

HW1: Mid-term assignment report

Luís Carlos Duarte Fonseca [89066], v2020-04-15

<https://github.com/luis99fonseca/TQS-first-practical-project->

1	Introduction	1
2	The Proposal	1
3	Technologies Stack	2
4	Implemented Tests	2
5	Quality metrics and Other Points	2
6	Overview of the work	2
7	Limitations	3
8	Product specification	3
9	Functional scope and supported interactions	3
10	System architecture	4
11	17.2.1 Software Ar	4
12	API for developers	4
13	Quality assurance	4
14	Overall strategy for testing	4
15	Unit and integration testing	4
16	Functional testing	4
17	Static code analysis	4
18	Continuous integration pipeline [optional]	5
19	References & resources	5

20 Introduction

20.1 The Proposal

O presente relatório tem como objetivo dar a conhecer o trabalho que foi feito no seguimento do primeiro projeto proposto na cadeira de TQS do presente ano letivo.

Neste contexto, foi sugerido o desenvolvimento de uma **multi-layer web application**, em Spring Boot provida de testes automáticos a vários níveis. Esta proposta enquadra-se assim com o plano de estudos da cadeira, na medida em que aquando da realização do mesmo, nos encontramos providos do conhecimento necessário a dita tarefa.

Era portanto, requerida a inclusão de 3 componentes principais:

- uma **web page** - minimalista, mas que permita aos utilizadores seleccionar uma localização e ver os respetivos valores de qualidade do ar;

- **integração com recurso externo** - permitindo a procura e uso de dados reais. Estes devem posteriormente ser copiados e guardados localmente recorrendo a um sistema de *cache*. A *cache* devem então não só servir como meio de acelerar o processo de pesquisa como também registar alguns dados estatísticos relativamente à sua operação, dos quais se pode apontar o número de hits/misses
- **criação de API (REST) própria** - de forma a que possa ser utilizada por clientes externos;

20.2 Technologies Stack

Como referido, a solução encontra-se implementada em Spring Boot no nível de serviços/*backend*. Havendo liberdade na escolha da ferramenta de *frontend*, optou-se pelo uso de Thymeleaf tendo em conta a facilidade de integração com o Spring Boot e o facto de não ser requerido uma plataforma muito complexa.

20.3 Implemented Tests

Na perspetiva de tudo o que foi lecionado e de todo o conhecimento adquirido, como referido, foi possível realizar aos mais variados níveis:

- **testes unitários** - bastante simples, com o intuito de verificar o bom funcionamento de componentes simples e de forma independente;
- **testes a nível de serviço** - um pouco mais complexos. uma vez que já requerem o uso de *Mocks* e isolamento relativamente a recursos externos;
- **testes de integração** - realizados das mais diversas formas, procurando confirmar o bom comportamento das componentes quando interagindo num ambiente mais próximo do real. Como se falará mais à frente, são testes mais custosos;
- **testes funcionais** - realizados no nível mais alto da aplicação, são o culminar de tudo o que foi revisto anteriormente, e são portanto a última barreira de testes a ser derrubada;

20.4 Quality metrics and Other Points

Fez-se uso de **métricas qualitativas** - recorrendo à ferramenta Sonar Qube, que fazendo verificação de código estático, permitia ganhar mais confiança na qualidade e *maintainability* do código gerado.

Como forma de tornar a plataforma mais *reliable*, fez-se uso de **2 APIs** externas aquando da procura de dados por cidade. Tendo em conta que uma delas necessitava de dados mais específicos - coordenadas do local - necessitou-se de uma outra API de forma a conseguir obter estes valores necessários, totalizando assim um uso total de **3 APIs**.

Finalmente, o projeto faz uso de uma ferramenta que suporta *Continuous Integration* - gitHub - na qual é importante referir não possuía muita experiência. Mais sobre isto à frente.

20.5 Overview of the work

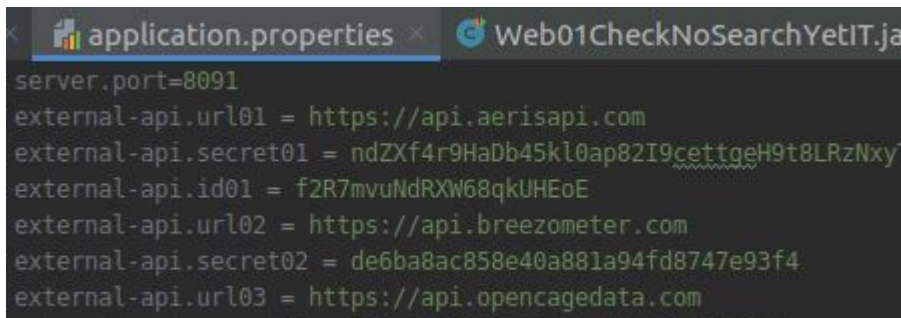
Desta forma, foi desenvolvida uma aplicação de seu nome "**Air Quality in Portuguese Cities**" ou **AQPC**. Esta permite a procura de dados gerais relativos à qualidade do ar em qualquer uma das principais cidades de Portugal, suportadas pelas 2 APIs. Graças à ferramenta de *caching* oferece a garantia de uma rápida e eficiente procura de locais previamente verificados.

Esta procura pode portanto ser feita a partir de uma página Web ou simplesmente recorrendo à API, conferindo portanto uma elevada flexibilidade ao produto. Finalmente, com a utilização de padrões de teste como os referidos anteriormente é possível garantir um produto de alta qualidade e confiança

20.6 Limitations

Não obstante, no contexto do projeto em si, existem algumas limitações, tanto deliberadas como inevitáveis:

- **não conversão de valores em $\mu\text{g}/\text{m}^3$** - uma vez que a 2a API não usa unidades fixas para a qualidade e tendo em conta que não se apresenta como uma prioridade, decidiu-se apenas apresentar as unidades nas quais os valores são conseguidos;
- **só cidades portuguesas** - mais uma vez, no scope do projeto considerou-se que não existia necessidade de expandir o alcance geográfico para além do nacional. Não obstante, uma eventual expansão não seria difícil de implementar;
- **chaves limitadas** - fez-se uso de chaves grátis/trial das APIs externas. Sendo 3 APIs distintas, estas têm portanto chaves diferentes e por consequência prazos diferentes. Desta forma recomenda-se a inserção das chaves do utilizador no ficheiro *application.properties*.



```
server.port=8091
external-api.url01 = https://api.aerisapi.com
external-api.secret01 = ndZXf4r9HaDb45kl0ap82I9cettgeH9t8LRzNxyT
external-api.id01 = f2R7mvuNdRXW68qkUHEoE
external-api.url02 = https://api.breezometer.com
external-api.secret02 = de6ba8ac858e40a881a94fd8747e93f4
external-api.url03 = https://api.opencagedata.com
```

Como podemos ver a partir da 2a linha, a informação encontra-se organizada praticamente como um conjunto de url-segredos. Desta forma, não só é aqui possível ver quais os endereços das APIs a ser usadas, como as respetivas chaves, que podem - e devem - ser substituídas. A aplicação encarrega-se posteriormente e sem qualquer configuração adicional de usar estes valores quando necessário, evidenciando assim um pouco do seu lado de *maintainability*.

Note-se ainda, o uso do *port* 8091 para hospedar a aplicação. A escolha desta porta foi completamente arbitrária, tendo contudo o intuito de evitar algum conflito com outras aplicações - que normalmente se encontram no port 8000.

21 Product specification

21.1 Functional scope and supported interactions

Para utilizar esta aplicação não existe qualquer restrição, o que noutras palavras indica que qualquer utilizador que pretenda obter informações sobre a qualidade do ar em Portugal, pode utilizar esta aplicação esperando sucesso em tal ação.

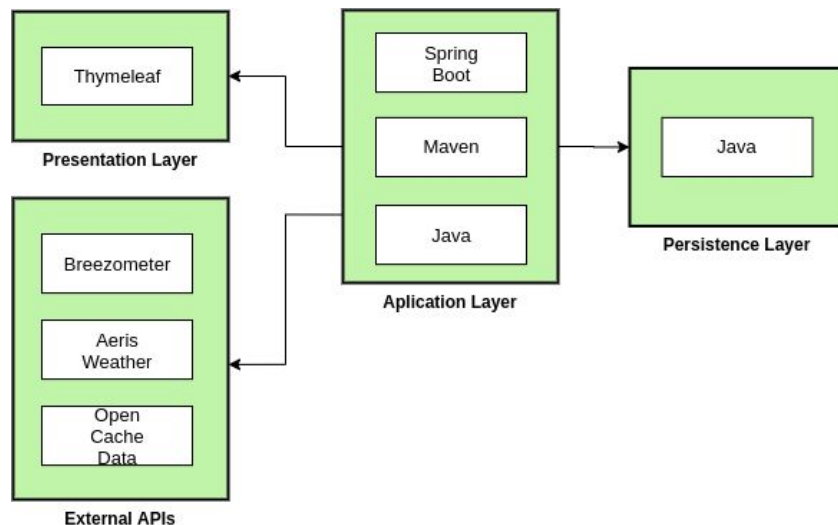
A pesquisa requer apenas a inserção do nome de uma cidade portuguesa. Tudo o resto é tratado “debaixo dos panos” pela aplicação: conversão do nome inserido para coordenadas geográficas, falhas nas APIs externas, etc. No final o utilizador recebe apenas os dados que deseja.

Tendo em conta o facto de que pode ser considerada uma API por si mesma, esta pode facilmente ser integrada noutros projetos como um recurso externo. Devido à simples “interface” que possui, apresenta-se como uma API bastante simples e rápida de trabalhar com.

Por outro lado, ao possuir uma página Web, permite uma utilização mais casual por clientes que apenas desejam ver o estado do ar - antes de sair de casa, por exemplo.

21.2 System architecture

Software Architecture



Aqui é apresentado o diagrama da arquitetura de software de forma simplificada. Como se pode observar, esta é constituída:

- **camada de persistência** - a cache usada nesta aplicação. Observe-se que embora tenha uma pequena parte constituída por Spring Boot, decidiu-se omitir a mesma, uma vez que a sua aplicação foi apenas com o pretexto de permitir compatibilidade com a *layer* adjacente. Desta forma, reforça-se assim que a mesma está praticamente implementada apenas em Java.
- **application layer** - a “cérebro” da aplicação e onde toda a maquinação é feita. Assenta sobre os três pilares representados e é portanto o *core* de toda a aplicação. Spring Boot, a principal componente da mesma, e que permite abstrair bastantes conceitos e implementações ao desenvolvedor, facilitando imenso o processo. Maven, que permite facilmente gerir o projeto, através de goals definidos. Finalmente Java como a linguagem que suporta as componentes anteriores.
- **External APIS** - as APIs usadas no projeto.
- **presentation layer** - a camada com a qual o utilizador pode interagir através da web. Como referido anteriormente, está implementada recorrendo a Thymeleaf, o que permite fácil integração com Spring Boot.

Class Diagram

(devido ao grande tamanho, encontra-se em reportfiles/classDia.png)

No mesmo sentido, apenas há a referir o facto de se ter usado o padrão Null Object para evitar erros no lado do utilizador.

Além disso, existem algumas classes POJO com o propósito apenas de servir de container para as mensagens recebidas das APIs externas.

21.3 API for developers

```
@GetMapping("/airquality/{city}")
@ResponseBody
public AirQuality getAirQuality(@PathVariable String city) { return airQualityService.getAirQuality(city); }

@GetMapping("/airquality/stats")
@ResponseBody
public List<Integer> getCacheStats() { return airQualityService.getStats(); }

@GetMapping(value = "/index")
public String frontPage(Model model){...}
```

Aqui se encontra sumariada API desenvolvida. Note-se que existem dois métodos (os primeiros) que se enquadram no contexto da API - daí a tag `@ResponseBody`. Enquanto que o último parâmetro é relativo à página Web.

22 Quality assurance

22.1 Overall strategy for testing

A aproximação inicial ao projeto foi feita recorrendo à estratégia de TDD. Foi interessante começar um projeto verdadeiramente do zero com este método, pois é uma experiência bastante diferente da que se tem durante guiões práticos - realçando-se assim a importância e mais valias do presente trabalho - porque surgem questões e dúvidas que não existem noutros momentos, e que exigem algum trabalho de pesquisa autónomo.

Não obstante, muito culposamente devido ao fraco planeamento, este método não prevaleceu a partir de certo ponto inicial do projeto. Uma vez que o planeamento foi quase não existente, os testes que foram feitos acabaram por não se tornar úteis no que diz respeito à verificação de código, uma vez que estavam preparados para lidar com uma arquitetura que acabou por não se tornar desejável. Não obstante, mostraram a sua utilidade no sentido em que revelaram esta referida falha, e permitiram uma melhor planificação do sistema. Desta forma, após uma reformulação do sistema, estes testes foram refeitos com o paradigma de TDD em mente.

Note-se que esta situação veio a revelar-se uma mais valia, uma vez que alterou positivamente a aproximação ao problema.

Embora não relacionado com testing, a título de exemplo meramente, pode referir-se o facto de que para chamadas a APIs externas, não se ter usado *RestTemplate* em prol de *WebClient*, uma vez que a primeira implementação está prestes a ser descontinuada; ou que para obter os valores para as APIs em *application.properties*, se optou por criar uma *class* própria para essa gestão em vez de simplesmente usar a tag `@value`, procurando seguir boas práticas. Realça-se assim o planeamento e trabalho de pesquisa feito para assegurar que situações como a referida anteriormente não se repetissem.

Além disso, e reforçando a importância do projeto em si, houveram alguns testes mais avançados em que não se fez uso de TDD. Isto porque devido à complexidade dos mesmos, era difícil antever o que deveria acontecer para testar o comportamento dos mesmos. Omitindo estas situações pontuais, os testes foram geralmente feitos antes do código, o que se tornou uma mais valia imensa. Foi mitigado o “medo” de testes falharem, porque inicialmente todos naturalmente o faziam, sendo posteriormente desenvolvida uma solução para que passassem.

Sendo que o uso de testes aplicado ao desenvolvimento de código nesta escala é uma experiência nova, optou-se por limitar o uso, como referido a TDD, fazendo uso de ferramentas como **Mockito**, **JUnit**, **Selenium** e **Spring Boot** testing facilities, e descartando portanto **Cucumber** e por consequência a aproximação BDD.

22.2 Unit and integration testing

No que diz respeito a **unit testing** algo interessante e surpreendente no início destes, foi um conceito que ainda não havia sido completamente consolidado: testes unitários devem ser bastante rápidos de forma a que possam ser corridos a qualquer momento com facilidade. Acontece que devido à inexperiência e falta de conforto prior à execução do projeto no que diz respeito a Spring Boot, inicialmente estes foram escritos aplicando *tags* de Spring Boot. Rapidamente isto revelou-se um problema grave, pois estava a alocar recursos usados tipicamente em testes de integração sem qualquer necessidade para tal, o se traduzia em testes simples relativamente demorados. Assim, ao corrigir-se este erro, passou-se ao uso apenas de JUnit e Mockito, o que traduziu num aumento significativo da velocidade de testing.

```
@Test
public void getStats_returnsStats(){
    List<Integer> integerList = new ArrayList<>(Arrays.asList(1,2,3));
    given (airQualityRepository.getStats()).willReturn(
        integerList
    );

    Assertions.assertThat(airQualityRepository.getStats()).containsExactly(1,2,3);
}
```

No exemplo acima, encontra-se um exemplo de um dos testes mais simples presentes neste projeto. Repare-se como se faz uso a um **Mock** de `airQualityRepository` de forma a definir e isolar o comportamento do mesmo neste contexto, sem ter de ter essa *class* ainda configurada, evidenciando as vantagens desta aproximação TDD.

Adicionalmente, surgiu a aparente necessidade de usar um **Spy**, com o intuito de só definir parcialmente o comportamento de uma class. Não obstante esta ideia foi rapidamente descartada, devido de não ser algo recomendável, uma vez que na maioria dos casos não faz sentido “mockar” apenas parte de uma class.

Neste paradigma apercebeu-se também de algo interessante: não é “possível testar tudo”- diga-se num projeto de maior dimensão. Isto porque há sempre situações as quais não se imaginam ou simplesmente não faz sentido testar certas coisas, como por exemplo classes idênticas a POJOs, sem métodos reais. Desta forma, conclui-se que testes não tornam o código a prova de falha, apenas aumentam consistentemente a robustez do mesmo.

Como último ponto há que referir que o sistema de caching não implementa um método de ttl baseado em tempo em si. Este tem uma capacidade máxima, a qual ao ser excedida fará com que o registo mais antigo seja eliminado. A razão para referir isto passa pelo facto em como se percebeu um pouco como testes com sensibilidade ao tempo devem ser tratados. Embora se tenha encontrado uma solução bastante mais simples, inicialmente tentou-se dar override no tempo do sistema só para testar o código, mas rapidamente se sentiu que não era uma opção inteligente. Adicionalmente é importante referir como embora o *approach* que se fez na cache tenho divergido ligeiramente, sente-se que o conceito de tempo ainda está presente e foi trabalhado na mesma. Apenas se escolheu deste modo uma vez que os dados das APIs externas demoram bastante tempo a atualizar - períodos de cerca de 1 hora - pelo que se considerou esta uma maneira melhor de gerir os dados. No que diz respeito a **integration tests**, estes obviamente revelaram-se mais complexos. e exigiram também um maior domínio de Spring Boot, para os mesmos.

```
@Test
public void whenGetAirQByInexistingCity_thenReturnNullAirQ() throws Exception {
    String no_existing_city = "no_city";

    mvc.perform(get( uriTemplate: "{no_existing_city}" ).contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.city").value( expectedValue: "unavailable"))
        .andExpect(jsonPath("$.pollutants").isArray())
    ;
}
```

Uma vez que necessitam de mais recursos do sistema, aquando do CI, foram colocados num workflow distinto - como veremos adiante. Estes testes permitiram testar o funcionamento de partes diferentes do programa interligadas, e encontrar erros não existentes nos testes unitários.

22.3 Functional testing

Para **testes funcionais** usou-se a ferramenta Selenium. neste sentido é importante referir o facto de que mesmo o código gerado pela mesma deve ser verificado, uma vez que nem sempre é robusto como desejamos, ou pode gerar erros desnecessários.

```
@Test
public void test02CheckSearchResult() throws Exception {
    driver.get("http://localhost:" + randomServerPort + "/");
    driver.findElement(By.id("name")).clear();
    driver.findElement(By.id("name")).sendKeys("ViSeU");
    driver.findElement(By.xpath("//input[@value='Go!']")).click();
    driver.findElement(By.xpath("//div/h4")).click();
    Assertions.assertThat(driver.findElement(By.xpath("//div/h4")).getText()).contains("for the city of viseu");
    // assertEquals("Taken at: 2020-04-08T16:00, for the city of viseu", driver.findElement(By.xpath("//div/h4")).getText());
}
```

Observando a imagem acima, num comentário posteriormente removido, o teste não era robusto o suficiente, não garantindo flexibilidade em função do tempo. Tal foi corrigido como se pode verificar. A driver usada permite definir tempos de espera, a partir dos quais se considera o teste falhado. Obviamente isto pode criar falsos negativos, no sentido em que não há nenhum erro, é apenas lentidão na rede, contudo são casos pontuais que devem ser analisados com cuidados. Adicionalmente é importante referir que estes testes tiveram de ser configurado para ser *headless* de forma a funcionar no contexto de CI.

22.4 Static code analysis

Usando a ferramenta SonarQube, estes testes foram feitos localmente, uma vez que para os fazer num contexto de CI, o repositório teria de ser público, algo que não se optou se forma a garantir a integridade do projeto.

Foi contudo interessante perceber o funcionamento desta ferramenta, e o facto de como esta apontava *smells* como *System.out.println()* como *logs*, comentários desnecessários, e até TODOS esquecidos. Não obstante, não é 1 ferramenta à “prova de bala”, no sentido em que a mesma deteta falsos positivos, dos quais se pode destacar variáveis de POJOs “não utilizadas”, quando na verdade o são indiretamente.

Esquecendo estes casos “pontuais”, considerou-se uma ótima ferramenta para manter um código limpo e organizado, conferindo mais *maintainability*. Ao detetar bugs e até falhas de segurança (como por exemplo a não verificação de parâmetros na main), revelou-se como uma mais valia no desenvolvimento do código.

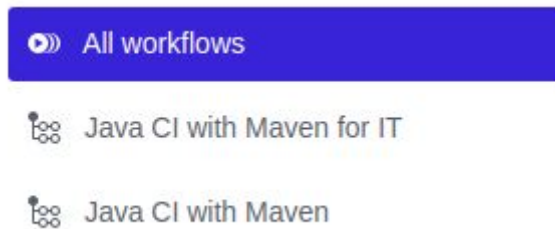
Finalmente, as medições da ferramenta (coverage/security/etc) não foram todas maximizadas, devido ao facto de que muitas vezes não estavam de acordo com o desejado.

22.5 Continuous integration pipeline

A nível de CI, este foi feito recorrendo ao GitHub Actions. Tal como os testes não possuía muita experiência prévia com os mesmos, pelo que foi uma experiência nova.

Tendo em conta que Actions também é 1 ferramenta nova, esta possui bastantes templates para ajudar “noobs” a desenvolver os primeiros workflows. Desta forma, foram adotados e adaptados templates Maven existentes a este projeto, de forma atingir os objetivos pretendidos.

Desenvolveram-se assim 2 workflows principais: um para testes unitários e outro para de integração.



No workflow Unitário, apenas os testes unitários eram executados, enquanto que no de IT ambos eram executados - uma vez que os unitários são rapidíssimos.

```
.github/workflows/maven-it.yml
1  # https://help.github.com/pt/actions/reference/events-that-trigger-workflows
2  # https://crontab.guru/
3  # Note: Had to be created on GitHub platform, else wasn't working, apa
4
5  name: Java CI with Maven for IT
6
7  on:
8    schedule:
9      - cron: '0 10,15,17,20,23,01 * * *'
10   push:
11     branches: [ master , noRepository]
12   pull_request:
13     branches: [ master , noRepository]
14
15   jobs:
16     i-test:
17
18       runs-on: ubuntu-latest
19
20       steps:
21         - uses: actions/checkout@v2
22         - name: Set up JDK 1.8
23           uses: actions/setup-java@v1
24           with:
25             java-version: 1.8
26         - name: Integration Testing
27           run: cd tqst01proj/ && mvn package failsafe:integration-test
28
```

Neste contexto de experimentar, e sabendo que testes IT não devem ser corridos sempre, apenas algumas vezes, devido ao custo computacional, criou-se adicionalmente um horário no qual os mesmos deviam ser executados, visando a criar uma experiência mais realista.

23 References & resources

Project resources

- Git repository: <https://github.com/luis99fonseca/TQS-first-practical-project->
- Video demo (no repositório)

Reference materials

<https://www.baeldung.com/configuration-properties-in-spring-boot/>

<https://stackoverflow.com/a/32066380>

(outros links importantes encontram-se no código, no local onde foram utilizados, para melhor compreensão)