HW1: Mid-term assignment report

Pedro Miguel Bastos Almeida [93150], v2021-05-14
Git: https://github.com/bastos-01/TQS-AirQuality

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 project was intended to be about air quality. It was intended to have a **REST API** that consumes and external API (in my case, AirVisual API). It was also supposed to have tests in multiple levels. Finally, it has a basic frontend Single Page Application that allows the user to search for the weather and pollution, as well as verify the cache details of the **REST API**. The product that I built is called **AirQuality** and it allows people to select The Country, State and City and receive information not only on the Pollution metrics, but also in the weather forecast. As said before, it is also possible to see the cache details.

1.2 Current limitations

The outside API chosen had some problems in specific cities. Unfortunately, I only found out that later, so I couldn't afford to change it. For example, the API says that Coimbra is available, but when searching for information on that city, it fails. Because it happens in very few cities, I didn't think that it would matter that much.

Besides that, the product has some unimplemented features:

- Second API – Although it was for extra points, I still consider that it was expected to integrate another API.
- Continuous Integration Framework – As the second API, this was also for extra points and it is not implemented.

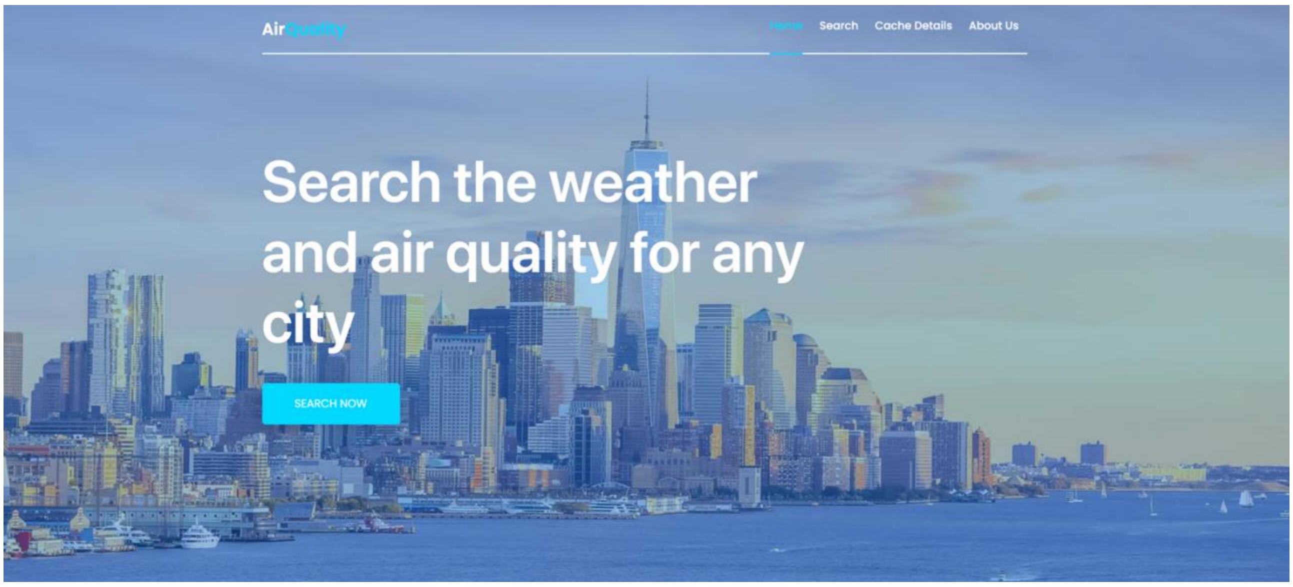
It also should have more tests on the **CityServiceImplementation**, that I didn't write due to lack of time.

2 Product specification

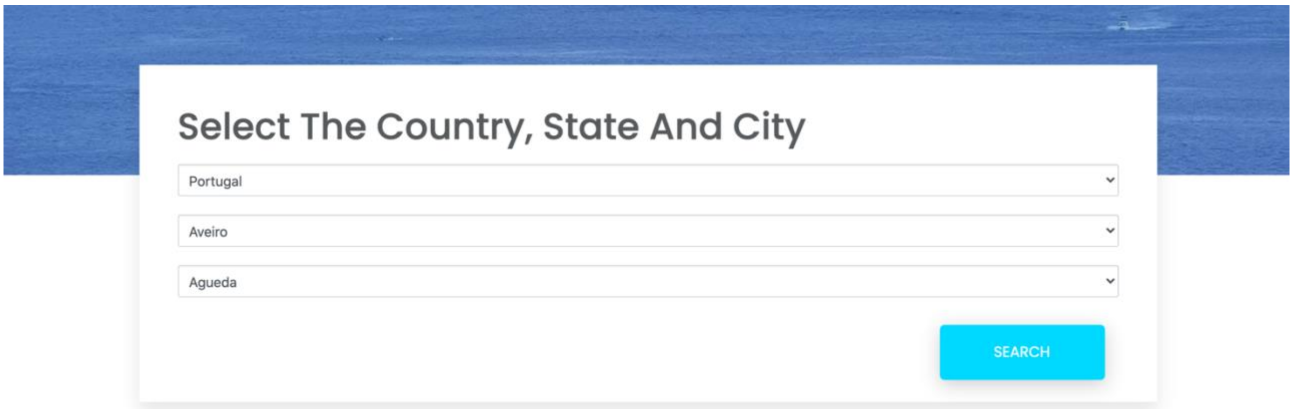
2.1 Functional scope and supported interactions

The application is pretty simple. The user needs to select the Country, State and City that he wants to search for the weather and pollution details. After clicking in the search button, the information will show. Besides that, it is possible to see the hits, misses and requests on the cache section. By clicking in the Update button, the stats will be updated.

The initial landing section is pretty simple. The user can clearly see what the application can do and by clicking in "Search Now" the page will scroll down to the search section.



The search section has 3 dropdowns to choose the Country, State and City to get the details.

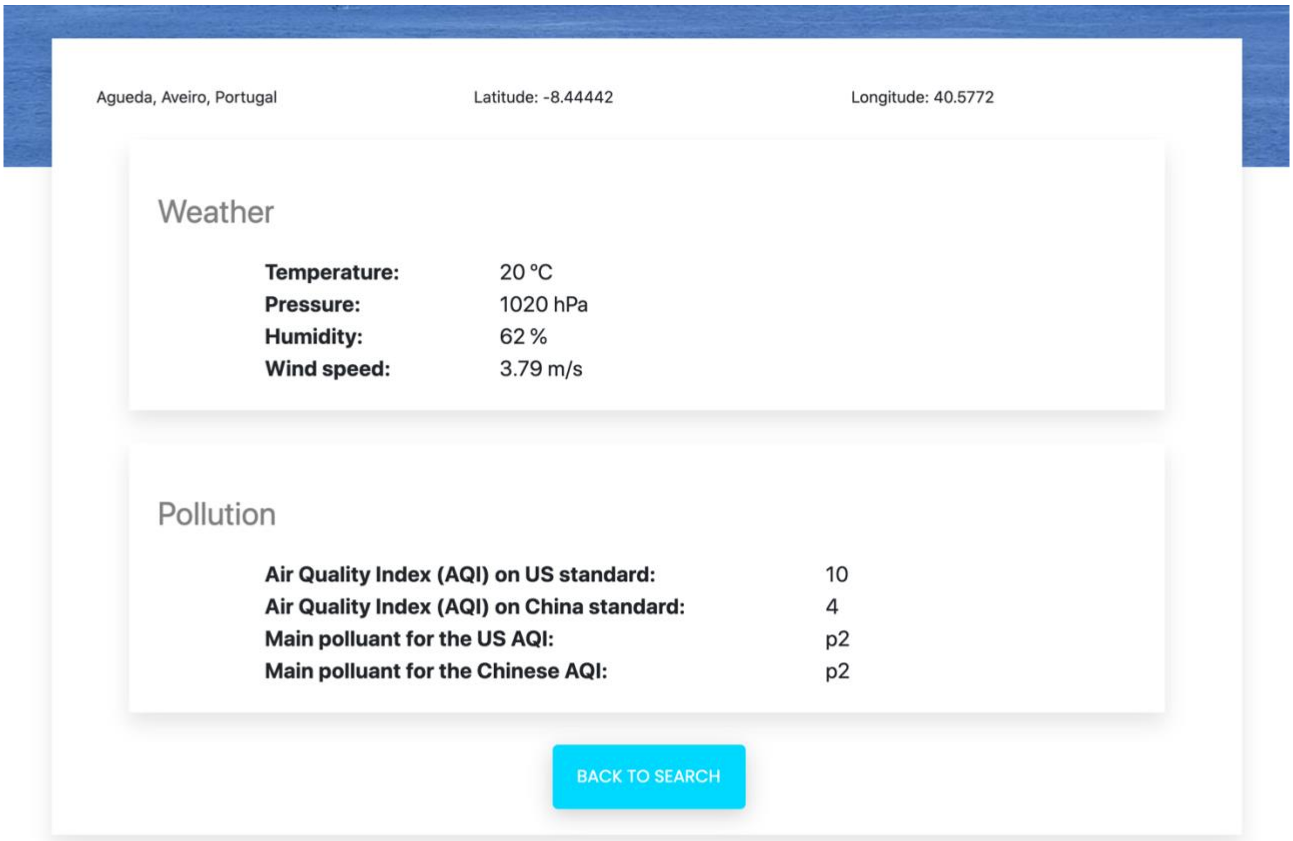

Select The Country, State And City

Portugal

Aveiro

Agueda

After choosing the intended regions and clicking on Search, the weather forecast and pollution details will show up, as well as the latitude and longitude of its city.



Agueda, Aveiro, Portugal Latitude: -8.44442 Longitude: 40.5772

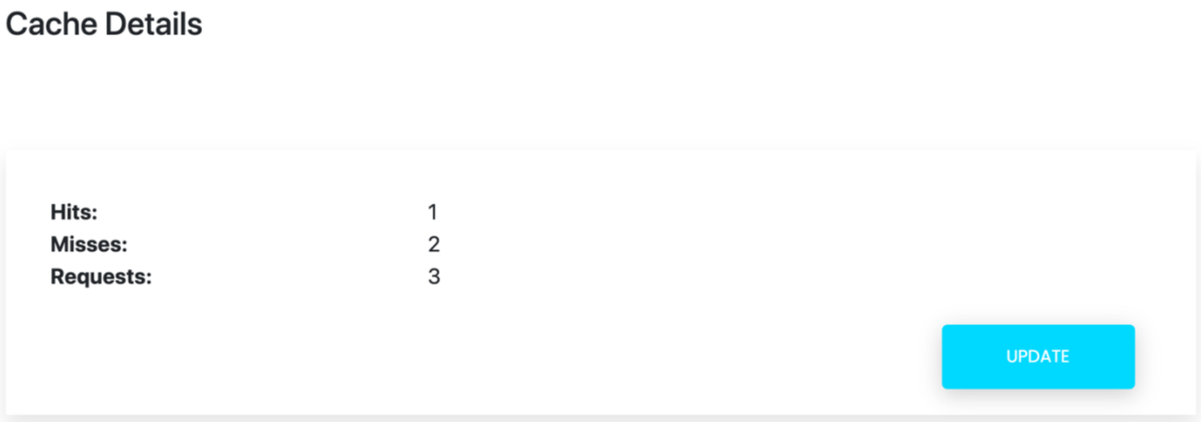
Weather

Temperature:	20 °C
Pressure:	1020 hPa
Humidity:	62 %
Wind speed:	3.79 m/s

Pollution

Air Quality Index (AQI) on US standard:	10
Air Quality Index (AQI) on China standard:	4
Main polluant for the US AQI:	p2
Main polluant for the Chinese AQI:	p2

As we can see, in the upper part it shows the latitude and longitude. On the Weather card, there is information about the temperature, pressure, humidity and wind speed. Finally, on the pollution card we have the AQI (Air Quality Index) on both US and China standards, followed by their main pollutants.

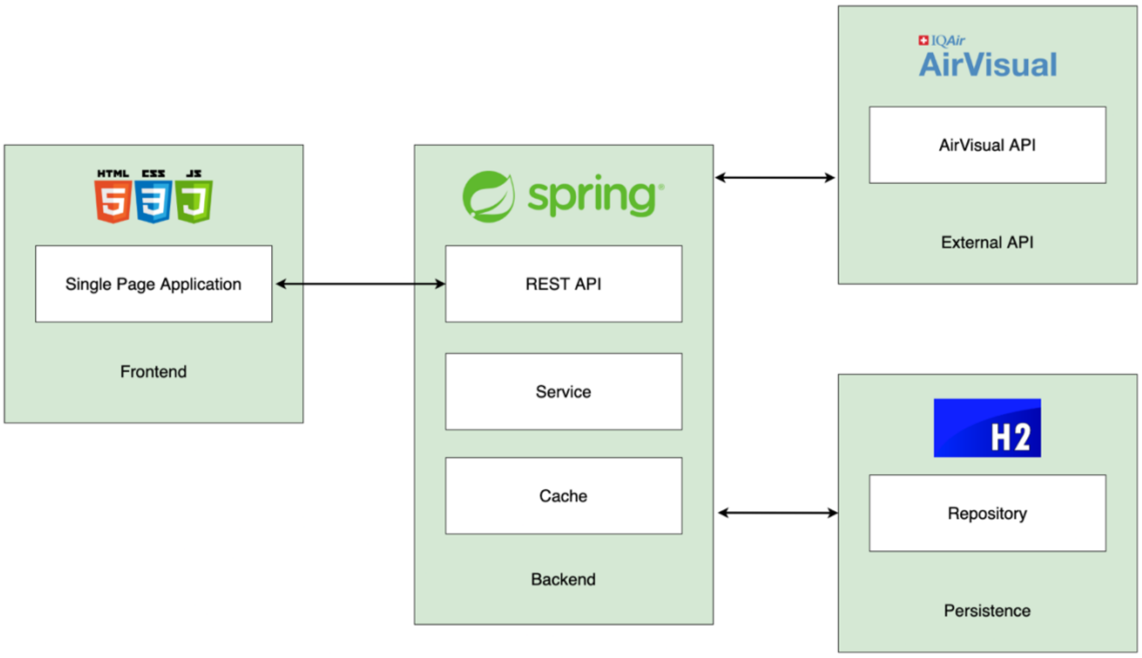


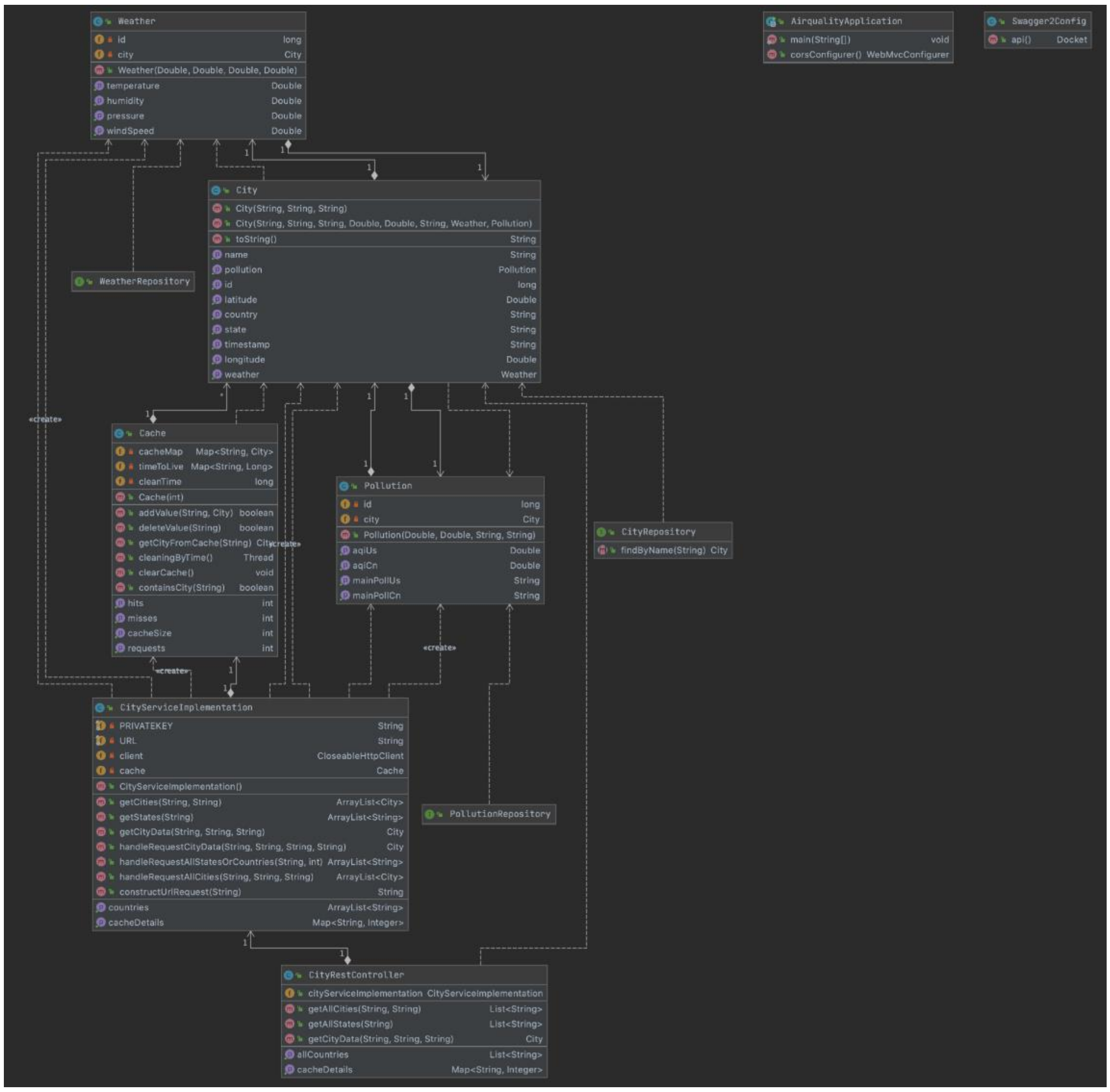
Cache Details

Hits:	1
Misses:	2
Requests:	3

Scrolling down, we get to the cache section, that informs us about the hits(Cache used), misses(Cache not used, requested to the external API) and requests(total number of requests).

2.2 System architecture

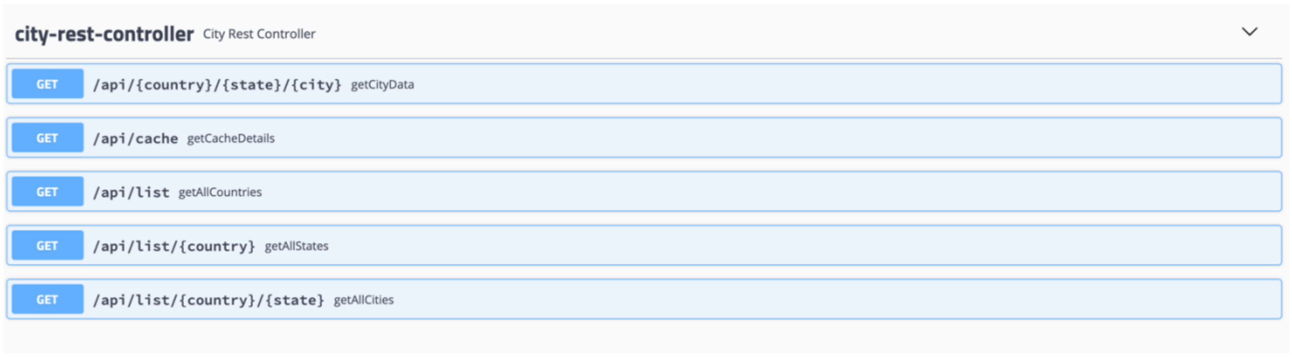



Weather
id long
city City
Weather(Double, Double, Double, Double)
temperature Double
humidity Double
pressure Double
windSpeed Double
AirqualityApplication
main(String[]) void
corsConfigurer() WebMvcConfigurer
Swagger2Config
api() Docket
City
City(String, String, String)
City(String, String, String, Double, Double, String, Weather, Pollution)
toString() String
name String
pollution Pollution
id long
latitude Double
country String
state String
timestamp String
longitude Double
weather Weather
WeatherRepository
«create»
Cache
cacheMap Map<String, City>
timeToLive Map<String, Long>
cleanTime long
Cache(int)
addValue(String, City) boolean
deleteValue(String) boolean
getCityFromCache(String) City
cleaningByTime() Thread
clearCache() void
containsCity(String) boolean
hits int
misses int
cacheSize int
requests int
Pollution
id long
city City
Pollution(Double, Double, String, String)
aqiUs Double
aqiCn Double
mainPollUs String
mainPollCn String
CityRepository
findByName(String) City
«create»
CityServiceImplementation
PRIVATEKEY String
URL String
client CloseableHttpClient
cache Cache
CityServiceImplementation()
getCities(String, String) ArrayList<City>
getStates(String) ArrayList<String>
getCityData(String, String, String) City
handleRequestCityData(String, String, String, String) City
handleRequestAllStatesOrCountries(String, int) ArrayList<String>
handleRequestAllCities(String, String, String) ArrayList<City>
constructUrlRequest(String) String
countries ArrayList<String>
cacheDetails Map<String, Integer>
PollutionRepository
CityRestController
cityServiceImplementation CityServiceImplementation
getAllCities(String, String) List<String>
getAllStates(String) List<String>
getCityData(String, String, String) City
allCountries List<String>
cacheDetails Map<String, Integer>

As we can see, the frontend was made with basic web technologies – HTML + CSS + JS + JQuery + AJAX.
The Bakcend was built in Spring Boot as asked, with a REST API, a service that calls the external API and the Cache implementation. Finally, the repository used was H2, with the connection made by JPA.

For better understanding of the backend, I decided to add the UML Class Diagram. The Rest controller is in the Class CityRestController, that receives the requests from the frontend and calls the appropriate methods of the service (Class CityServiceImplementation). Then, the service calls the external API (if the data is not already stored in cache). In addition, the class Swagger2Config is to have an endpoint with the documentation.

2.3 API for developers

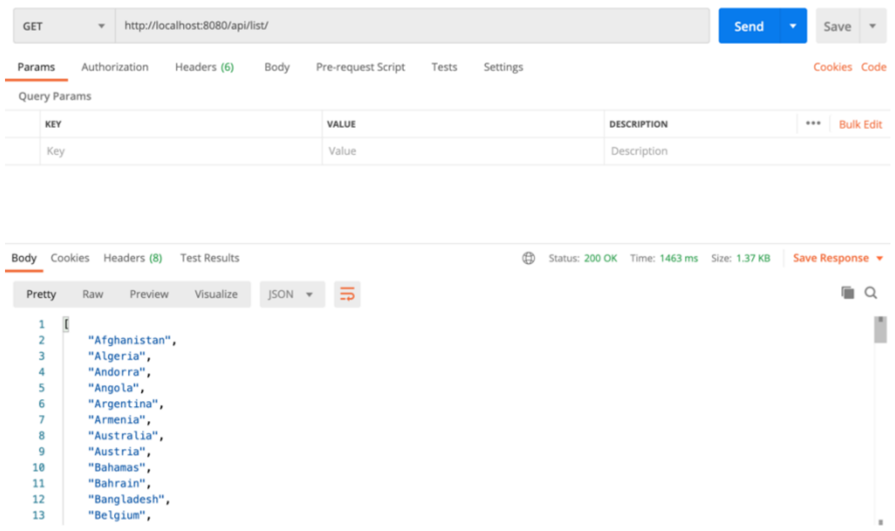


city-rest-controller City Rest Controller

- GET /api/{country}/{state}/{city} getCityData
- GET /api/cache getCacheDetails
- GET /api/list getAllCountries
- GET /api/list/{country} getAllStates
- GET /api/list/{country}/{state} getAllCities

The documentation of our API can be found in http://localhost:8080/swagger-ui.html.
As we can see, there are several endpoints for the developer to use. Basically, all the /list endpoints are to retrieve all available coutries/states/cities. Then there is one endpoint to get the forecast information and one to get the cache details.

- **/api/list/** (get all the countries available)
    - Type: GET

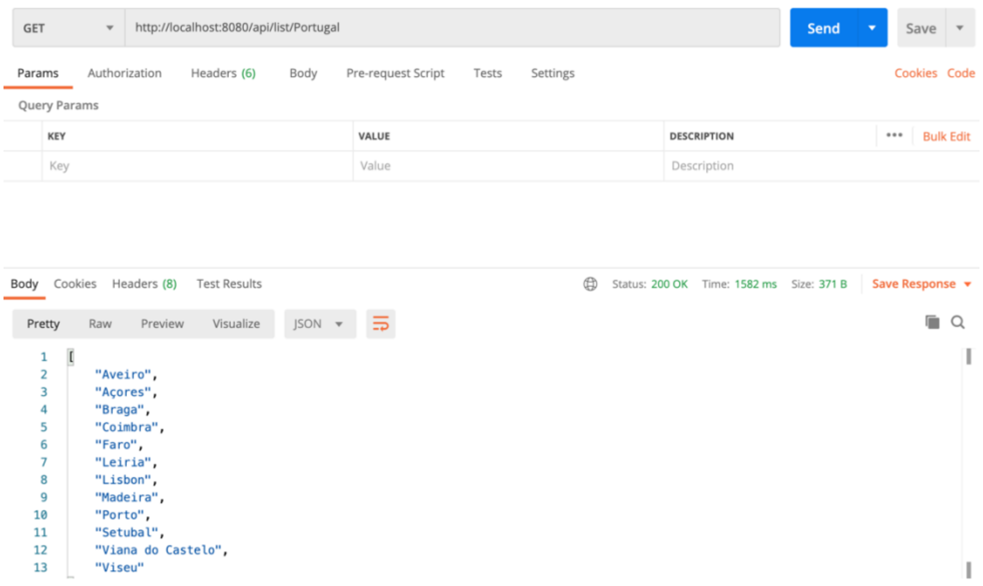


GET http://localhost:8080/api/list/

Status: 200 OK Time: 1463 ms Size: 1.37 KB

```
[
    "Afghanistan",
    "Algeria",
    "Andorra",
    "Angola",
    "Argentina",
    "Armenia",
    "Australia",
    "Austria",
    "Bahamas",
    "Bahrain",
    "Bangladesh",
    "Belgium",
```

- **/api/list/{country}** (returns the list of available states from the selected country)
    - TYPE: GET

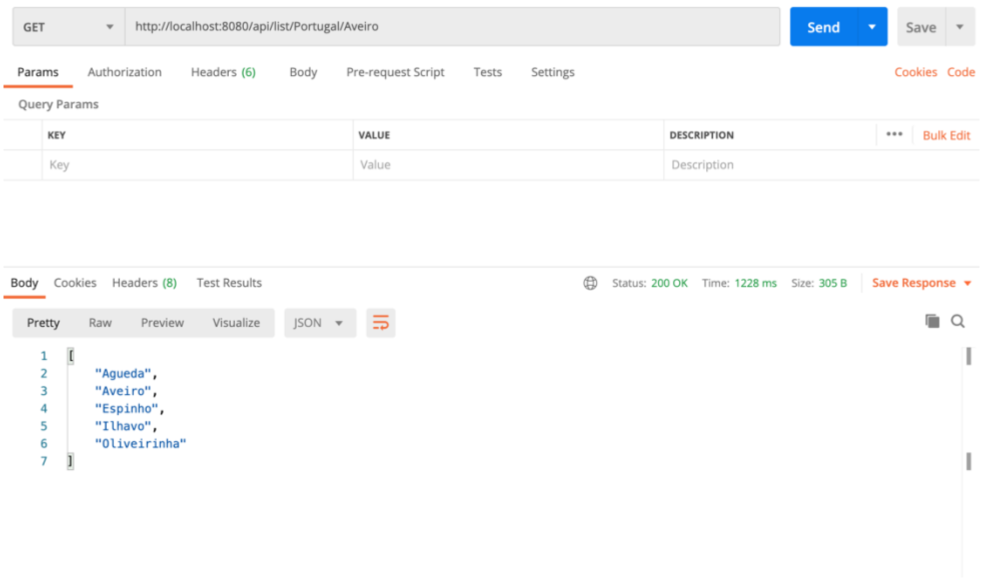


http://localhost:8080/api/list/Portugal

Status: 200 OK Time: 1582 ms Size: 371 B

```
[
    "Aveiro",
    "Açores",
    "Braga",
    "Coimbra",
    "Faro",
    "Leiria",
    "Lisbon",
    "Madeira",
    "Porto",
    "Setubal",
    "Viana do Castelo",
    "Viseu"
```

- **/api/list/{country}/{state}** (returns all the cities available from the selected state)
    - Type: GET

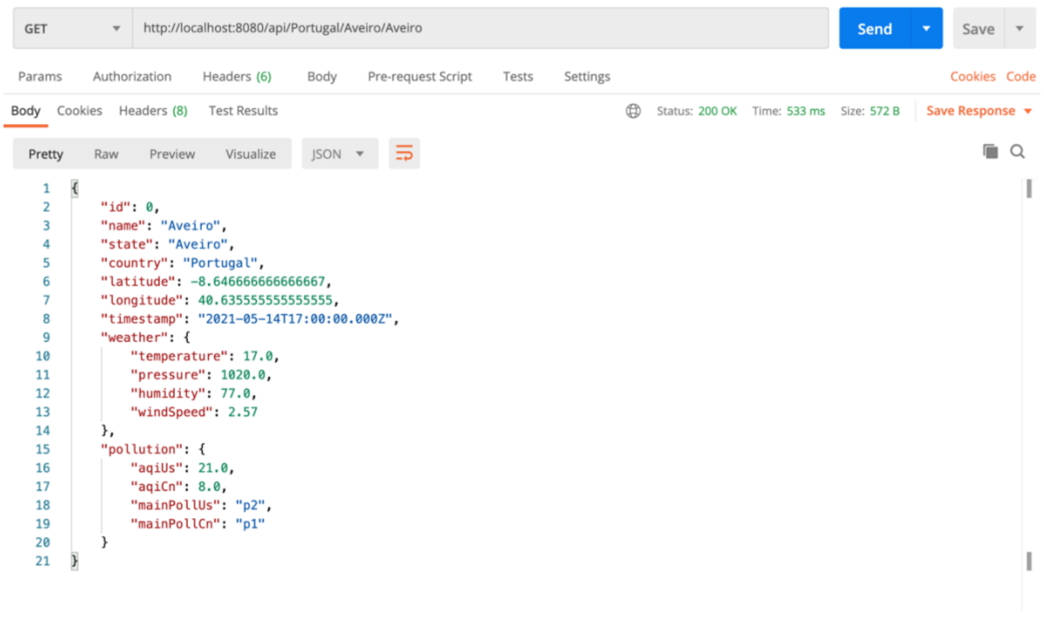


http://localhost:8080/api/list/Portugal/Aveiro

Status: 200 OK Time: 1228 ms Size: 305 B

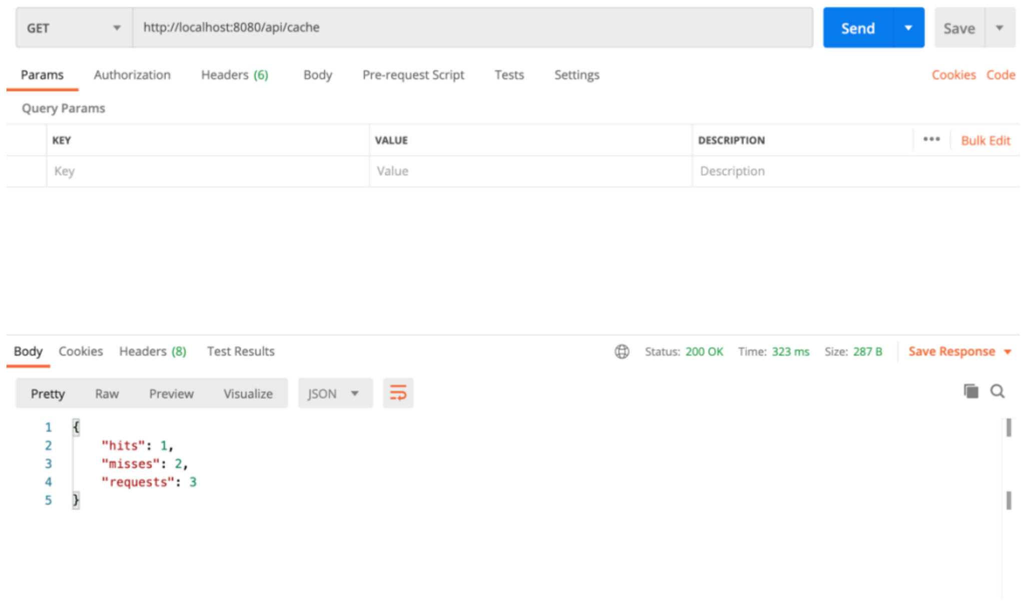
```
[
    "Agueda",
    "Aveiro",
    "Espinho",
    "Ilhavo",
    "Oliveirinha"
]
```

- **/api/{country}/{state}/{city} (**returns the weather and pollution forecast)
  - Type: GET



```
{
    "id": 0,
    "name": "Aveiro",
    "state": "Aveiro",
    "country": "Portugal",
    "latitude": -8.646666666666667,
    "longitude": 40.635555555555555,
    "timestamp": "2021-05-14T17:00:00.000Z",
    "weather": {
        "temperature": 17.0,
        "pressure": 1020.0,
        "humidity": 77.0,
        "windSpeed": 2.57
    },
    "pollution": {
        "aqiUs": 21.0,
        "aqiCn": 8.0,
        "mainPollUs": "p2",
        "mainPollCn": "p1"
    }
}
```

- **/api/cache** (returns the hits, misses and requests of the cache)
  - Type: GET



```
{
    "hits": 1,
    "misses": 2,
    "requests": 3
}
```

3 Quality assurance

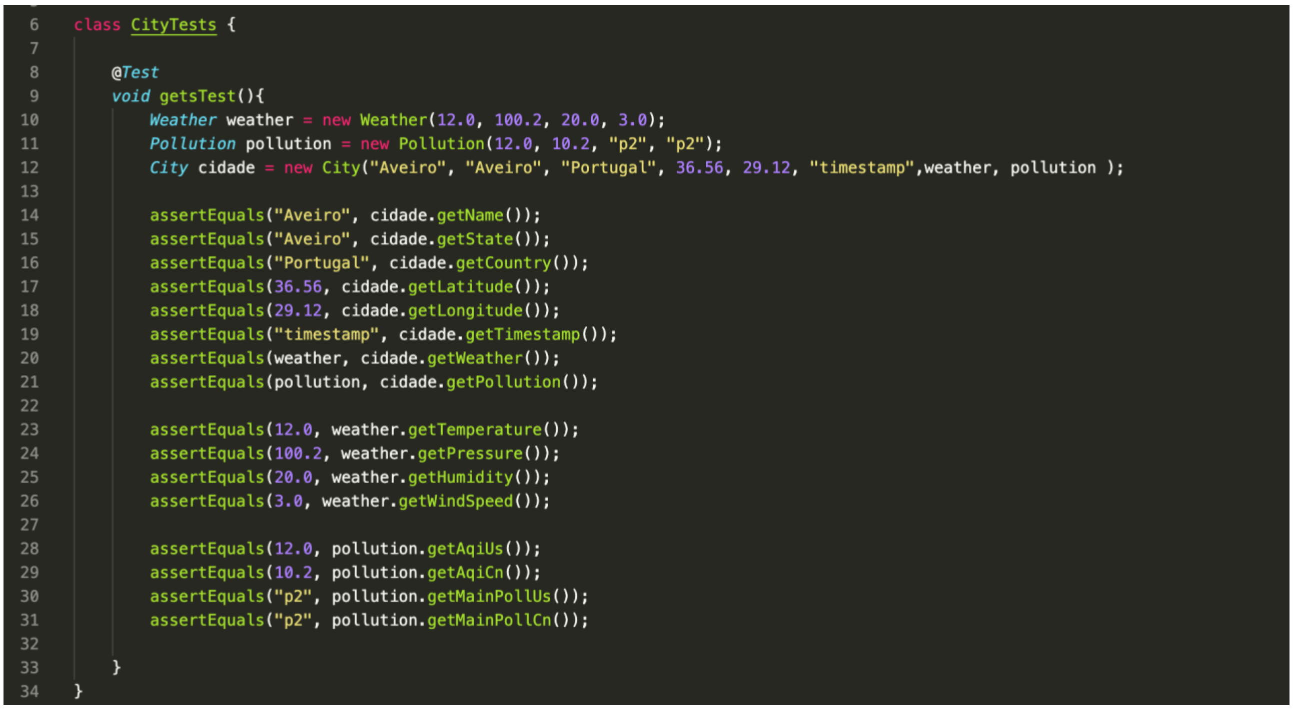
3.1 Overall strategy for testing

First, I decided to do TDD (Test Driven Development). But then I realized that the REST API and the frontend were quite simple to do. So, I decided to the skeleton of the REST API and then write the tests (Before making the API I already had the idea to most tests and wrote it on a paper).
Almost everything has tests implemented. For unit tests, I used **Junit5.** Then, for integration and service tests, **Mockito** and **SpringBoot MockMvc** was used. Finally, for functional tests on the web interface, **Selenium WebDriver** was used.

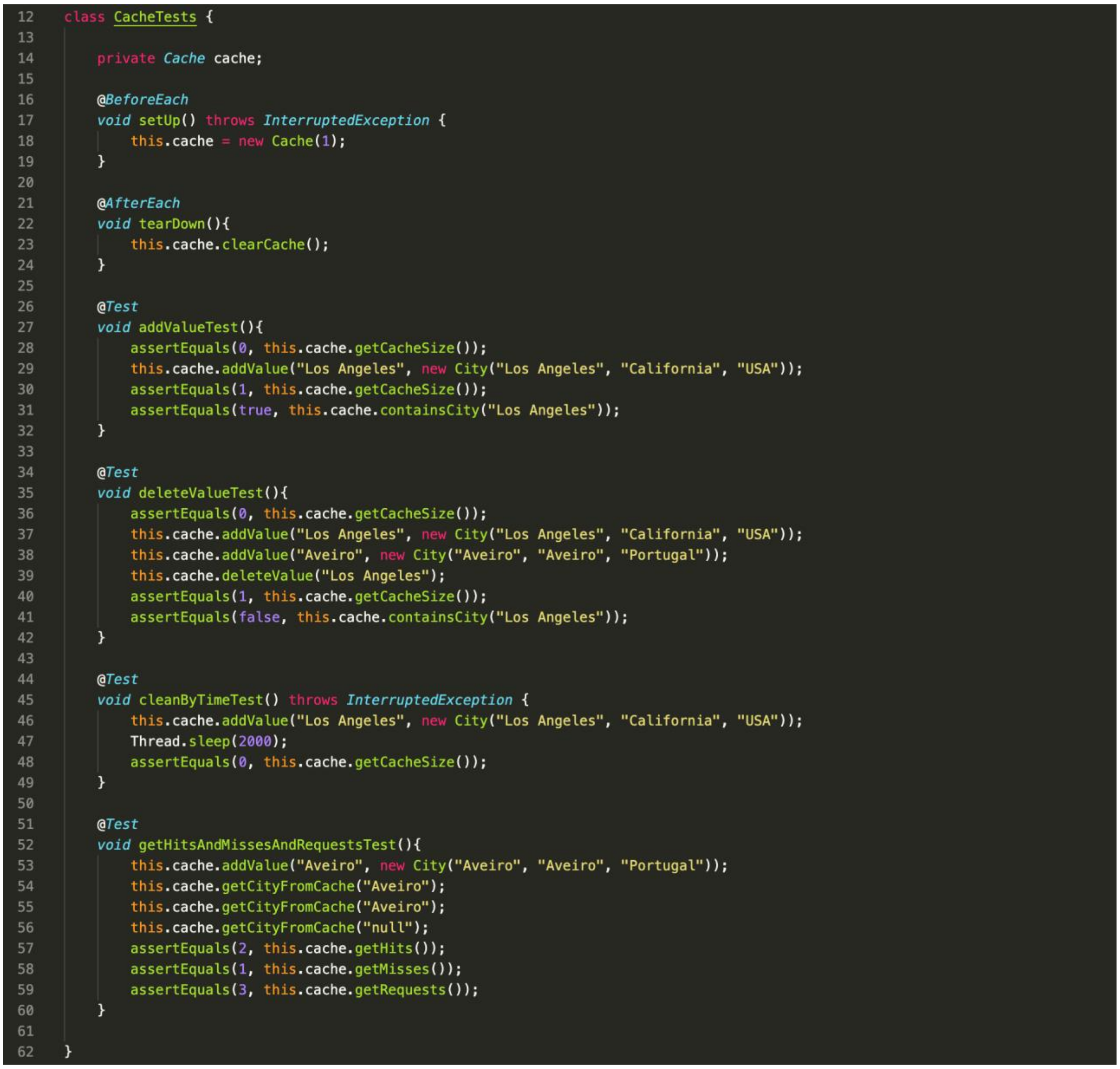
3.2 Unit and integration testing

Unit Tests

Starting by the Unit tests, I decided to write tests on the classes (**City, Weather, Pollution**) just because the % of code coverage was getting lower because of those classes. These are simple asserts on automatically generated get methods:

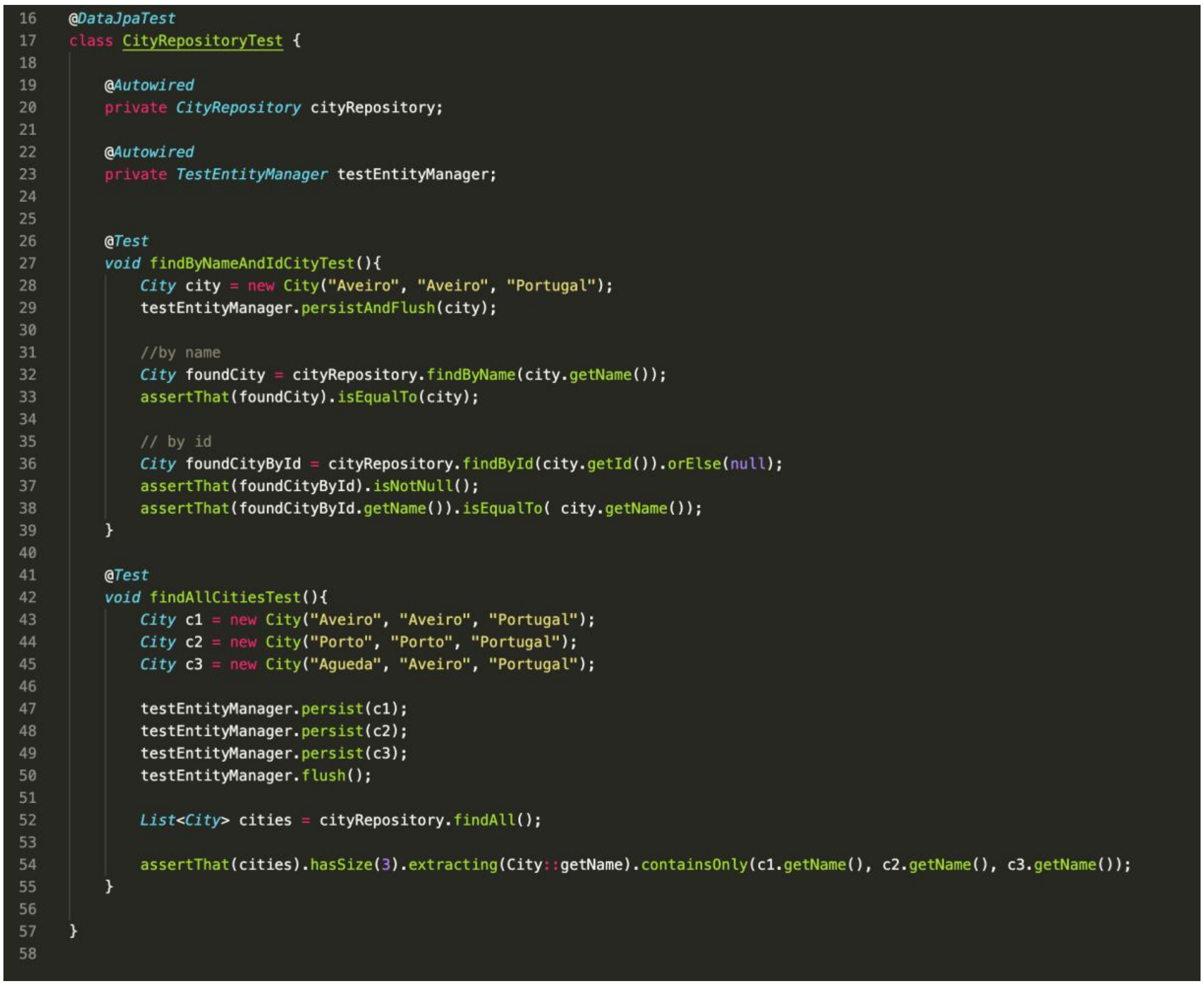

```java
class CityTests {

    @Test
    void getsTest(){
        Weather weather = new Weather(12.0, 100.2, 20.0, 3.0);
        Pollution pollution = new Pollution(12.0, 10.2, "p2", "p2");
        City cidade = new City("Aveiro", "Aveiro", "Portugal", 36.56, 29.12, "timestamp",weather, pollution );

        assertEquals("Aveiro", cidade.getName());
        assertEquals("Aveiro", cidade.getState());
        assertEquals("Portugal", cidade.getCountry());
        assertEquals(36.56, cidade.getLatitude());
        assertEquals(29.12, cidade.getLongitude());
        assertEquals("timestamp", cidade.getTimestamp());
        assertEquals(weather, cidade.getWeather());
        assertEquals(pollution, cidade.getPollution());

        assertEquals(12.0, weather.getTemperature());
        assertEquals(100.2, weather.getPressure());
        assertEquals(20.0, weather.getHumidity());
        assertEquals(3.0, weather.getWindSpeed());

        assertEquals(12.0, pollution.getAqiUs());
        assertEquals(10.2, pollution.getAqiCn());
        assertEquals("p2", pollution.getMainPollUs());
        assertEquals("p2", pollution.getMainPollCn());

    }
}
```

Then, most of my Unit tests were written for the **Cache** class. I made tests to add/delete an entry, a test to verify the Thread that implements the time-to-live policy (check whether the cache cleans over time or not) and a test to verify the hits, misses and requests:

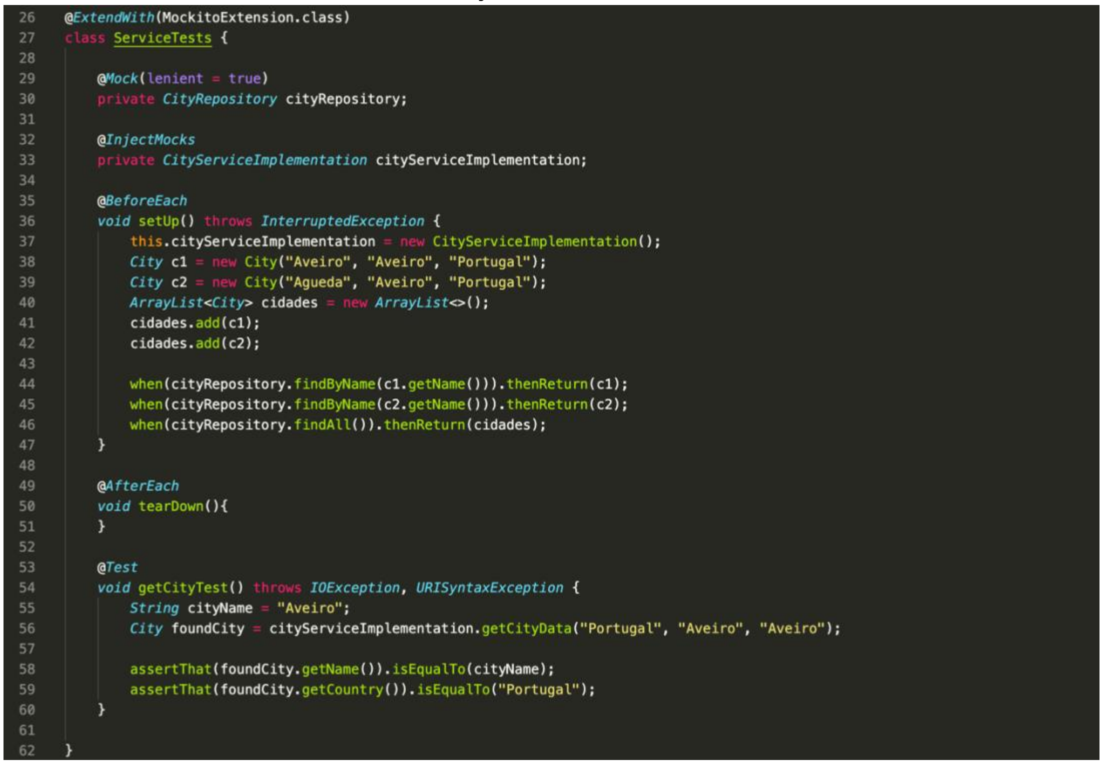


```java
class CacheTests {

    private Cache cache;

    @BeforeEach
    void setUp() throws InterruptedException {
        this.cache = new Cache(1);
    }

    @AfterEach
    void tearDown(){
        this.cache.clearCache();
    }

    @Test
    void addValueTest(){
        assertEquals(0, this.cache.getCacheSize());
        this.cache.addValue("Los Angeles", new City("Los Angeles", "California", "USA"));
        assertEquals(1, this.cache.getCacheSize());
        assertEquals(true, this.cache.containsCity("Los Angeles"));
    }

    @Test
    void deleteValueTest(){
        assertEquals(0, this.cache.getCacheSize());
        this.cache.addValue("Los Angeles", new City("Los Angeles", "California", "USA"));
        this.cache.addValue("Aveiro", new City("Aveiro", "Aveiro", "Portugal"));
        this.cache.deleteValue("Los Angeles");
        assertEquals(1, this.cache.getCacheSize());
        assertEquals(false, this.cache.containsCity("Los Angeles"));
    }

    @Test
    void cleanByTimeTest() throws InterruptedException {
        this.cache.addValue("Los Angeles", new City("Los Angeles", "California", "USA"));
        Thread.sleep(2000);
        assertEquals(0, this.cache.getCacheSize());
    }

    @Test
    void getHitsAndMissesAndRequestsTest(){
        this.cache.addValue("Aveiro", new City("Aveiro", "Aveiro", "Portugal"));
        this.cache.getCityFromCache("Aveiro");
        this.cache.getCityFromCache("Aveiro");
        this.cache.getCityFromCache("null");
        assertEquals(2, this.cache.getHits());
        assertEquals(1, this.cache.getMisses());
        assertEquals(3, this.cache.getRequests());
    }

}
```

Then, some basic unit tests were made for **CityRepository**. The class **TestEntityManager** was used to make these tests possible. Also, the annotation **@DataJpaTest**. There are only 2 tests written for this class, as most of it is generated by **JPA**:

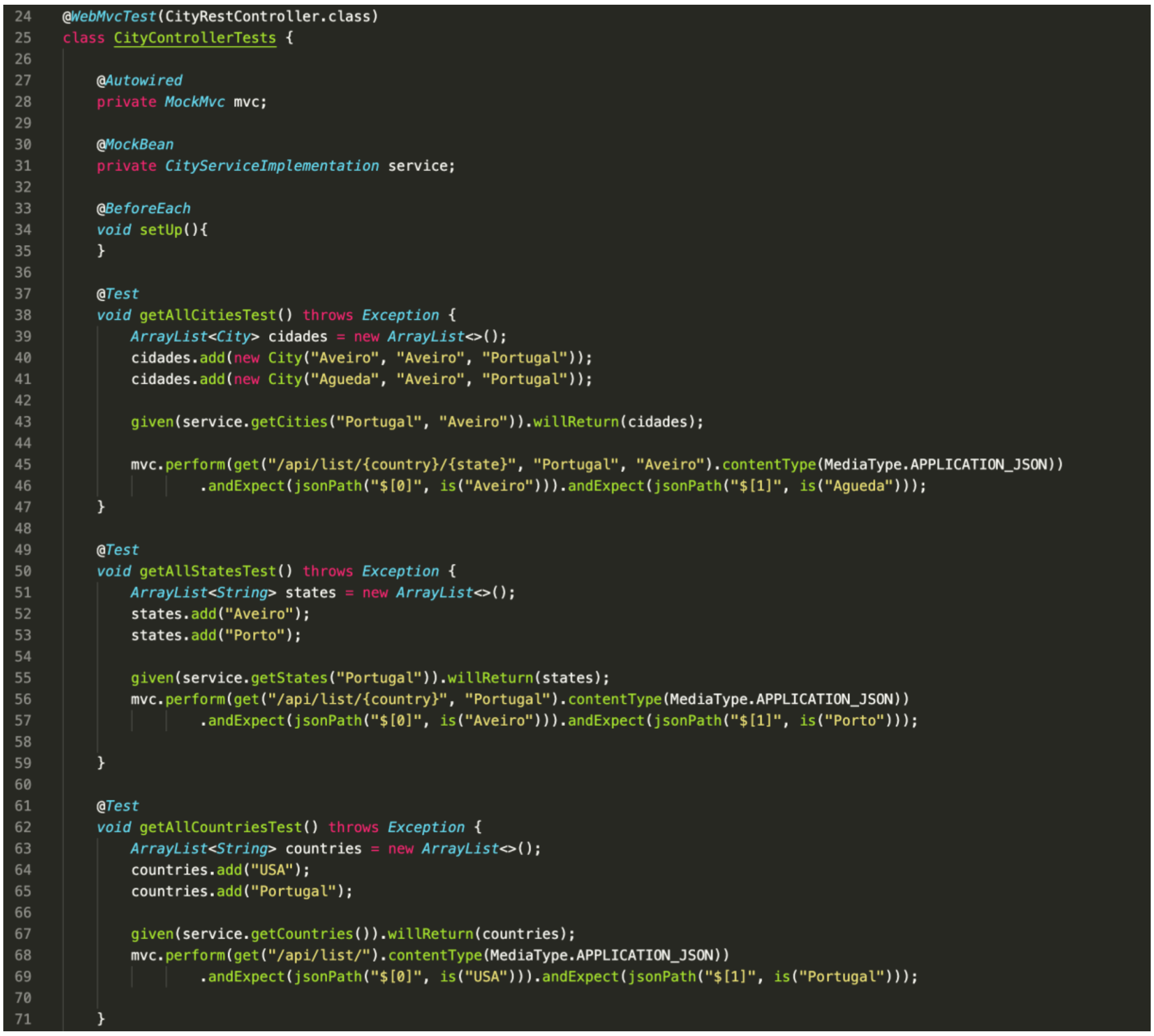

```java
@DataJpaTest
class CityRepositoryTest {

    @Autowired
    private CityRepository cityRepository;

    @Autowired
    private TestEntityManager testEntityManager;


    @Test
    void findByNameAndIdCityTest(){
        City city = new City("Aveiro", "Aveiro", "Portugal");
        testEntityManager.persistAndFlush(city);

        //by name
        City foundCity = cityRepository.findByName(city.getName());
        assertThat(foundCity).isEqualTo(city);

        // by id
        City foundCityById = cityRepository.findById(city.getId()).orElse(null);
        assertThat(foundCityById).isNotNull();
        assertThat(foundCityById.getName()).isEqualTo( city.getName());
    }

    @Test
    void findAllCitiesTest(){
        City c1 = new City("Aveiro", "Aveiro", "Portugal");
        City c2 = new City("Porto", "Porto", "Portugal");
        City c3 = new City("Agueda", "Aveiro", "Portugal");

        testEntityManager.persist(c1);
        testEntityManager.persist(c2);
        testEntityManager.persist(c3);
        testEntityManager.flush();

        List<City> cities = cityRepository.findAll();

        assertThat(cities).hasSize(3).extracting(City::getName).containsOnly(c1.getName(), c2.getName(), c3.getName());
    }

}
```

Then, I wrote **service level** tests using dependency isolation **Mocks**. As was said earlier, I should have written more tests for the service, but didn't due to lack of time. Still, I wrote a test to the method that returns the city:

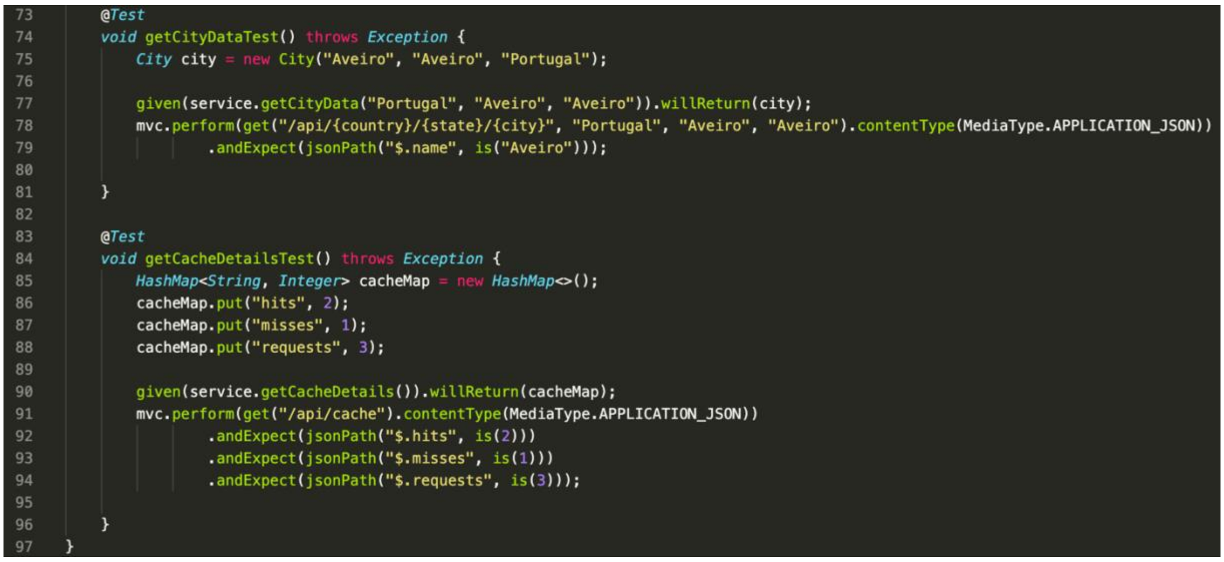

```java
@ExtendWith(MockitoExtension.class)
class ServiceTests {

    @Mock(lenient = true)
    private CityRepository cityRepository;

    @InjectMocks
    private CityServiceImplementation cityServiceImplementation;

    @BeforeEach
    void setUp() throws InterruptedException {
        this.cityServiceImplementation = new CityServiceImplementation();
        City c1 = new City("Aveiro", "Aveiro", "Portugal");
        City c2 = new City("Agueda", "Aveiro", "Portugal");
        ArrayList<City> cidades = new ArrayList<>();
        cidades.add(c1);
        cidades.add(c2);

        when(cityRepository.findByName(c1.getName())).thenReturn(c1);
        when(cityRepository.findByName(c2.getName())).thenReturn(c2);
        when(cityRepository.findAll()).thenReturn(cidades);
    }

    @AfterEach
    void tearDown(){
    }

    @Test
    void getCityTest() throws IOException, URISyntaxException {
        String cityName = "Aveiro";
        City foundCity = cityServiceImplementation.getCityData("Portugal", "Aveiro", "Aveiro");

        assertThat(foundCity.getName()).isEqualTo(cityName);
        assertThat(foundCity.getCountry()).isEqualTo("Portugal");
    }

}
```

Integration Tests

For the integration tests, **SpringBoot MockMvc** was used. Every endpoint has a test written for it, including the cache endpoint. Basically, I wrote a given statement and prepared a response from the service. Then, performed a request to the endpoint and verified whether it succeeds or not:



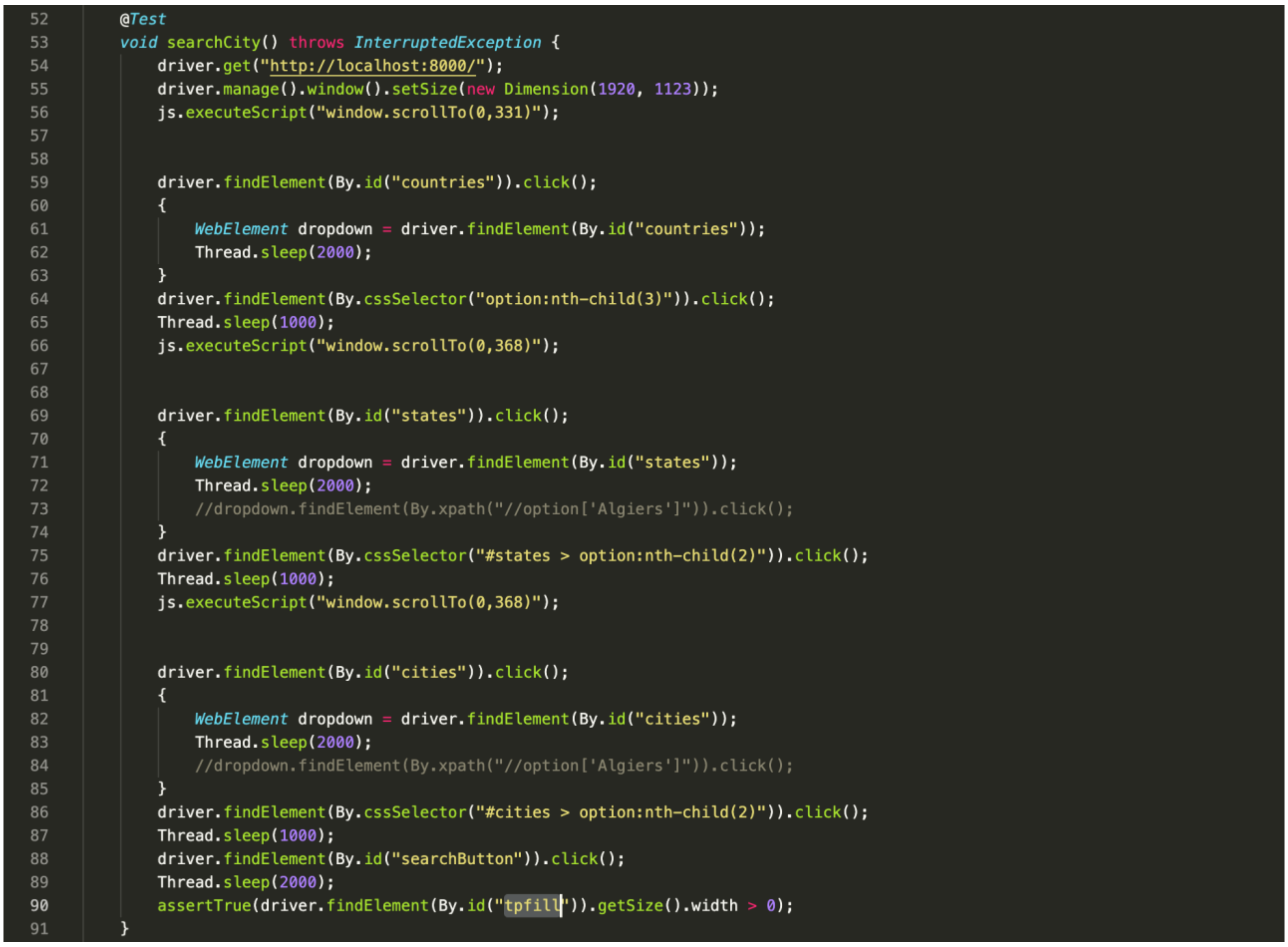
```java
@WebMvcTest(CityRestController.class)
class CityControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private CityServiceImplementation service;

    @BeforeEach
    void setUp(){
    }

    @Test
    void getAllCitiesTest() throws Exception {
        ArrayList<City> cidades = new ArrayList<>();
        cidades.add(new City("Aveiro", "Aveiro", "Portugal"));
        cidades.add(new City("Agueda", "Aveiro", "Portugal"));

        given(service.getCities("Portugal", "Aveiro")).willReturn(cidades);

        mvc.perform(get("/api/list/{country}/{state}", "Portugal", "Aveiro").contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$[0]", is("Aveiro"))).andExpect(jsonPath("$[1]", is("Agueda")));
    }

    @Test
    void getAllStatesTest() throws Exception {
        ArrayList<String> states = new ArrayList<>();
        states.add("Aveiro");
        states.add("Porto");

        given(service.getStates("Portugal")).willReturn(states);
        mvc.perform(get("/api/list/{country}", "Portugal").contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$[0]", is("Aveiro"))).andExpect(jsonPath("$[1]", is("Porto")));

    }

    @Test
    void getAllCountriesTest() throws Exception {
        ArrayList<String> countries = new ArrayList<>();
        countries.add("USA");
        countries.add("Portugal");

        given(service.getCountries()).willReturn(countries);
        mvc.perform(get("/api/list/").contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$[0]", is("USA"))).andExpect(jsonPath("$[1]", is("Portugal")));

    }
```



```java
    @Test
    void getCityDataTest() throws Exception {
        City city = new City("Aveiro", "Aveiro", "Portugal");

        given(service.getCityData("Portugal", "Aveiro", "Aveiro")).willReturn(city);
        mvc.perform(get("/api/{country}/{state}/{city}", "Portugal", "Aveiro", "Aveiro").contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.name", is("Aveiro")));

    }

    @Test
    void getCacheDetailsTest() throws Exception {
        HashMap<String, Integer> cacheMap = new HashMap<>();
        cacheMap.put("hits", 2);
        cacheMap.put("misses", 1);
        cacheMap.put("requests", 3);

        given(service.getCacheDetails()).willReturn(cacheMap);
        mvc.perform(get("/api/cache").contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.hits", is(2)))
                .andExpect(jsonPath("$.misses", is(1)))
                .andExpect(jsonPath("$.requests", is(3)));

    }
}
```

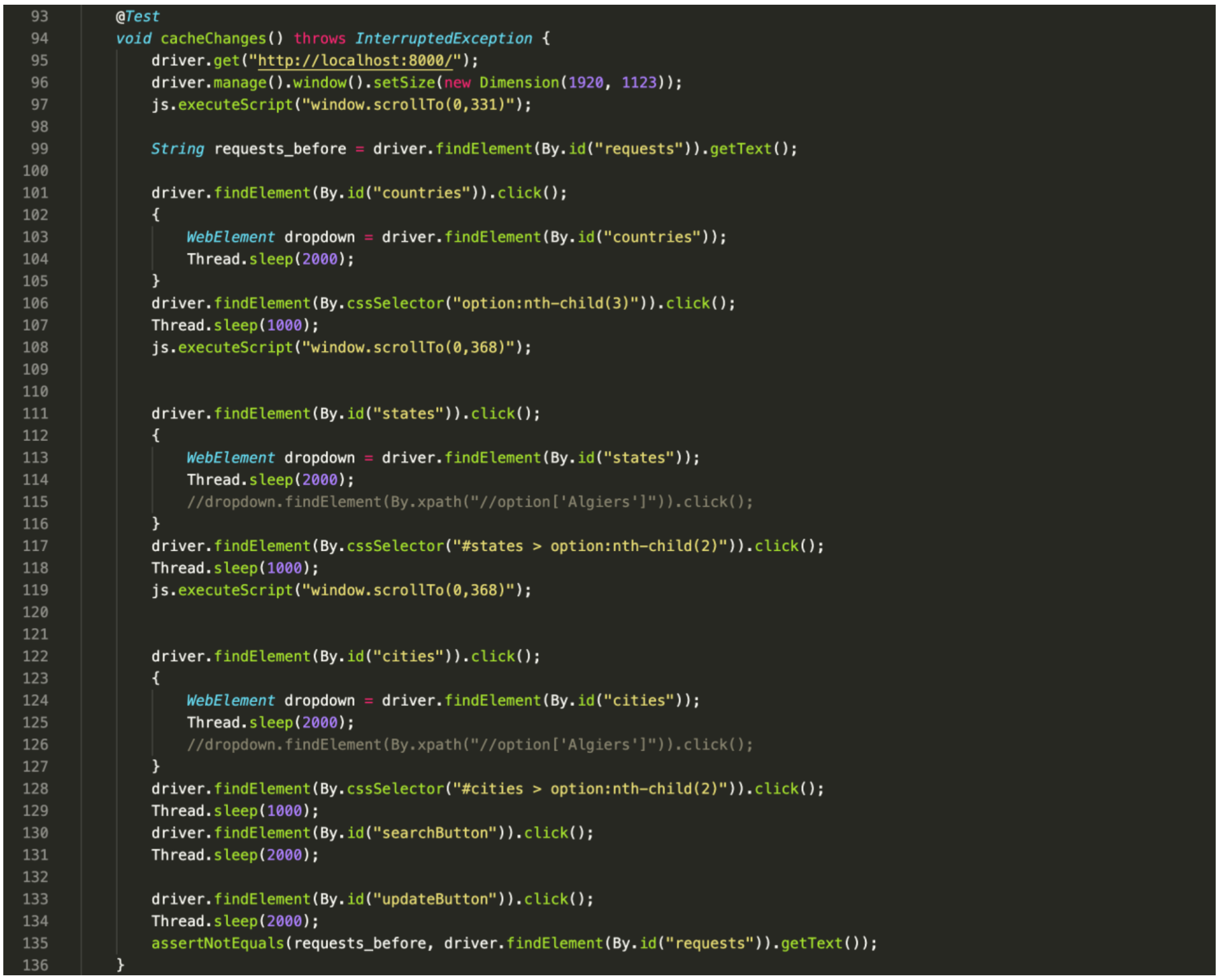
3.3 Functional testing

As I said earlier, I used the **Selenium WebDriver** to make these tests (FireFox Driver). A note that I want to add here is that I had to put a lot of sleeps in between the clicks so that the selenium could find the html items. Although most of this code is generated by de Selenium IDE extension, I had to make several changes to it in order to make it find the HTML elements. The tests made here are pretty much the interaction of the user with the webpage. The first test, chooses a city, state and country, searches and verifies if the data is showing:



```java
    @Test
    void searchCity() throws InterruptedException {
        driver.get("http://localhost:8000/");
        driver.manage().window().setSize(new Dimension(1920, 1123));
        js.executeScript("window.scrollTo(0,331)");


        driver.findElement(By.id("countries")).click();
        {
            WebElement dropdown = driver.findElement(By.id("countries"));
            Thread.sleep(2000);
        }
        driver.findElement(By.cssSelector("option:nth-child(3)")).click();
        Thread.sleep(1000);
        js.executeScript("window.scrollTo(0,368)");


        driver.findElement(By.id("states")).click();
        {
            WebElement dropdown = driver.findElement(By.id("states"));
            Thread.sleep(2000);
            //dropdown.findElement(By.xpath("//option['Algiers']")).click();
        }
        driver.findElement(By.cssSelector("#states > option:nth-child(2)")).click();
        Thread.sleep(1000);
        js.executeScript("window.scrollTo(0,368)");


        driver.findElement(By.id("cities")).click();
        {
            WebElement dropdown = driver.findElement(By.id("cities"));
            Thread.sleep(2000);
            //dropdown.findElement(By.xpath("//option['Algiers']")).click();
        }
        driver.findElement(By.cssSelector("#cities > option:nth-child(2)")).click();
        Thread.sleep(1000);
        driver.findElement(By.id("searchButton")).click();
        Thread.sleep(2000);
        assertTrue(driver.findElement(By.id("tpfill")).getSize().width > 0);
    }
```

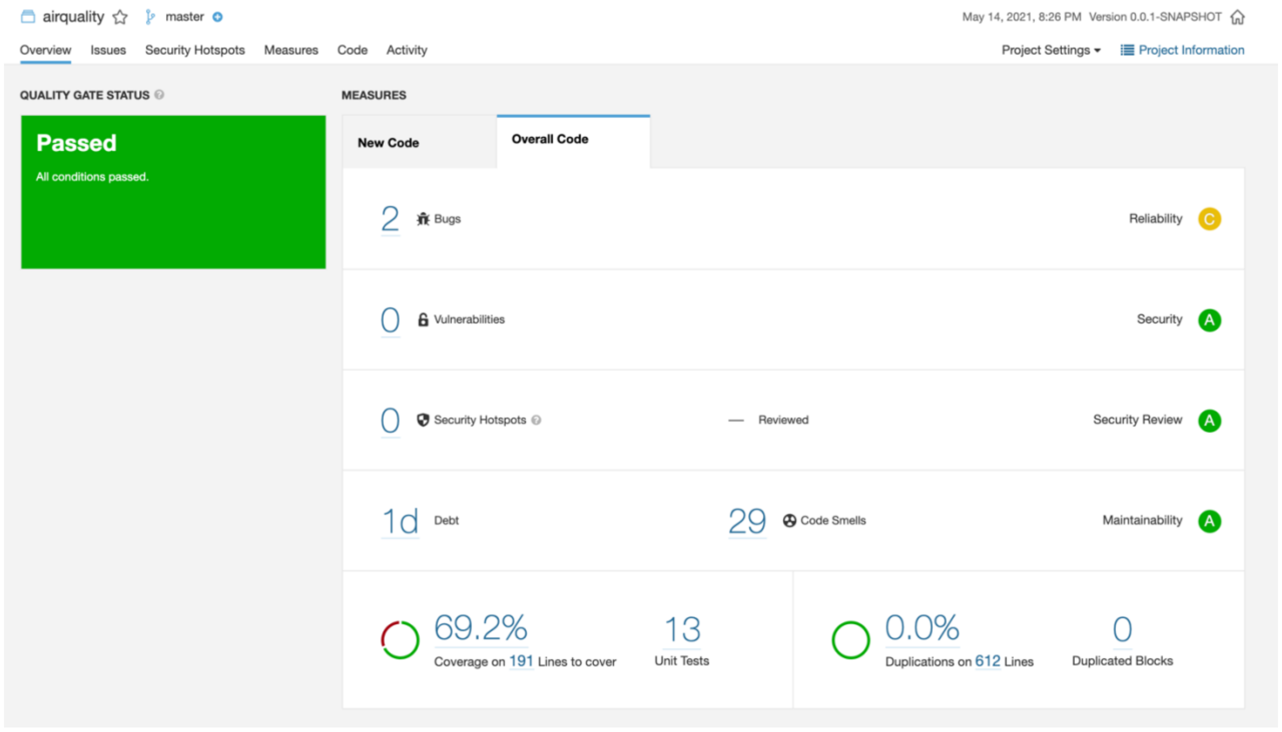
The second one, does the same but verifies if the cache values are updated when clicking the update button. It verifies if the initial number of requests is not the same as after clicking it:



```java
    @Test
    void cacheChanges() throws InterruptedException {
        driver.get("http://localhost:8000/");
        driver.manage().window().setSize(new Dimension(1920, 1123));
        js.executeScript("window.scrollTo(0,331)");

        String requests_before = driver.findElement(By.id("requests")).getText();

        driver.findElement(By.id("countries")).click();
        {
            WebElement dropdown = driver.findElement(By.id("countries"));
            Thread.sleep(2000);
        }
        driver.findElement(By.cssSelector("option:nth-child(3)")).click();
        Thread.sleep(1000);
        js.executeScript("window.scrollTo(0,368)");


        driver.findElement(By.id("states")).click();
        {
            WebElement dropdown = driver.findElement(By.id("states"));
            Thread.sleep(2000);
            //dropdown.findElement(By.xpath("//option['Algiers']")).click();
        }
        driver.findElement(By.cssSelector("#states > option:nth-child(2)")).click();
        Thread.sleep(1000);
        js.executeScript("window.scrollTo(0,368)");


        driver.findElement(By.id("cities")).click();
        {
            WebElement dropdown = driver.findElement(By.id("cities"));
            Thread.sleep(2000);
            //dropdown.findElement(By.xpath("//option['Algiers']")).click();
        }
        driver.findElement(By.cssSelector("#cities > option:nth-child(2)")).click();
        Thread.sleep(1000);
        driver.findElement(By.id("searchButton")).click();
        Thread.sleep(2000);

        driver.findElement(By.id("updateButton")).click();
        Thread.sleep(2000);
        assertNotEquals(requests_before, driver.findElement(By.id("requests")).getText());
    }
```

3.4 Static code analysis

For static code analysis, **SonarQube** was used. Here are the results:


airquality master
May 14, 2021, 8:26 PM Version 0.0.1-SNAPSHOT
Overview Issues Security Hotspots Measures Code Activity
Project Settings Project Information

QUALITY GATE STATUS
Passed
All conditions passed.

MEASURES
New Code | Overall Code

2 Bugs — Reliability C
0 Vulnerabilities — Security A
0 Security Hotspots — Reviewed — Security Review A
1d Debt — 29 Code Smells — Maintainability A
69.2% Coverage on 191 Lines to cover — 13 Unit Tests
0.0% Duplications on 612 Lines — 0 Duplicated Blocks

Most of the code smells are the sleeps that I added between the clicks on the Selenium tests, which was explained before. Because of that, I had no choice but to keep them. The code coverage is lowered by the Swagger2Config class and by the service – as I said earlier, should have more tests in this class.
With this, I actually learned that it is better to write good code right from the beginning. This is because the first time I tested the project on SonarQube, it had failed with 111 code smells (A lot of unused imports and bad variable names). I also learned that there are some issues that sonar reports that might not be valid.

4 References & resources

Project resources

- Video demo https://www.youtube.com/watch?v=8K9fgKWcC-Y and on repository
- Ready to use application: Although it is not deployed anywhere, there is a docker-compose.yml in the root of the repository. To run the application:

  docker-compose up –build

  The application will be running on localhost:8000, the backend on localhost:8080

Reference materials

External API used:

https://api-docs.iqair.com/?version=latest