HW1: Mid-term assignment report

Mário Francisco Costa Silva [93430], v2021-05-07

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 application built for this project is named AirCheck, a place where a person can easily find out about the current air quality of a location, it also provides an API for users that might want to use it in their applications/services.

1.2 Current limitations

One of the biggest limitations of this product is the cities list. The cities are loaded from a CSV the first time a request is made to the service and loaded into an in-memory database. They are only cities from Portugal because it would take too much time to load a CSV much bigger, also the API does not provide any pagination feature for this list, if the API were to have more cities, later on, a pagination feature would be necessary for the API.

Another limitation is the small number of use cases it has. Implementing another use case in this API would be simple given the code organization and that new service would be very similar to the existing services already provided.

2 Product specification

2.1 Functional scope and supported interactions

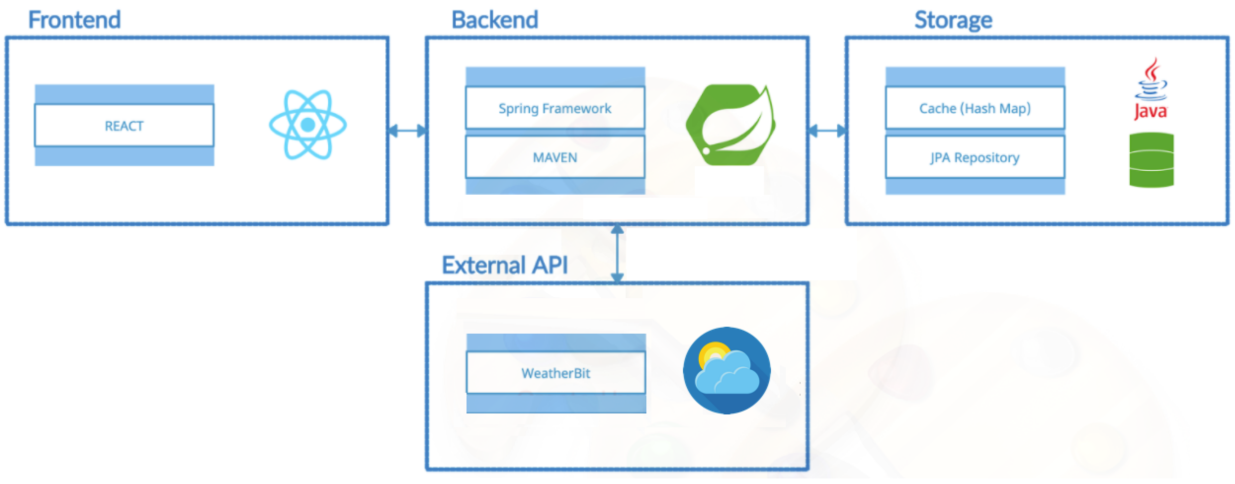
A user can type on a search bar or browse through a list of cities and interact with them to see the air indicators of that city.

Every request made to the API is cached to provide a faster response time in data retrieval, users can also verify cache statistics such as the number of requests, hits, misses, and the time-to-live of requests.

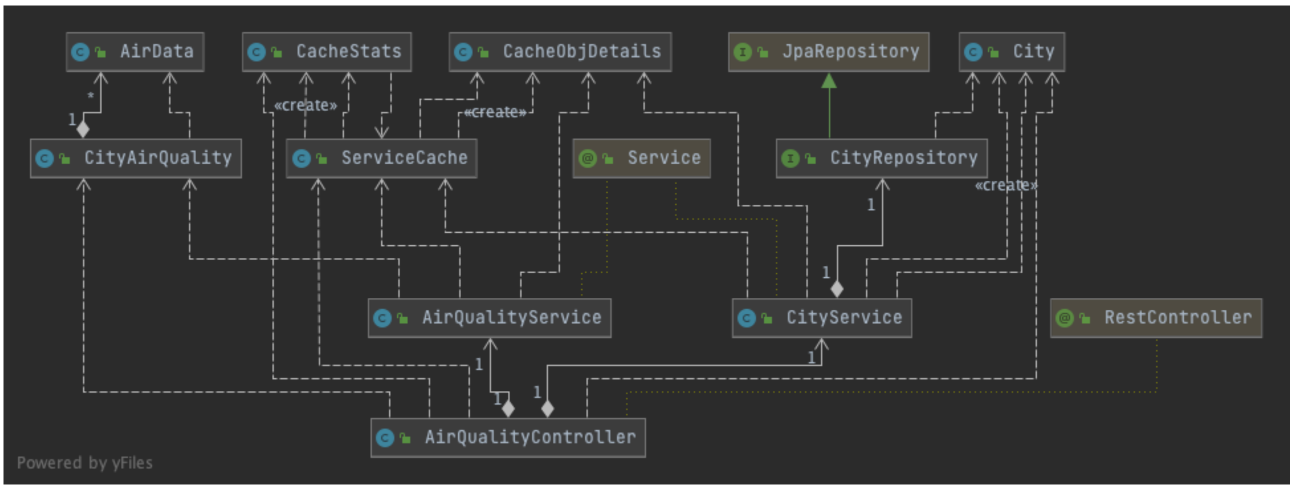
One use case for this application would be, for example, Pedro is a scientist that is investigating how pollution is traveling through a region and wants to find some pollution hot spots so he started monitoring the air quality of these regions through this product.

2.2 System architecture

The frontend of this application was built with React that makes HTTP requests to the backend API. The backend was developed with Spring Framework and Maven for dependency injection. The cache is a hashmap stored in memory that maps the request made to the output value of that request. For getting the air quality it was used an external API Weatherbit (https://www.weatherbit.io/).



The backend structure followed the MVC architecture, with a Controller receiving the requests from the clients and calling the corresponding services/functions, then returns to the user a response accordingly to what was returned from the services.

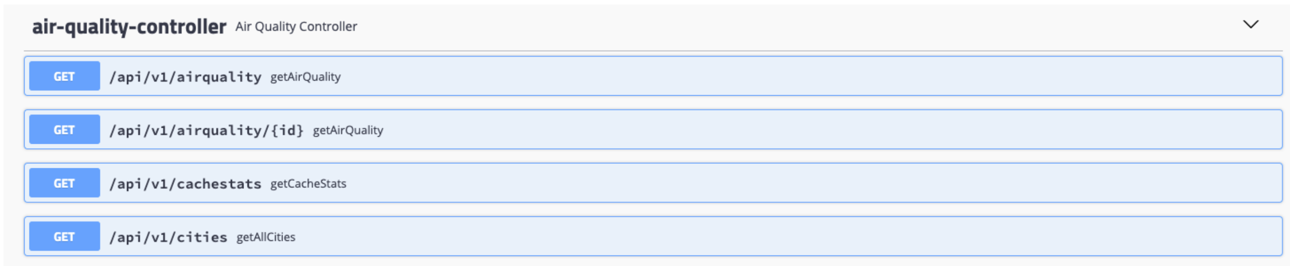


2.3 API for developers

This application has a simple API for developers, and it is documented with the help of the Swagger tool.

There's an endpoint to get the air quality of a city by providing a parameter 'city' and it also allows to provide another optional parameter 'country'. Then there is another endpoint that also allows to get the air quality of a city but by providing the path variable id, this id can be obtained from the Weatherbit API that contains a dataset with the cities and their corresponding ids, or there is also an option in this API to get the cities, however, we only provide 112 cities of Portugal since the number of cities is really large.

Finally, there's an endpoint for getting the cache statistics of the application like the number of hits, requests, misses, and the time to live of requests.


air-quality-controller Air Quality Controller

GET /api/v1/airquality getAirQuality

GET /api/v1/airquality/{id} getAirQuality

GET /api/v1/cachestats getCacheStats

GET /api/v1/cities getAllCities

3 Quality assurance

3.1 Overall strategy for testing

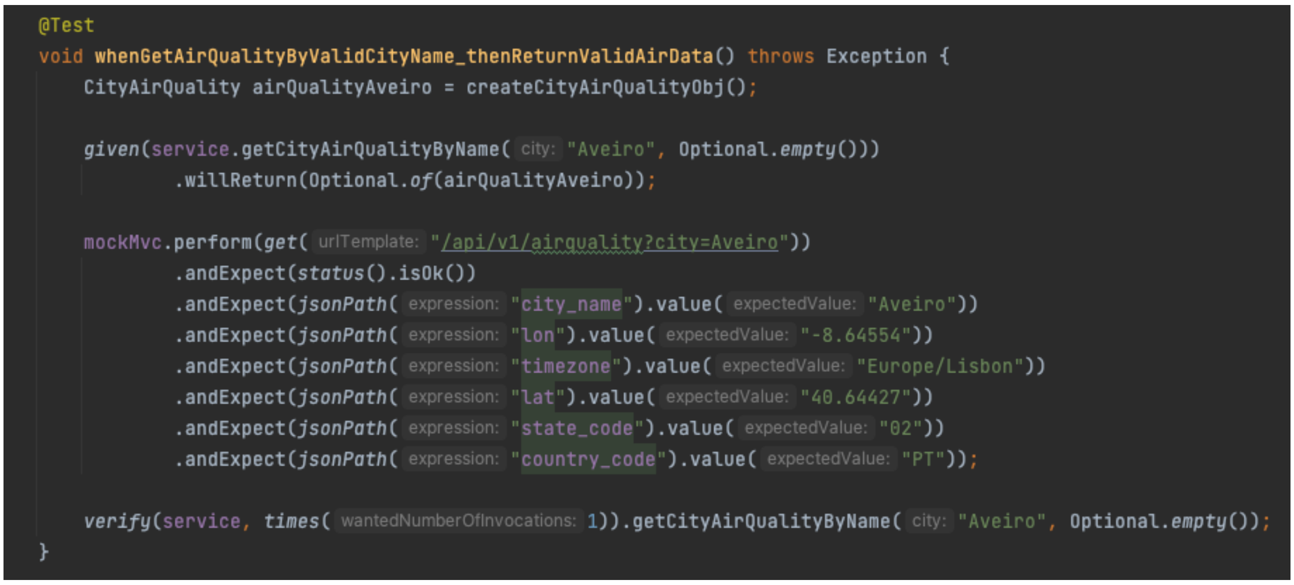
The strategy followed for this project was a little of TDD. Initially, I did tests to a controller and a service, only after I started to code the implementation to pass the requirements of the tests developed. This way it allowed me to build code with a proper structure and also be certain of the code implemented.

I mixed up different testing tools, I used Cucumber features with selenium for functional testing, REST-Assured for Integration testing, and for unit tests JUnit, Mockito, and MockMvc.

3.2 Unit and integration testing

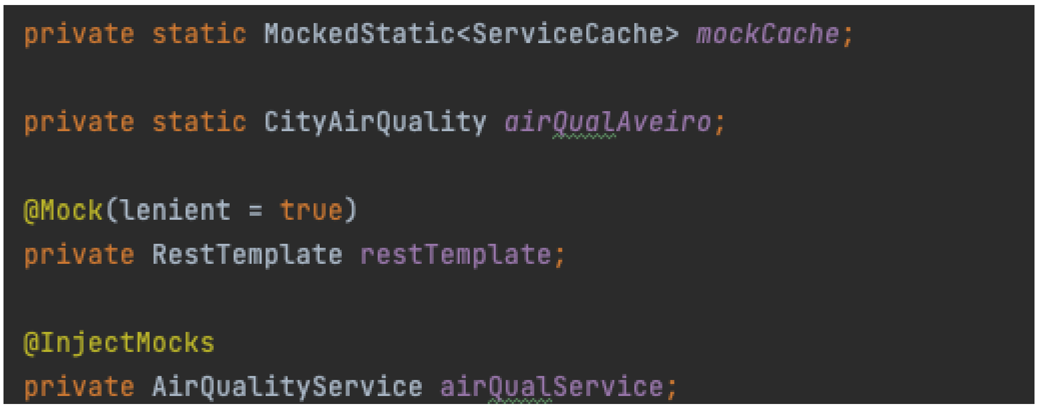
I decided not to do unit tests on models since they don't have functionalities that are critical for the other classes.

For the Controller, I did tests to the endpoints with MockMvc and mocked what the services returned to be able to check if the controller returns accordingly and only executes the functions necessary.



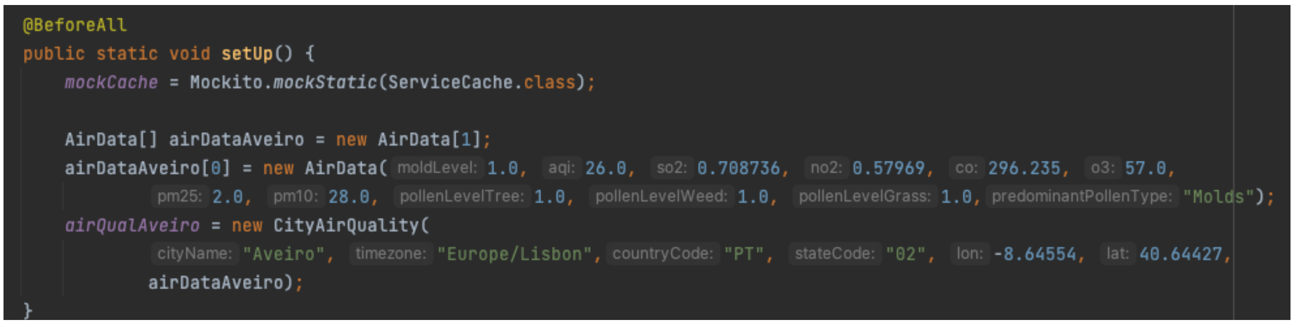
```
@Test
void whenGetAirQualityByValidCityName_thenReturnValidAirData() throws Exception {
    CityAirQuality airQualityAveiro = createCityAirQualityObj();

    given(service.getCityAirQualityByName( city: "Aveiro", Optional.empty()))
            .willReturn(Optional.of(airQualityAveiro));

    mockMvc.perform(get( urlTemplate: "/api/v1/airquality?city=Aveiro"))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "city_name").value( expectedValue: "Aveiro"))
            .andExpect(jsonPath( expression: "lon").value( expectedValue: "-8.64554"))
            .andExpect(jsonPath( expression: "timezone").value( expectedValue: "Europe/Lisbon"))
            .andExpect(jsonPath( expression: "lat").value( expectedValue: "40.64427"))
            .andExpect(jsonPath( expression: "state_code").value( expectedValue: "02"))
            .andExpect(jsonPath( expression: "country_code").value( expectedValue: "PT"));

    verify(service, times( wantedNumberOfInvocations: 1)).getCityAirQualityByName( city: "Aveiro", Optional.empty());
}
```

For the Air Quality Service, I had to mock the cache to be able to control what it returns, and since it is a static object I had to use MockedStatic.



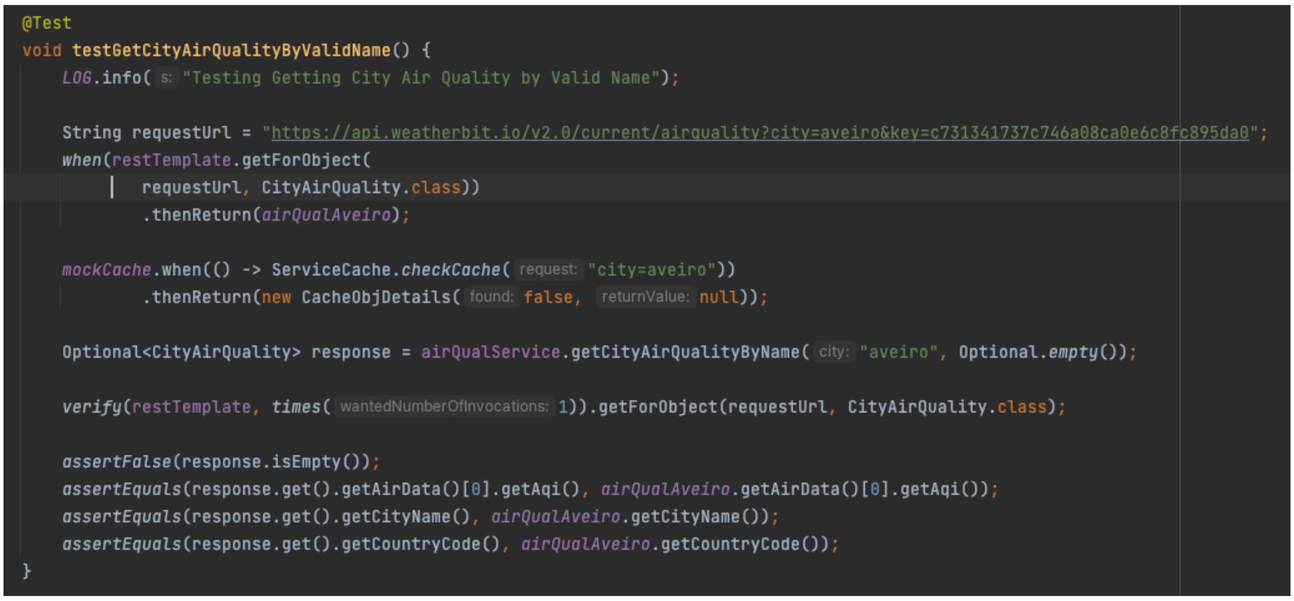
```
private static MockedStatic<ServiceCache> mockCache;

private static CityAirQuality airQualAveiro;

@Mock(lenient = true)
private RestTemplate restTemplate;

@InjectMocks
private AirQualityService airQualService;
```

Before doing the tests I initialize the cache mock and also create an object that I want to use.



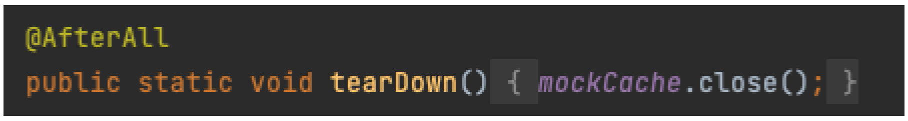
```
@BeforeAll
public static void setUp() {
    mockCache = Mockito.mockStatic(ServiceCache.class);

    AirData[] airDataAveiro = new AirData[1];
    airDataAveiro[0] = new AirData( moldLevel: 1.0,  aqi: 26.0,  so2: 0.708736,  no2: 0.57969,  co: 296.235,  o3: 57.0,
            pm25: 2.0,  pm10: 28.0,  pollenLevelTree: 1.0,  pollenLevelWeed: 1.0,  pollenLevelGrass: 1.0,  predominantPollenType: "Molds");
    airQualAveiro = new CityAirQuality(
            cityName: "Aveiro",  timezone: "Europe/Lisbon", countryCode: "PT",  stateCode: "02",  lon: -8.64554,  lat: 40.64427,
            airDataAveiro);
}
```

Then I test the service by returning the object created before when it makes a call to the external API and also when it verifies the cache to return empty, after that, I check if the service behaved properly and returned the right object.



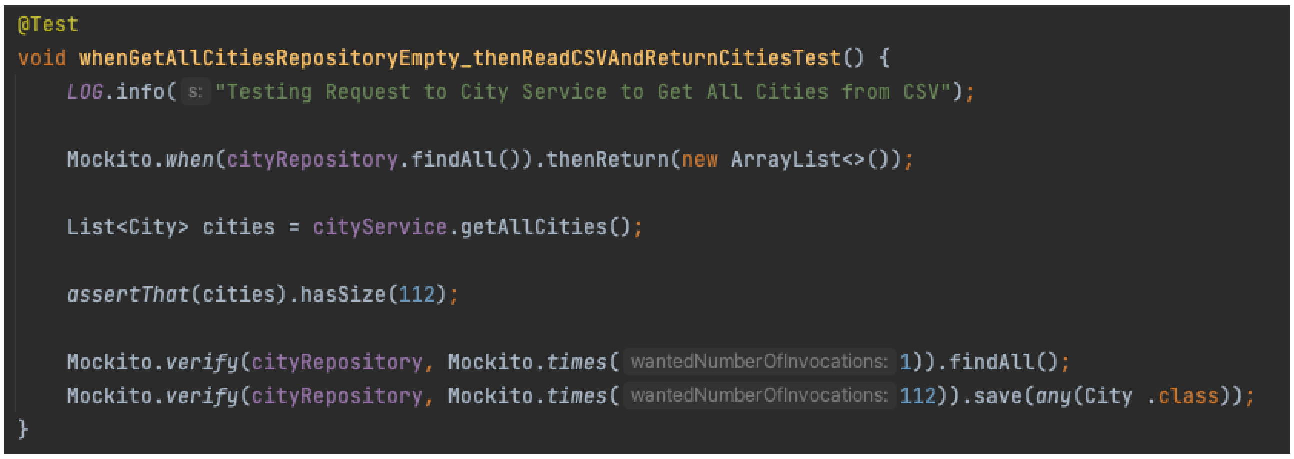
```java
@Test
void testGetCityAirQualityByValidName() {
    LOG.info( s: "Testing Getting City Air Quality by Valid Name");

    String requestUrl = "https://api.weatherbit.io/v2.0/current/airquality?city=aveiro&key=c731341737c746a08ca0e6c8fc895da0";
    when(restTemplate.getForObject(
            requestUrl, CityAirQuality.class))
            .thenReturn(airQualAveiro);

    mockCache.when(() -> ServiceCache.checkCache( request: "city=aveiro"))
            .thenReturn(new CacheObjDetails( found: false,  returnValue: null));

    Optional<CityAirQuality> response = airQualService.getCityAirQualityByName( city: "aveiro", Optional.empty());

    verify(restTemplate, times( wantedNumberOfInvocations: 1)).getForObject(requestUrl, CityAirQuality.class);

    assertFalse(response.isEmpty());
    assertEquals(response.get().getAirData()[0].getAqi(), airQualAveiro.getAirData()[0].getAqi());
    assertEquals(response.get().getCityName(), airQualAveiro.getCityName());
    assertEquals(response.get().getCountryCode(), airQualAveiro.getCountryCode());
}
```

After the tests, the cache mock has to be closed since it is a static mock.



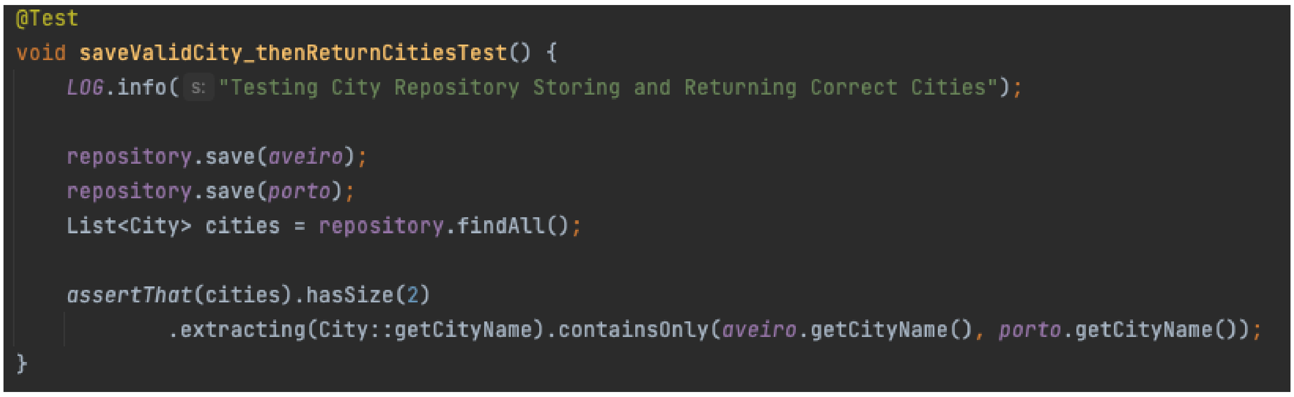
```java
@AfterAll
public static void tearDown() { mockCache.close(); }
```

For the Cities Service, it was pretty much the same as the previous service, but I also decided to check if the cities from the CSV were being read and saved correctly on the city repository.



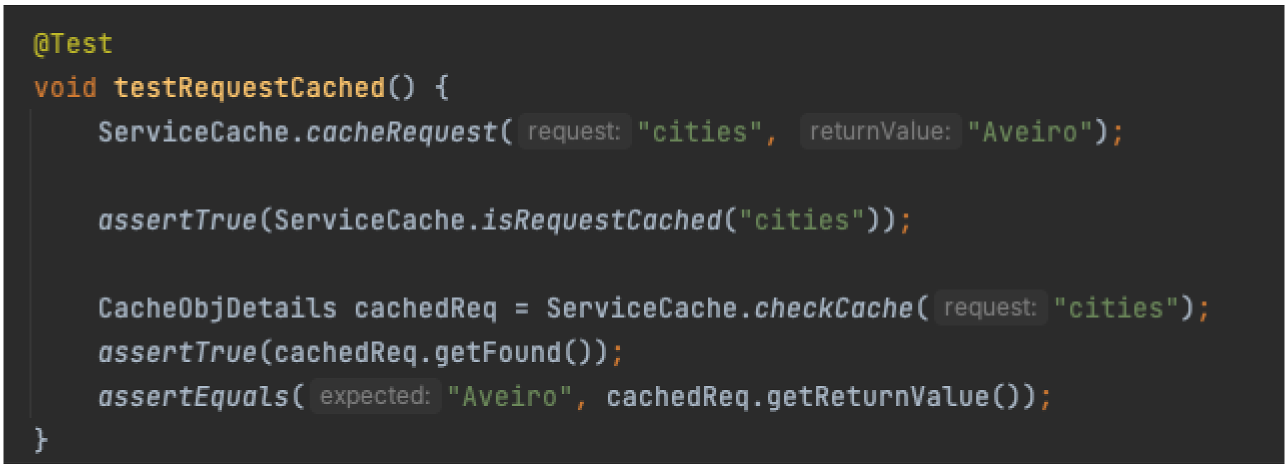
```java
@Test
void whenGetAllCitiesRepositoryEmpty_thenReadCSVAndReturnCitiesTest() {
    LOG.info( s: "Testing Request to City Service to Get All Cities from CSV");

    Mockito.when(cityRepository.findAll()).thenReturn(new ArrayList<>());

    List<City> cities = cityService.getAllCities();

    assertThat(cities).hasSize(112);

    Mockito.verify(cityRepository, Mockito.times( wantedNumberOfInvocations: 1)).findAll();
    Mockito.verify(cityRepository, Mockito.times( wantedNumberOfInvocations: 112)).save(any(City .class));
}
```

In the City Repository, I made simple tests of saving cities and also returning the cities.



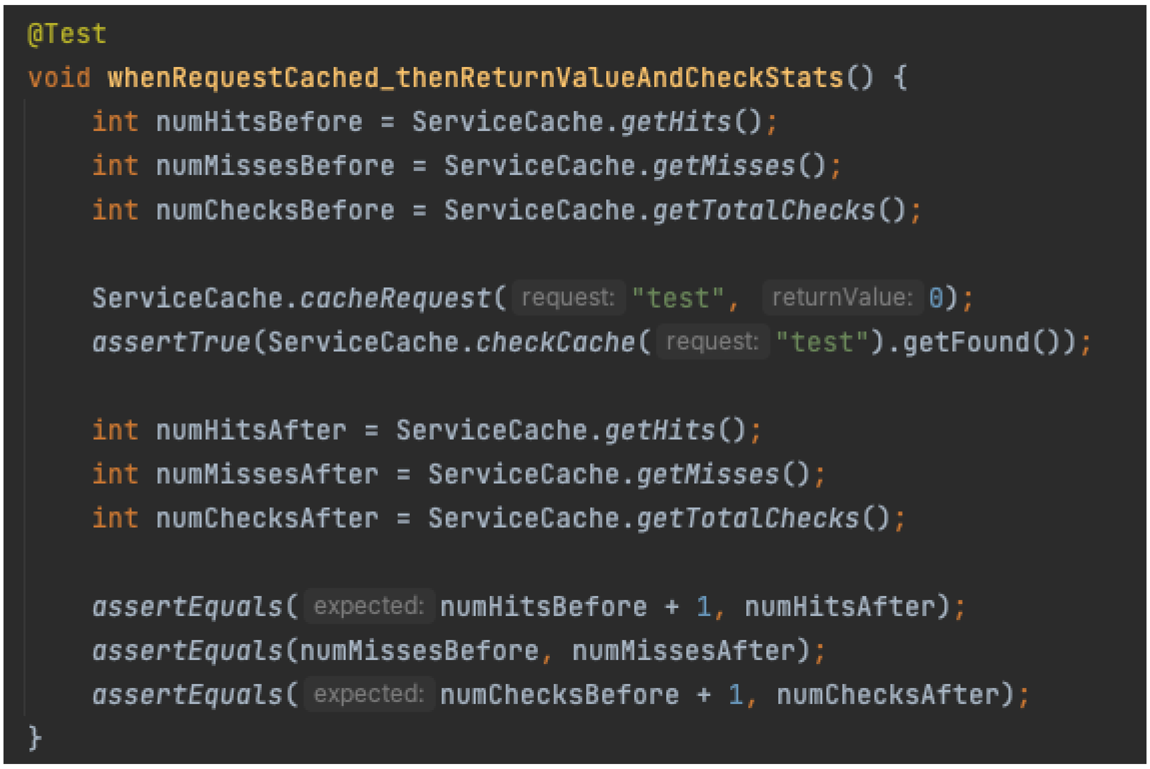
```
@Test
void saveValidCity_thenReturnCitiesTest() {
    LOG.info( s: "Testing City Repository Storing and Returning Correct Cities");

    repository.save(aveiro);
    repository.save(porto);
    List<City> cities = repository.findAll();

    assertThat(cities).hasSize(2)
            .extracting(City::getCityName).containsOnly(aveiro.getCityName(), porto.getCityName());
}
```

Finally, for the Cache, I tested several things such as caching a request and then checking if it is indeed cached.



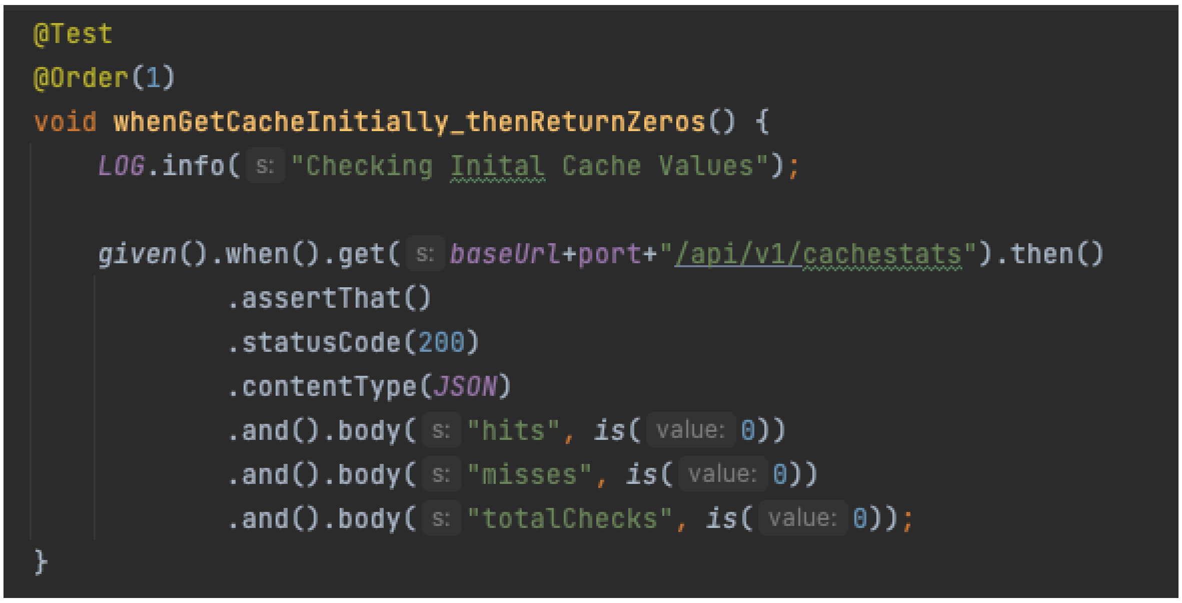
```
@Test
void testRequestCached() {
    ServiceCache.cacheRequest( request: "cities",  returnValue: "Aveiro");

    assertTrue(ServiceCache.isRequestCached("cities"));

    CacheObjDetails cachedReq = ServiceCache.checkCache( request: "cities");
    assertTrue(cachedReq.getFound());
    assertEquals( expected: "Aveiro", cachedReq.getReturnValue());
}
```

Also tested if the cache statistics updated accordingly to the number of requests made to it and also tested the time to live of requests as will be shown in the section ahead in the static code analysis.



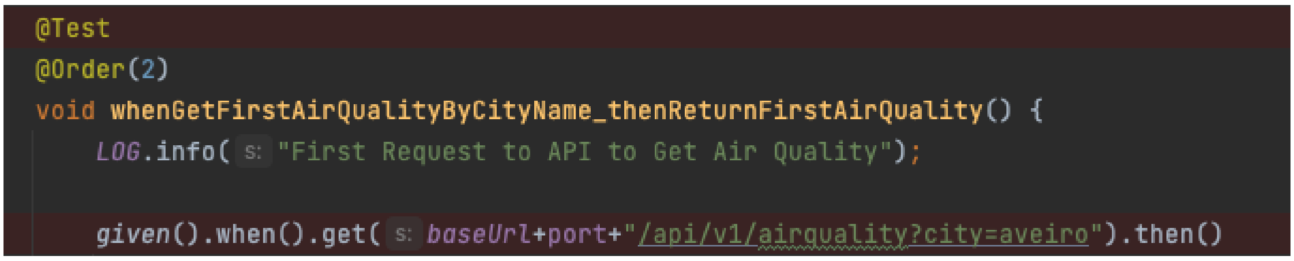
```
@Test
void whenRequestCached_thenReturnValueAndCheckStats() {
    int numHitsBefore = ServiceCache.getHits();
    int numMissesBefore = ServiceCache.getMisses();
    int numChecksBefore = ServiceCache.getTotalChecks();

    ServiceCache.cacheRequest( request: "test",  returnValue: 0);
    assertTrue(ServiceCache.checkCache( request: "test").getFound());

    int numHitsAfter = ServiceCache.getHits();
    int numMissesAfter = ServiceCache.getMisses();
    int numChecksAfter = ServiceCache.getTotalChecks();

    assertEquals( expected: numHitsBefore + 1, numHitsAfter);
    assertEquals(numMissesBefore, numMissesAfter);
    assertEquals( expected: numChecksBefore + 1, numChecksAfter);
}
```

For the integration tests, I decided to use Rest Assured since I like the syntax more and I find it easier to code. I also decided to order the tests in which the integration test run, because I wanted to test the values of cache in a sequence that made sense, so initially check if cache values are equal to zero.



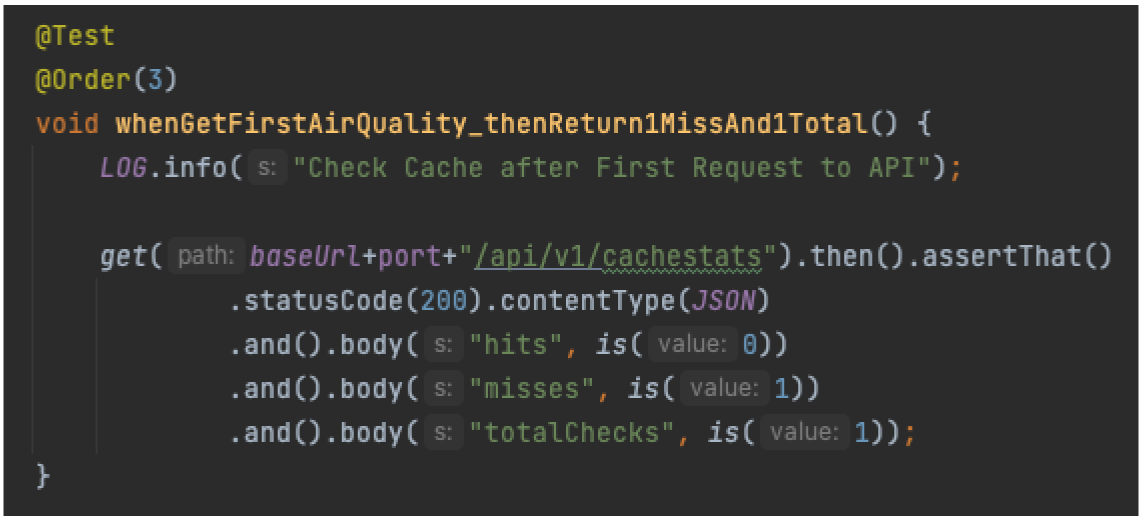
```java
@Test
@Order(1)
void whenGetCacheInitially_thenReturnZeros() {
    LOG.info( s: "Checking Inital Cache Values");

    given().when().get( s: baseUrl+port+"/api/v1/cachestats").then()
            .assertThat()
            .statusCode(200)
            .contentType(JSON)
            .and().body( s: "hits", is( value: 0))
            .and().body( s: "misses", is( value: 0))
            .and().body( s: "totalChecks", is( value: 0));
}
```

Then I make a request to an endpoint.



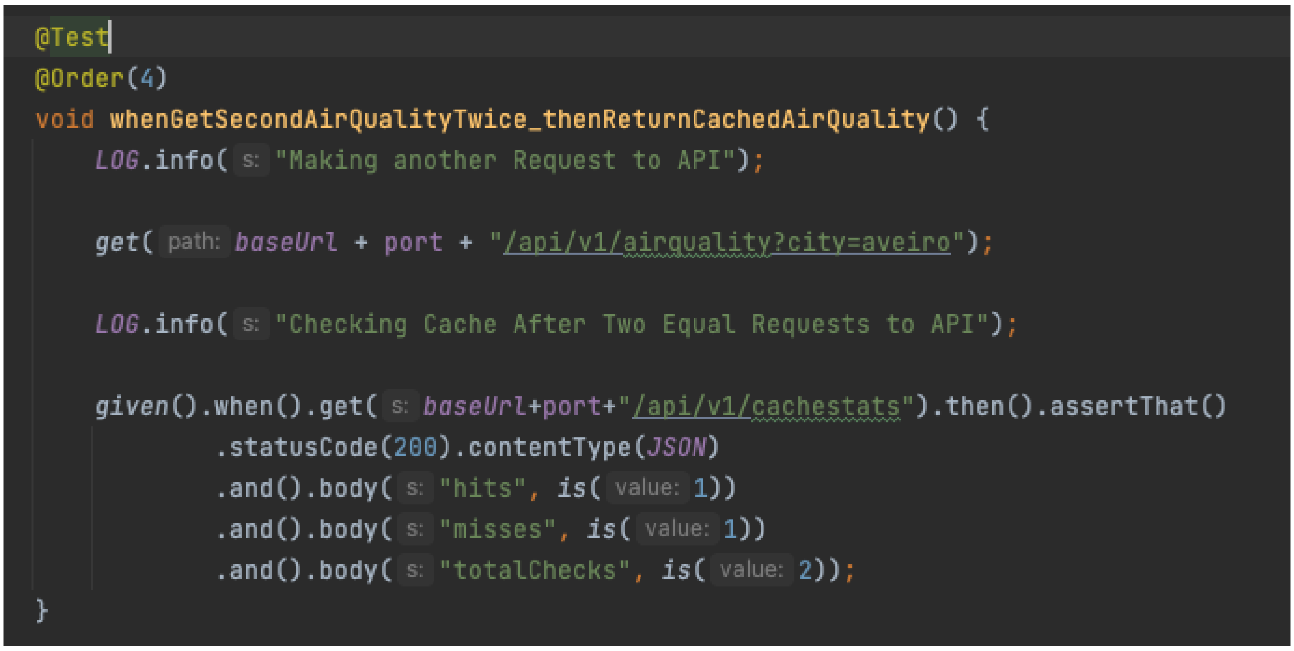
```java
@Test
@Order(2)
void whenGetFirstAirQualityByCityName_thenReturnFirstAirQuality() {
    LOG.info( s: "First Request to API to Get Air Quality");

    given().when().get( s: baseUrl+port+"/api/v1/airquality?city=aveiro").then()
```

And then I check the cache statistics numbers make sense or not.



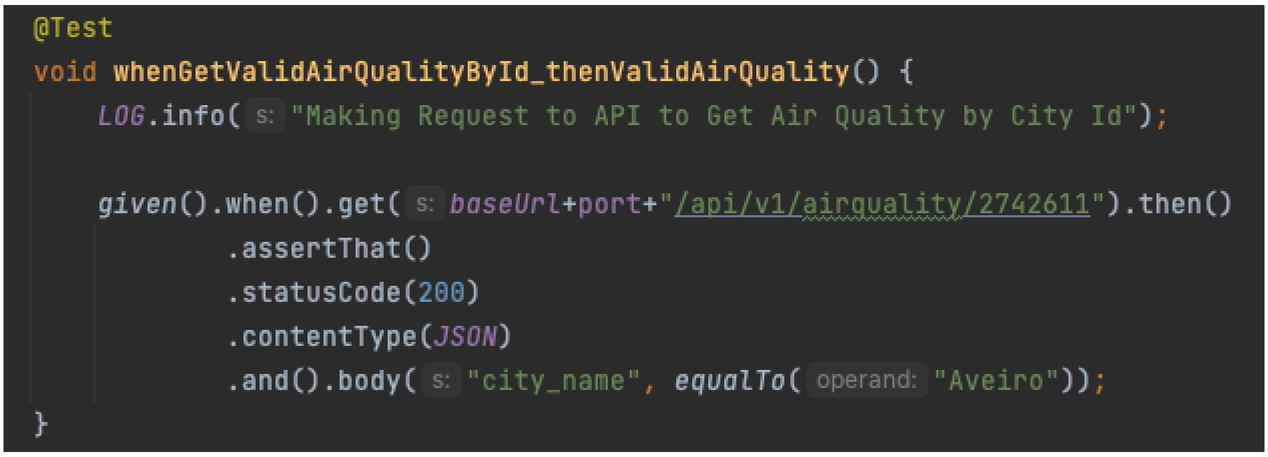
```java
@Test
@Order(3)
void whenGetFirstAirQuality_thenReturn1MissAnd1Total() {
    LOG.info( s: "Check Cache after First Request to API");

    get( path: baseUrl+port+"/api/v1/cachestats").then().assertThat()
            .statusCode(200).contentType(JSON)
            .and().body( s: "hits", is( value: 0))
            .and().body( s: "misses", is( value: 1))
            .and().body( s: "totalChecks", is( value: 1));
}
```

Finally, I repeat the request done in step 2 and check the cache statistics numbers again.



```
@Test
@Order(4)
void whenGetSecondAirQualityTwice_thenReturnCachedAirQuality() {
    LOG.info( s: "Making another Request to API");

    get( path: baseUrl + port + "/api/v1/airquality?city=aveiro");

    LOG.info( s: "Checking Cache After Two Equal Requests to API");

    given().when().get( s: baseUrl+port+"/api/v1/cachestats").then().assertThat()
            .statusCode(200).contentType(JSON)
            .and().body( s: "hits", is( value: 1))
            .and().body( s: "misses", is( value: 1))
            .and().body( s: "totalChecks", is( value: 2));
}
```

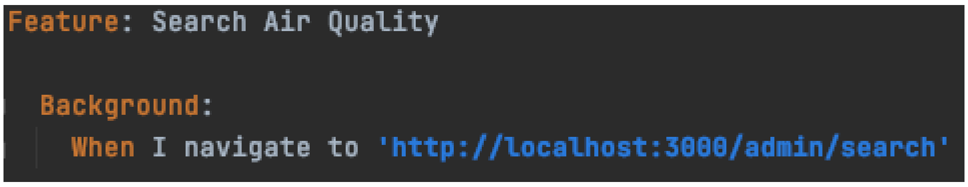
For other integration tests, the order wasn't necessary and I just tested the endpoints with valid and invalid parameters and check the return values from the request were correct.



```
@Test
void whenGetValidAirQualityById_thenValidAirQuality() {
    LOG.info( s: "Making Request to API to Get Air Quality by City Id");

    given().when().get( s: baseUrl+port+"/api/v1/airquality/2742611").then()
            .assertThat()
            .statusCode(200)
            .contentType(JSON)
            .and().body( s: "city_name", equalTo( operand: "Aveiro"));
}
```

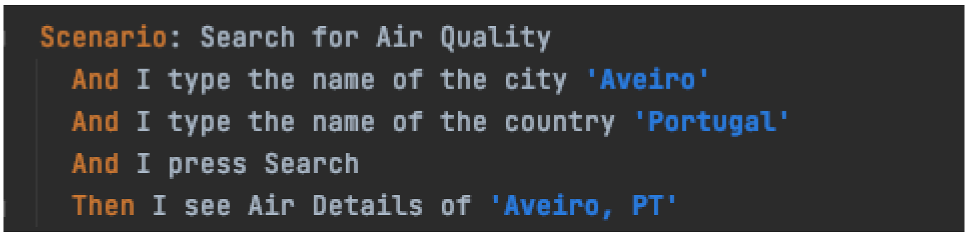
3.3 Functional testing

For functional tests, I decided to use Selenium with the Firefox driver.
For the feature created I created a background to start at the search tab of the application.



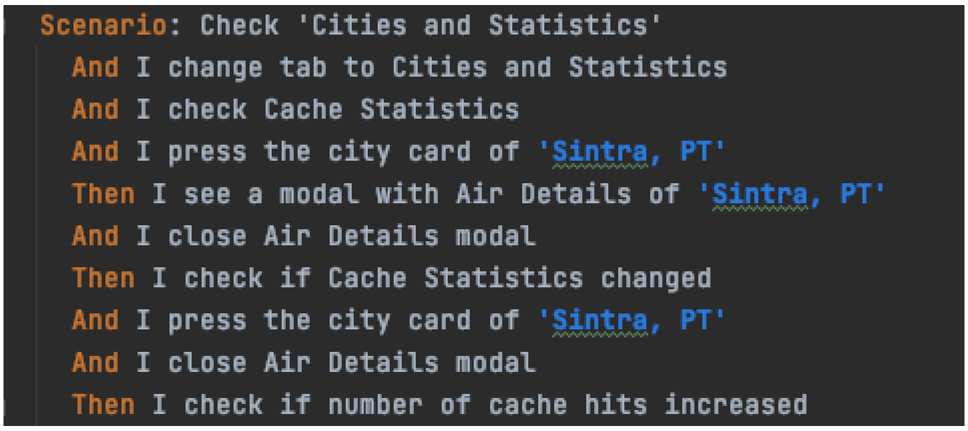
```
Feature: Search Air Quality

  Background:
    When I navigate to 'http://localhost:3000/admin/search'
```

I made two scenarios, the first simply tests the search bar on the main tab by typing a city and country name and then verifying that the air details are there.



```
Scenario: Search for Air Quality
  And I type the name of the city 'Aveiro'
  And I type the name of the country 'Portugal'
  And I press Search
  Then I see Air Details of 'Aveiro, PT'
```

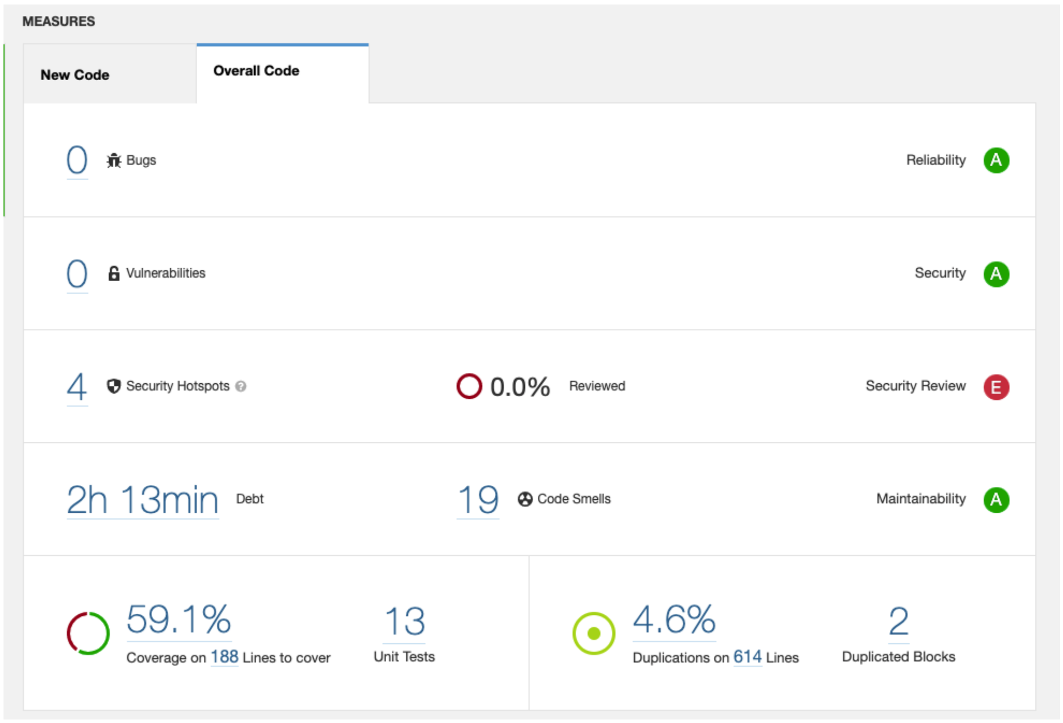
The second scenario tests the cities and statistics tab, here it will store the values present in the statistics like the number of hits, misses and requests, then presses on a city card and verifies the air details are there, then checks again the cache statistics and sees if either hits or misses changed and if the number of requests increased, after that it presses the same city card, and verifies that the air details are there again, and finally checks if the number of hits increased since this is definitely a cached value.



```
Scenario: Check 'Cities and Statistics'
  And I change tab to Cities and Statistics
  And I check Cache Statistics
  And I press the city card of 'Sintra, PT'
  Then I see a modal with Air Details of 'Sintra, PT'
  And I close Air Details modal
  Then I check if Cache Statistics changed
  And I press the city card of 'Sintra, PT'
  And I close Air Details modal
  Then I check if number of cache hits increased
```

3.4 Static code analysis

The first time Sonar Qube analysis was run against the code:



MEASURES

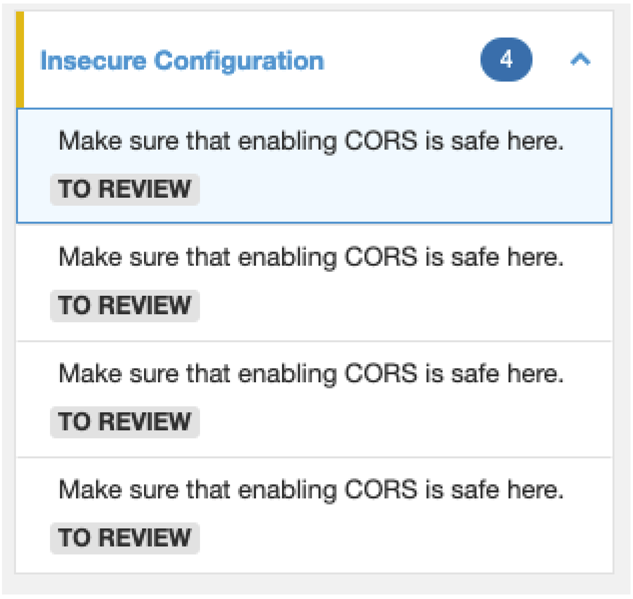
New Code | Overall Code

0 Bugs		Reliability A
0 Vulnerabilities		Security A
4 Security Hotspots	0.0% Reviewed	Security Review E
2h 13min Debt	19 Code Smells	Maintainability A

59.1% Coverage on 188 Lines to cover — 13 Unit Tests

4.6% Duplications on 614 Lines — 2 Duplicated Blocks

It had four security hotspots, however, they are because CORS was enabled in order to be able to successfully get requests from the browser, I later reviewed it as safe.


Insecure Configuration 4

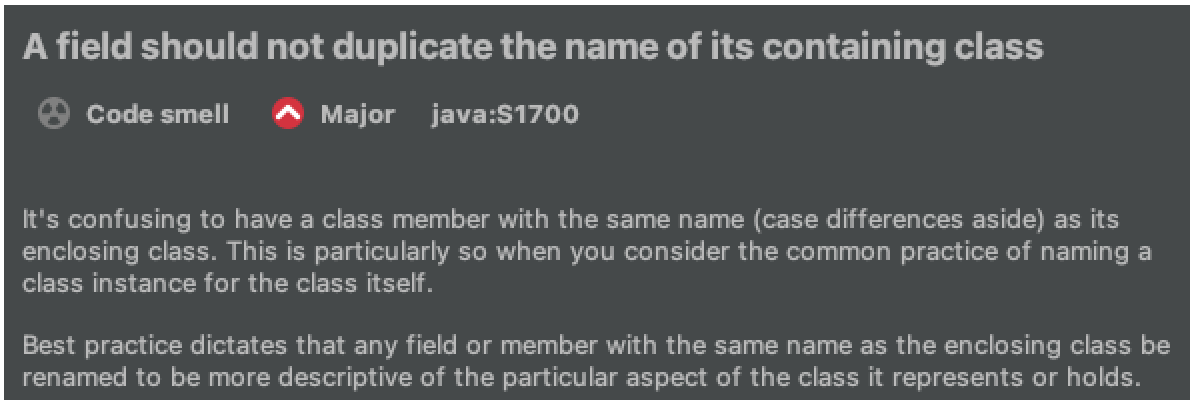
Make sure that enabling CORS is safe here.
TO REVIEW

Make sure that enabling CORS is safe here.
TO REVIEW

Make sure that enabling CORS is safe here.
TO REVIEW

Make sure that enabling CORS is safe here.
TO REVIEW

I also decided to install Sonar Lint in IntelliJ idea and that gave me on-the-fly suggestions and detected code smells.


A field should not duplicate the name of its containing class

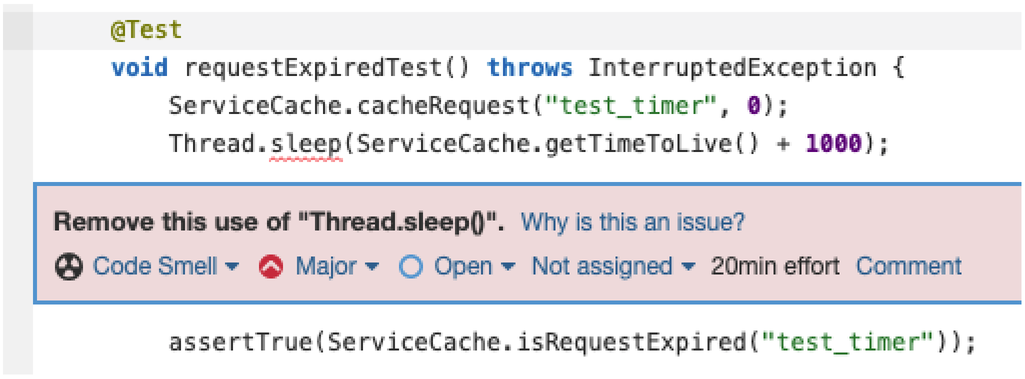
Code smell Major java:S1700

It's confusing to have a class member with the same name (case differences aside) as its enclosing class. This is particularly so when you consider the common practice of naming a class instance for the class itself.

Best practice dictates that any field or member with the same name as the enclosing class be renamed to be more descriptive of the particular aspect of the class it represents or holds.

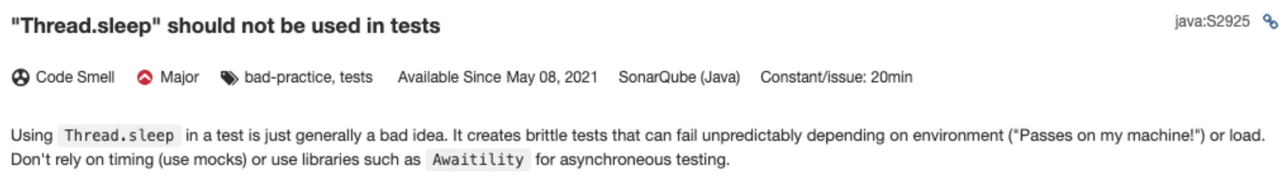
For example, this happened because my cache class was named Cache and the hash map was also called cache. To fix this I changed the class name to ServiceCache.

To test the cache request's expiration I was using a Thread function to sleep the current operation and wait a little more than the time to live of the cache. However, Sonar Qube detected this and suggested a better approach:


```
@Test
void requestExpiredTest() throws InterruptedException {
    ServiceCache.cacheRequest("test_timer", 0);
    Thread.sleep(ServiceCache.getTimeToLive() + 1000);
```

Remove this use of "Thread.sleep()". Why is this an issue?
Code Smell Major Open Not assigned 20min effort Comment

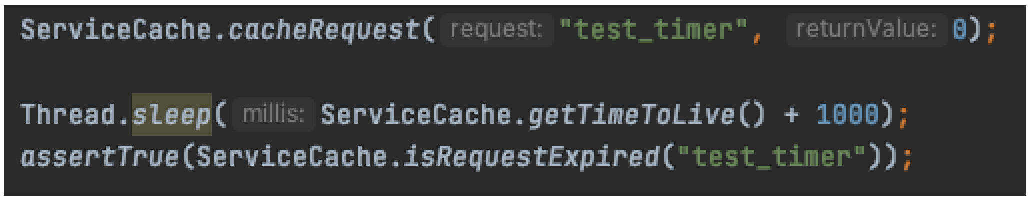
```
    assertTrue(ServiceCache.isRequestExpired("test_timer"));
```



"Thread.sleep" should not be used in tests java:S2925

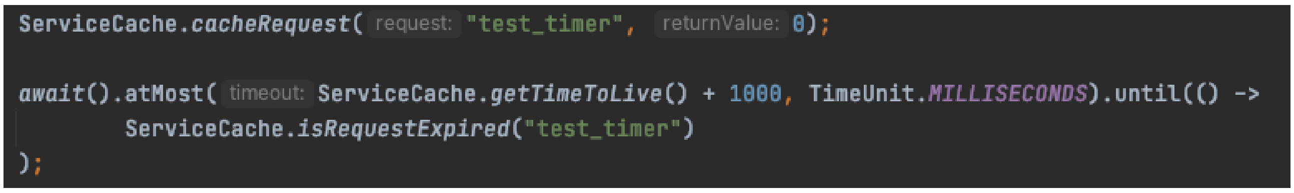
Code Smell · Major · bad-practice, tests · Available Since May 08, 2021 · SonarQube (Java) · Constant/issue: 20min

Using `Thread.sleep` in a test is just generally a bad idea. It creates brittle tests that can fail unpredictably depending on environment ("Passes on my machine!") or load. Don't rely on timing (use mocks) or use libraries such as `Awaitility` for asynchroneous testing.



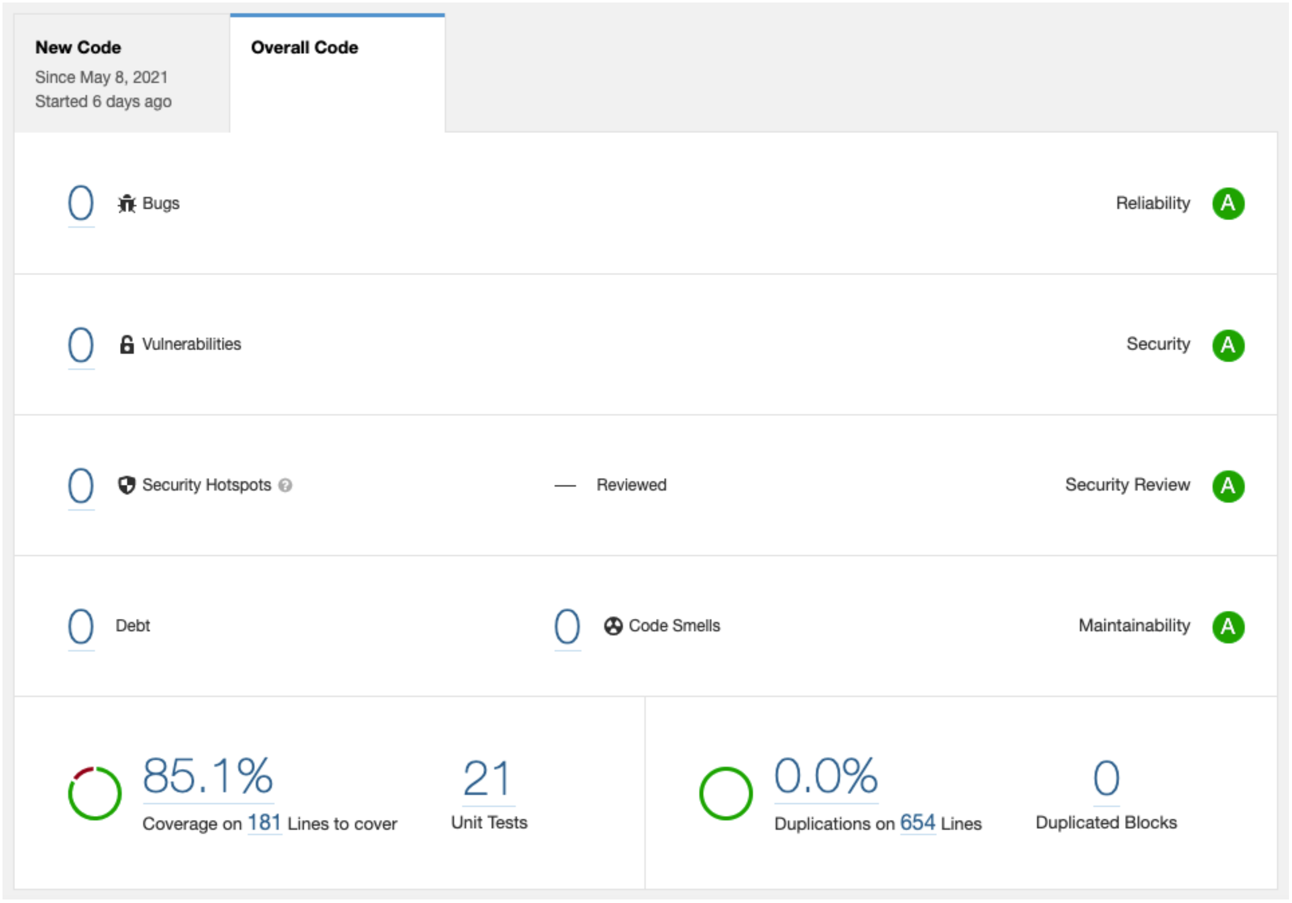
```
ServiceCache.cacheRequest( request: "test_timer",  returnValue: 0);

Thread.sleep( millis: ServiceCache.getTimeToLive() + 1000);
assertTrue(ServiceCache.isRequestExpired("test_timer"));
```

After seeing this, I searched for this library Awaitility and found out it is perfect for ensuring that a specific condition occurs in a specific amount of time or not. I believe in this case there wouldn't be much difference, but in other cases, this would be crucial to make sure something happens before moving on the test instead of sleeping which might be different sometimes. It also has an easy and fluent API.



```
ServiceCache.cacheRequest( request: "test_timer",  returnValue: 0);

await().atMost( timeout: ServiceCache.getTimeToLive() + 1000, TimeUnit.MILLISECONDS).until(() ->
        ServiceCache.isRequestExpired("test_timer")
);
```

In the end, the results were pretty good with 85% coverage, and 0 code smells, note that I reviewed Security Hotspots and some Code Smells that I thought weren't critical.



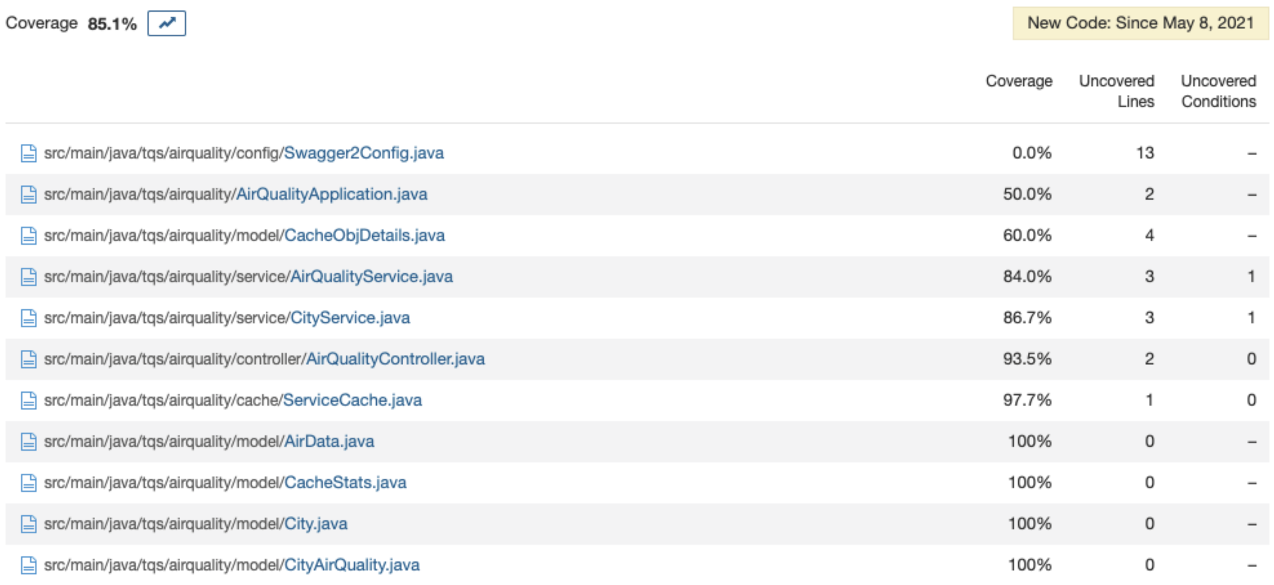
New Code
Since May 8, 2021
Started 6 days ago

Overall Code

0 Bugs		Reliability	A
0 Vulnerabilities		Security	A
0 Security Hotspots	— Reviewed	Security Review	A
0 Debt	0 Code Smells	Maintainability	A

85.1% Coverage on 181 Lines to cover · 21 Unit Tests · 0.0% Duplications on 654 Lines · 0 Duplicated Blocks

By analyzing the individual file coverage, we can see that a file that is only used to configure Swagger has 0% code coverage and that is bringing the overall code coverage down, all other files have pretty high coverages.

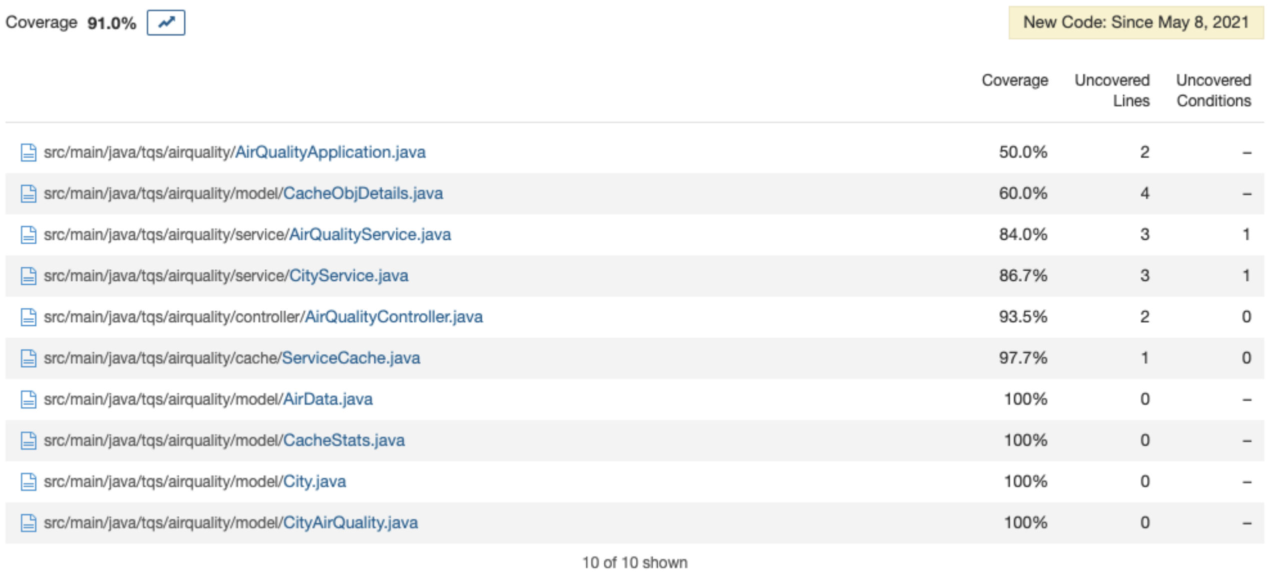


Coverage **85.1%**

New Code: Since May 8, 2021

	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/tqs/airquality/config/Swagger2Config.java	0.0%	13	–
src/main/java/tqs/airquality/AirQualityApplication.java	50.0%	2	–
src/main/java/tqs/airquality/model/CacheObjDetails.java	60.0%	4	–
src/main/java/tqs/airquality/service/AirQualityService.java	84.0%	3	1
src/main/java/tqs/airquality/service/CityService.java	86.7%	3	1
src/main/java/tqs/airquality/controller/AirQualityController.java	93.5%	2	0
src/main/java/tqs/airquality/cache/ServiceCache.java	97.7%	1	0
src/main/java/tqs/airquality/model/AirData.java	100%	0	–
src/main/java/tqs/airquality/model/CacheStats.java	100%	0	–
src/main/java/tqs/airquality/model/City.java	100%	0	–
src/main/java/tqs/airquality/model/CityAirQuality.java	100%	0	–

After removing the Swagger2Config from the analysis I got a percentage of 91% overall code coverage.



Coverage **91.0%**

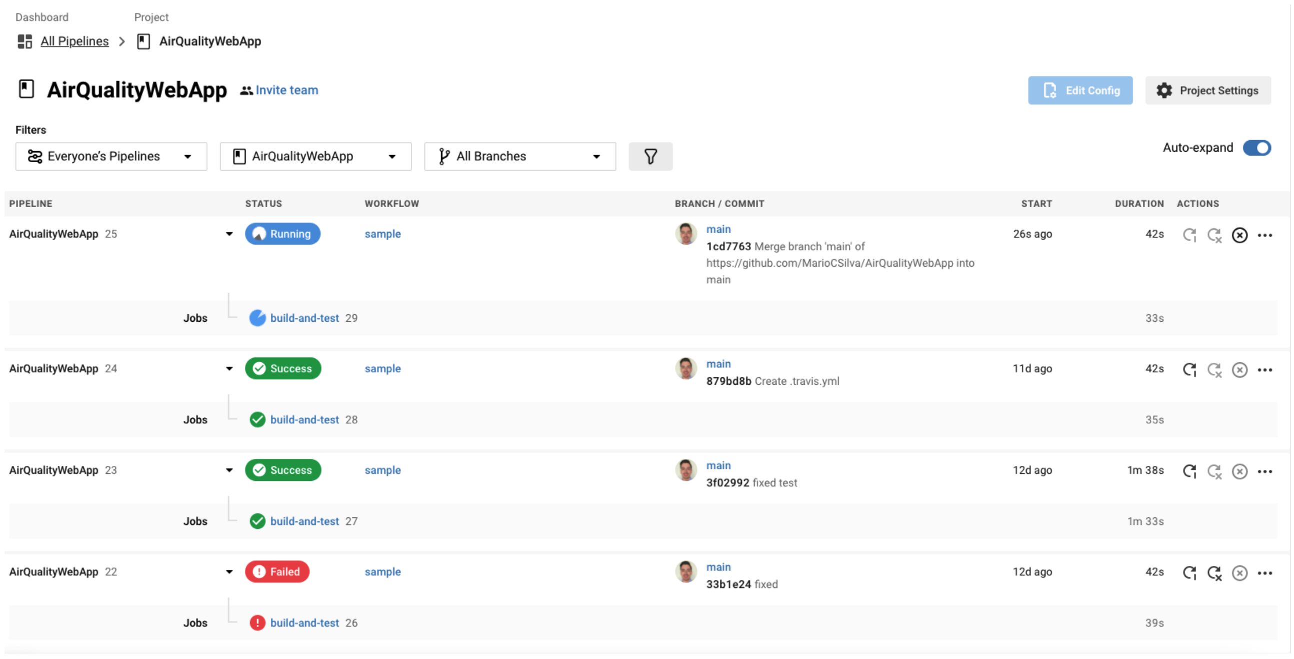
New Code: Since May 8, 2021

	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/tqs/airquality/AirQualityApplication.java	50.0%	2	–
src/main/java/tqs/airquality/model/CacheObjDetails.java	60.0%	4	–
src/main/java/tqs/airquality/service/AirQualityService.java	84.0%	3	1
src/main/java/tqs/airquality/service/CityService.java	86.7%	3	1
src/main/java/tqs/airquality/controller/AirQualityController.java	93.5%	2	0
src/main/java/tqs/airquality/cache/ServiceCache.java	97.7%	1	0
src/main/java/tqs/airquality/model/AirData.java	100%	0	–
src/main/java/tqs/airquality/model/CacheStats.java	100%	0	–
src/main/java/tqs/airquality/model/City.java	100%	0	–
src/main/java/tqs/airquality/model/CityAirQuality.java	100%	0	–

10 of 10 shown

3.5 Continuous integration pipeline

I also implemented a Continuous integration pipeline in Circle-CI early on the start-up of the project. Since I usually work on different days bit by bit this was really useful since sometimes I had failures on some tests that I hadn't noticed and also allows me to just commit without worrying.


Dashboard Project
All Pipelines > AirQualityWebApp

AirQualityWebApp Invite team   Edit Config   Project Settings

Filters
Everyone's Pipelines | AirQualityWebApp | All Branches   Auto-expand

PIPELINE		STATUS	WORKFLOW	BRANCH / COMMIT	START	DURATION	ACTIONS
AirQualityWebApp 25		Running	sample	main 1cd7763 Merge branch 'main' of https://github.com/MarioCSilva/AirQualityWebApp into main	26s ago	42s	
	Jobs	build-and-test 29				33s	
AirQualityWebApp 24		Success	sample	main 879bd8b Create .travis.yml	11d ago	42s	
	Jobs	build-and-test 28				35s	
AirQualityWebApp 23		Success	sample	main 3f02992 fixed test	12d ago	1m 38s	
	Jobs	build-and-test 27				1m 33s	
AirQualityWebApp 22		Failed	sample	main 33b1e24 fixed	12d ago	42s	
	Jobs	build-and-test 26				39s	

In the end I got a total of 28 runs in the CI pipeline. This pipeline was very simple and all it did was run 'mvn clean package' and 'mvn test'.

```
- run:
    name: Build
    command: cd ./backend-spring-boot/air-quality && mvn -B -DskipTests clean package
- run:
    name: Test
    command: cd ./backend-spring-boot/air-quality && mvn test
```

4 References & resources

Project resources

- Video demo - https://drive.google.com/file/d/1onYrT5VadueGoAsSPyv59svgAKUQnh0t/view?usp=sharing / Video File also in github: https://github.com/MarioCSilva/AirQualityWebApp/blob/main/AirCheckDemo.mov
- GitHub Repository: https://github.com/MarioCSilva/AirQualityWebApp/

Reference materials

Weatherbit API - https://www.weatherbit.io/api