incrementHit() void
incrementMiss() void
incrementRequestTotal() void
isExpired(Long) boolean
setTtl(Long, Calendar) void
allEntries List<CachedCity>
hit int
miss int
requestTotal int
CityData
CityData(Long, String, String, Map<String, Double>)
equals(Object) boolean
hashCode() int
idx Long
name String
no2 Double
o3 Double
pm10 Double
pm25 Double
so2 Double
timestamp String
«create»
RequestsController
service RequestServiceImpl
cache CacheManager
RequestsController()
getCityById(Long) ResponseEntity<CityData>
getCityByName(String) ResponseEntity<CityData>
searchCityByName(String) Map<Long, String>
cacheStats Map<String, Object>
WebController
requestsController RequestsController
WebController()
cache(Model) String
getCityById(Long, String, Model) String
index(String, Model) String
Powered by yFiles

Figure 1: Class Diagram of AirQuality web application

The project was developed in SpringBoot and Thymeleaf and, instead of using an in-memory database to store the cached cities as persistent data, a simpler solution using a HashMap was implemented. The communication between the frontend and the backend was through a Controller (WebController) that, in turn, calls the RestController of the API (RequestController).

The communication between the system's API and the external one is done via a RequestServiceImpl, that implements from an interface called RequestService. This allows it to easily fetch data from the external data source as well as making the system more flexible if the service were to be replaced to handle another API. This system also contains a Basic Http Connection, that executes the GET requests through a RestTemplate.

There is also a cache manager that deals with the retrieving and management of objects in cache as well as the expirations, and this management class is initialized and called in the RestController itself, where first the system verifies if the requested city is in the cache and, if no data is found, then the external API is called.

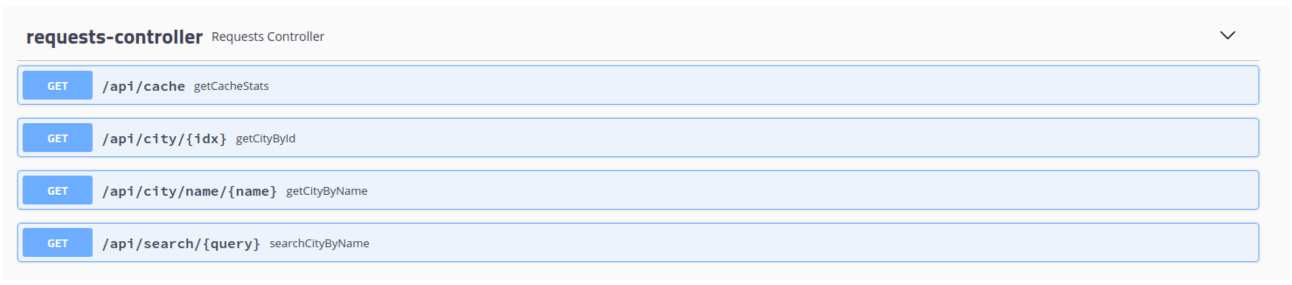
Furthermore there are two entities, one that contains all the information about the air quality of a city (CityData) and another one that is stored in cache, with a time to live associated that represents the time at which the entity was cached, and it is also used to verify if a cache's entry is valid or expired.

2.3 API for developers

The API for developers can be divided into three main functionalities: get air quality data, search for cities and get cache statistics. In the first one, there are two endpoints available, one to get a city data based on their unique city id (/api/city/{idx}) that receives a required parameter idx, of the type Long, representative of the city id, and another one based on the city name (/api/city/name/{name}) that receives a required parameter String representative of the city name. As mentioned previously, only one of these endpoints is used on the web application.

The search endpoint is /api/search/{query}, that receives a required String parameter that is the search query to execute, and it is through this endpoint that a developer can get a list of cities that match the parameter query, as well as the city id. Lastly, there is also an endpoint to get some cache statistics (/api/cache) that retrieves useful information such as hits, misses, requests number and all cached cities in the API.

The documentation of this APi was made using Swagger2 and it is available at /swagger-ui.html. The following image contains the endpoints available listed in the Request Controller section of the API documentation.



requests-controller Requests Controller

GET /api/cache getCacheStats

GET /api/city/{idx} getCityById

GET /api/city/name/{name} getCityByName

GET /api/search/{query} searchCityByName

Figure 2 - Request Controller endpoints in Swagger2 documentation

3 Quality assurance

3.1 Overall strategy for testing

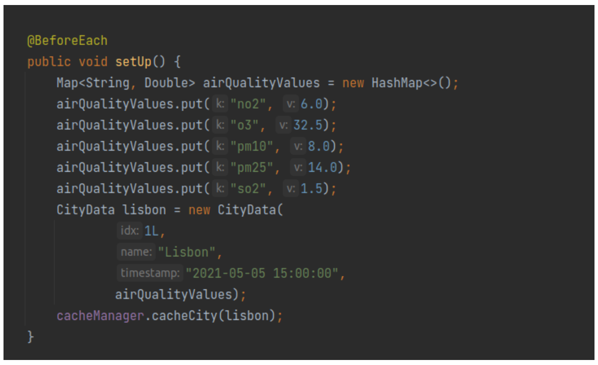
The overall strategy for test used was Test Driven Development, due to it being the more viable option, even more so with SonarQube (that was implemented early on) providing important feedback on the coverage and lack thereof, as well as in code quality, making refactoring easier. The tests implemented range from unit tests using Junit and mock tests with Mockito to integration tests using webMVC and Selenium functional tests.

3.2 Unit and integration testing

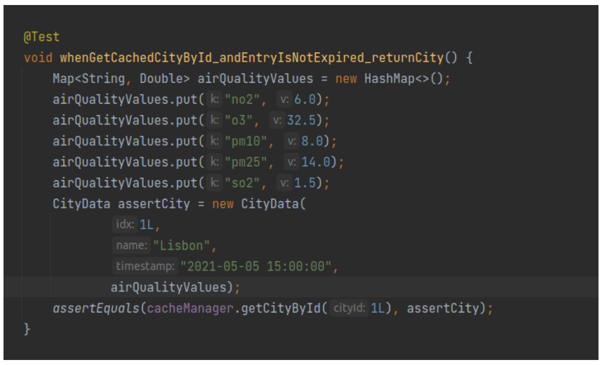
Unit tests were implemented to verify the business logic in the cache manager class, and were achieved by using JUnit. Test cases considered include:

- getting a city in cache by id if the entry is not expired, should return that city's data;
- getting a city in cache by id if the entry does not exist or the entry is expired, should return null;
- getting a city in cache by name if the entry is not expired should return that city's data;
- getting a city in cache by name if the entry does not exist or is expired, should return null;
- assert if the cache statistics (hits, misses, request number) are being incremented as they should;
- assert if, when we pull all cities from cache, a list is retrieved with no expired entries.

All these relied on the same strategy: adding a city to cache on the set up method before each test and then proceed to test the scenarios listed above by asserting the results, as the code snippet below exemplifies.



```java
@BeforeEach
public void setUp() {
    Map<String, Double> airQualityValues = new HashMap<>();
    airQualityValues.put( k: "no2",  v: 6.0);
    airQualityValues.put( k: "o3",  v: 32.5);
    airQualityValues.put( k: "pm10",  v: 8.0);
    airQualityValues.put( k: "pm25",  v: 14.0);
    airQualityValues.put( k: "so2",  v: 1.5);
    CityData lisbon = new CityData(
            idx: 1L,
            name: "Lisbon",
            timestamp: "2021-05-05 15:00:00",
            airQualityValues);
    cacheManager.cacheCity(lisbon);
}
```



```java
@Test
void whenGetCachedCityById_andEntryIsNotExpired_returnCity() {
    Map<String, Double> airQualityValues = new HashMap<>();
    airQualityValues.put( k: "no2",  v: 6.0);
    airQualityValues.put( k: "o3",  v: 32.5);
    airQualityValues.put( k: "pm10",  v: 8.0);
    airQualityValues.put( k: "pm25",  v: 14.0);
    airQualityValues.put( k: "so2",  v: 1.5);
    CityData assertCity = new CityData(
            idx: 1L,
            name: "Lisbon",
            timestamp: "2021-05-05 15:00:00",
            airQualityValues);
    assertEquals(cacheManager.getCityById( cityId: 1L), assertCity);
}
```

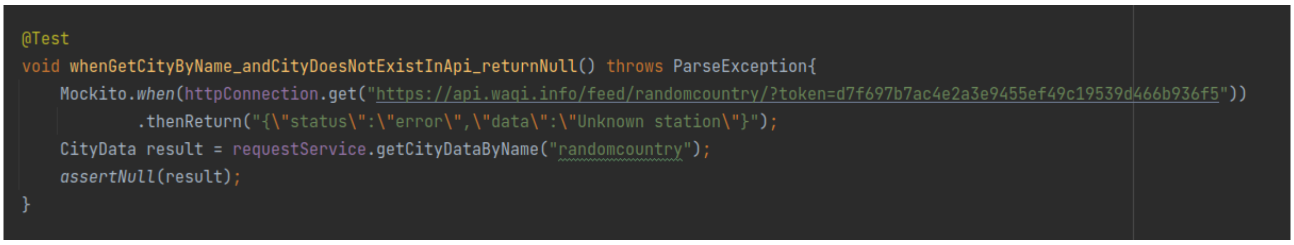
Figure 3 - CacheManager_UnitTest.java setup and test example.

To test the request service, a mock test was implemented using mockito, where we mock the external API and based on those mocked responses assert if the service is performing both the search and the retrieval of information correctly. These service mock tests include:

- getting an existing city by id and by name, should return that city's data;
- getting a non existing city by id and by name, should return null;
- getting a city by name and by id with empty air quality fields, should return that city's data where some or all fields related to air quality are null;
- searching for a city and if there are results, then it should return a Hashmap with the cities id and the cities name;
- searching for a city and if there are no results, then it should return an empty Hashmap;

- searching for a city and if an internal error occurs (such as token expired or request quota surpassed), then it should return null.

These tests were tackled by first mocking the httpConnection *get* method to retrieve a fixed response and then testing the request service methods, as it can be seen below.



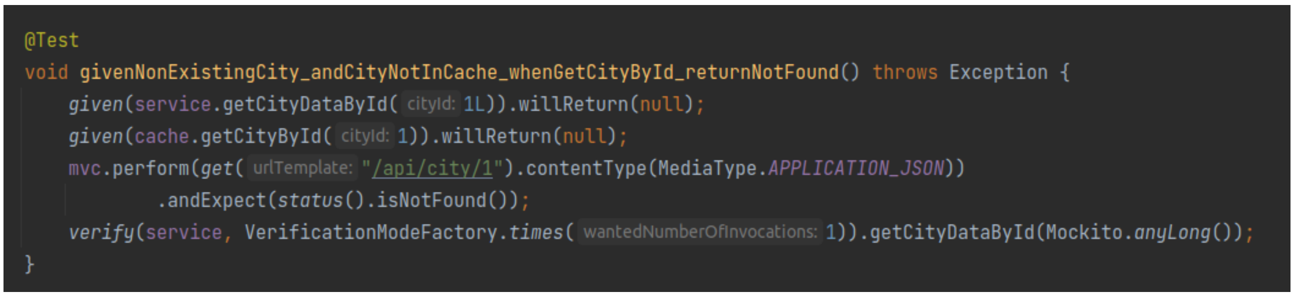
```java
@Test
void whenGetCityByName_andCityDoesNotExistInApi_returnNull() throws ParseException{
    Mockito.when(httpConnection.get("https://api.waqi.info/feed/randomcountry/?token=d7f697b7ac4e2a3e9455ef49c19539d466b936f5"))
            .thenReturn("{\"status\":\"error\",\"data\":\"Unknown station\"}");
    CityData result = requestService.getCityDataByName("randomcountry");
    assertNull(result);
}
```

Figure 4 - TestService_withMockProvider.java test example

To test the integration of the Request Controller it was used a WebMvcTest and, to make it apply only to the controller, some dependencies were mocked using @MockBean such as the service and the cache manager. These controller tests' scenarios include:

- getting a city by id and by name that does not exist on cache, should return that city's data fetched from the external API;
- getting a city by id and by name that exists on cache, should return that city's data fetched from cache;
- getting a non existing city by id and by name, should return with a status of 404 Not Found;
- getting the cache stats when the cache is empty, should return cache stats with an empty InCache list;
- getting the cache stats when there is cached data, should return cache stats with a InCache list of cities stores in cache in that moment;
- searching for a city when there are results should return a Hashmap with the matching cities id and name, and if there are no results should return an empty Hashmap;
- searching for a city without a search query, should return with a status of 404 Not Found.

The general testing strategy used for the controller was mocking the service and cache dependency, and then using mvc.perform() method to test the expected results, as well as verifying the number of calls to a service's method, exemplified below.



```java
@Test
void givenNonExistingCity_andCityNotInCache_whenGetCityById_returnNotFound() throws Exception {
    given(service.getCityDataById( cityId: 1L)).willReturn(null);
    given(cache.getCityById( cityId: 1)).willReturn(null);
    mvc.perform(get( urlTemplate: "/api/city/1").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isNotFound());
    verify(service, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).getCityDataById(Mockito.anyLong());
}
```

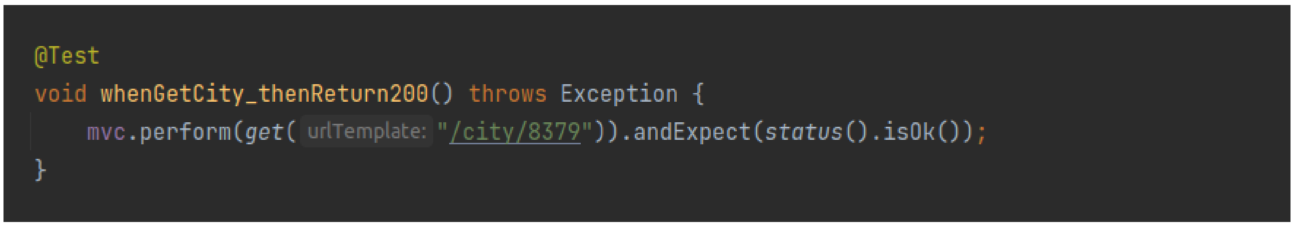
Figure 5 - An example of a test to RequestController

Lastly, there are simple integration tests on the WebController class, where MockMvc was used too, but without mocked dependencies. These tests simply verify if the response status of the web controller is 200 (OK) in the following cases:

- when the index page is displayed, and when a search is executed in that page;

- when a city's air quality information page is displayed, and when a search is executed in that page;
- when the cache statistics page is displayed.

These tests were mainly to see if the controller responds correctly to the calls and assuring that the response status is always the correct one, the remaining tests were made with Selenium and are described in the following section.



```java
@Test
void whenGetCity_thenReturn200() throws Exception {
    mvc.perform(get( urlTemplate: "/city/8379")).andExpect(status().isOk());
}
```

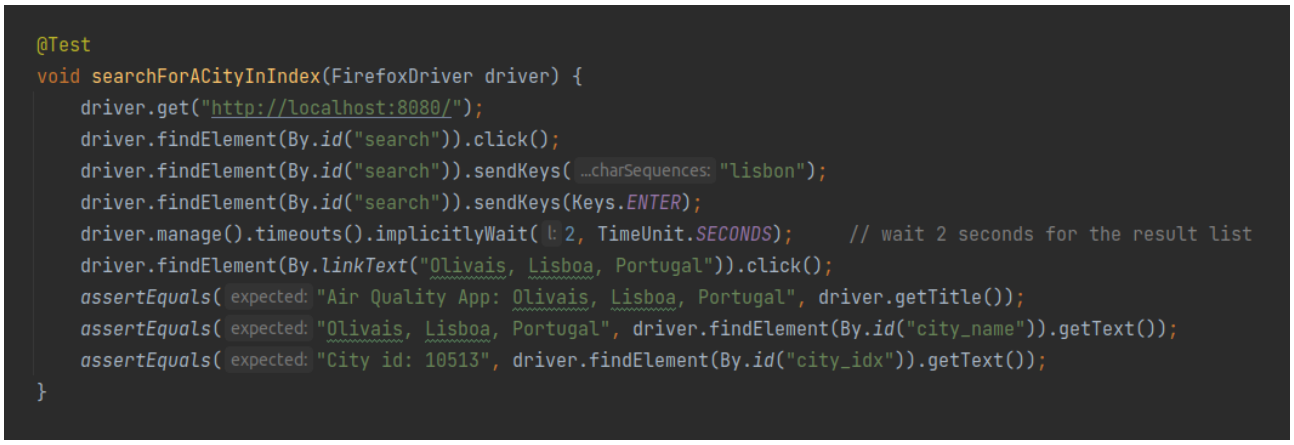
Figure 6 - Example of the Integration tests on the WebController

3.3 Functional testing

Functional tests were done using SeleniumIDE, where a Selenium project was created and the interaction with the frontend were recorded and then exported to a java class, and consist in three mais tests: search for a city in the index page, search for a city in another city's information page and getting the cache statistics.

In the first one, we load the index page, search for "Lisbon" in the search bar and select a result. Similarly, in the second one we load the browser already in a city's information page and proceed to click the search bar, typing "London", and selecting a result. Lastly, we load the index page, click on the cache info button and are redirected to the cache statistics page.

In all the mentioned tests, when the desired page is reached, some basic information that should be presented is asserted such as the page title and some content, which is not based on any air quality values once these are prone to change as the days go by. Furthermore it is worth mentioning that, if the external API for some reason removes the cities tested in these functional tests, they will fail.

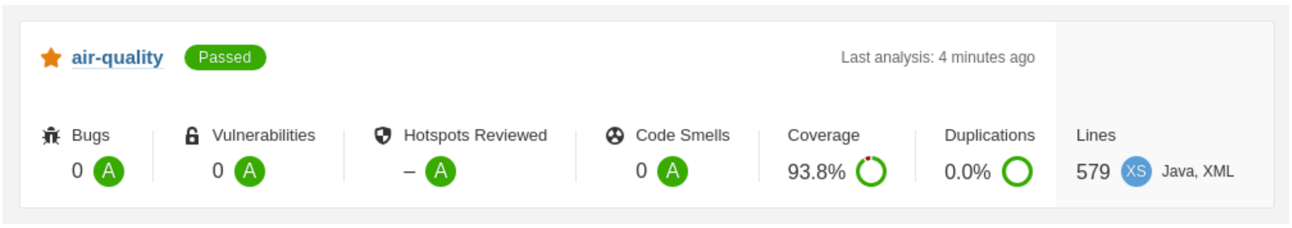


```java
@Test
void searchForACityInIndex(FirefoxDriver driver) {
    driver.get("http://localhost:8080/");
    driver.findElement(By.id("search")).click();
    driver.findElement(By.id("search")).sendKeys( ...charSequences: "lisbon");
    driver.findElement(By.id("search")).sendKeys(Keys.ENTER);
    driver.manage().timeouts().implicitlyWait( l: 2, TimeUnit.SECONDS);     // wait 2 seconds for the result list
    driver.findElement(By.linkText("Olivais, Lisboa, Portugal")).click();
    assertEquals( expected: "Air Quality App: Olivais, Lisboa, Portugal", driver.getTitle());
    assertEquals( expected: "Olivais, Lisboa, Portugal", driver.findElement(By.id("city_name")).getText());
    assertEquals( expected: "City id: 10513", driver.findElement(By.id("city_idx")).getText());
}
```

Figure 7 - Selenium test example for a search on the home page

3.4 Static code analysis

For static code analysis the chosen tool was SonarQube running locally due to it being used in the lab classes, in conjunction with JaCoCo to generate the reports on the coverage. This allowed not only to highlight code smells, bugs and vulnerabilities, but also which lines of code and conditions were not being covered by tests, which greatly simplified the whole process.



air-quality Passed
Last analysis: 4 minutes ago

Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines
0 A	0 A	– A	0 A	93.8%	0.0%	579 XS Java, XML

Figure 8 - SonarQube project overview

Despite passing all the quality gates imposed by SonarQube, some lines are not covered mostly in the AirQualityApplication class, in the ErrorController and in the equals method of the CityData entity. Likewise there are some conditions not covered on the request service, due to the fact that realistically they would never happen, taking into consideration the response structure of the external API used.

4 References & resources

Project resources

- Video demo: github.com/davidgmorais/air-quality/blob/main/AirQualityDemo.mp4.
- The source code of the application is in the project's github repository (github.com/davidgmorais/air-quality). It was not deployed but it could easily be done using a docker container.
- QA dashboard: As mentioned before, SonarQube was runned locally in opposition to SonarCloud.

Reference materials

API documentation;

JUnit5 and Mockito documentation;

TQS lab activities;

Setting Up Swagger 2 with a Spring REST API.