deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

HW1: MID-TERM ASSIGNMENT REPORT

AirQuality And Weather

Licenciatura em Engenharia Informática
Testes e Qualidade de Software

Mariana Andrade Nº103823

deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

HW1: MID-TERM ASSIGNMENT REPORT

AirQuality And Weather

Licenciatura em Engenharia Informática
Testes e Qualidade de Software
Prof. Ilídio Oliveira

Mariana Andrade Nº103823

Aveiro, 10 de abril de 2023

CONTEUDO

ÍNDICE FIGURAS

Aveiro, 10 de abril de 2023

INTRODUÇÃO

No âmbito da Unidade Curricular de Testes e Qualidade de Software, do 3º ano do curso de Licenciatura de Engenharia Informática, da UA – Universidade de Aveiro –, foi solicitada a elaboração de um trabalho no qual se desenvolva uma multi-layer web application, com recurso à framework *Spring Boot*, com testes automatizados.

A **Air Quality and Weather** foi a aplicação desenvolvida que comunica com APIs externas – *Air Visual, Open Weather Current e Open Weather Geocoding* – que visa fornecer informações precisas e atualizadas sobre a qualidade do ar e a meterologia, dada uma determinada cidade ou região.

Posto isto, os objetivos que se pretendem alcançar com este trabalho são:

1. Aprofundar o conhecimento da ferramenta *Spring Boot;*
2. Detalhar o conhecimento e desenvolvimento de teste automatizados;
3. Aprofundar as Garantias de Qualidade e Design de Software.

O trabalho é constituído por quatro partes: Especificação do Produto; Arquitetura; API e, por último, Garantia de Qualidade.

A metodologia utilizada para realização deste trabalho incidiu-se em pesquisas bibliográficas de artigos científicos, comunicações orais, livros e apontamentos em sala de aula. As referências inerentes às pesquisas realizadas encontram-se normalizadas de acordo com a Norma APA (American Psychological Association) 7th, com as devidas adaptações inerentes à língua portuguesa.

AIR QUALITY AND WEATHER

Air Quality And Weather é projetada a todos aqueles que desejam obter informações sobre a meteorologia e/ou a qualidade do ar para uma dada região. Apenas com a informação da cidade e do país o utilizador a uma destas informações. Posto isto, os principais caso de usos identificado são: informações de Qualidade do ar e informações meteorológicas.

Relativamente às informações de Qualidade do ar, como atleta, quando se faz uma viagem para outra cidade e países, é necessário ter em conta a qualidade do ar, de forma a planear os meus treinos, de acordo a atmosfera em que vou me encontrar. Através da funcionalidade de *Search Air Quality*, o utilizador tem a possibilidade de pode comparar a qualidade do ar, entre a sua localização e a sua cidade destino, de forma a perceber que diferenças pode encontrar a nível da qualidade do ar. Desta forma, o utilizador seleciona o botão de pesquisa, no *card* de Air Quality, onde é redirecionado para um *form*, onde preenche com os dados do: cidade e país. Ao clicar no botão *Search* serão expostos dados como Concentrações de ozono, dióxido de carbono entre outros compostos.

Relativamente às informações meteorológicas, um residente de Aveiro. Aveiro é um distrito muito chuvoso e, por isso, uma pessoa as informações meteorológicas ajuda na prevenção de um guarda-chuva. Neste caso, a aplicação oferece, assim, a informação da meteorologia dado as informações da localização com os parâmetros país, distrito e cidade, de uma dada lista. Selecionados os dados, o utilizador é exibido os dados meteorológicos, como a direção do vento e a temperatura.

Ainda assim, *Air Quality and Weather*, oferece dados de estatísticas com informações sobre certos detalhes da cache do serviço fornecido, tal como de chamadas às APIs externas, através da navegação da *navabar* ou dos botões presentes em *card* na homepage da aplicação.

ARQUITETURA

A arquitetura planeada é feita de maneira modular, permitindo a rápida prototipagem e desenvolvimento para a adição de novas funcionalidades. Constituída por 2 componentes: a componente da interface Web – escrita na framework *Next.js* -, a componente de backend – usando a framework *Spring Boot.*

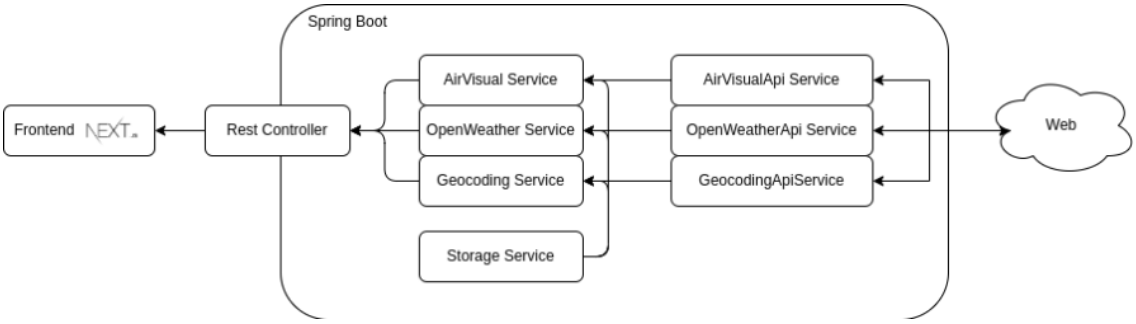

Figura 1- Diagrama da Arquitetura

De acordo com o diagrama, é possível perceber que o serviço foi devido em 4 fases: *Rest Controller, Services, Data Collection e Cache.*

Rest Controller

Esta camada lida com oss pedidos de recursos feitas por um utilizador, o pedido recebido, executa os pedidos internos de modo que a resposta seja deligente e retorna a resposta mais apropriada. Trata-se de uma camada importante, pois fornece de forma padrão e escalável a manipulação dos recursos no serviço.

Spring Boot Services e Caching

Estas duas camadas são onde o principalmente processamento ocorre, desde o gerenciamento da cache ao acesso às APIs externas. Os *services* são componentes que englobam a distribuição do pedido para a API correspondente ou a cache e agregam dados das várias APIs de forma a permitir que qualquer número de APIs consiga compartilhar a mesma função de saída. Enquanto, a cache é um mecanismo de armazenamento temporário que é usado de forma a armazenar dados frequentemente utilizados. Desta maneira, reduzimos o tempo de resposta do aplicativo e evitando a sobrecarga de fontes

de dados, neste caso às APIs externas. Com a combinação destas duas camadas de numa só é possível criar um serviço mais rápido e escalável.

Interação entre modelo de dados

A arquitetura como já referido consiste na interação de vários módulos com um núcleo principal em Spring Boot. Segue-se dois diagramas de interações de dois métodos diferentes entre os diferentes componentes existentes.

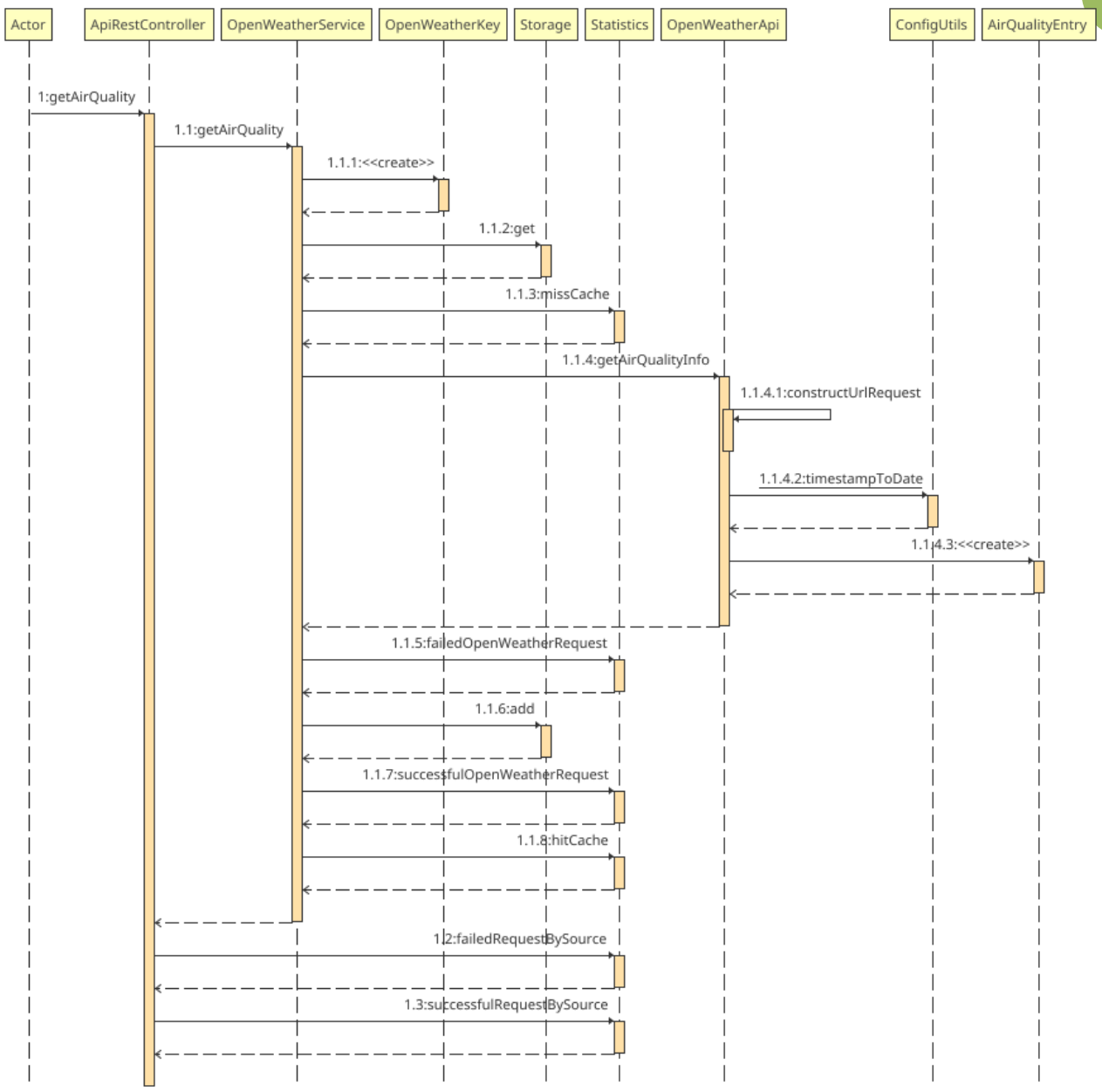

Figura 2- Diagram de Sequência, getAirQuality

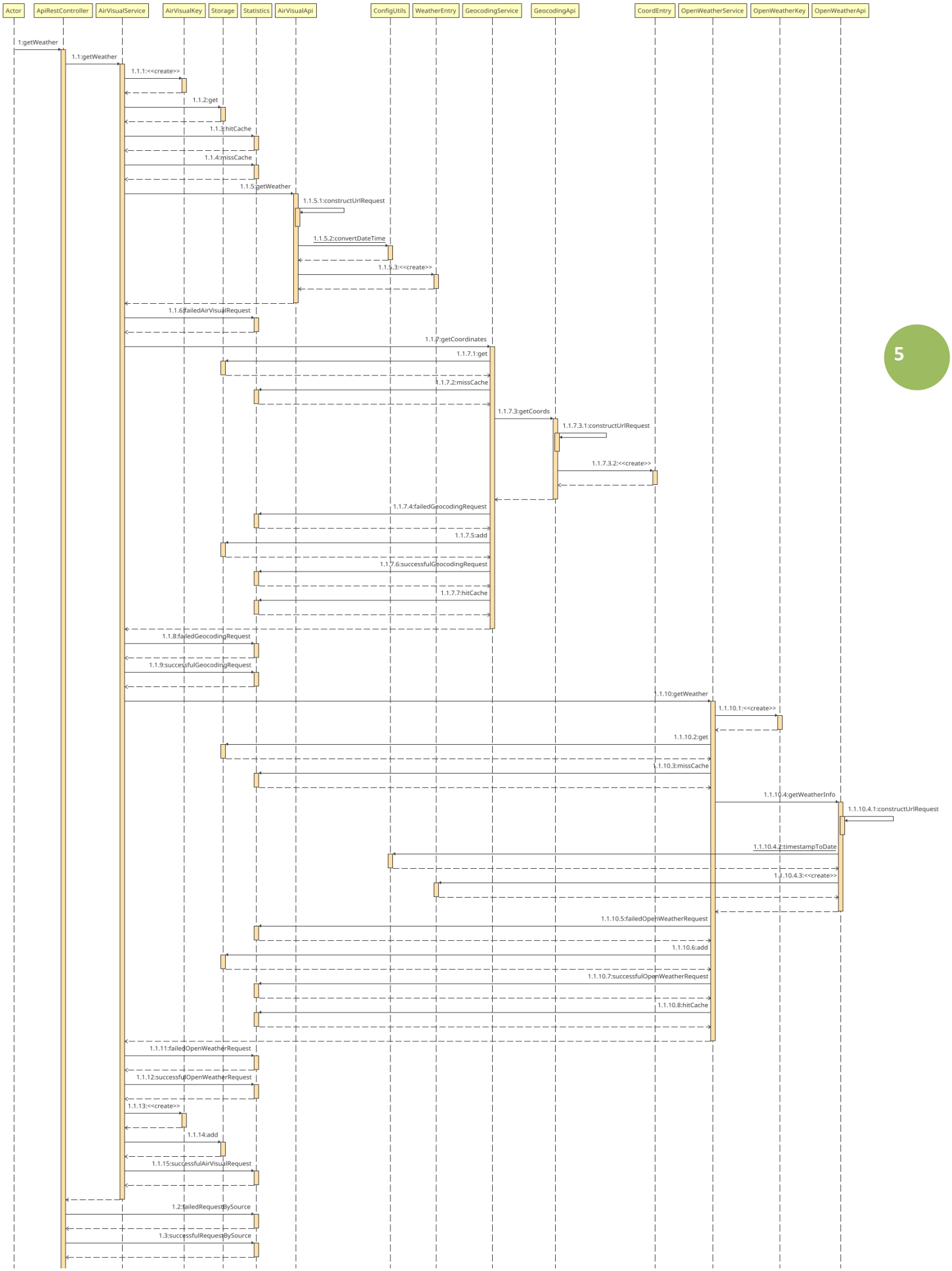

Figura 3- Diagrama de sequência, GetWeather

API para desenvolvedores

As documentações dos endpoints da API estão disponíveis no endpoint **/swagger-ui/index.html,** utilizando a ferramenta *Swagger*.

Figura 4- Endpoint Da API

GARANTIA DA QUALIDADE

O termo qualidade evoluiu ao longo do tempo de maneira a enquadrar-se no mercado. A qualidade é percebida pelas pessoas de forma subjetiva e pessoal, sobressaindo a impossibilidade de ser medida e a dificuldade de se compararem produtos entre si (Neto, P., 2021). A qualidade, segundo a ISO9001 (2015) é o nível de perfeição de um processo, serviço ou produto entregue pela sua empresa

Estratégia utilizada nos testes

Para testagem da implementação e das funcionalidades do produto, inicialmente recorreu-se a estratégia TDD (Test Driven Development). Esta estratégia de desenvolvimento de software envolve a escrita testes automatizados antes da implementação do código em si, permitindo verificar se o código que está a ser implementado atende aos requisitos específicos e garantir que as alterações feitas não afetem as funcionalidades existentes. No entanto, para a clarificação dos testes esperados, optei por criar uma base do serviço antes de implementar os testes. Isto ainda permitiu testar o máximo possível de componentes e seus comportamentos.

Para especificação foram criados teste unitários em *Junit5*, testes de integração e serviços, utilizando as ferramentas *Mockito*, *SpringBoot MockMvc* e *RestAssurence*. E para a testagem do frontend, aplicou-se uma *BEhavior-Driven-Development (BBD)*, recorrendo ao *Selenium WebDriver*.

Testes unitários e de integração

Os testes unitários e de integração são duas abordagens de testes comuns, cada um deles com seus próprios objetivos e momentos de execução diferentes no ciclo de vida do desenvolvimento de software.

Os testes unitários têm como objetivo testar unidades individuais de código, como funções ou métodos, em isolamento do resto do sistema. Normalmente, são executados durante o processo de desenvolvimento para garantir que cada unidade de código esteja funcionando como esperado. Neste caso, foram implementados teste unitários na classe *Storage*, onde são testadas as funcionalidades da cache, adicionar e procurar um elemento, e depois a verificação do funcionamento do TTL (Time To live) de um elemento. E na classe *ConfigUtils*, que contém testes de conversão de *timestamp* and *LocalTime* para o formato *dd-mm-aaaa* da data.


```
class StorageTest {

    5 usages
    private Storage<String, String> storage;

    mariana
    @BeforeEach
    void setup() { storage = new Storage<>( max_time: 4); // 4 ms }

    mariana
    @Test
    void testAddAndGet(){
        storage.add( item: "key", entry: "value");
        assertEquals( expected: "value", storage.get("key"));
    }

    mariana *
    @Test
    void testTTL() throws InterruptedException {
        storage.add( item: "key", entry: "value");
        Thread.sleep( l: 5);
        assertNull(storage.get("key"));
    }
}
```

Figura 5- Testes unitários da class Storage

```
class ConfigUtilsTest {

    mariana
    @Test
    void  TestConvertFromDataTimeToDate() {
        String dateTime = "2020-01-01T00:00:00.000Z";
        String expected = "01-01-2020";
        String result = ConfigUtils.convertDateTime(dateTime);
        assertEquals(expected, result);
    }

    mariana
    @Test
    void TestConvertFromTimestampToDate() {
        long timestamp = 1577836800;
        String expected = "01-01-2020";
        String result = ConfigUtils.timestampToDate(timestamp);
        assertEquals(expected, result);
    }
```

Figura 6- Testes unitários da class ConfigUtils

Ainda, com testes unitários foi testado o comportamento dos *controllers* usando *RestAssurance e Spring Boot MockMvc*, avaliando o seu comportamento fase a resposta a cada pedido, feito em cada endpoint.

```java
@WebMvcTest(ApiRestController.class)
@Disabled // REMOVE Or Comment THIS LINE TO RUN THE TESTS But if you do, the tests will fail, because the number of requests to the external APIs is limited
class ApiRestController_withMockServiceTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private AirVisualService airVisualService;

    @MockBean
    private GeocodingService geocodingService;

    @MockBean
    private OpenWeatherService openWeatherService;

    @BeforeEach
    void setUp() {
    }

    @Test
    @DisplayName("When the service returns a list of countries, then returns 200 and the list")
    void whenGetCountries_thenReturns200AndReturnListCountries() throws Exception {
        List<String> countries = Arrays.asList("Portugal", "Spain");
        when(airVisualService.countries()).thenReturn(countries);

        mockMvc.perform(get("/api/v1/countries")
                        .contentType("application/json"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(2)))
                .andExpect(jsonPath("$[0]", is("Portugal")))
                .andExpect(jsonPath("$[1]", is("Spain")));

        verify(airVisualService, times(1)).countries();
        verifyNoMoreInteractions(openWeatherService);
        verifyNoInteractions(geocodingService);
    }

    @Test
    @DisplayName("When the service fails, then returns 404")
    void whenGetCountries_thenReturns404() throws Exception {
        when(airVisualService.countries()).thenReturn(null);

        mockMvc.perform(get("/api/v1/countries")
                        .contentType("application/json"))
                .andExpect(status().isNotFound());

        verify(airVisualService, times(1)).countries();
        verifyNoMoreInteractions(openWeatherService);
        verifyNoInteractions(geocodingService);
    }
```

Figura 7- Testes ao Controller , endpoint GET countries, com Spring Boot MockMvc

```java
@WebMvcTest(ApiRestController.class)
@ExtendWith(MockitoExtension.class)
class ApiRestController_withRestAssuredTest {
    @Autowired
    private MockMvc mvc;

    @MockBean
    private AirVisualService airVisualService;

    @MockBean
    private GeocodingService geocodingService;

    @MockBean
    private OpenWeatherService openWeatherService;

    @BeforeEach
    public void setUp() {
        RestAssuredMockMvc.mockMvc(mvc);
    }

    @Test
    @DisplayName("When the service returns a list of countries, then returns 200 and the list")
    void whenGetCountries_thenReturns200AndReturnListCountries() throws Exception {
        List<String> countries = List.of("Country1", "Country2");
        when(airVisualService.countries()).thenReturn(countries);

        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/countries")
                .then()
                .statusCode(200)
                .body("size()", is(2))
                .body("get(0)", is("Country1"))
                .body("get(1)", is("Country2"));

        verify(airVisualService, times(1)).countries();
        verifyNoInteractions(geocodingService);
        verifyNoInteractions(openWeatherService);
    }

    @Test
    @DisplayName("When the service fails, then returns 404")
    void whenGetCountriesAndServiceFails_thenReturns404() throws Exception {
        when(airVisualService.countries()).thenReturn(null);

        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/countries")
                .then()
                .statusCode(404);

        verify(airVisualService, times(1)).countries();
        verifyNoInteractions(geocodingService);
        verifyNoInteractions(openWeatherService);
```

Figura 8- Teste ao controller, endpoint GET countries, com RestAssured

Tal como, os *controller*s foi nos demais serviços e classes de suporte, para cada uma das suas funções.


```java
@SpringBootTest
class GeocodingServiceTest {

    @MockBean
    private GeocodingApi api;

    @MockBean
    private Storage<String, CoordEntry> storage;

    private GeocodingService service;

    @BeforeEach
    void setUp() {
        service = new GeocodingService(api, storage);
    }

    @Test
    void testGetCoordWhenInCache() throws IOException, URISyntaxException {
        when(storage.get("Portugal:Ovar")).thenReturn(new CoordEntry(42.85, -8.62));
        CoordEntry entry = service.getCoordinates("Ovar", "Portugal");

        assertThat(entry.getLat(), is(42.85));
        assertThat(entry.getLon(), is(-8.62));

        verify(storage, times(1)).get("Portugal:Ovar");
        verify(api, times(0)).getCoords("Ovar", "Portugal");
    }


    @Test
    void testGetCoordWhenNotInCache() throws IOException, URISyntaxException {
        when(storage.get("Portugal:Ovar")).thenReturn(null);
        when(api.getCoords("Ovar", "Portugal")).thenReturn(new CoordEntry(42.85, -8.62));
        CoordEntry entry = service.getCoordinates("Ovar", "Portugal");
        assertEquals(42.85, entry.getLat());
        assertEquals(-8.62, entry.getLon());

        verify(storage,times(1)).get("Portugal:Ovar");
        verify(api,times(1)).getCoords("Ovar", "Portugal");
    }

    @Test
    void testGetCoordNotInCacheOrApi() throws IOException, URISyntaxException {
        when(storage.get("Portugal:Ovar")).thenReturn(null);
        when(api.getCoords("Ovar", "Portugal")).thenReturn(null);
        CoordEntry entry = service.getCoordinates("Ovar", "Portugal");
        assertNull(entry);

        verify(storage,times(1)).get("Portugal:Ovar");
        verify(api,times(1)).getCoords("Ovar", "Portugal");
    }
}
```

Figura 9- Testes unitários, para o serviço Geocoding

Os testes de integração são usados para testar a integração entre diferentes componentes ou módulos dos sistemas. Este tipo de testes tem como objetivo garantir que todos os componentes do sistema funcionem juntos corretamente e atendam aos requisitos de negócios, normalmente, são menos executados comparativamente com os testes unitários e correm-se no final do ciclo de desenvolvimento. Tal como, anteriormente foi implementado este tipo de testes aos *controllers*, não restringindo o seu comportamento com os outros componentes.


```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, classes = AirQualityApplication.class)
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
@AutoConfigureMockMvc
@Disabled // Remove this line to run the integration tests
class ApiRestControllerIT {
    @Autowired
    private MockMvc mvc;

    @BeforeEach
    void setUp() {
        RestAssuredMockMvc.mockMvc(mvc);
        Statistics.getInstance().reset();
    }

    @Test
    @DisplayName("Test if the API returns 200 when getting the list of countries")
    void whenGetCountries_thenStatus200() {
        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/countries")
                .then()
                .statusCode(200);

        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/statistics")
                .then()
                .statusCode(200)
                .body("cacheMisses", is(1))
                .body("cacheHits", is(0))
                .body("successfulRequests", is(2))
                .body("failedRequests", is(0))
                .body("successfulGeocodingRequests", is(0))
                .body("failedGeocodingRequests", is(0))
                .body("successfulAirVisualRequests", is(1))
                .body("failedAirVisualRequests", is(0))
                .body("successfulOpenWeatherRequests", is(0))
                .body("failedOpenWeatherRequests", is(0));
    }

    @Test
    @DisplayName("Test if the API returns 200 when getting the list of states")
    void whenGetStates_thenStatus200() {
        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/Portugal/states")
                .then()
                .statusCode(200);

        RestAssuredMockMvc.given()
                .when()
                .get("/api/v1/statistics")
                .then()
                .statusCode(200)
                .body("cacheMisses", is(1))
                .body("cacheHits", is(0))
                .body("successfulRequests", is(2))
                .body("failedRequests", is(0))
                .body("successfulGeocodingRequests", is(0))
                .body("failedGeocodingRequests", is(0))
                .body("successfulAirVisualRequests", is(1))
                .body("failedAirVisualRequests", is(0))
                .body("successfulOpenWeatherRequests", is(0))
                .body("failedOpenWeatherRequests", is(0));
    }
```

Figura 10- Teste de integração, endpoint Get countries

Testes funcionais

Os testes funcionais são uma abordagem de teste de software que se concentra em avaliar o sistema como um todo, em relação a requisitos de negócios ou especificações do utilizador. Ao contrário dos testes unitários ou de integração, os testes funcionais são executados do ponto de vista do usuário final e testam a funcionalidade do sistema de uma perspetiva mais ampla.

Neste caso, em específico, foi utilizada a ferramenta *Selenium*, que permite verificar se o software atende aos requisitos de negócios e às expectativas do utilizador, convertendo para *page Object.*

```
@Test
@DisplayName("Test Search Air Quality")
void testSearchAirQuality() {
    driver.get("http://frontend:3000/");
    driver.manage().window();
    {
        List<WebElement> elements = driver.findElements(By.cssSelector(".card:nth-child(1) > .card-body > .btn"));
        assertTrue(elements.size() > 0);
    }
    {
        List<WebElement> elements = driver.findElements(By.linkText("Air Quality"));
        assertTrue(elements.size() > 0);
    }
    driver.findElement(By.cssSelector(".card:nth-child(1) > .card-body > .btn")).click();
    driver.findElement(By.name("city")).click();
    driver.findElement(By.name("city")).clear();
    driver.findElement(By.name("city")).sendKeys("Ovar");
    driver.findElement(By.name("country")).click();
    driver.findElement(By.name("country")).clear();
    driver.findElement(By.name("country")).sendKeys("Portugal");
    driver.findElement(By.cssSelector(".btn-primary")).click();
    driver.findElement(By.cssSelector(".btn-primary")).click();
    {
        List<WebElement> elements = driver.findElements(By.cssSelector(".text-2xl"));
        assertTrue(elements.size() > 0);
    }
    {
        List<WebElement> elements = driver.findElements(By.cssSelector(".stats:nth-child(1) > .stat-figure .swap-on"));
        assertTrue(elements.size() > 0);
    }
}
```

Figura 11- script test junit in Selenium, adaptado a Docker

```
@Test
@DisplayName("Test Search Air Quality Page Object")
void testSearchAirQualityPageObject() {
    driver.get("http://frontend:3000/");
    driver.manage().window();
    HomePage homePage = new HomePage(driver);
    homePage.open();
    assertTrue(homePage.isAirQualityButtonDisplayed());
    assertTrue(homePage.isAirQualityLinkDisplayed());

    AirQualityPage airQualityPage = homePage.clickAirQualityButton();
    airQualityPage.search("Ovar", "Portugal");
    assertTrue(airQualityPage.getAirQualityIndex());
    assertTrue(airQualityPage.getSubTitle());
}
```

Figura 12- Test in Selenium, converte in page Object

Outra abordagem, ao qual se recorreu, foi a Behavior-Driven Development, cujo foco é descrever o comportamento do software em termos de cenários de uso. Os cenários básicos correspondentes que responde ao caso de uso dado, estão especificados num em um ficheiro, *feature*, onde os testes são testados todas as tarefas necessárias para conclusão deste.


```
Feature: Search for Air Quality and Weather statistics
  As a user
  I want to search for air quality and weather statistics
  So that I can view the current conditions

  Background:
    Given I am on the homepage

  Scenario: Search for Air Quality
    When I click on the Search Air Quality button
    And I enter "Ovar" in the city field
    And I enter "Portugal" in the country field
    And I click the search button
    Then I should see "Ovar" and "Portugal" in the air quality data

  Scenario: Search for Weather
    When I click on the search weather button
    And I select "Portugal" in the country field
    And I select "Aveiro" in the state field
    And I select "Agueda" in the city field
    And I search for the weather
    Then I should see "Agueda", "Aveiro" and "Portugal" in the weather data
```

Figura 13- Cenários em BBD

```
public class frontendSteps {
    private static final DockerComposeContainer environment = new DockerComposeContainer(new java.io.File("../../docker-compose.test.yml"))
            .withBuild(true);

    private RemoteWebDriver driver;
    private HomePage homePage;
    private AirQualityPage airQualityPage;
    private WeatherPage weatherPage;

    @Given("I am on the homepage")
    public void iAmOnTheHomepage() throws MalformedURLException {
        environment.stop();
        environment.start();
        URL url = new URL("http://localhost:4444/wd/hub");
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setBrowserName("firefox");
        capabilities.setCapability("enableVNC", true);
        driver = new RemoteWebDriver(url, capabilities);

        driver.get("http://frontend:3000");

        homePage = new HomePage(driver);
        airQualityPage = new AirQualityPage(driver);
        weatherPage = new WeatherPage(driver);
    }

    @When("I click on the Search Air Quality button")
    public void iClickOnTheButton() {
        homePage.clickAirQualityButton();
    }

    @And("I enter {string} in the city field")
    public void iEnterInTheCity(String city) {
        airQualityPage.insertCity(city);
    }

    @And("I enter {string} in the country field")
    public void iEnterInTheCountry(String country) {
        airQualityPage.insertCountry(country);
    }

    @And("I click the search button")
    public void iTwoClickTheButton() {
        airQualityPage.search();
    }

    @Then("I should see {string} and {string} in the air quality data")
    public void iShouldSeeTheAirQualityData(String city, String country) {
        airQualityPage.assertData(city, country);
    }
```

Figura 14- Teste cucumber implementado para o cenário de search Air Quality

Análise da qualidade do código

Para uma análise de código, foi utilizada a ferramenta *JaCoCo*, para uma análise de cobertura de código. Desta forma foi possível identificar quais as partes de código que estavam e não estavam a ser testadas, ajudando a melhorar a qualidade do código e identificar possíveis pontos fracos.

airQuality

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
pt.ua.deti.tqs.airquality.extapi		3%		0%	25	31	191	203	10	16	0	3
pt.ua.deti.tqs.airquality.pageobject		0%		0%	58	58	117	117	55	55	6	6
pt.ua.deti.tqs.airquality.model.openweather		21%		5%	72	102	4	22	10	40	0	2
pt.ua.deti.tqs.airquality.model.airvisual		50%		27%	74	108	2	23	8	42	0	2
pt.ua.deti.tqs.airquality.model.geocoding		22%		0%	24	36	2	12	8	20	0	2
pt.ua.deti.tqs.airquality.model		93%		70%	3	24	2	84	0	19	0	2
pt.ua.deti.tqs.airquality		85%		n/a	1	4	2	9	1	4	0	2
pt.ua.deti.tqs.airquality.services		100%		100%	0	29	0	169	0	13	0	3
pt.ua.deti.tqs.airquality.boundary		100%		100%	0	16	0	73	0	9	0	1
Total	2 590 of 4 499	42%	284 of 380	25%	257	408	320	712	92	218	6	23

Figura 16 - Resultados do JaCoCo para teste unitários

airQuality

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
pt.ua.deti.tqs.airquality.model.openweather		23%		1%	72	102	3	22	10	40	0	2
pt.ua.deti.tqs.airquality.model.airvisual		29%		2%	77	108	2	23	11	42	0	2
pt.ua.deti.tqs.airquality.pageobject		56%		66%	32	58	57	117	30	55	3	6
pt.ua.deti.tqs.airquality.services		63%		43%	19	29	71	169	3	13	0	3
pt.ua.deti.tqs.airquality.model.geocoding		9%		0%	30	36	7	12	14	20	1	2
pt.ua.deti.tqs.airquality.model		69%		40%	10	24	22	84	6	19	0	2
pt.ua.deti.tqs.airquality.extapi		90%		63%	11	31	24	203	0	16	0	3
pt.ua.deti.tqs.airquality.boundary		71%		50%	7	16	21	73	0	9	0	1
pt.ua.deti.tqs.airquality		85%		n/a	1	4	2	9	1	4	0	2
Total	2,010 of 4,499	55%	327 of 380	13%	259	408	209	712	75	218	4	23

Figura 15 - Resultados do JaCoCo para teste de integração

Pela análise destes resultados entende-se que os testes unitários têm uma coverage de 44% do código, enquanto o teste de integração tem 55% de coverage.

Ainda, com o recurso a ferramenta *SonarCloud*, é possível identificar problemas de código em projetos de software, como segurança, conformidade com padrões de codificação, desempenho e manutenibilidade. Com ajuda das suas métricas e indicadores ajudaram a identificar e entender melhorias necessárias a nível da qualidade de código.

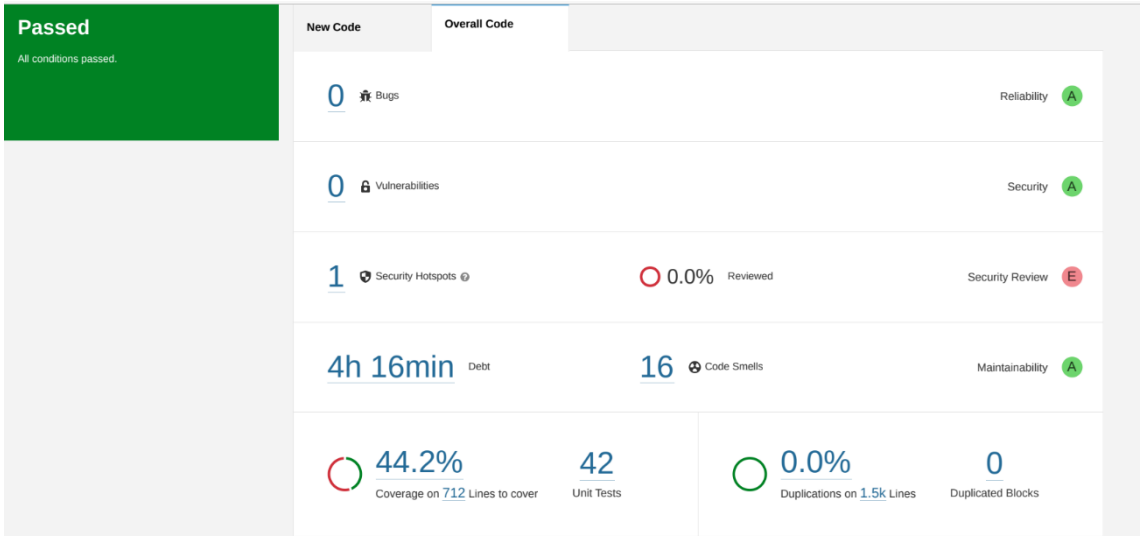

Figura 17- Resultados do SonarCloud

Pela análise destes resultados, verifica-se que há 0 bugs, 0 vulnerabilidades, 1 *Secutiry Hostsposts*, devido à anotação @CrossOrigin, e inicialmente 69 *code smells*, mais tarde corrigidos e ficaram apenas 16 *code smells*. No total, foram realizados 57 testes, 42 teste unitários e os restantes são testes de integração, porém e notório que coverage não seja total, mas semelhantes à do *Jacoco* para testes unitários, pois o *SonarClouds* apena avalia os testes Unitários.

Pipeline de integração contínua

No contexto de testes e qualidade de software, o *GitHub Actions* é usado para criar pipeline de integração contínua (CI) que automatiza a execução de testes em um repositório de software, com o objetivo de garantir que o código esteja a funcionar corretamente e de acordo com as especificações e expetativas iniciais.

O seu funcionamento baseia-se em um fluxo de trabalho que permite a execução de tarefas personalizadas em resposta de eventos específicos, como *push* no repositório. Esta pipeline de CI, usando o *GitHub Actions*, para testes pode ser dividida em várias etapas. Primeiro, o código-fonte é clonado do repositório para uma máquina virtual em que as tarefas de teste serão executadas. Em seguida, as dependências são instaladas e os testes são executados em um ambiente controlado e isolado. Os resultados dos testes são então coletados e relatados para os desenvolvedores analisarem.

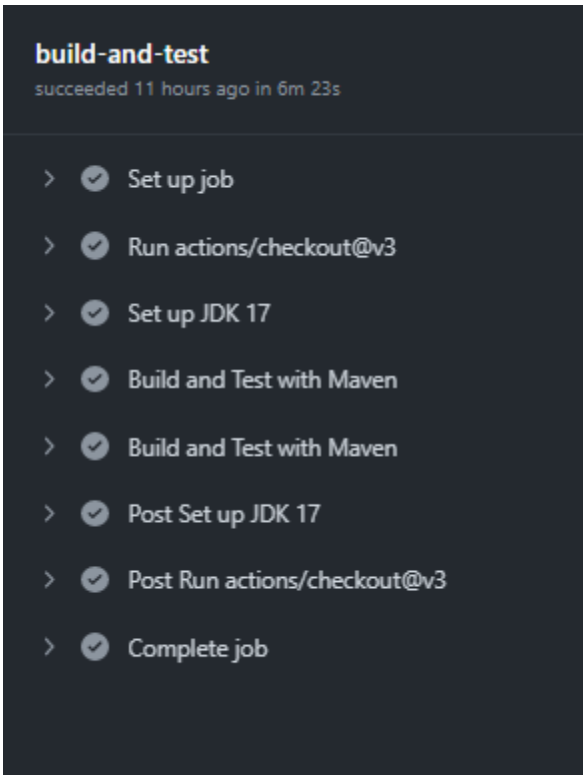

Figura 18 - steps the github actions

CONCLUSÃO

Air Quality And Weather, surgiu com resposta a uma multi-layer web application, com informações de meteorológicas e de qualidade de ar. A sua implementação comunica com diferentes APIs externas, com algumas limitações que levaram a decisões importantes no projeto e levantaram a aplicação de novos requisitos. Por exemplo, o número limitado de request à API, num determinado período, levou a implementa de API secundário no processo de forma a conseguir obter dados, sem ter de afetar o utilizador, caso não seja possível obter dados da API principal.

De uma forma geral considera-se que a construção desta web application revelou-se construtiva para a aprendizagem da qualidade e garantias de um projeto de software. Desde a experiência de framework *Spring Boot,* que se revelou essencial para a agilidade e simplicidade na escrita de código, com o recurso a dependências. A aprendizagem da utilização de ferramentas como *SonarCloud, Selenium e Jacoco,* levou a aprofundamento da qualidade de código e de garantias deste. A necessidade de teste notou-se ser importante, permitindo entender a sua necessidade para projetos futuros.

Relativamente aos resultados, considera-se que existe ainda medidas e melhoramentos necessários. Contudo, perspetivando os objetivos inicialmente pretendidos, entende-se que estes foram alcançados uma vez que foi possível:

- Aprofundar o conhecimento já adquirido da ferramenta *Spring Boot;*
- Aplicar e melhorar o conhecimento e desenvolvimento de teste automatizados;
- Compreender e aprofundar as Garantias de Qualidade e Design de Software.

Em suma, apesar de pequenas limitações mencionadas, este trabalho revelou-se um estímulo positivo, a aquisição e aprendizagem de conhecimento na sua realização foi notória, contribuído para futuros trabalhos e projetos.

REFERÊNCIAS E RECURSOS

APIs Externas

[1] OpenWeatherMap, "Geocoding API," OpenWeatherMap, 2021. [Online]. Available: https://openweathermap.org/api/geocoding-api. [Acedida: 10-04 -2023].

[2] OpenWeatherMap, "Current weather data," OpenWeatherMap, 2021. [Online]. Available: https://openweathermap.org/current. [Acedida: 10-04 -2023].

[3] IQAir, "Air Quality API documentation," IQAir, 2021. [Online]. Available: https://api-docs.iqair.com/?version=latest. [Acedida: 10-04 -2023].

Repositório

tqs_103823/HW1 at main · MarianaAndrad/tqs_103823 (github.com)