HW1: Mid-term assignment report

Bruno de Sousa Bastos[93302], v2021-05-14

1 Introduction

1.1 Overview of the work

The application provides an API that can give the quality of the air of a given city. It also provides a WebPage in order to interact and see the results in an easier way.

1.2 Current limitations

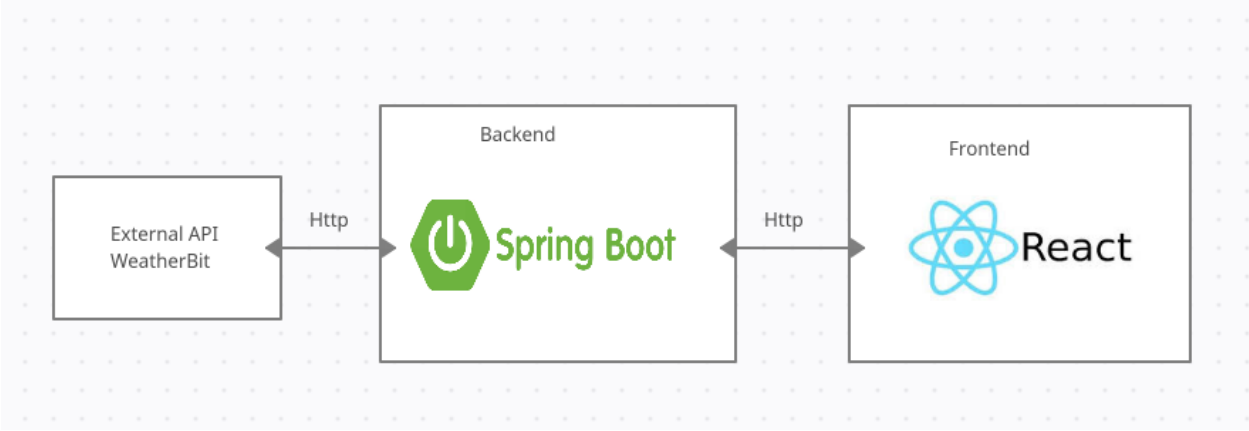
The application utilizes an external API in order to obtain the data of the air for a city. However, if the external API fails or does not have an answer to that city, it would be nice to integrate another external API to solve some of the issues of the first one. Due to the lack of time, this was not possible.

2 Product specification

2.1 Functional scope and supported interactions

Users can interact with the application through a web page that allows them to search and check the air quality of a city by giving the id of the city or the city name and the corresponding country code. They can also check how their requests influence the cache system, they can see the total of requests made as well as the number of times the request was in cache and how many times it wasn't. Users can also make direct requests to the API without going through the frontend.

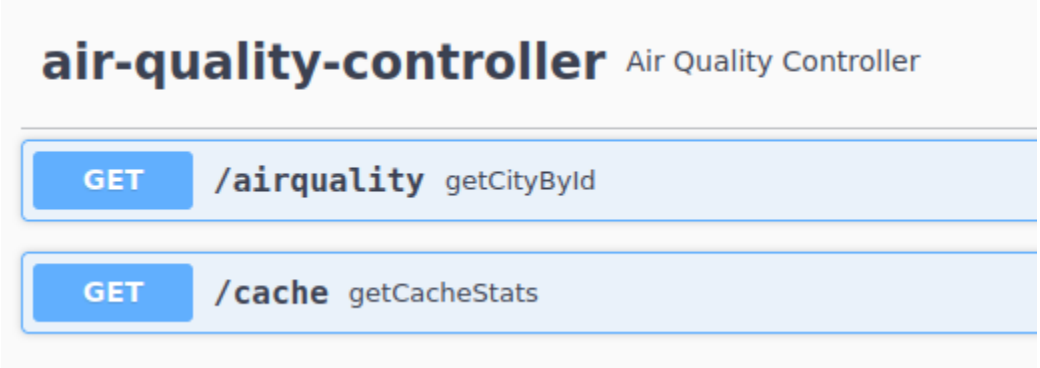
2.2 System architecture



For the backend it was used, as suggested, Spring boot. The frontend was a simple Web page made with react. The air quality data was obtained through the external API WeatherBit.

2.3 API for developers

The API has 2 groups of endpoints, one group to get the air quality data and the other to obtain the cache statistics. To obtain the air quality data, there is a single endpoint, it can be passed different request parameters. Developers can search by id or by city name and country.



The above image does not show both the possibilities for the /airquality endpoint.
In order to get the air quality by city id:
“/airquality?id=123”
In order to obtain the air quality by city name and country:
“/airquality?city=Aveiro&country=PT”

3 Quality assurance

3.1 Overall strategy for testing

For the controller, repository, service and cache tests it was used a TDD approach. Although, during the implementation of the cache some problems appear in which the structure of the cache had to be changed.
The frontend web page tests made with selenium were integrated with cucumber which made it easier to read and understand what the tests were supposed to do.

3.2 Unit and integration testing

My implementation of a repository was mainly the call of the external API in order to get the data. The results provided by the external API were not tested but I wanted to test if the it was returning the right data after a request. In order to do so, it was made a mock to the “rest template” object which was used to make the http requests to the external API. Using this mock it was possible to test the repository without making any request to the external API and without taking into consideration its return value.

```java
@ExtendWith(MockitoExtension.class)
class WeatherBitRepositoryTest {


    @Mock(lenient = true)
    private RestTemplate restTemplate;

    @InjectMocks
    private WeatherBitRepository repository;
```

```java
@Test
void test_getInValidCityInfoById_thenReturnNull(){
    String url = BASE_URL + "?city_id=-1&key=" + KEY;
    when(restTemplate.getForObject(
            url, City.class))
            .thenReturn(null);
    assertThat(repository.getDetailsByCityId(-1L)).isNull();
    verify(restTemplate, times( wantedNumberOfInvocations: 1)).getForObject(url, City.class);
}
```

It was made a service that uses that data retrieved by the repository and provides it to the controller. This service is also responsible for managing the cache. The strategy was to use a mock of the repository so that the tests were independent of the repository behaviour. Then it was tested the returning values and the expected behaviour of the service when providing a mocked value for the repository.

```java
@ExtendWith(MockitoExtension.class)
class WeatherbitServiceTest {


    @Mock( lenient = true)
    private WeatherBitRepository repository;

    @InjectMocks
    private WeatherbitService service;
```

```java
@Test
void test_whenInValidId_thenReturnEmpty() throws Exception{

    Optional<City> fromDb = service.getCityById(-1L);

    assertThat(fromDb.isEmpty());

    verify(repository, VerificationModeFactory.times( wantedNumberOfInvocations: 1))
            .getDetailsByCityId(anyLong());
}
```

To test the cache, it was only tested if the methods did what they were supposed to. If a value was added it was supposed to return that value when asked. If the value had expired it was expected to not return that value.

```java
@Test
void test_whenEmptyCache_returnEmpty(){
    assertThat(cacheManager.getCache()).isEmpty();
}
```

The last unit tests were made to the controller by mocking the service responses and by using MockMvc to perform the requests to the controller. The return of the controller was tested based on the mocked values of the service. For all the endpoints it was tested if the result provided by the service was empty and if it was the case, it was expected that the controller responded with Not Found. In case that the service returned a value it was expected that the controller responded with an OK and returned the entity provided by the service. It was also tested the value of the cache stats by mocking the service method that returned the cache stats.

```java
@WebMvcTest(AirQualityController.class)
class AirQualityControllerTest {


    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private WeatherbitService service;
```

```java
@Test
void test_getCityByInvalidNameCountry_thenReturnNotFound() throws Exception{
    given(service.getCityByNameAndCountry( city: "Aveiro",  country: "ES"))
            .willReturn(Optional.empty());

    mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/airquality?city=Aveiro&country=ES"))
            .andExpect(status().isNotFound());

    verify(service, times( wantedNumberOfInvocations: 1)).getCityByNameAndCountry( city: "Aveiro",  country: "ES");

}
```

For the Integration tests, it tested the overall behaviour of the application by using MockMvc to test the response of the controller, this time without any mocks. It was tested if the endpoints were returning the values when the parameters were correct and if it was returning Not found whenever the parameters were not found. It was also tested the cache behaviour by calling an endpoint multiple times and checking the cache values. Also, because the cache was shared between all integration tests, I had 2 options. Making the tests and giving them an order so that the cache behaviour could be tested or restart the test application for each of the integration tests. Although the second might take a bit more time and probably is not the best idea for bigger projects, I decided to choose that anyway because it allows me to keep the tests that I did, saving me time.

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, classes = AirqualityApplication
@AutoConfigureMockMvc
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_EACH_TEST_METHOD)
public class ControllerIT {

    @Autowired
    private MockMvc servlet;
```

```java
@Test
void test_whenGetCorrectId_thenReturnData() throws Exception {
    this.servlet.perform(get( urlTemplate: "/airquality?id=" + AVEIRO_ID))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "city_name", is( value: "Aveiro")))
            .andExpect(jsonPath( expression: "country_code", is( value: "PT")));
}
```

The code that was shown only provides the smallest tests so that they can fit in the report. However, the idea behind them should be the same.

3.3 Functional testing

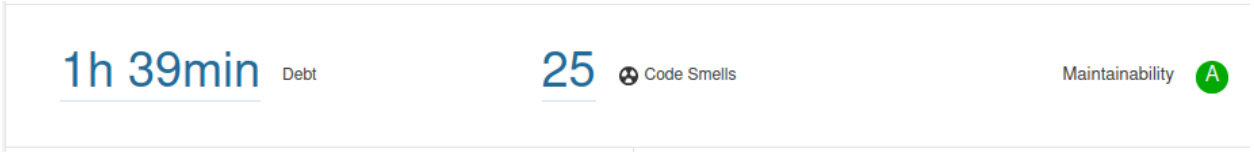
Functional tests were implemented with selenium with the help of cucumber in order to make the tests more readable. It tested the behaviour of the frontend when search by id or name and country as well as the behaviour of the cache. To run these tests it is required that the frontend is running on “localhost:3000”.

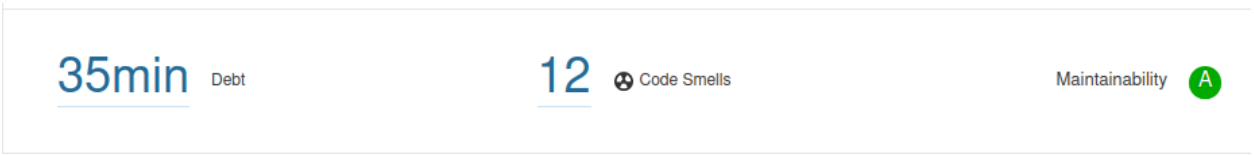
```
Scenario: Search by Wrong ID
  When I navigate to "http://localhost:3000"
  And I insert "1" in "city_id"
  And I press "search_by_id"
  Then I should see an error message
```

```
Scenario: Search Aveiro by Name and Country and check Cache
  When I navigate to "http://localhost:3000"
  And I check that "hits" value is equal to 0
  And I check that "misses" value is equal to 0
  And I check that "requests" value is equal to 0
  And I insert "Aveiro" in "name"
  And I insert "PT" in "country"
  And I press "search_by_name"
  Then I should see that result contains "Aveiro"
  And I press "refresh"
  Then I should see that cache has changed
  And I check that "hits" value is equal to 0
  And I check that "misses" value is equal to 1
  And I check that "requests" value is equal to 1
  And I press "search_by_name"
  And I press "refresh"
  Then I should see that cache has changed
  And I check that "hits" value is equal to 1
  And I check that "misses" value is equal to 1
  And I check that "requests" value is equal to 2
```

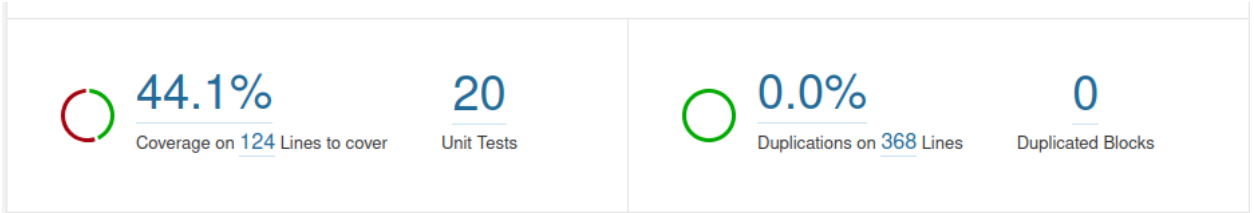
3.4 Static code analysis

It was used a local instance of SonarQube as well as the Sonalint plugin for the IDE. This way it was possible to fix some code smells without needing to check on SonarQube. Jacoco was used to provide the test coverage for the application.





These are the results for the code coverage.



Although the overall coverage is not that great when analysing the files one by one we can see that some of the classes we indeed have not been tested which decrease the code coverage by a lot.

	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/tqs/airquality/model/AirQualityData.java	0.0%	13	36
src/main/java/tqs/airquality/model/City.java	11.4%	0	62
src/main/java/tqs/airquality/AirqualityApplication.java	50.0%	2	–
src/main/java/tqs/airquality/repository/WeatherBitRepository.java	50.0%	6	–
src/main/java/tqs/airquality/service/WeatherbitService.java	75.0%	6	2
src/main/java/tqs/airquality/cache/CacheManager.java	85.7%	5	0
src/main/java/tqs/airquality/controller/AirQualityController.java	100%	0	–
src/main/java/tqs/airquality/cache/CachePair.java	100%	0	–
src/main/java/tqs/airquality/model/CacheStats.java	100%	0	–
src/main/java/tqs/airquality/config/Swagger2Config.java	100%	0	–

10 of 10 shown

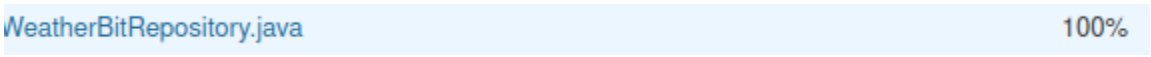
For the most part the classes tested are above 75%, however the WeatherBitRepository was only at 50%.

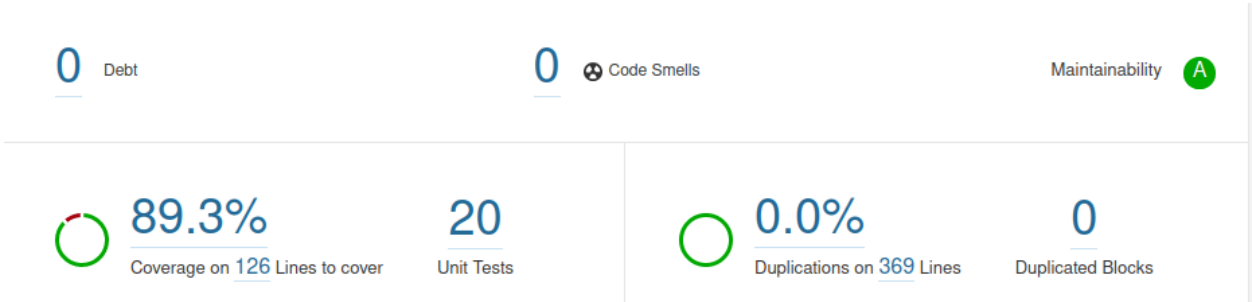
```
    public City getDetailsByCityId(long cityId){
        try {
            String url = BASE_URL + "?city_id=" + cityId + "&key=" + KEY;
            return this.restTemplate.getForObject(url, City.class);
        }catch (Exception e) {
            // change to logger?
            LOGGER.info(e.toString());
        }
        return null;
    }

    public City getDetailsByCityNameAndCountry(String cityName, String country){
        try{
            String url = BASE_URL + "?city=" + cityName + "&country=" + country + "&key=" + KEY;
            return this.restTemplate.getForObject(url, City.class);
        }catch (Exception e){
            LOGGER.info(e.toString());

        }
        return null;
    }
```

This print shows the lines that were not covered. After some modifications to the tests, I was able to get 100% coverage for this class.





Final results after some debugging. Some of the code smells were removed because they were related to the variable names, however because the external API required that the names were like that, there wasn't much to be done there. There is also a Security Hotspot related to the CrossOrigin from spring boot but since it is not a major problem for this work I didn't bother fixing it.

4 References & resources

Project resources

GitHub Repo: https://github.com/BrunosBastos/air-quality-api

WeatherBit API: https://www.weatherbit.io/api/airquality-current