deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Alexandre Antunes Rodrigues [92951], 2021-05-14

1 Introduction

1.1 Overview of the work

No âmbito da disciplina de Teste e Qualidade de Software (TQS) foi proposto a realização de um pequeno projeto com vista a numa só aplicação aplicar os conhecimentos adquiridos em maioria das aulas práticas.
Assim, foi criada uma aplicação web que tem como objetivo apresentar os componentes poluentes do ar dada uma localização ou uma localização e uma data. Esta comunica com uma third-party API, neste caso foi escolhida uma API nomeada de Waqi que contrariamente a outras disponíveis, esta apresenta mais dados relevantes para o que é proposto analisar.
Como back-end, este foi criado em java com a ajuda do Spring Boot e o front-end foi realizado em htlm com a ajuda do thymeleaf.
Por fim foram realizados testes unitários, service level, integração e funcionais.

1.2 Current limitations

Como limitações ou como trabalho futuro, por um lado, poderia existir a comunicação com uma segunda third-party API visto que caso a primeira falhe teriamos a segunda como salva-guarda e, por outro lado, poderia existir uma pesquisa mais extensa com vários filtros tais como intervalos de tempo, vários locais, locais com maior incidência de determinado poluente de ar assim como outros filtros interessantes para o utilizador.

2 Product specification

2.1 Functional scope and supported interactions

A aplicação web permite, tal com demonstrado no vídeo auxiliar, as seguintes pesquisas e observações.

1. Pesquisa por localização ou/e por data;
2. Visualização de dados da localização tais como nome, latitude, longitude, data e timezone;
3. Vizualização de poluentes do ar assim como o seu valor médio, valor mínimo e máximo antendendo à pesquisa realizada (localização e/ou data);
4. Vizualização do número total de requests, misses e hits ocorridos na cache através de dados gráficos estatísticos;

2.2 System architecture

As classes criadas foram organizadas pelos packages: controller, model e service.
O *back-end* como já referido foi realizado usando *Spring Boot* e *Maven*.

Package Model

Dentro do package model encontram-se as classes:

AirPollutionForecast: Esta representa os dados de uma dado tipo de poluente referentes a um determinado momento de localização (data) assim como representa o valor médio, máximo e mínimo de um dado poluente.

Location: Representa uma localização. Tem como atributos o nome, latitude, longitude, time e timezone.

CacheMemory: Apresenta uma estrutura, um LinkedHashMap, que permite guardar como key uma Location e como value uma lista de AirPollutionForecast.
Também permite obter informação da mesma como o número de misses, hits e requests. O número máximo do tamanho da LinkedHashMap pode ser alterado.

CacheHistory: Apresenta uma parte do histórico da cache. É uma classe que tem com atributos tempo, número de hits, misses e requests.

Package Controller

HomeController: Contém o acesso a várias páginas html como o index logo de início assim como todas aquelas em que é usado um botão.

LocationController: Permite as ligações à API waqi assim como a procura e obtenção de dados da cache caso esta contenha logo os dados pesquisados (location e/ou date). É nesta classe que principalmente são definidos os endpoints assim como a maneira que cada request irá ser consumido no back-end.

APIController: Contem alguns métodos que foram utilizados para realização de testes.

Package Service

ConsumeWaqiAPI: Permite fazer chamadas à API waqi dependendo do tipo de request (location e/ou date).

DataAccess: Permite aceder aos dados, isto é, caso não exista a localização pedida na cache, irá ser feita uma chamada à API quer devolve os dados, caso a localização exista e se assim for são guardados na cache.

Package Swagger

SpringFoxConfig: Basicamente contém um Bean que permite a realização da documentação da API.

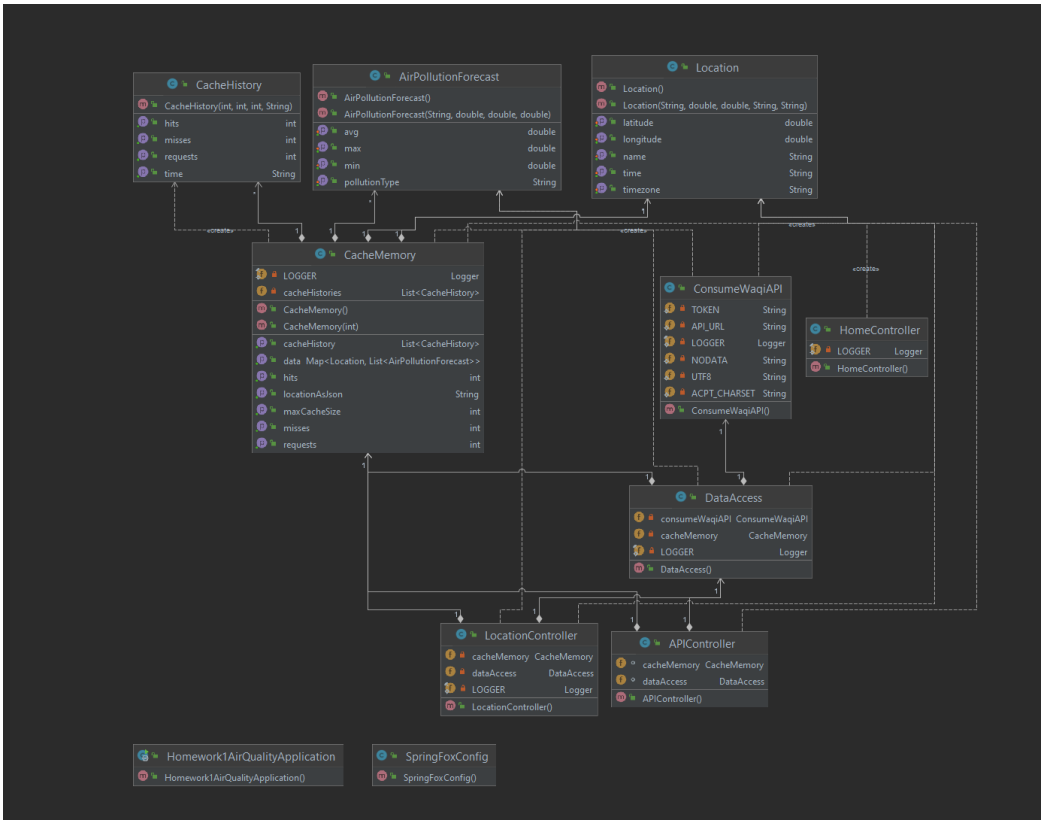

Fig1. Diagrama de classes.

2.3 API for developers

Foram criados 4 endpoints principais e que devem ser especificados.

1) /cacheHistory

Permite obter todo o histórico da cache (lista com o número de hits, misses e requests).

2) /cacheStatistics

Permite obter todas as estatísticas da cache através de análise grafical do número de hits, misses e requests.

3) /location

Dada uma localização permite obter todos os dados da mesma (atributos mencionados em cima) assim como todos os poluentes do ar que estão presentes na mesma.
Tem como parâmetos: latitude, longitude, name, time e timezone.

4) /locationAndDate

Dada uma localização e uma data permite obter todos os dados da mesma (atributos mencionados em cima) assim como todos os poluentes do ar que estão presentes na mesma.
Tem como parâmetos: latitude, longitude, name, time e timezone.

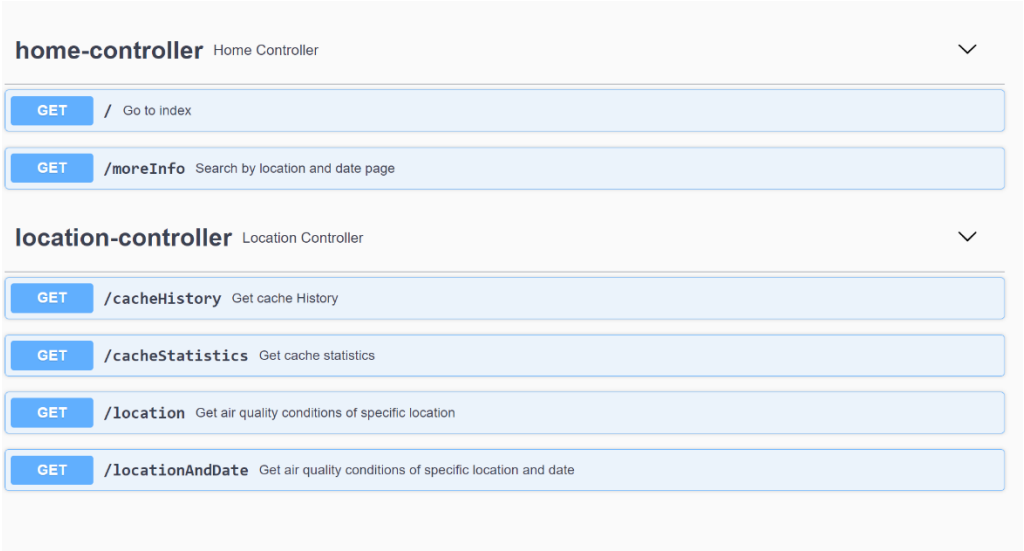

Fig2. Documentação da API

Pode ser visualizado em: http://localhost:8080/swagger-ui/index.html#

3 Quality assurance

3.1 Overall strategy for testing

Foi usada uma estratégia TLD (Test Last Development) apesar de não ser tão recomendado como TDD (Test Driver Development).
Ao fim de todo o desenvolvimento passei a realizar os testes unitários, seguindo a ordem dos testes pedidos no enunciado.
Na realizaçãoo dos testes foi usado JUnit5, Mockito, MockMvc e por fim o Selenium Jupiter para a realização dos testes funcionais.

3.2 Unit and integration testing

Com base no foco das classes que têm como base o modelo do negócio, isto é, todas as classes importantes e que têm algum comportamento de risco, foram realizados testes unitários.

CacheMemoryTest

Na classe caheMemory foram realizados testes unitários com o objetivo de verificar as condições iniciais da cache, isto é, o número de misses, hits e de requests ser igual a 0, testes ao número de misses, hits e requests após uma alteração dos elementos da cache e testes sobre o tamanho da cache após inserção de alguns elementos, verificado se dado o limite de tamanho (neste caso size 2) se o terceiro elemento implicaria que o elemento "mais velho" fosse retirado e desse a vez.

Nos testes unitários sobre as classes *Location* e *AirPollutionForecast* foi testado se os objetos eram iguais ou diferentes, isto porque é necessários que a key do HashMap presente na cache Memory não permita que objetos iguais (key) sejam colocados.

3.3 Service Level Tests

Foi usado a extenção do *Mockito*, fazendo "*Mock*" à classe *CacheMemory* e ao *ConsumeWaquiApi*, realizado *InjectMocks* na classe *DataAccess.*
Sobre os *service level tests* foram realizados testes quando dado uma localização (neste caso foi a Irlanda), quando é realizado um "*bad request*" à API, sobre localizações com detalhe e sobre localizações da cache.

3.4 Integration Tests

Foi usado *Spring Boot MockMvc* isto porque permite simular o comportamento de determinados objetos para a realização de testes de integração na API.
Foram realizados alguns testes de integração, usado o *MockBean* por forma a simular o comportamento da **cache** assim como o *dataAccess*.
Os testes de integração realizados foram nomeadamente na inserção de uma localização inválida, nos dados estatísticos da cache e nos dados estatísticos da cache quando retorna os dados em *JSON*.

3.5 Functional testing

Nos testes funcionais, com recurso ao *Selenium IDE*, foram gravadas as ações efetuadas na interface com a inserção de *asserts* propositados para a realização dos testes.
Assim, foi seguida uma dada ordem: estado inicial da cache, pesquisa de uma localização, estado da cache após a pesquisa, pesquisa de uma outra localização e estado da cache após essa pesquisa.

3.6 Static code analysis

Na análise do código estático foi usado o *Sonar Qube*. Foi introduzidas todas as dependências indicadas na documentação do *Sonar Qube.*
Assim, de início tinha 1 bug e 144 *code smells*. O bug que aparecia foi de imediato corrigido visto que era uma coisa simples. Consegui reduzir para 38 *code smells*. Estes 38 verificam-se nomeadamente no uso de métodos de outras classes e no uso do *try/catch* que a meu ver não pode ser alterado para a recomendação dada visto que deixaria de funcionar corretamente.

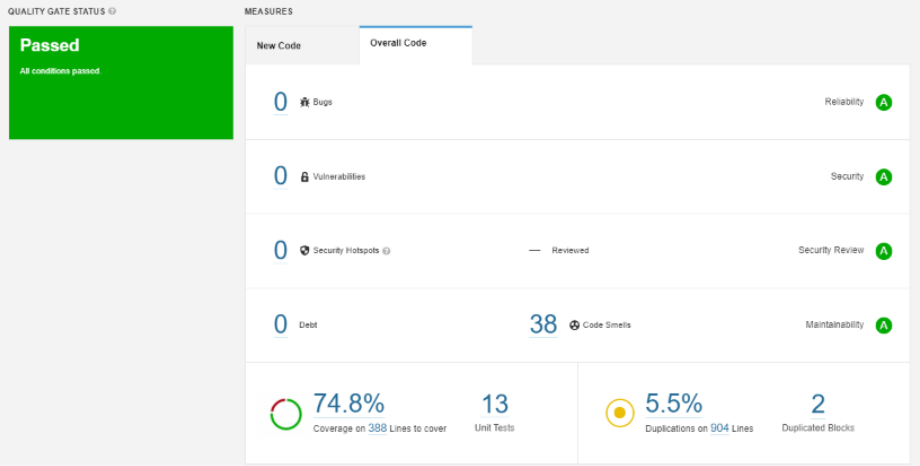

Fig2. Sonar Qube

4 Conclusion

A realização deste projeto permitiu-me rever e aplicar metodologias importantes aprendidas nas aulas práticas. O pensamento da realização dos testes deve ser feito primeiro antes do desenvolvimento. Na prática verifiquei isso visto que primeiro desenvolvi e só depois realizei os testes o que é muito mais complicado.
A realização dos diferentes tipos de testes num desenvolvimento realizado de raiz foi muito importante visto que me permitiram identificar e corrigir determinados erros no desenvolvimento e verificar se todos os requisitos foram implementados corretamente.
Nomeadamente os testes unitários em que verifiquei a funcionalidade de pedaços de código, testes de integração em que verifiquei se a integração correta dos diferentes componentes interagem com as suas interfaces, testes funcionais em que verifiquei a validade dos requisitos e regras de negócio.
Concluindo consegui aprofundar e pôr em prática a matéria aprendida e usada nas atividades laboratoriais num desenvolvimento realizado de raiz.

Project resources

- Video demo presente no repositório;
  Projeto: https://github.com/alex-pt01/tqs_portfolio/tree/main/Homework

Reference materials

JUnit5 documentação
https://junit.org/junit5/docs/current/user-guide/#running-tests

JUnit Cheat Sheet
https://www.jrebel.com/blog/junit-cheat-sheet

Mockito documentation
https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#1

Mockito tutorial
https://javacodehouse.com/blog/mockito-tutorial/

Selenium IDE
https://www.selenium.dev/selenium-ide/

ChromeDriver
https://sites.google.com/a/chromium.org/chromedriver/

Selenium-Jupiter
https://bonigarcia.github.io/selenium-jupiter/

Cucumber
https://cucumber.io/docs

SonarQube
https://docs.sonarqube.org/latest/setup/get-started-2-minutes/