

HW1: Relatório do trabalho intermediário

Mariana Rosa [98390], v2022-04-22

Introdução	1
Visão geral do trabalho	1
Limitações	2
Especificação do produto	2
Functional scope e interações possíveis	2
Arquitetura do sistema	2
API para o desenvolvimento	3
Quality assurance	3
Estratégia para a testagem	3
Testes de integração e unitários	4
Testes funcionais	7
Análise da qualidade do código	9
Referências & Recursos	12

1 Introdução

1.1 Visão geral do trabalho

Este relatório tem como objetivo descrever o trabalho individual da disciplina de Testes e Qualidade de Serviço, cobrindo as *features* do *software* do produto e assegurando uma garantia de qualidade.

O projeto denomina-se **Covid Tracking** e permite ao utilizador ver as estatísticas sobre os dados relativos à COVID-19 em todo o mundo.

1.2 Limitações

Quando comecei o projeto, esperava conseguir ter no site mais estatísticas em relação a cada país, nomeadamente ao longo dos seis meses e comparações entre países. Porém, tive receio de não conseguir implementar os testes pedidos, dando assim máxima prioridade à testagem do projeto. A Cache também não ficou implementada como gostaria, sendo que os números de *hits* e *misses* não estão a contabilizar na totalidade, porém individualmente funcionam.

2 Especificação do produto

2.1 *Functional scope* e interações possíveis

Esta aplicação, como referido anteriormente, irá permitir que todos os utilizadores interessados em saber como está a situação atual da doença de coronavírus no mundo possam o fazer.

Podem pesquisar por cada país (de todo o mundo) ou por estatísticas globais.

Alguns possíveis cenários são:

Scenario: Pesquisar por estatísticas globais sobre o COVID

Dado o url "<http://localhost:8080/>"

Quando o cliente quer ver as estatísticas do mundo

E ele clica no botão "All Countries"

Então ele vê a mensagem "Today World's Data" com as devidas informações

Scenario: Ver as estatísticas da Croácia

Dado o url "<http://localhost:8080/countrydata>"

Quando o cliente quer ver as estatísticas da Croácia

E ele clica no botão "Specific Country"

E pesquisa "Croatia"

Então ele vê uma tabela com as devidas informações

2.2 Arquitetura do sistema

Para este projeto utilizei a tecnologia Spring Boot, uma ferramenta bastante poderosa que permite criar aplicações Spring diretamente embutidas num servidor *web* (TomCat). Para a visualização do trabalho feito no backend, utilizei um [template](#) gratuito em HTML/JavaScript.

Assim a sua arquitetura consiste:

- O cliente tem contacto com a camada *web* construída em HTML/Javascript;
- Esta camada irá fazer os pedidos à API (GET) que está documentada através do Postman (irei falar em mais detalhe posteriormente);
- Os pedidos passam pelos Controllers (*PlaceController.java* e *StatisticsController.java*) para mapear os pedidos e chamam os serviços associados (*StatisticsService.java* e *PlaceService.java*);
- Estes valores vão ser transmitidos pela API [Vaccovid](#);

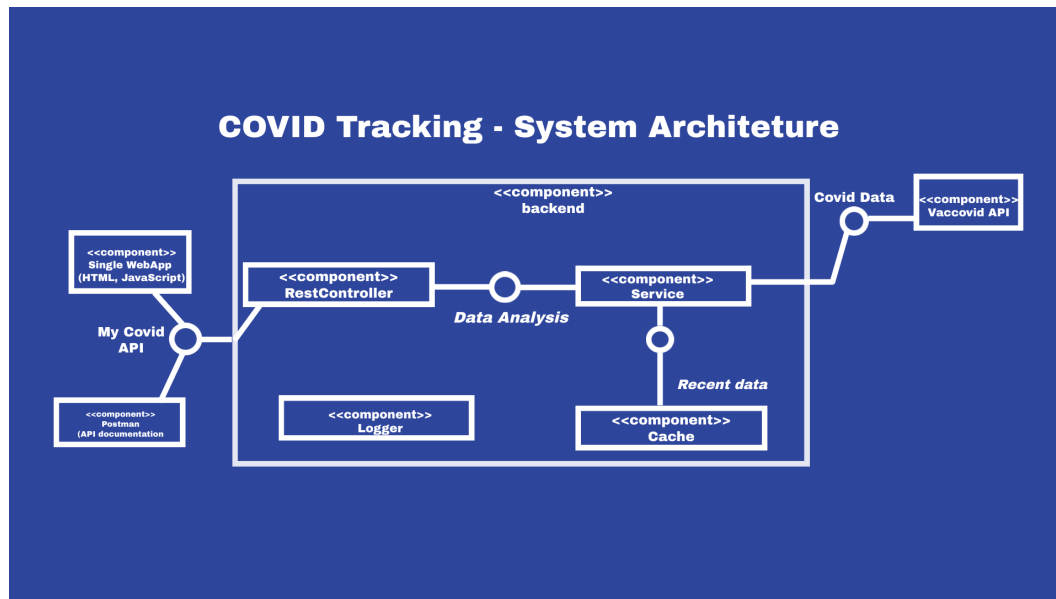


Figura 1 - Arquitetura do Sistema

2.3 API para o desenvolvimento

A documentação da API desenvolvida está presente através de uma documentação feita na plataforma *Postman* e pode ser consultada [aqui](#) onde mostra em detalhe em alguns exemplos como se pode aceder às diferentes informações:

- Sobre os países;
- Sobre as estatísticas diárias dos países;
- Sobre as estatísticas diárias do mundo;
- Sobre as estatísticas semestrais de certo país.

3 Quality assurance

3.1 Estratégia para a testagem

Embora saiba que a estratégia TTD é a mais favorável, não consegui realizar a mesma. Comecei a testar a aplicação assim que consegui ligar a API construída com o *frontend*. Tal como foi realizado nas aulas da disciplina TQS os testes unitários foram desenvolvidos através do *Junit5*, os testes de integração e serviços em *Mockito* e *SpringBoot Mock Mvc*. Para os

testes do *frontend* apliquei o BDD (Behavior-driven development) através do *Selenium WebDriver*.

A ordem de testagem foi a seguinte: comecei por escrever os testes unitários da *Cache* e dos *Models* (*Place.java* e *Statistic.java*), depois passei para os serviços, *RestController* e o *frontend* através dos testes *Selenium Web driver* com suporte da ferramenta *Cucumber*. No final, utilizei o *SonarQube* e os relatórios *JaCoCo* para completar *code smells* e corrigir certos erros que não eram tão notórios.

3.2 Testes de integração e unitários

3.2.1 Testes aos Models (*PlaceTest.java* e *StatisticTest.java*)

Foram realizados testes unitários na *Cache* e nos *Models*.

Exemplo, para ver se criava uma localidade de maneira correta:

```
public class PlaceTest {

    Place france = new Place("France", "fra", "Europe", 65533058);

    @Test
    void getPlaceTest() {

        assertEquals("France", france.getCountry());
        assertEquals("fra", france.getIso());
        assertEquals("Europe", france.getContinent());
        assertEquals(65533058, france.getPopulation());

    }
}
```

3.2.2 Testes à Cache (*CacheTest.java*)

Foi realizado um teste unitário com inúmeros *asserts* para verificar se a *cache* funcionava como pretendido, se adicionava/removia o objeto no momento certo e outros aspectos. Aqui deixo um excerto do teste com todos os *asserts*:

```
public class CacheTest { //unit testing
    @Test
    public void testMapAssert() throws InterruptedException {
        Cache.cacheMap.clear(); // cleaning cache
        HashMap<String, Object> expectedCashMap = new HashMap<>();

        // Places

        Place france = new Place("France", "fra", "Europe", 65533058);
        Place brasil = new Place("Brazil", "bra", "South America", 215274575);
```

```

Place croatia = new Place("Croatia", "hrv", "Europe", 4059781);
Place azores = new Place("az", "Azores", "Europe", 4059781);

Cache.cacheMap.put("country_name_brazil_return_place", brasil);
Cache.cacheMap.put("country_name_croatia_return_place", croatia);
Cache.cacheMap.put("country_name_france_return_place", france);

expectedCashMap.put("country_name_brazil_return_place", brasil);
expectedCashMap.put("country_name_croatia_return_place", croatia);
expectedCashMap.put("country_name_france_return_place", france);

// Statistics

Statistics franceStats = new Statistics("France", 7845245, 14548, 7845512, 56, 488954, 11235, 785200, 7.85);
Statistics brasilStats = new Statistics("Brazil", 1225445, 48481, 123358788, 8855, 54454, 111555, 87455265,
    9.85);
Statistics croatiaStats = new Statistics("Croatia", 1445565225, 1454, 12345845, 10, 4847747, 454, 4545454,
    2.48);
Statistics azoresStats = new Statistics("Azores", 157854, 1235, 201, 2, 10, 200, 14744, 1.48);

Cache.cacheMap.put("country_france_statistics", franceStats);
Cache.cacheMap.put("country_brazil_statistics", brasilStats);
Cache.cacheMap.put("country_croatia_statistics", croatiaStats);

expectedCashMap.put("country_france_statistics", franceStats);
expectedCashMap.put("country_brazil_statistics", brasilStats);
expectedCashMap.put("country_croatia_statistics", croatiaStats);

// Assert that cache worked perfectly fine
assertThat(Cache.cacheMap, is(expectedCashMap));

// Assert cache size
assertThat(Cache.cacheMap.size(), is(6));

// Cache has the right inputs, doesn't contain any value that wasn't add
assertThat(Cache.cacheMap, IsMapContaining.hasEntry("country_france_statistics",
franceStats));
    assertThat(Cache.cacheMap, IsMapContaining.hasEntry("country_name_france_return_place",
france));
    assertThat(Cache.cacheMap,
not(IsMapContaining.hasEntry("country_name_france_return_place", azores)));
    assertThat(Cache.cacheMap, not(IsMapContaining.hasEntry("country_azores_statistics",
azoresStats)));

// Cache is able to save countries and its statistics

assertThat(Cache.cacheMap, IsMapContaining.hasEntry("country_name_france_return_place",
france));
    assertThat(Cache.cacheMap, IsMapContaining.hasEntry("country_france_statistics",
franceStats));

// Cache has the right keys (country and statistics) and its expected values
assertThat(Cache.cacheMap, IsMapContaining.hasKey("country_name_croatia_return_place"));
assertThat(Cache.cacheMap, IsMapContaining.hasKey("country_croatia_statistics"));
assertThat(Cache.cacheMap, IsMapContaining.hasValue(croatia));
assertThat(Cache.cacheMap, IsMapContaining.hasValue(croatiaStats));

// Cache doesn't have the keys that weren't add
assertThat(Cache.cacheMap, not(IsMapContaining.hasValue(azores)));
assertThat(Cache.cacheMap, not(IsMapContaining.hasValue(azoresStats)));

```

```

//After 10 seconds an object is removed from a cache

ClockRemoveFromCache("country_name_croatia_return_place");
TimeUnit.SECONDS.sleep(10); // Waiting 10 seconds for the object being removed. Although it was only needed 1, + safety
assertThat(Cache.cacheMap, not(IsMapContaining.hasKey("country_name_croatia_return_place")));

// Test if hits and miss are working correctly
//its expected 2 hits and 2 misses
Object obj = Cache.cacheMap.get("country_name_france_return_place"); //hit
Object obj2 = Cache.cacheMap.get("country_name_azores_return_place"); //miss
Object obj3 = Cache.cacheMap.get("country_name_brazil_return_place"); //hit
Object obj4 = Cache.cacheMap.get("country_name_anotherplace_return_place"); //miss
Object obj5 = Cache.cacheMap.get("country_name_croatia_return_place"); //miss, was just removed from cache

Status st = new Status(0, 0);

if (obj == null) {
    st.setMiss();
} else{
    st.setHit();
}
if (obj2 == null) {
    st.setMiss();
} else {
    st.setHit();
}
if (obj3 == null) {
    st.setMiss();
} else {
    st.setHit();
}
if (obj4 == null) {
    st.setMiss();
} else {
    st.setHit();
}
if (obj5 == null) {
    st.setMiss();
} else {
    st.setHit();
}

assertThat(st.getHit(), is(2));
assertThat(st.getMiss(), is(3));

}

```

3.2.3 Testes aos Controllers (*ControllerTest.java*)

Através das anotações *MockMvn* e *MockBean* foi possível testar para cada endpoint que informação provinha dos serviços simulados.

Um exemplo:

```

@Test
void getWorldDataTest() throws Exception {
    Statistics worldData = new
Statistics("World",507955550,255334,6237108,933,41733,459875123,907573, 0.0);

    when( service.getStatsWorld() ).thenReturn( worldData );
}

```

```

mvc.perform(
    get("/api/statistics/world").contentType(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.*", hasSize(9)))
    .andExpect(jsonPath("$.place", is(worldData.getPlace())))
);

verify(service, times(1)).getStatsWorld();
}

```

3.2.4 Testes aos Serviços (*StatisticServiceTest.java*)

Neste caso utilizei a anotação `@Mock` para simular o serviço e os elementos a serem gerados. Depois comparei se as componentes estavam a ser criados de maneira correta.

Exemplo:

```

@ExtendWith(MockitoExtension.class)
public class StatisticServiceTest {

    @Mock
    private HandlingRequestsService handler;

    @Mock
    private StatisticsService service;

    @Mock
    Statistics worldData = new Statistics("World", 507955550, 255334, 6237108, 933, 41733, 459875123, 907573,
        0.0);

    @Test
    void getStatsWorldTest() throws IOException, InterruptedException {

        Mockito.when(service.getStatsWorld()).thenReturn(worldData);

        Statistics info = service.getStatsWorld();

        assertEquals(worldData.toString(), info.toString());
    }
}

```

3.3 Testes funcionais

Para testar a interface com que o utilizador interage, foi utilizado a abordagem *Behavior-Driven Development* recorrendo ao *Selenium WebDriver*. Tal como realizamos nas aulas práticas, cenários foram escritos para a realização dos testes. Um excerto do ficheiro *trackincovid.feature*:

Feature: Tracking

Scenario: Search for all world covid statistics

When I'm accessing "http://localhost:8080/"

And the user clicks on "All Countries" button

Then the user should see the following message "Today's World Data" with the informations

Scenario: See Croatia statistics

When I'm accessing "http://localhost:8080/countrydata"

And the user clicks on "Specific Country" button

And the user searches for "Croatia" on the search bar

And the user clicks on "search" button

Then "Croatia" statistics are presented in a table

Scenario: User searches for a nonexistent country

When I'm accessing "http://localhost:8080/countrydata"

And the user searches for "Azores" on the search bar

And the user clicks on "search" button

Posteriormente há que realizar um teste para cada um destes *steps*. Exemplo de dois testes do ficheiro FrontendSteps.java:

```
@When("I'm accessing {string}")
public void browseTo(String url) {
    driver = new ChromeDriver();
    driver.get(url);
    driver.manage().window().setSize(new Dimension(1479, 837));
}

@When("the user clicks on {string} button")
public void checkButtonClick(String button) {
    switch (button) {
        case "search":
            driver.findElement(By.id("searchButton")).click();
            break;
        case "Specific Country":
            driver.findElement(By.id("specificCountry")).click();
            break;
        case "All Countries":
            driver.findElement(By.linkText("All countries")).click();
            break;
    }
}
```

Uma pequena nota, no ficheiro das *features* encontra-se em falta um passo que era o final do último cenário, onde o utilizador vê um aviso a informar que não foi encontrado nenhum país com aquele nome. Porém, não consegui implementar o teste então ficou comentado.

3.4 Análise da qualidade do código

3.4.1 SonarQube

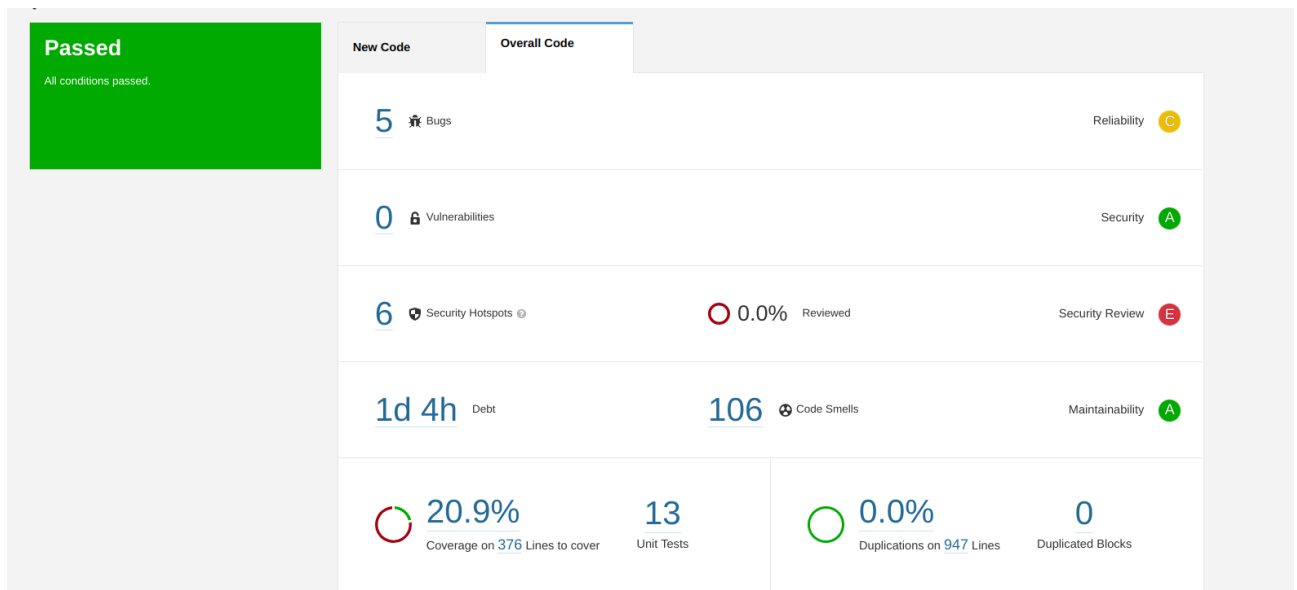


Figura 2 - Análise inicial

Para a análise do código utilizei o SonarQube para uma revisão aprofundada, que consegue ver com muito detalhe o código e apontar pequenos ou grandes erros/code smells (más práticas de programação) que sozinha nunca teria dado conta. Inicialmente o relatório era o demonstrado na figura 2.

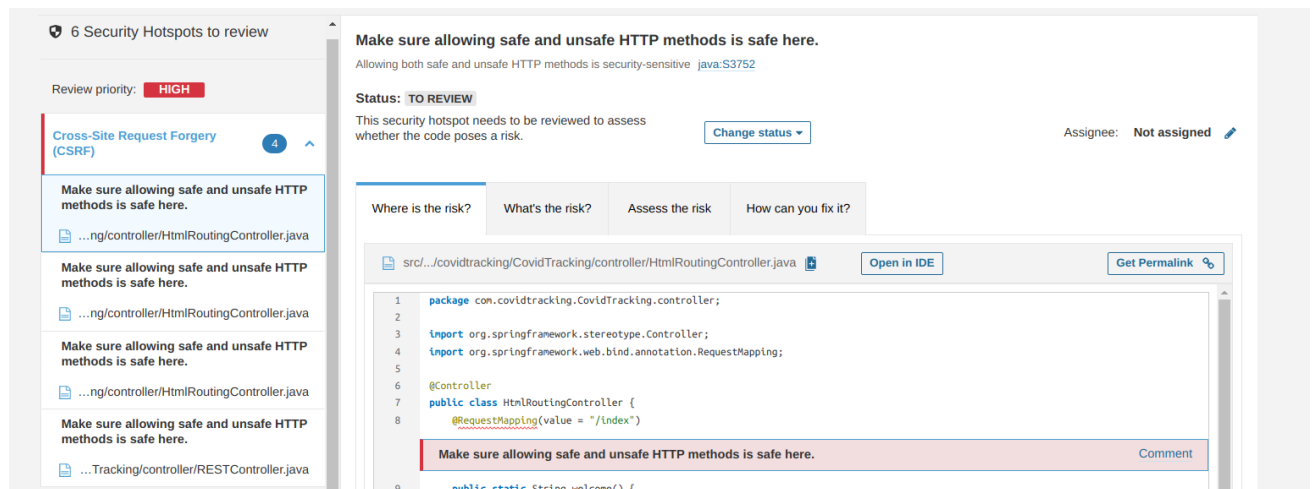


Figura 3 - Security Hotspots

Decidi analisar o mais grave, os *Security Hotspots*, que estava avaliado com um **E**. Para resolver foi bastante fácil uma vez que o SonarQube fornece uma explicação do problema e sugestões de como resolvê-lo. Assim rapidamente percebi que tinha de explicitar nas anotações *@RequestMapping* o valor e o método que iria utilizar, por exemplo:

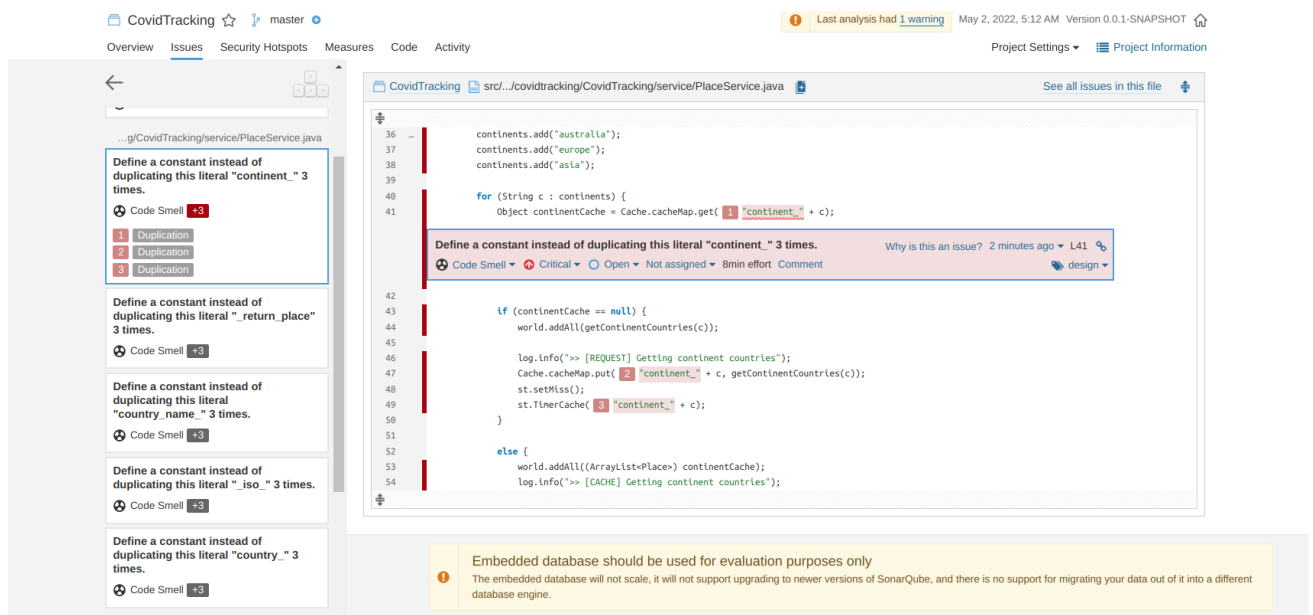
```
@RequestMapping(value = "/index", method = RequestMethod.GET)
public static String welcome() {
    return "index"; }
```

De seguida, analisei e corriji alguns dos critical e major bugs. Mais uma vez, foi muito fácil com ajuda da ferramenta identificar o problema e a sua solução (Fig. 4 e 5)

Figura 4 - Bug - Comparar Strings de maneira correta

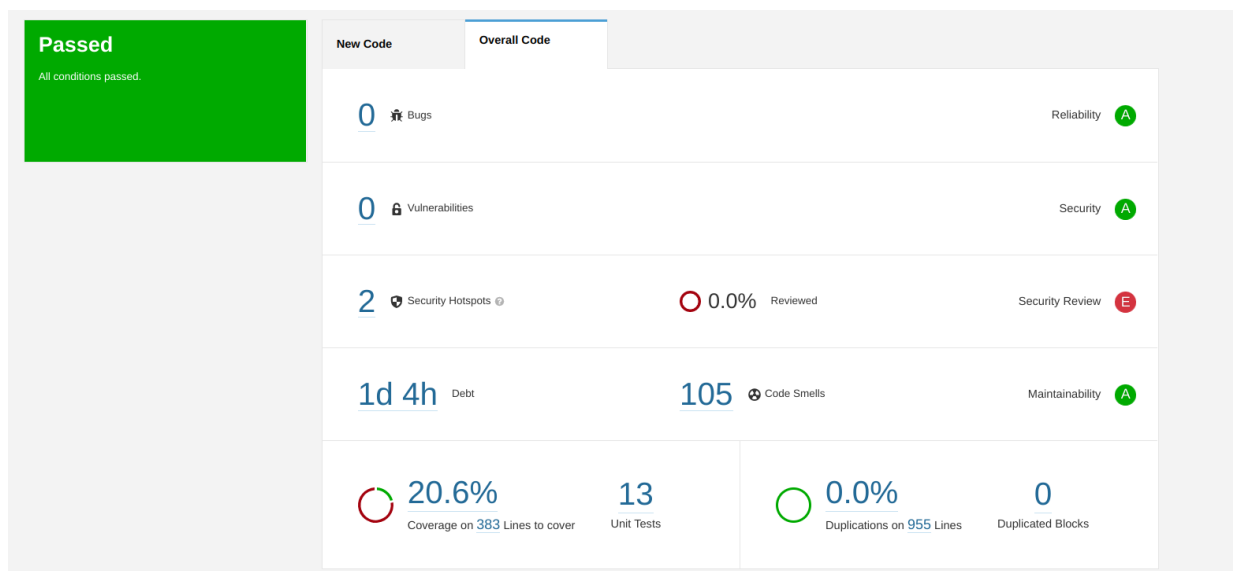
The screenshot displays the SonarQube interface for a bug report. On the left, a sidebar lists several bugs, with the selected one being 'Strings and Boxed types should be compared using "equals()"'. The main area shows the code snippet for the bug. The code is in Java and defines a service class `SixMonthsStatisticsService`. It includes a `getStatisticsData` method that takes a `String iso` parameter and returns a `ArrayList<SixMonthsStatistics>`. The code uses `String` and `JSONArray` objects. The bug report indicates that the comparison of `String` and `Boxed` types should be done using `"equals()"`. The bug is categorized as 'Major' and is currently 'Open'.

Figura 5 - Bug - Criar constantes para não repetir novas Strings ao longo do código



The screenshot shows the SonarQube web interface for the 'CovidTracking' project. The left sidebar displays a list of issues, all categorized as 'Code Smell' with a severity of 'Critical'. The main panel shows the source code for 'PlaceService.java' with a highlighted issue: 'Define a constant instead of duplicating this literal "continent_" 3 times.' The code snippet shows a loop where the literal 'continent_' is repeated multiple times. A warning banner at the bottom states: 'Embedded database should be used for evaluation purposes only. The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.'

Figura 6 - Resultado final



The screenshot shows the JaCoCo test results dashboard. On the left, a green box indicates 'Passed' with 'All conditions passed.' The main dashboard is divided into two tabs: 'New Code' and 'Overall Code'. The 'Overall Code' tab is active, showing the following metrics:

- Bugs:** 0
- Vulnerabilities:** 0
- Security Hotspots:** 2 (0.0% Reviewed)
- Debt:** 1d 4h
- Code Smells:** 105
- Coverage:** 20.6% (Coverage on 383 Lines to cover)
- Unit Tests:** 13
- Duplications:** 0.0% (Duplications on 955 Lines)
- Duplicated Blocks:** 0

Reliability and Security are both rated 'A'.

No total o código ficou apenas com 2 *Security Hotspot*, 105 *code smells* e 13 testes, sendo que há coverage em 20.6% do código.

3.4.2 JaCoCo

Para gerar o relatório JaCoCo bastou correr o comando `mvn clean test jacoco:report`

após instaladas as dependências. Cria automaticamente um relatório na pasta target com a seguinte página a ilustrar o quão testado está o código e o que escapou, sendo que a cor verde representa que os *branches* foram *covered* durante o teste e o vermelho o contrário.

CovidTracking

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.covidtracking.CovidTracking.service	<div><div></div></div>	7%	<div><div></div></div>	0%	31	37	185	202	11	17	0	4
com.covidtracking.CovidTracking.models	<div><div></div></div>	37%		n/a	36	52	75	112	36	52	1	3
com.covidtracking.CovidTracking.exception	<div><div></div></div>	4%		n/a	7	8	13	14	7	8	2	3
com.covidtracking.CovidTracking.cache	<div><div></div></div>	20%		n/a	10	11	14	20	10	11	2	3
com.covidtracking.CovidTracking.controller	<div><div></div></div>	73%	<div><div></div></div>	60%	9	19	11	32	6	14	0	3
com.covidtracking.CovidTracking		37%		n/a	1	2	2	3	1	2	0	1
Total	1,135 of 1,405	19%	44 of 50	12%	94	129	300	383	71	104	5	17

Figura 7 - Resultado geral

Agora vendo um exemplo de uma classe, neste caso a *RestController*, conseguimos ver que a maior parte das funções foram bem testadas.

Figura 8 - Análise da classe *RestController*.

RestController

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getPlaceByCountry(String)	<div><div></div></div>	0%	<div><div></div></div>	0%	2	2	4	4	1	1
getCacheStatistics()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
getCountriesByContinent(String)	<div><div></div></div>	93%	<div><div></div></div>	75%	1	3	1	8	0	1
getAllCountries()	<div><div></div></div>	93%	<div><div></div></div>	75%	1	3	1	8	0	1
RestController()	<div><div></div></div>	100%		n/a	0	1	0	2	0	1
getCountriesStatistics()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
getStatsByCountry(String)	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
getWorldStatistics()	<div><div></div></div>	100%		n/a	0	1	0	1	0	1
Total	19 of 98	80%	4 of 10	60%	5	13	7	26	2	8

4 Referências & Recursos

Recursos:	URL
Repositório	https://github.com/marianarosa01/CovidTracking
Demonstração em vídeo	https://www.youtube.com/watch?v=5CtyD5JG4Pw
Documentação da API (Postman)	https://documenter.getpostman.com/view/18915431/UyrAFwiy

Referências e material utilizado

Para a realização deste trabalho optei por utilizar a API [VACCOVID](#) pois tinha diversas informações, pois o meu objetivo inicial era fazer estatísticas sobre os dados atuais e de há 6 meses atrás, contudo como já foi explicado anteriormente não consegui realizar até ao final.

A nível de investigação utilizei o suporte fornecido pelo docente Ilídio Oliveira, quer seja material das aulas teóricas e os guiões práticos. Também utilizei a documentação presente no *website* [Baeldung](#).