HW1: Mid-term assignment report

Inês Leite [92928], 2021-05-14

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 made, Air Quality, has the goal of providing the Air Quality data of a certain location, given its geographical coordinates.

GitHub link: https://github.com/inespl/TQS

1.2 Current limitations

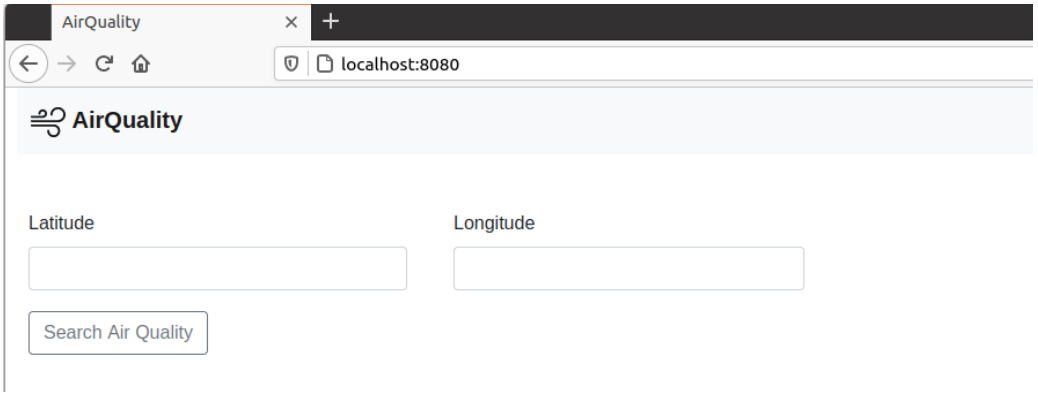
There isn't validation of the coordinates data which leads to Whitelabel Error Pages. Additionally, it should check if the API response has relevant data for certain coordinates (some are in the middle of the Artic, and don't have data).

2 Product specification

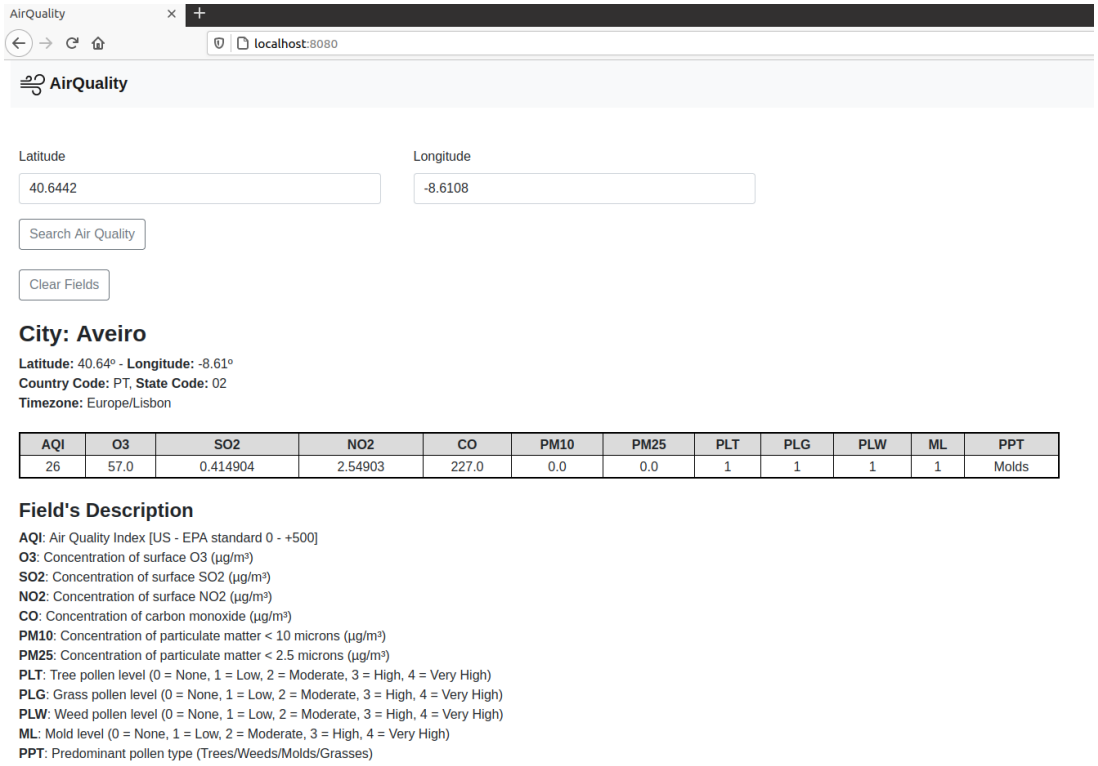
2.1 Functional scope and supported interactions

The application has 2 ways of interaction: directly, using the front-end, and using de API endpoints created by the application.

Through the front-end there is a simple interface that allows you to input two values, one for latitude and another for longitude, obtaining the wanted location's geographical coordinates.



After filling those fields with adequate values and clicking on the ‘Search Air Quality’ button, appears a section below this one with some information about the place of the coordinates inserted and a table with several data related to Air Quality. There is also a description of the initials of the table.



The second way of interacting with this application is by using the API endpoints. [See 2.3]

The REST API invoked to get the air quality data is by WeatherBit.io[1] and it's called with the function **callGetAirQualityByLatLon()** and requires two doubles, one for the latitude another for the longitude. This method returns a string of the air quality data in a JSON format.

```
public String callGetAirQualityByLatLon(double lat, double lon){
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));

    HttpEntity<String> entity = new HttpEntity<>( body: "parameters", headers);

    String url = AIR_QUALITY_DATA + "lat=" + lat + "&lon=" + lon + KEY;
    ResponseEntity<String> result = restTemplate.exchange(url, HttpMethod.GET, entity, String.class);

    return result.getBody();
}
```

The in-memory Cache is implemented by a class called Cache, it has 2 attributes, an integer time-to-live (TTL) and a HashMap<String, CacheObject>. The CacheObject is a class with 2 attributes also, a long that is created when the class is instanced and a string that contains the air quality data in JSON format.

```
public class Cache {
    private final Map<String, CacheObject> cacheMap;
    private final long timeToLive;

    public static class CacheObject {
        private final long lastAccessed = System.currentTimeMillis();
        private final String quality;

        public CacheObject(String quality) {
            this.quality = quality;
        }

        private String getQuality(){
            return this.quality;
        }
    }

    public Cache(long ttl) {
        this.timeToLive = ttl;
        cacheMap = new HashMap<>();
    }
```

The Cache class has two main methods, put and get values. The get function checks if it contains that key and, if so, if the TTL has passed, if it has, removes the object from the map, if it hasn't passed, returns the value.

When the user gives the coordinates the home1() function is called. First, it checks if the cache has that air quality data (**cache.get()**) , if it doesn't the **callGetAirQualityByLatLon()** function is called. Then with the response from one of the previous methods it's created an

[1] WeatherBit.io: https://www.weatherbit.io/api/airquality-current

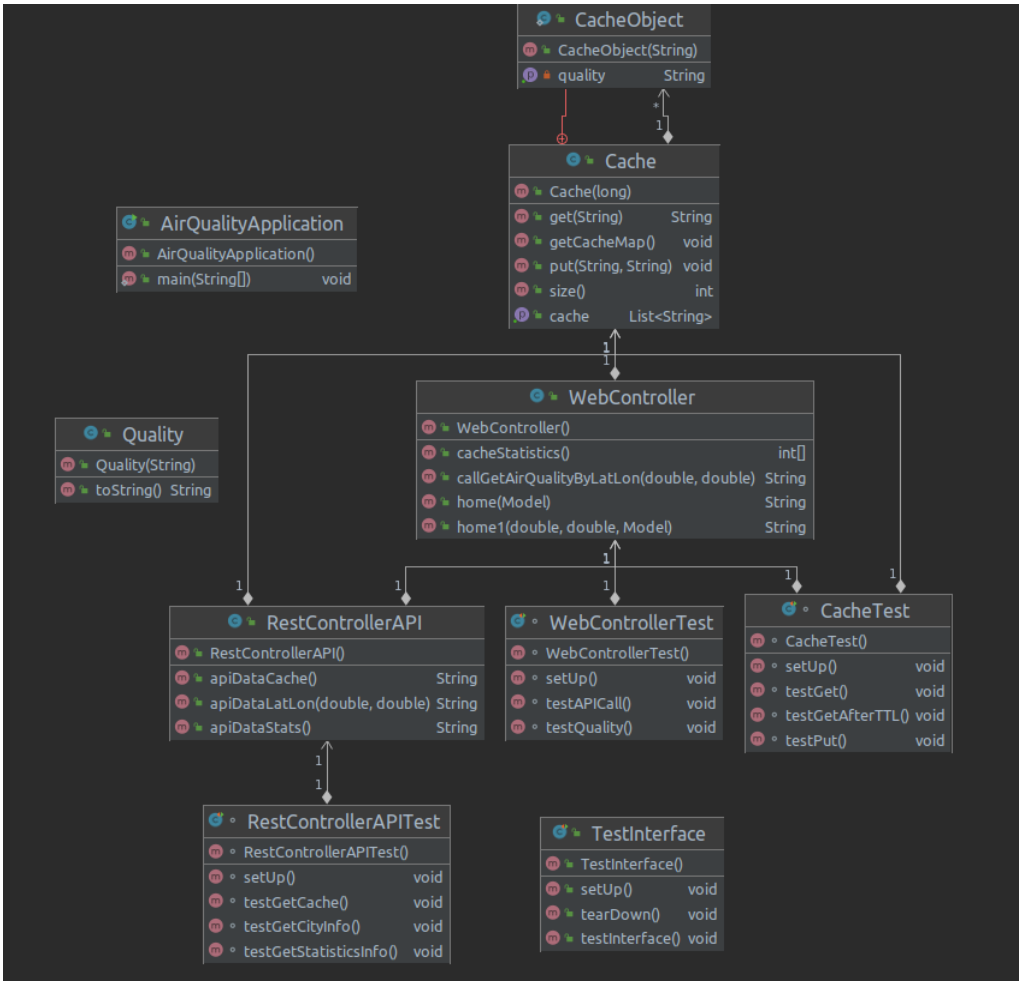
instance of Quality (the class that has attributes for each of the fields in the JSON string) and sent to the HTML file.

```java
@PostMapping("/")
public String home1(@RequestParam(name = "lat") double lat, @RequestParam(name = "lon") double lon, Model model) throws IOException {
    bf = new BufferedWriter(fw);
    bf.write(String.format("%s %s \t INFO \t\t Request for Air Quality of Lat: %f, Lon: %f%n", localDateTime[0], localDateTime[1], lat, lon));
    Quality quality;
    String latlon = lat + "," + lon;
    String qualityJson = cache.get(latlon);

    cache.getCacheMap();
    if (qualityJson == null){
        misses ++;
        qualityJson = callGetAirQualityByLatLon(lat, lon);
        cache.put(latlon, qualityJson);
        bf.write(String.format("%s %s \t SUCCESS \t Request Accepted, API returned the result (not found in cache)%n", localDateTime[0], localDateTime[1]));
    }else {
        hits++;
        bf.write(String.format("%s %s \t SUCCESS \t Request Accepted and found in cache%n", localDateTime[0], localDateTime[1]));
    }
    quality = new Quality(qualityJson);
```

2.2 System architecture

The software architecture is presented in the following image, most of these classes were already presented and its details along with them.

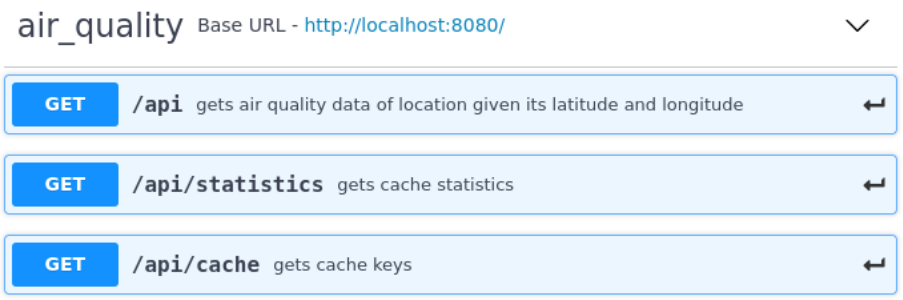


2.3 API for developers

The application's API endpoints can be accessed with the Base URL of localhost:8080/api and has three possible GET requests:

- one for the air quality data given a location's latitude and longitude
- another for the statistics of the cache usage, information on the number of requests, hits and misses and the time to live (TTL) in milliseconds
- and lastly, one that returns the current cache keys

The data for the first endpoint is obtained by the same function used by the WebController, callGetAirQualityByLatLon(),



3 Quality assurance

3.1 Overall strategy for testing

The tests were made with some of the technologies presented in class, like Mockito, Selenium, obviously unit tests. I tried to do several different tests to have the best results, unit tests more to test the Cache class, mocks for the APIs and Selenium for the front-end functionalities.

Note: I made some tests with REST-Assured on the API created, but it had to be in a different project and i didn't find that really helpful because I couldn't call on some of the methods of the main project.

3.2 Unit and integration testing

There are unit tests to verify the cache functionalities. It was made using 3 tests:

- **testPut()**
  - It simulates the insertion after the cache didn't return the location data. **+++**
- **testGet()**
  - It simulates the request of the location's data.
- **testGetAfterTTL()**
  - It simulates the request of the location's data after the TTL.

The first is a simple put, I inserted 3 items in an empty cache and tested that it's size became 3. The next is a get, after the insertion of 3 items it checks if get requests returns the value for its key. The last one is similar to testGet, but here, the test checks if the cache is removing the items after its TTL has passed.

```java
@Test
void testPut() {
    System.out.println("Size before put" + c.size());
    c.put( key: "38.7452,-9.1604", s1);
    c.put( key: "41.1333,-8.6167", s2);
    c.put( key: "41.1495,-8.6108", s3);
    System.out.println("Size after put" + c.size());
    assertThat(c.size(), is( value: 3));
}
```

```java
@Test
void testGetAfterTTL() throws InterruptedException{
    c.put( key: "38.7452,-9.1604", s1);

    TimeUnit.MILLISECONDS.sleep( timeout: 500);
    c.put( key: "41.1333,-8.6167", s2);

    TimeUnit.MILLISECONDS.sleep( timeout: 505);
    assertThat(c.get("38.7452,-9.1604"), is(nullValue()));
    assertThat(c.get("41.1333,-8.6167"), is(s2));

    TimeUnit.MILLISECONDS.sleep( timeout: 500);
    assertThat(c.get("41.1333,-8.6167"), is(nullValue()));
}
```

Regarding the WebController tests, mocks were use to simulate the API call, and 2 tests were made, if the mock was correct and if the Quality instance created had the right values.

```java
class WebControllerTest {

  double lat = 40.4;
  double lon = -8.5;
  @Mock
     WebController wc;

  @BeforeEach
  void setUp(){
      wc = mock(WebController.class);
      when(wc.callGetAirQualityByLatLon(lat, lon)).thenReturn("{\"data\":[{\"mold_level\":1,\"aqi\":41,\"pm10\":3," +
              "\"co\":235.5,\"o3\":88,\"predominant_pollen_type\":\"Molds\",\"so2\":0,\"pollen_level_tree\":1," +
              "\"pollen_level_weed\":1,\"no2\":1,\"pm25\":0.874739,\"pollen_level_grass\":1}],\"city_name\":\"Mealhada\"," +
              "\"lon\":-8.5,\"timezone\":\"Europe\\/Lisbon\",\"lat\":40.4,\"country_code\":\"PT\",\"state_code\":\"02\"}\n");
  }
```

```java
@Test
void testAPICall(){
    assertThat(wc.callGetAirQualityByLatLon(lat, lon), is( value: "{\"data\":[{\"mold_level\":1,\"aqi\":41,\"pm10\":3," +
            "\"co\":235.5,\"o3\":88,\"predominant_pollen_type\":\"Molds\",\"so2\":0,\"pollen_level_tree\":1," +
            "\"pollen_level_weed\":1,\"no2\":1,\"pm25\":0.874739,\"pollen_level_grass\":1}],\"city_name\":\"Mealhada\"," +
            "\"lon\":-8.5,\"timezone\":\"Europe\\/Lisbon\",\"lat\":40.4,\"country_code\":\"PT\",\"state_code\":\"02\"}\n"));
}

@Test
  void testQuality(){
    Quality q = new Quality(wc.callGetAirQualityByLatLon(lat, lon));
    assertThat(q.moldLevel, is( value: 1));
    assertThat(q.aqi, is( value: 41));
    assertThat(q.pm10, is( value: 3.0));
    assertThat(q.co, is( value: 235.5));
    assertThat(q.predominantPollenType, is( value: "Molds"));
    assertThat(q.pollenLevelTree, is( value: 1));
    assertThat(q.pollenLevelWeed, is( value: 1));
    assertThat(q.no2, is( value: 1.0));
    assertThat(q.pm25, is( value: 0.874739));
    assertThat(q.pollenLevelGrass, is( value: 1));
    assertThat(q.cityName, is( value: "Mealhada"));
    assertThat(q.lon, is( value: -8.5));
    assertThat(q.timezone, is( value: "Europe/Lisbon"));
    assertThat(q.lat, is( value: 40.4));
    assertThat(q.countryCode, is( value: "PT"));
    assertThat(q.stateCode, is( value: "02"));
}
```

For the integration tests of the API created, its was used MockMVC and 3 tests were made, for each one of the the endpoints:

- **testGetCityInfo()**
  - Checks if the requests made has the city information correct (as the current values of the air quality keep changing, it was only tested the city information)
- **testGetStatisticsInfo()**
  - Checks if is has the fields “requests”, “hits” and “misses”
- **testGetCache()**
  - checks if it has the field “data”.

```
class RestControllerAPITest {

    MockMvc mockMvc;

    @Mock
    RestControllerAPI rest_controller;

    @BeforeEach
    void setUp() throws IOException {
        rest_controller = new RestControllerAPI();

        mockMvc = MockMvcBuilders.standaloneSetup(rest_controller).build();
    }
```

```
@Test
void testGetCityInfo() throws Exception {
    mockMvc.perform(get( urlTemplate: "http://localhos:8080/api")
            .param( name: "lat", ...values: "40.4")
            .param( name: "lon",  ...values: "-8.5"))
            .andExpect(content().string(containsString( substring: "\"city_name\":\"Mealhada\",\"lon\":-8.5," +
                    "\"timezone\":\"Europe\\/Lisbon\",\"lat\":40.4,\"country_code\":\"PT\",\"state_code\":\"02\"")))
            .andExpect(status().isOk());
}

@Test
void testGetStatisticsInfo() throws Exception {
    mockMvc.perform(get( urlTemplate: "http://localhos:8080/api")
            .param( name: "lat", ...values: "40.4")
            .param( name: "lon",  ...values: "-8.5"));
    mockMvc.perform(get( urlTemplate: "http://localhos:8080/api/statistics"))
            .andExpect(content().string(containsString( substring: "\"hits\"")))
            .andExpect(content().string(containsString( substring: "\"misses\"")))
            .andExpect(content().string(containsString( substring: "\"requests\"")))
            .andExpect(content().string(containsString( substring: "\"ttl\"")))
            .andExpect(status().isOk());
}

@Test
void testGetCache() throws Exception {
    mockMvc.perform(get( urlTemplate: "http://localhos:8080/api/cache"))
            .andExpect(content().string(containsString( substring: "\"data\"")))
            .andExpect(status().isOk());

}
```

3.3 Functional testing

For the functional tests it was used Selenium Web Driver, because the front-end is very simple, it was only asserted if the values of the coordinates were the same.

```
@Test
public void testInterface() {
  driver.get("http://localhost:8080/");
  driver.manage().window().setSize(new Dimension( width: 840,  height: 730));
  driver.findElement(By.id("latID")).click();
  driver.findElement(By.id("latID")).sendKeys( ...charSequences: "40.4");
  driver.findElement(By.id("lonID")).click();
  driver.findElement(By.id("lonID")).sendKeys( ...charSequences: "-8.5");
  driver.findElement(By.cssSelector(".btn")).click();
  driver.findElement(By.cssSelector(".col-12:nth-child(4) > .btn")).click();

  assertThat(driver.findElement(By.id("latID1")).getText(), is( value: ""));
  assertThat(driver.findElement(By.id("lonID1")).getText(), is( value: ""));

  driver.findElement(By.id("latID1")).click();
  driver.findElement(By.id("latID1")).sendKeys( ...charSequences: "37.6");
  driver.findElement(By.id("lonID1")).click();
  driver.findElement(By.id("lonID1")).sendKeys( ...charSequences: "-9.6");
  driver.findElement(By.cssSelector(".col-12:nth-child(3) > .btn")).click();
  driver.findElement(By.cssSelector("span:nth-child(1)")).click();
  driver.findElement(By.cssSelector("span:nth-child(1)")).click();

  {
    WebElement element = driver.findElement(By.cssSelector("span:nth-child(1)"));
    Actions builder = new Actions(driver);
    builder.doubleClick(element).perform();
  }

  assertThat(driver.findElement(By.cssSelector("h3 > b")).getText(), is( value: "City: Sines"));
  assertThat(driver.findElement(By.id("latID1")).getText(), is(not( value: "37.6")));
  assertThat(driver.findElement(By.id("lonID1")).getText(), is(not( value: "-9.6")));
}
```

3.4 Static code analysis

I had the SonarLint plugin on IntelilJ and i fixed several code smells that way.

4 References & resources

Project resources

- The video demo is on GitHub (if having trouble watching use https://www.onlinemp4parser.com/

Reference materials

As mentioned before the API used was from WeatherBit.io.