HW1: Mid Term Assignment Report

Lucius Vinicius Rocha Machado Filho
96123

1 Introdução

1.1 Visão Geral do Trabalho

O objetivo principal era criar uma aplicação Maven em que servia de middleware entre uma API externa, filtrando os campos desta, e outros utilizadores. Além disso, um frontend em que demonstrasse alguns dos pedidos possíveis para a nossa própria API criada também era necessário. Tudo isso sempre seguido de testes e garantia de qualidade.

O produto é uma API em que você pode obter informação sobre covid através de endpoints.

1.2 Limitações

A API infelizmente não lida muito bem com grandes períodos de tempo, uma vez que nenhuma das APIs externas tinham a possibilidade desse filtro. Por essa razão, pedidos que envolvam muitos dias tendem a demorar, principalmente se nenhum desses pedidos estiverem na Cache.

2 Especificação do Produto

2.1 Interações Suportadas

É possível utilizar a API para obter informações sobre o COVID-19 em determinados países, assim como ver o seu histórico ao longo de um determinado tempo. Além disso, também é possível obter informações da Cache usada internamente.

2.2 Arquitetura do Sistema

Para o backend foram utilizados projetos Maven em conjunto com o Spring Boot para fazer a conexão entre API externas, serviços e endpoints customizáveis. Para os testes foram utilizados JUnit para os testes unitários, em conjunto com a biblioteca *mockito* para quando houvesse a necessidade da criação de Mocks. Já para os testes de integração foram utilizados o *springbootest* em conjunto com *Rest Templates* para realizar a simulação de pedidos a nossa API. Por último, os testes funcionais foram feitos através do Selenium e Cucumber e foi utilizado o Sonarqube como garantia de bom código.

O frontend já foi feito em ReactJS, para modificações em tempo real quando os pedidos eram feitos, e com a biblioteca Material UI para estilizar levemente os inputs do site, mas ainda contento o design mínimo esperado.

No projeto foram usadas duas API externas sendo a primeira o COVID-19, e a segunda o COVID-19 Statistics, ambas APIs estavam na hiperligação citada no enunciado do projeto. A aplicação chama uma API ou outra dependendo do endpoint, assim providenciando valores que podem divergir a partir de uma determinada parametrização,

além de providenciar uma possível troca de pedido se por acaso uma das APIs externas forem abaixo.

Por último, foi também utilizada uma cache com TTL para armazenar pedidos repetidos às APIs externas, demonstrando ser bastante útil quando eram feitos os pedidos de um período de dias, uma vez que ambas as APIs não providenciam esse tipo de parâmetro, tendo que ser feito então vários pedidos.

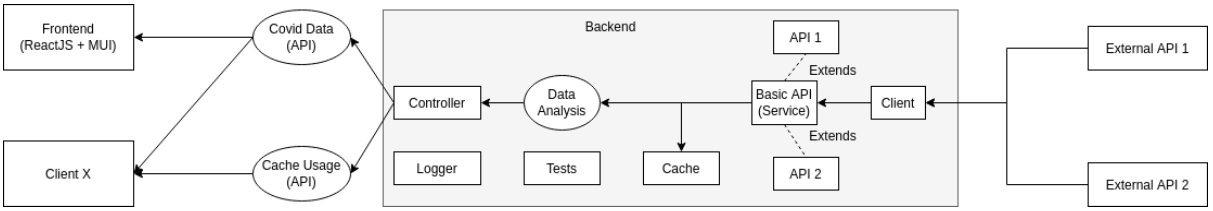

Fig. 1: Diagrama da estrutura

2.3 API para developers

Fig. 2: Documentação da API

Os endpoints da API são separados em “api1” e “api2”, cada um desses representando as APIs externas utilizadas. Já que nada entre elas têm em comum, foi decidido que a separação em seu uso seria simplesmente através da escolha do utilizador final.

A API consegue retornar, para cada API externa, uma lista com os nomes dos países disponíveis nelas, assim como filtrar por esse nome. Esse último filtro ainda possui argumentos opcionais, sendo eles a data inicial e final, para assim conseguir obter um histórico de um período de tempo.

Por último, a API consegue retornar os dados de uso da Cache, sendo eles o número de acertos/falhas, número de pedidos, e até mesmo seu TTL.

3 Garantia de Qualidade

3.1 Estratégia no Geral para Testes

A estratégia a princípio não foi baseada em TDD, o que na verdade causou problemas (mais de uma vez) acerca da estrutura. Depois, quando esses problemas foram resolvidos, o projeto passou a seguir essa técnica, havendo bem menos alterações no futuro.

Para o BDD foi utilizado Cucumber com o Selenium e para os testes Unitários foram utilizados majoritariamente JUnit, Mockito e RestTemplate.

3.2 Testes Unitários e de Integração

3.2.1 Cache

O primeiro teste a ser produzido foi o unitário sobre a Cache, uma vez que era o mais simples de se implementar (já que não envolvia o uso de Mocks), e também era necessário confirmar se os inserts/getters estavam funcionando como esperado, assim como o TTL. Nele, foram aplicados testes quando a Cache era criada, inserida algo, seus status (que são dados para o endpoint na API), links inexistentes e o TTL em si.

Ênfase no teste do TTL, uma vez que é necessário esperar um determinado tempo para testar corretamente. Então foi utilizado o método await().until() para esperar até que a cache não contivesse mais a chave, que é o URL do pedido.

```
assertTrue(cache.contains(response));
assertThat(cache.get(uri), is(response));

await().until(requestIsExpired(uri));

assertTrue(cache.isEmpty());
```

Fig. 3: Chamada da função de espera no teste unitário da Cache

```
1 usage  itstar
private Callable<Boolean> requestIsExpired(String key) {
    itstar
    return new Callable<Boolean>() {
        itstar
        public Boolean call() { return !cache.containsKey(key); }
    };
}
```

Fig. 4: Função de espera

Como mostrado na Figura 2, existiam asserts para verificar que o conteúdo existia na cache, porém após a espera, a cache se encontrava vazia.

3.2.2 Controller (Unitário)

O teste do Controller já utiliza objetos Mocks, ao contrário da Cache. No Controller utilizamos o MockMvc para simular pedidos aos endpoints na API e com isso simular também os resultados vindo dos Services.

As funções testadas foram as de obter a lista de nomes de todos os países, obter algum país através do nome, obter um histórico de um país através do nome e um período de datas, a tentativa de obter um país através de um nome não existente e a obtenção dos status da Cache. Todas essas funções foram igualmente testadas para ambas as APIs que estão disponíveis através dos Services.

```
itstar
@Test
void getSpecificCountryNotFoundForAPI1() throws Exception {

    String country = "not_existent";

    when(client1.getCountryByRegion(country)).thenReturn(new ArrayList<>());

    mvc.perform(
        get("/api1/countries/not_existent")
    )
    .andExpect(jsonPath("$", hasSize(0)));

    verify(client1, times(1)).getCountryByRegion(country);
}
```

Fig. 5: Exemplo do teste no Controller sobre um país com um nome não existente na API 1.

3.2.3 Controller (Integração)

O único teste de integração foi realizado no controller, uma vez que ele é o responsável pelo input mais externo da aplicação (os endpoints da API feita) e esses endpoints percorrem a arquitetura até os mais externos, assim realizando um caminho que percorre tudo.

A estratégia para fazer os testes de integração usada foi a de utilizar um RestTemplate e o @SpringBootTest para simular pedidos aos próprios endpoints da API.

```
itstar *
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class CovidControllerIT {

    @LocalServerPort
    int randomServerPort;

    5 usages
    @Autowired
    private TestRestTemplate restTemplate;
```

Fig. 6: Setup das configurações para os testes de integração.

Depois de configurado, apenas foi necessário converter os testes unitários já existentes. Esse efeito foi realizado após trocar a condição do Mock pelo RestTemplate.

```
@Test
void getSpecificCountryForAPI2() {
    validateSpecificCountry("api2");
}

2 usages  itstar
private void validateSpecificCountry(String api) {
    itstar
    ResponseEntity<List<DataOutput>> response = restTemplate.exchange("/" + api + "/countries/brazil", HttpMethod.GET, null, new ParameterizedTypeReference<List<DataOutput>>() {

    });

    List<DataOutput> countries = response.getBody();
    assertThat(countries.size(), is(1));
    assertThat(countries.get(0).getCountry(), is("Brazil"));
}
```

Fig. 7: Exemplo de teste feito na integração. No caso, o exemplo é para obter um país específico através da API 2.

3.2.4 APIs

Assim como no teste do Controller unitário, os testes das APIs utilizam Mocks, porém no caso das APIs é para simular respostas dos endpoints externos para cada uma através da classe Client e a notação *@Mock*. Apesar de serem ficheiros de testes diferentes, ambas as APIs seguem uma estrutura de testes muito semelhante, não havendo então muita razão para separar a explicação dela.

As funções testadas foram: obter a lista dos nomes de países; obter dados de um país em específico através do nome; obter dados de um país através do nome e um período de data; obter dados de um país que não existe no endpoint externo; obter dados através de uma data do futuro. Ademais, o único teste exclusivo para uma das APIs foi feito para a API 1, uma vez que a resposta do endpoint externo pode conter valores iguais a null em alguns campos específicos, sendo então realizado um teste para fazer essa verificação.

```java
@Test
void getSpecificCountryWithNullNewDeathsAndNullCases() throws IOException, InterruptedException, ParseException {
    when(client.doRequest(HISTORY_URL + "?country=angola&day=2022-04-21", cache, api.getHeaderHost(), api.getHeaderKey()))
            .thenReturn(jsonMeth.generateJSONObject(SPECIFIC_COUNTRY_WITH_NULL_DEATHS_AND_CASES));

    List<DataOutput> response = api.getCountryByRegionAndDate("angola", "2022-04-21", "2022-04-21");

    assertThat(response.get(0).getCountry(), is("Angola"));
    assertThat(response.get(0).getNewDeaths(), is(0l));
    assertThat(response.get(0).getNewActive(), is(0l));
}
```

Fig. 8: Função exclusiva a API 1 para testar os campos iguais a null.

3.3 Testes Funcionais

Uma vez que os testes funcionais entram em contato com o frontend, antes é preciso saber de como ele é estruturado:

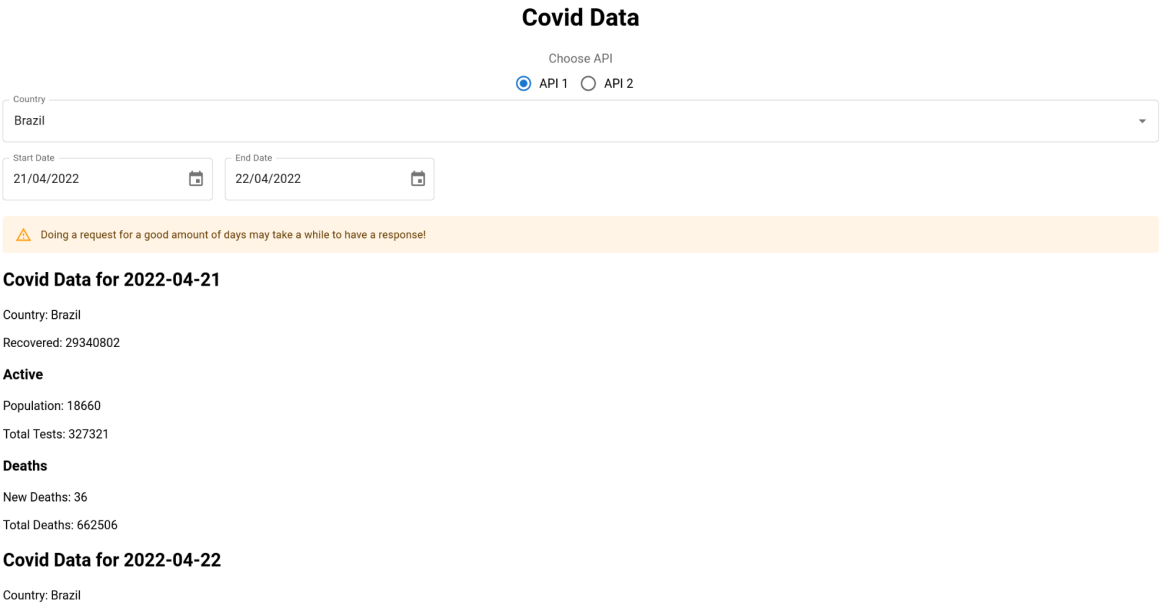

Fig. 9: Vista do frontend.

A partir da figura acima, é possível ver que o site é composto de 4 inputs principais, sendo eles sendo a seleção da API, o nome do país, e dois inputs para definir o período de tempo para fazer a busca.

Após os inputs existe uma mensagem alertando que fazer pedidos com um período de tempo demasiado grande pode causar uma certa demora da API, uma vez que, como dito anteriormente, esses pedidos são feitos manualmente, já que os endpoints externos não possuem essa função acoplada. Depois, é possível analisar que ele demonstra os dados para cada dia especificado nos inputs.

A partir dessa informação, é possível a criação de cenários Cucumber sobre o site. Esses cenários, já que eram bastante semelhantes, foram então convertidos em Scenario Outline, para conseguir replicar o uso da mesma estrutura diversas vezes:

```
Feature: API
    Background: The Website
        Given the website just opened

    Scenario Outline: Search with API
        When I choose for the <api> API
        And I choose the country <country>
        And I choose the date range between 12 and 18 of April
        Then new deaths on day <day> is <number_deaths>

        Examples:
            | api       | country       | day | number_deaths |
            | "Api 1"   | "Brazil"      | 12  | 104           |
            | "Api 1"   | "Argentina"   | 18  | 17            |
            | "Api 2"   | "Brazil"      | 14  | 139           |
```

Fig. 10: Cenários do Cucumber.

Com a estrutura do cenário já pronta, foi necessário fazer a configuração nas classes de teste em Java para conseguir preparar os testes, assim como o uso de Selenium para operar com o frontend automaticamente.

Felizmente, com o Selenium IDE é possível gravar interações com o website e a partir daí conseguir a equivalência em Java. Com o export do Selenium IDE já pronto, foi necessário separar em funções em relação ao Cucumber (Given, When, And e Then) e aplicar parâmetros dinâmicos.

```java
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("ies/hw/hw1")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "ies.hw.hw1")
public class FrontendTest {

}
```

Fig. 11: Estrutura do ficheiro base de teste, que faz comunicação com a feature Cucumber.

```java
@Given("the website just opened")
public void setUp() {
    driver = new FirefoxDriver();
    js = (JavascriptExecutor) driver;

    driver.get("http://localhost:3000/");
}
```

Fig. 12: Função que utiliza o método *@Given* para inicializar o web driver.

Com essas implementações, os testes automáticos do frontend passaram a funcionar.

3.4 Análise da Qualidade do Código

Para obter qualidade no código, foi utilizado o Sonarqube, em conjunto com o JaCoCo para obter também a cobertura dos testes. É importante citar que o Sonarqube não começou a ser utilizado desde o início do projeto.

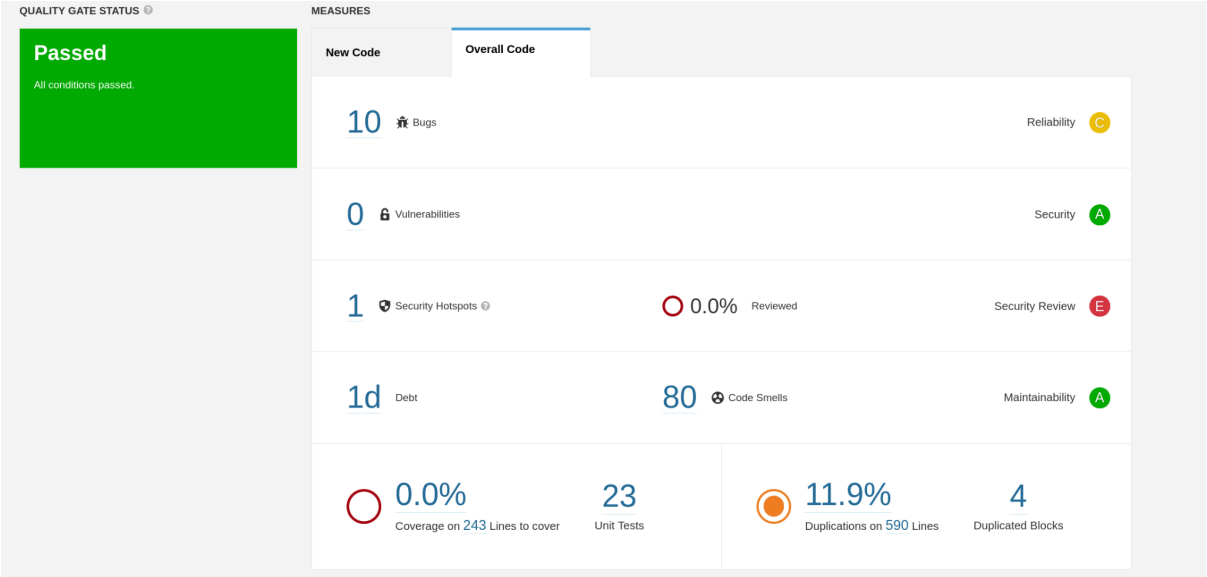

Fig. 13: Primeiro Dashboard mostrado pelo Sonarqube.

Observação: A princípio o JaCoCo não estava acoplado, então por isso a cobertura dos testes estava a 0.0%.

Como é possível observar, existe um número considerável de bugs e code smells, além de demasiadas linhas repetidas de código. Essas linhas se referiam à API 1 e 2, em que ainda não existia uma superclasse que abrangesse ambas; apenas existia uma simples interface.

Alguns Code Smells interessantes obtidos foram os que indicavam que os métodos de teste não precisavam ser *private*, o retorno padrão quando não se tinha informação (ao invés do return null que tinha anteriormente), o reforço de uso de logs ao invés de prints usuais e *protected* no construtor da superclasse. De resto, a maioria dos code smells foram importações não utilizadas ou falha na convenção dos nomes das variáveis.

Fig. 14: Code Smell sobre as classes de teste públicas.

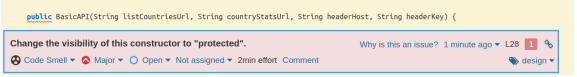

Fig. 15: Code Smell sobre o uso de um construtor público em uma superclasse abstrata.

Fig. 16: Code Smell sobre o uso de prints ao invés de logs.

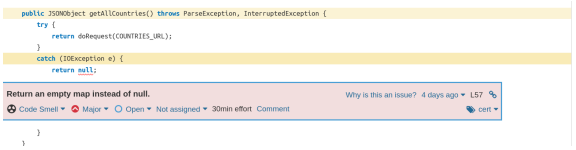

Fig. 17: Code Smell sobre valor default null, ao invés de ser uma Lista/Objeto vazio.

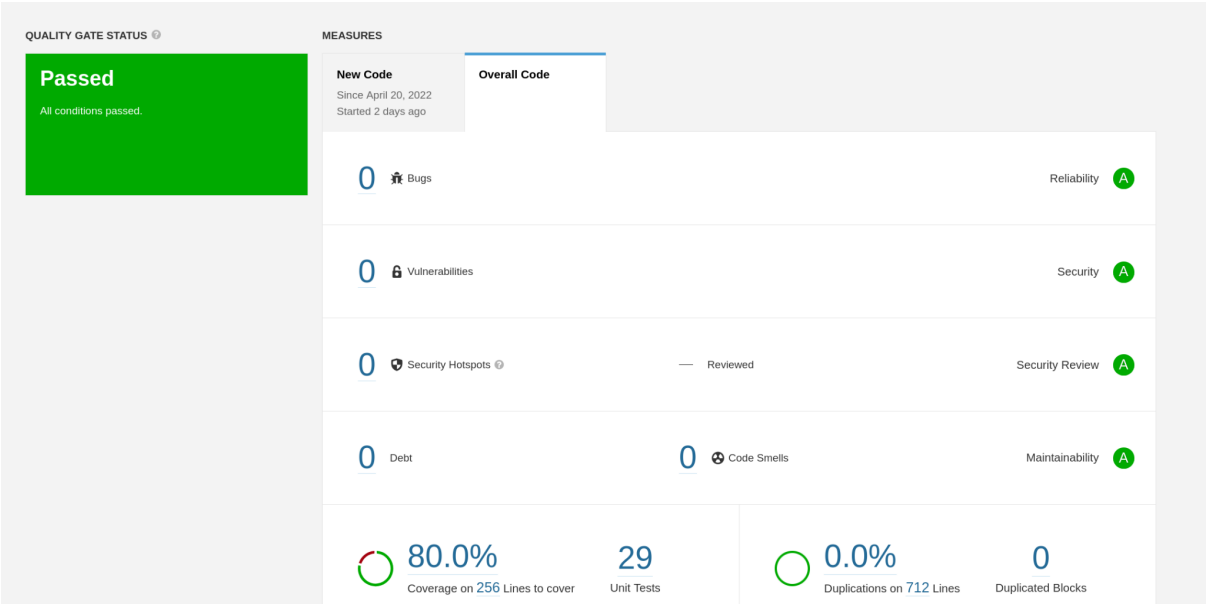

Fig. 18: Dashboard final do Sonarqube.

É importante dizer que houve uma alteração nas métricas do quality gate. Já que existiam muitos testes em que não testavam setters/getters de determinadas classes ou até mesmo a condição de o log ser capaz de dar informação (que é sempre verdade), houve uma diminuição da cobertura de testes do novo código ser 75%, ao invés de 80%. Porém, ainda assim no código em geral foi garantido 80% de cobertura nos testes.

4 Referências e Recursos

Repositório Git: https://github.com/luciusvinicius/tqs_96123

Video demo: A demo do vídeo se encontra no repositório, em /report/video

Materiais de referência:

API Externa 1 - COVID-19

API Externa 2 - COVID-19 Statistics

Material UI

ReactJS