Teste e Qualidade de Software

Professor:
Ilídio Oliveira

HW1: Qualidade do Ar

Hugo Paiva de Almeida, 93195

DETI
Universidade de Aveiro
14-05-2021

Índice

1 Introdução

1.1 Visão Geral do Trabalho

Este trabalho consistiu no desenvolvimento de uma aplicação em *Spring Boot* de modo a fornecer o serviço e respetiva *interface Web*, em *Angular*, com informações atuais sobre a qualidade do ar de uma determinada localização bem como estatísticas da *cache* implementada. O maior foco do trabalho traduziu-se nos testes, com cerca de 70 desenvolvidos, bem como a implementação em *pipelines* de integração contínua de modo a assegurar a execução de todos os testes automaticamente.

Para se obter as informações da qualidade do ar, que incluem valores de diversos poluentes do ar, bem como a qualidade geral do ar, foram utilizadas duas *APIs*: a disponibilizada pela empresa *OpenWeather Ltd* e a disponibilizada pelo projeto *The World Air Quality*. Ao realizar um pedido de informações através de coordenadas, o serviço irá primeiramente verificar se existe alguma entrada na *cache* com essas coordenadas e, em caso negativo, irá aceder à última *API*, uma vez que geralmente possui mais dados e, em caso de erro, tentará aceder à primeira. Já no caso do pedido de informações através de localização, o serviço primeiramente acede à primeira *API* através do serviço de resolução de localização disponibilizado e, após receber as coordenadas do local, realiza o pedido normalmente, como descrito anteriormente.

Em relação à *cache*, esta foi implementada com base nas coordenadas, servindo estas como chave de acesso, ficando na base de dados de memória, por defeito, durante 120 segundos. Como forma de auxílio na visualização dos dados da *cache*, para além das métricas pedidas, foi também implementado um histórico de pedidos à *API*, cada um com a respetiva referência de acesso ou não à *cache*.

1.2 Limitações

Uma vez que a *cache* foi preparada para guardar informações relativas a pedidos de informações sobre a qualidade do ar com coordenadas, sempre que um utilizador realiza um pedido para obter informações sobre uma dada localização, é sempre necessário o acesso à *API* externa de resolução para coordenadas, antes do acesso à *cache*.

Apesar de existir vontade para tal, visto que ao implementar todos os testes em *pipelines* de Integração Contínua com configuração de *Selenium* em *Docker* e execução destes testes automaticamente se perdeu bastante tempo, a aplicação cingiu-se à funcionalidade pedida de informações atuais da qualidade do ar, acabando por se explorar pouco as *APIs* em troca de maior foco nos testes.

2 Especificação do Produto

2.1 Espetro funcional e Interações Suportadas

Como utilizador da *interface Web*, é possível aceder às informações atuais da qualidade do ar através de pesquisa por coordenadas e localização. Para além disso, todo o histórico de pedidos e estatísticas referentes à *cache* estão disponíveis, permitindo um maior controlo sobre o funcionamento da mesma.

Como utilizador da *REST API*, é possível obter as mesmas informações que um utilizador da *interface Web* mas, de forma mais crua, em *JSON*, permitindo a integração destes dados em outras plataformas.

2.2 Arquitetura do Sistema

Como mencionado anteriomente, a arquitetura baseia-se em três componentes chave:

- Aplicação Web em *Angular*
- Serviço em *Spring Boot*
- *APIs* Externas

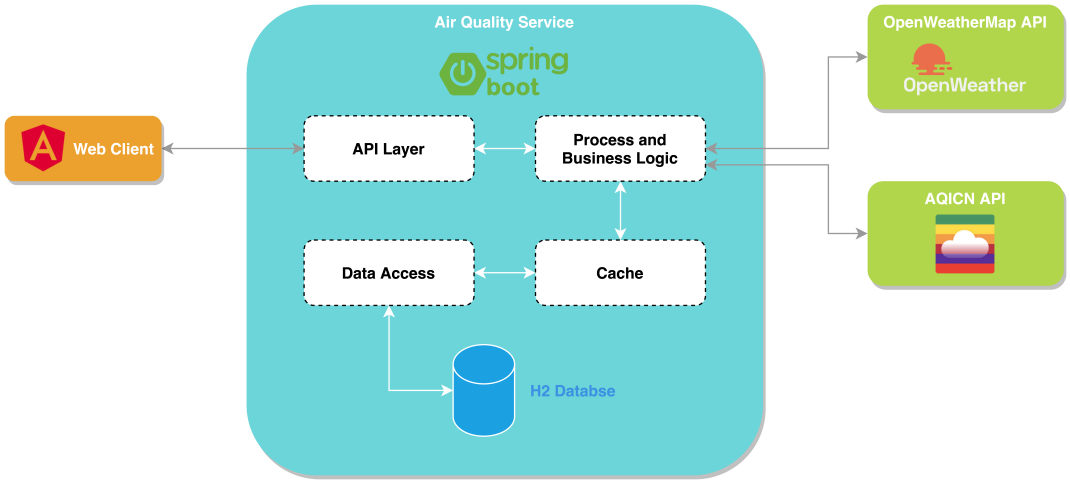

Figure 1: Arquitetura do Sistema

O fluxo deste sistema centra-se no serviço, onde são recebidos os pedidos do cliente, guardados os dados de *cache* e feitos os pedidos às *APIs* externas. Caso o utilizador queira obter informações relativas à qualidade atual do ar, fará um pedido à *REST API* do serviço que, caso necessite de resolver a localização, irá aceder à *API* externa *OpenWeatherMap*, de modo a obter as coordenadas. Tendo as coordenadas, quer seja através da resolução da localização ou através de envio direto do cliente, o serviço verifica se existem informações referentes às mesmas em *cache*, através de uma base de dados em memória *H2*. Caso isto se verifique, estas são retornadas, em caso contrário, são feitos pedidos às *APIs* externas, primeiramente à *AQICN* e, em caso de erro, à *OpenWeatherMap*.

Em relação às informações de estatísticas da *cache*, sempre que o serviço recebe um pedido, irá verificar à base de dados as informações guardadas e retornará a lista de pedidos anteriores ou o número de pedidos, acessos à *cache* e falhas de acessos à *cache*.

Para visualizar melhor o serviço, é possível observar o seu diagrama *UML* aqui.

2.3 *API* de Documentação

Um desenvolvedor pode obter informações relativas à qualidade do ar através de coordenadas ou localização, bem como aceder a informações referentes ao funcionamento da *cache*.

De modo a documentar os *endpoints* da *API*, recorreu-se à ferramenta *Swagger*, ficando esta disponível no *endpoint /api/swagger-ui/index.html*:

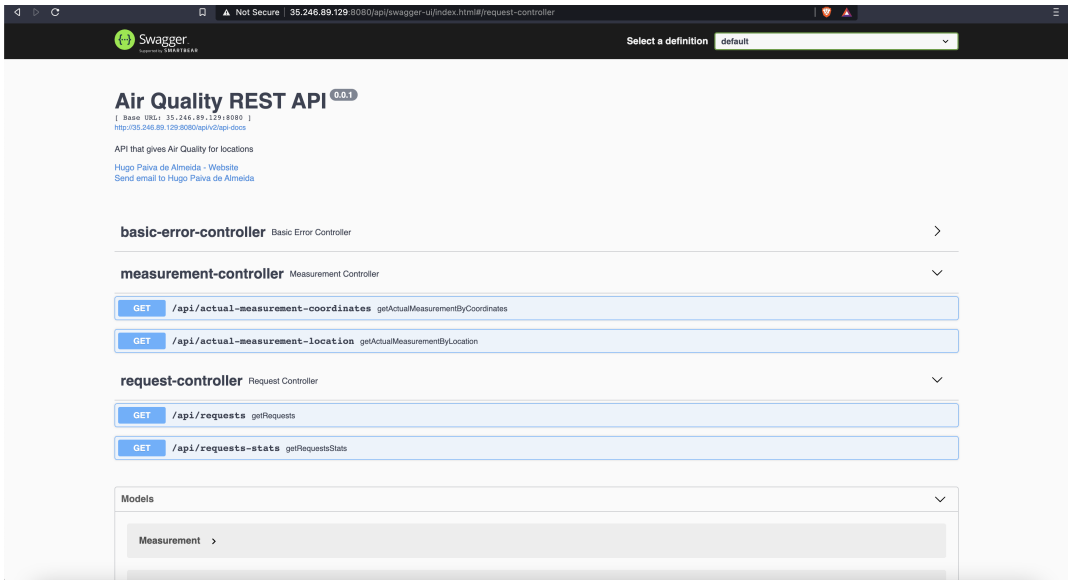

Figure 2: Documentação da *REST API* através do *Swagger*

3 Garantia de Qualidade

3.1 Estratégia de testes

Tentou-se seguir uma aproximação *Test-Driven Development (TDD)* sempre que possível, no entanto nem sempre se conseguiu fazer os testes antes de uma versão funcional da componente em que se estava a trabalhar. Apesar disto, reconhece-se que esta abordagem é excelente para se perceber logo desde o início todos os cenários que uma determinada componente irá ter de enfrentar.

3.2 Testes Unitários e de Integração

Foram feitos testes unitário para todas as classes referentes à funcionalidade do serviço, com objetos *Mock*, sempre que possível. Para definir os testes, pensou-se em todos os cenários possíveis de falha do funcionamento da classe, assegurando que existiria o levantamento de exceções nestes casos, e nos casos normais de uso. Tentou-se sempre descrever o objetivo dos testes através do título da função de teste.

Um bom exemplo destes testes são os testes da *Cache* e dos *Services*, onde em ambos se utilizam vários objetos *Mock* que são injetados na instância da classe que se vai testar:

```java
@ExtendWith(MockitoExtension.class)
class MeasurementServiceTest {
    private Double latitude;
    private Double longitude;
    private Long id;
    private Date date;
    private Integer airQualityIndex;
    private Double pm25;
    private Double pm10;
    private Double wind;
    private Measurement measurement;

    @Mock
    private Cache cache;

    @Mock
    private AQICNMeasurementResolver aqicnMeasurementResolver;

    @Mock
    private OpenWeatherMeasurementResolver openWeatherMeasurementResolver;

    @InjectMocks
    private MeasurementService measurementService;

    @BeforeEach
    void setUp() {
        Random rand = new Random();
        Measurement measurement = new Measurement();
        this.measurement = measurement;
        this.latitude = rand.nextDouble() * 90;
        measurement.setLatitude(this.latitude);
        this.longitude = rand.nextDouble() * 180;
        measurement.setLongitude(this.longitude);
        this.id = rand.nextLong();
        measurement.setId(this.id);
        this.date = new Date(System.currentTimeMillis());
        measurement.setDate(this.date);
        this.airQualityIndex = rand.nextInt( bound: 501);
        measurement.setAirQualityIndex(this.airQualityIndex);
```

Figure 3: Classe de testes referente à classe *MeasurementServices* com a exitência de vários *Mocks*

Em relação aos testes de integração, foram feitos testes por cada componente *Controller*, de modo a testar:

- Comportamento do *Controller* sem a utilização de uma *HTTP-REST framework* com os testes *MeasurementControllerMockMvcMockServiceTest* e *RequestControllerMockMvcMockServiceTest*, utilizando como objeto *Mock* a classe dos *Services* de cada um
- Comportamento do *Controller* com a *REST API* a funcionar do lado do servidor e todos componentes a participar com os testes *MeasurementControllerMockMvcIT* e *RequestControllerMockMvcIT*
- Comportamento do *Controller* com a *REST API* a funcionar do lado do servidor, todos os componentes a participar e um cliente *HTTP* envolvido com os testes *MeasurementControllerTemplateIT* e *RequestControllerTemplateIT*

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT, classes = AirQualityServiceApplication.class)
@AutoConfigureMockMvc
@AutoConfigureTestDatabase
class MeasurementControllerMockMvcIT {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private MeasurementRepository measurementRepository;

    @AfterEach
    public void resetDb() { measurementRepository.deleteAll(); }

    @Test
    public void testCoordinatesWhenInvalidLatMin_thenBadRequest() throws Exception {
        mvc.perform(get( urlTemplate: "/actual-measurement-coordinates")
                .param( name: "lat", String.valueOf(-182.903213))
                .param( name: "lon", String.valueOf(90.213212))
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isBadRequest());
    }
```

Figure 4: Exemplo de teste do *MeasurementController* com a *REST API* a funcionar do lado do servidor e um cliente *HTTP* envolvido

3.3 Testes Funcionais

Uma vez que se queria envolver os testes funcionais no processo automático de testes com Integração Contínua, perdeu-se bastante tempo de modo a configurar o *Selenium* para utilizar um *DockerBrowser*. Nestes testes, além de se testar casos normais de uso que um cliente teria, também se assegurou que a aplicação *Web* responderia com mensagens de erro indicativas do tipo de problema encontrado.

O endereço utilizado para fazer os pedidos é o *172.17.0.1* uma vez que, estando o *browser* a correr num *container Docker*, para este aceder à *API* que está a ser executada explicitamente para estes testes de acordo com a anotação *@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)*, o mesmo tem de aceder ao *localhost* da máquina que lançou o *container*, o que apenas é possível com aquele endereço em sistemas *Linux*. Também foi necessária uma configuração especial a nível de endereço para onde vão os pedidos à *API* no *Angular*, de modo a que estes testes corram sem problemas.

```
// SpringBootTest to run the REST API
@ExtendWith(SeleniumJupiter.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class ActualMeasurementTest {

    private String webApplicationBaseUrl = "172.17.0.1";

    @Test
    void testGetActualMeasurementCoordinates(@DockerBrowser(type = FIREFOX, version = "84") RemoteWebDriver driver) {
        driver.get("http://" + webApplicationBaseUrl + ":4200/");
        driver.manage().window().setSize(new Dimension( width: 1792,  height: 1025));
        {
            WebDriverWait wait = new WebDriverWait(driver,  timeOutInSeconds: 30);
            wait.until(ExpectedConditions.presenceOfElementLocated(By.cssSelector("h3")));
        }
        assertEquals( expected: "Air Quality", driver.getTitle());
        assertEquals( expected: "Search by Coordinates", driver.findElement(By.cssSelector("h3")).getText());
        driver.findElement(By.cssSelector(".form-group:nth-child(1) > .ng-pristine")).click();
        driver.findElement(By.cssSelector(".form-group:nth-child(1) > .ng-pristine")).sendKeys( ...charSequences: "42");
        driver.findElement(By.cssSelector(".form-group:nth-child(2) > .form-control")).click();
        driver.findElement(By.cssSelector(".form-group:nth-child(2) > .form-control")).sendKeys( ...charSequences: "56");
        driver.findElement(By.cssSelector(".fa-search")).click();
        {
            WebDriverWait wait = new WebDriverWait(driver,  timeOutInSeconds: 30);
            wait.until(ExpectedConditions.presenceOfElementLocated(By.cssSelector(".card:nth-child(1) > .ng-star-inserted")));
        }
        assertThat(driver.findElement(By.cssSelector(".card:nth-child(1) > .ng-star-inserted")).getText(), is( value: "42, 56°"));
        assertThat(Integer.parseInt(driver.findElement(By.cssSelector(".main-content > div:nth-child(1) > div:nth-child(1) > div:nth-child(2) > div:nth-child(2) > h5:nth-child(2)"))
                .getText()), is(greaterThanOrEqualTo( value: 0)));
    }
```

Figure 5: Exemplo de testes do *Selenium* com *DockerBrowser*

3.4 Análise de Código Estático

Para a análise de código estático, foi criado um servidor próprio do *SonarQube* na *Google Cloud*. Este servidor encontra-se no endereço http://34.89.73.181:9000 com as seguintes credenciais:

- **Login:** admin
- **Password:** tqs-p1

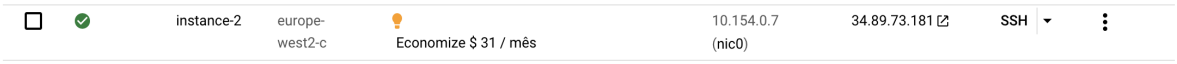

Figure 6: *VM* com uma instância do *SonarQube*

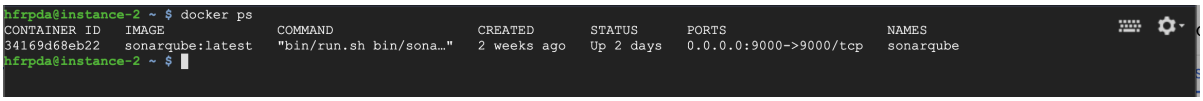

Figure 7: *SonarQube* em um *container Docker*

Através desta ferramenta foi possível ir verificando a qualidade do código bem como, por meio da Integração Contínua, se o novo código adicionado seguia os requisitos mínimos de qualidade de código, sendo que foram utilizados os que existem por defeito:

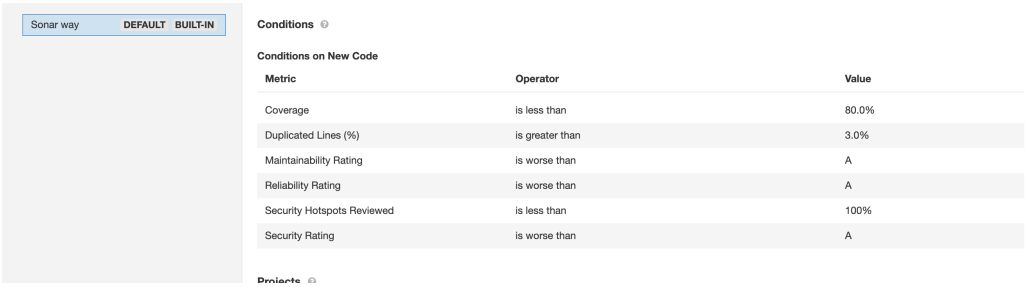

Figure 8: Requisitos mínimos de qualidade de código definidos no *SonarQube*

Ao longo do desenvolvimento foram detetados alguns *Code Smells* mais graves que rapidamente foram resolvidos, comprovando a utilidade desta ferramenta. O resultado final, após a execução de todos os testes e resolução dos *Code Smells* mais graves é o seguinte:

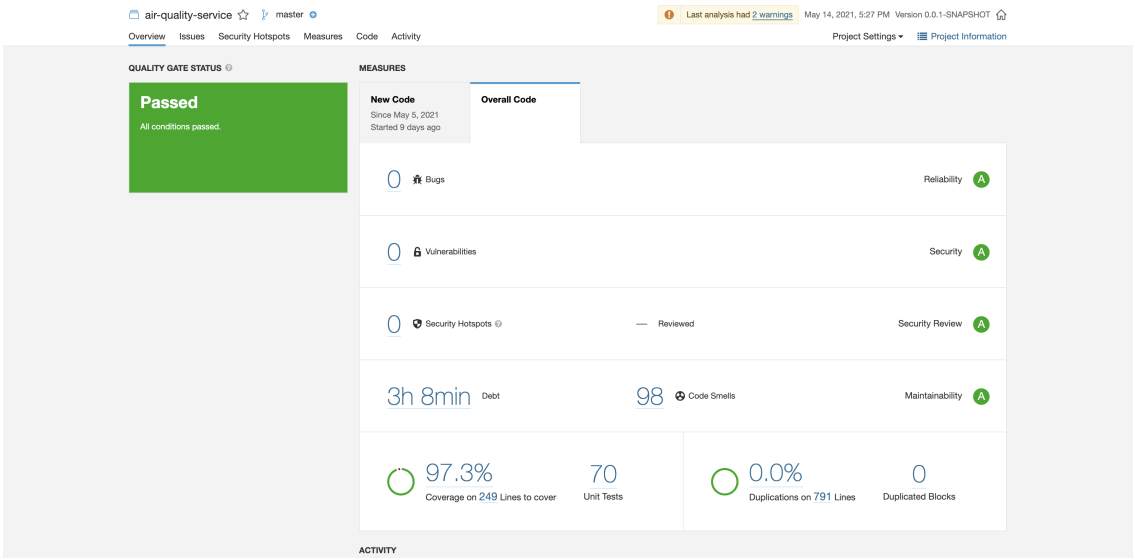

Figure 9: *SonarQube Overview*

Em suma, excedeu-se por muito os requisitos mínimos, tendo um *Code Coverage* de praticamente de 100%, o que revela a eficácia dos testes produzidos.

3.5 *Pipeline* de Integração Contínua

Em relação à *Pipeline* de Integração Contínua foram definidas duas *Pipelines*, uma para testar a *build* do *Angular* e outra para testar o código do *Spring Boot*, enviar o código para análise no *SonarQube* e aprovação segundo os requisitos mínimos. No fundo, sempre que se fazia *push* para a *branch main* ou se criava um *pull request* com destino nessa branch, estes testes eram executados, sendo que a *pull request* só era aceite quando no último *commit* os testes fossem executados com sucesso.

```yaml
name: Build Angular

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [14.x]

    steps:
      - uses: actions/checkout@v1

      - name: Cache node modules
        uses: actions/cache@v1
        with:
          path: ~/.npm
          key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
          restore-keys: |
            ${{ runner.os }}-node-
      - name: Node ${{ matrix.node-version }}
        uses: actions/setup-node@v1
        with:
          node-version: ${{ matrix.node-version }}
      - name: npm install and npm run build
        run: |
          npm i
          npm run build
        working-directory: ./web-application
```

Figure 10: *Pipeline Build Angular*

```yaml
name: Spring Boot CI Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  verify:
    runs-on: ubuntu-latest
    name: Running Verify Lifecycle
    steps:
    - uses: actions/checkout@v2
    - name: Set up Java 11
      uses: actions/setup-java@v2
      with:
        distribution: 'adopt'
        java-version: '11'
    - name: Build Docker-Compose
      run: docker-compose build angular
    - name: Start Angular on Docker
      run: docker-compose run -p 4200:80 -e API_BASE_URL=http://172.17.0.1:8080/api/ -d angular
    - name: Prepare Docker for Selenium
      run: docker pull aerokube/selenoid:1.10.1 && docker pull selenoid/vnc:firefox_84.0
    - name: Verify code executing each default lifecycle phase and IT tests
      run: mvn -B clean verify
      working-directory: ./air-quality-service
  sonarQube:
    runs-on: ubuntu-latest
    name: SonarQube Code Analysis
    steps:
    - uses: actions/checkout@v2
    - name: Set up Java 11
      uses: actions/setup-java@v2
      with:
        distribution: 'adopt'
        java-version: '11'
    - name: Build Docker-Compose
      run: docker-compose build angular
    - name: Start Angular on Docker
      run: docker-compose run -p 4200:80 -e API_BASE_URL=http://172.17.0.1:8080/api/ -d angular
    - name: Prepare Docker for Selenium
      run: docker pull aerokube/selenoid:1.10.1 && docker pull selenoid/vnc:firefox_84.0
    - name: Send code to SonarQube server and wait for quality gate
      run: mvn -B clean verify sonar:sonar -Dsonar.host.url=${{ secrets.SONARQUBE_HOST }} -Dsonar.login=${{ secrets.SONARQUBE_TOKEN }} -Dsonar.qualitygate.wait=true
      working-directory: ./air-quality-service
```

Figure 11: *Pipeline* testes *Spring Boot*

De notar que, na reta final do trabalho, para permitir a execução dos testes funcionais, a *build* e execução da aplicação *web* em *Angular* em *container Docker* também teve de ocorrer na *pipeline* dos testes do *Spring Boot*.

4 Conclusão

Para terminar, pensa-se que, de acordo com as metas estabelecidas pelo docente, o trabalho foi bem sucedido. Foi possível desenvolver uma aplicação *Web* e testá-la intensivamente quer manualmente, quer automaticamente através de uma *Pipeline* de Integração Contínua. A eficácia dos testes construídos foi demonstrada, com o auxílio do *SonarQube*, que também alertou para situações relevantes através dos *Code Smells*.

Considera-se que os conhecimentos para a realização de testes, bem como de outras ferramentas como *Docker*, *Spring Boot* e *Angular*, foram aprofundados para além da percepção da importância dos testes uma vez que em alguns casos na implementação faltava código para lidar com situações. Segui-se a metáfora do teste de pirâmide, começando pelo desenvolvimento dos testes unitários, seguindo-se os de integração e por fim os funcionais.

Este trabalho focou-se principalmente na componente da realização de testes.

5 Resources

Demo: https://www.youtube.com/watch?v=4_sAMCed5b4

Aplicação Web: http://35.246.89.129/

REST API: http://35.246.89.129:8080/

Documentação REST API: http://35.246.89.129:8080/api/swagger-ui/index.html

Repositório: https://github.com/hugofpaiva/tqs-p1

SonarQube: http://34.89.73.181:9000 com as seguintes credenciais:

- **Login:** admin
- **Password:** tqs-p1

6 Referências

[1] Material de apoio do Docente

[2] T. W. A. Q. I. project, "API - Air Quality Programmatic APIs" aqicn.org. https://aqicn.org/api/

[3] "Air Pollution - OpenWeatherMap," openweathermap.org. https://openweathermap.org/api/air-pollution

[4] "Automate Your Development Workflow With Github Actions - DZone DevOps," dzone.com. https://dzone.com/articles/automate-your-development-workflow-with-github-act-1

[5] "Dockerize a Spring Boot app in 3 minutes," Docker cloud hosting. Cheap, simple and fast to deploy and manage. https://dockerize.io/guides/docker-spring-boot-guide

[6] "Try Out SonarQube | SonarQube Docs," docs.sonarqube.org. https://docs.sonarqube.org/latest/setup/get-started-2-minutes/