Universidade de Aveiro

# HW1: Mid-term assignment report

Pedro Miguel Nicolau Escaleira *[88821]*

`https://github.com/oEscal/tqs_project_1`

15 de abril, 2020

# Indice

# Indice de imagens

# 1. Introdução

## 1.1 Contextualização do trabalho

Este projeto, proposto pelo professor da disciplina de **Teste e Qualidade de Software**, teve como principal objetivo a consolidação dos conhecimentos adquiridos durante as aulas da mesma tidas até ao momento.

Desta forma, foi sugerida a criação duma aplicação *web* simples para obtenção de dados sobre a qualidade do ar num dado sitio fornecido. Para isso, a solução criada possui um *back-end* sob a forma de *REST API*, feita em *java* com a ajuda de *Spring Boot* e um *fron-end* feito em *python* com a ajuda de *Flask* e *Jinja 2*. Fazendo jus ao nome da disciplina, claramente toda esta plataforma foi criada com o intuito de serem feitos testes, a vários níveis, para a mesma, pelo que foi, duma forma geral, usado o *JUnit* para a criação de testes para a *api* e *Selenium WebDriver* para a criação de testes para a interface.

## 1.2 Limitações

Apesar do trabalho ter sido concluído com sucesso e de ter ido de encontro aos requisitos pedidos, houveram algumas *features* que ficaram por implementar, mas que teriam sido uma adição que o autor gostaria de ter criado. De seguida, são apresentadas as principais:

- **Pesquisa dum lugar pelo nome**: no resultado final, apenas dá para pesquisar a qualidade de ar quando dadas as coordenadas da localização pretendida. Contudo, seria mais *user friendly* fazer a mesma pesquisa por nome.
- **Utilização doutra *API* remota**: na solução final, apenas é usada um serviço remoto para obtenção dos dados necessários. Contudo, tal como é sugerido nos pontos extra do guião do trabalho, seria mais *reliable* a utilização de mais que um serviço, para o caso do primeiro falhar. Esta aproximação não foi usada já que iria adicionar uma grande quantidade de sobrecarga sobre o trabalho feito dada a dificuldade desta adição quando concluído grande parte do código feito.
- **Testes da interface em que houvesse alteração do código *HTML***: Seria algo de interesse de se fazer testes, por exemplo, sob *inputs* com atributos alterados, para testar o comportamento da plataforma quando não apresentada, por exemplo, uma entrada na forma de número ou a submissão do formulário sem quaisquer *inputs* obrigatórios preenchidos. Contudo, após alguma pesquisa, o *Selenium IDE*, ferramenta usada na criação dos testes da interface, não parece apresentar documentação de como fazer alterações no código fonte da página testada.

# 2. Especificações do produto

## 2.1 Funcionalidades e interações suportadas

Como demonstrado na figura 2.1, esta plataforma é uma aplicação simples que permite aos seus utilizadores obterem a qualidade do ar para um determinado lugar, indicando as coordenadas. Permite não só saber a qualidade atual, mas também a passada e futura. As métricas que ela disponibiliza são a **data referente à qualidade do ar**, o **valor escalar e textual da qualidade do ar**, o **poluente dominante** e a **concentração e valor escalar e textual da qualidade do ar referente a cada um dos poluentes principais**.

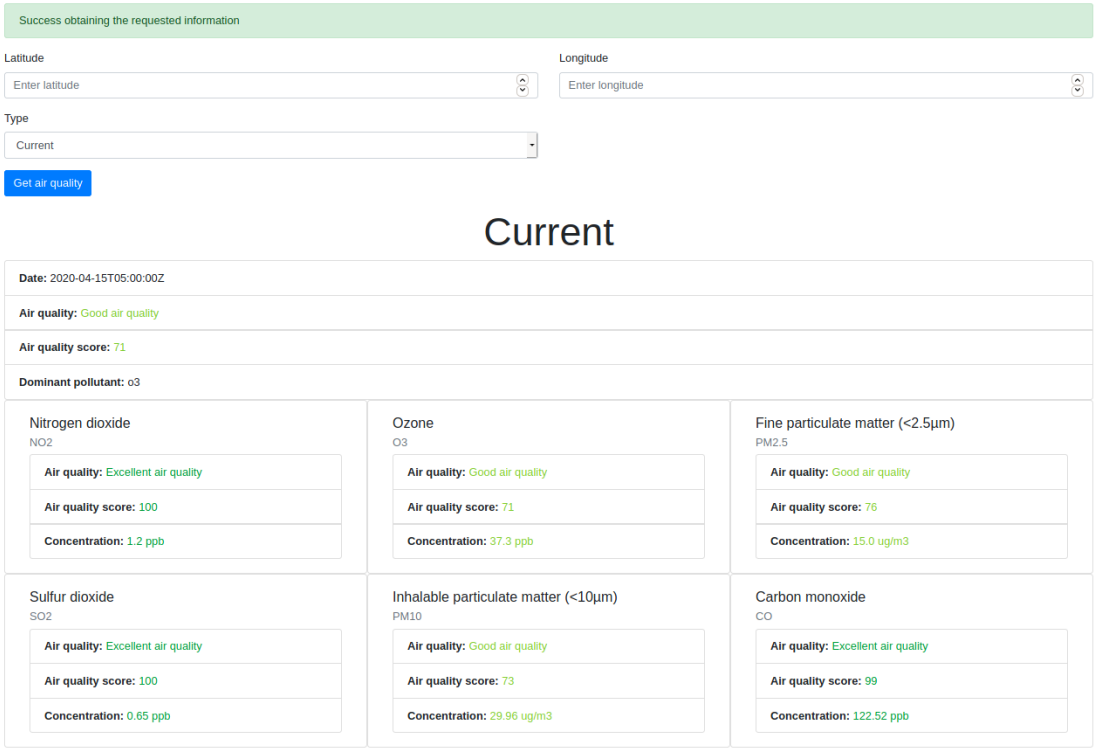


**Figure 2.1:** *Print da interface quando feito um pedido da qualidade do ar atual.*

Os possíveis utilizadores e cenários da plataforma criada são:

- **População de risco**: dado o estado debilitado desta fração de população, é do interesse de algumas saber a qualidade do ar que respiram, principalmente as que possuem problemas respiratórios, de forma a melhor controlarem o seu estado de saúde. Desta forma, uma pessoa

nestas condições poderá dirigir-se à interface desta aplicação, introduzir as coordenadas do local onde se encontra ou se vai encontrar nos próximos tempos, selecionar a obtenção de dados sobre o estado atual (*Type Current*) ou sobre o estado previsto no futuro (*Type Forecast*, selecionando também o número de horas seguintes sobre as quais pretende obter os dados) e, sendo assim, obter o estado da qualidade do ar atual ou nas horas seguintes, respetivamente.

- **Estudiosos**: profissionais que tenham interesse em estudar a qualidade de ar de acordo com o local, como por exemplo o estudo da evolução num determinado lugar. Sendo assim, um utilizador deste tipo pode dirigir-se à página *web*, selecionar um determinado lugar introduzindo as correspondentes coordenadas, se pretende os dados de previsões passadas (*Type History*) ou futuras (*Type Forecast*) e o número de horas de dados deste o momento atual pretende obter. A partir dos resultados obtidos, poderá copiar cada um deles e fazer o correspondente estudo.

## 2.2 Arquitetura do sistema

O *back-end* do projeto foi feito usando *java* com *Maven* e *Spring Boot*. Quanto á arquitetura, é apresentado um diagrama de classes simples da mesma na figura 2.2 (este diagrama de classes apenas contém as classes criadas e as relações entre elas, sendo que os detalhes de cada uma se encontrarão definidos em diagramas expostos em subsecções seguintes, de forma a que este não fique demasiado confuso). Estas classes foram organizadas, de acordo com a sua complexidade em 4 *packages*: **controller**, **model**, **serializers** e **service**. Nas subsecções seguintes

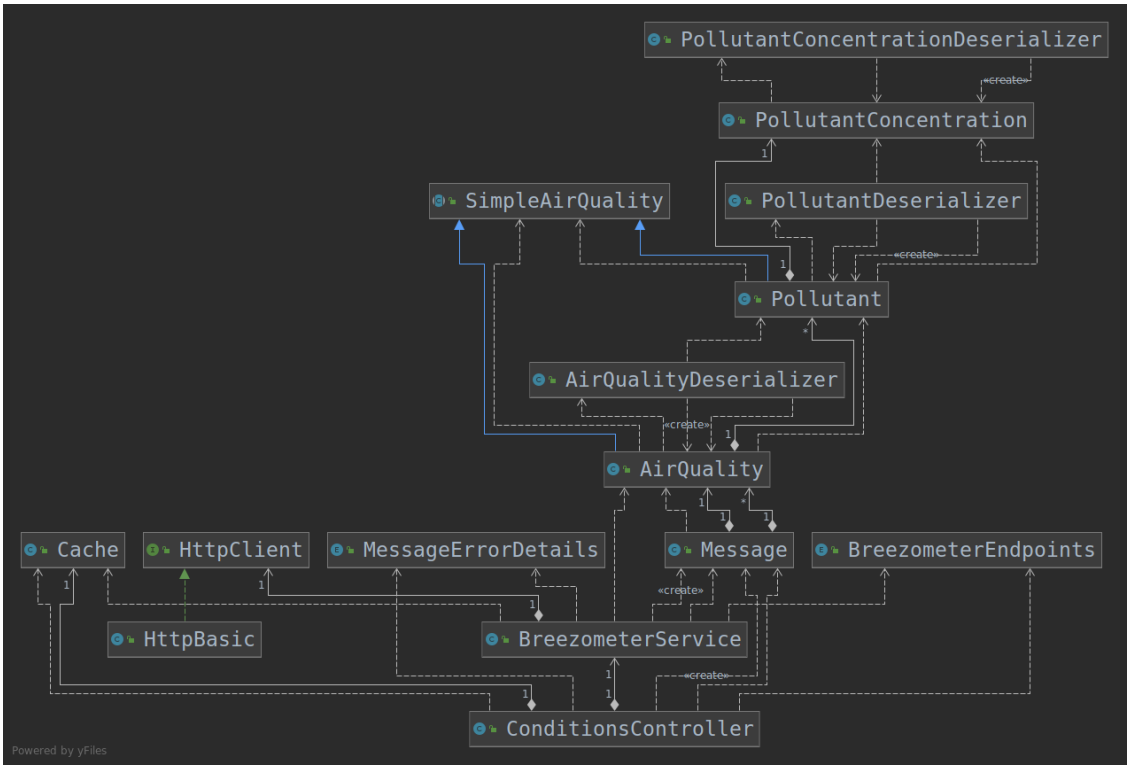


**Figure 2.2:** *Diagrama de classes simples do projeto.*

### 2.2.1 Package controller

Este *package* é constituído apenas por uma classe, ***ConditionsController***, que é a responsável por lidar com as ligações feitas à *API* criada. Desta forma, nesta classes são definidos os *endpoints* do nosso serviço e é definida a forma como cada *request* vai ser consumido no *back-end*. Na figura 2.3 é possível encontrar o esquema da classe descrita.

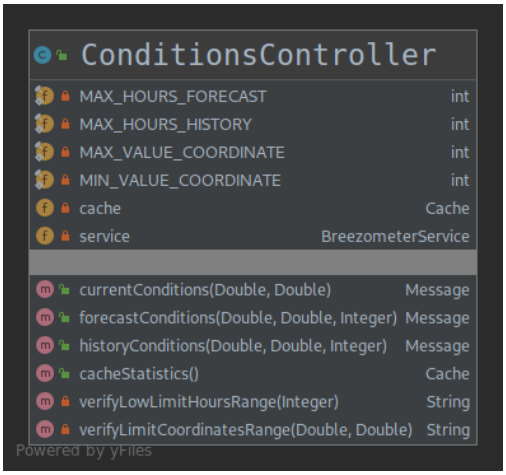


**Figure 2.3:** *Diagrama das classes do package* ***controller****.*

### 2.2.2 Package model

Todas as entidades usadas pelo nosso servido são representadas pelas classes deste *package*. Desta forma, nela estão incluídas as classes:

- **Que representam os dados da qualidade do ar**:
  - ***AirQuality***: representa a qualidade do ar referente a um determinado momento. Sendo assim, para além dos objetos desta classe possuírem informações básicas sobre a qualidade do ar, contêm também uma lista de poluentes (classe ***Pollutant***) que se encontram no ar no instante que esse objeto retrata.
  - ***Pollutant***: classe que contém informações sobre um determinado poluente, como a forma como o respetivo poluente afeta a qualidade do ar e a concentração deste num determinado momento (classe ***PollutantConcentration***).
  - ***PollutantConcentration***: classe que representa a informação da concentração dum poluente.
- **Que modelam o sistema de cache**:
  - ***Cache***: classe que permite criar objetos que simulam um sistema de cache simples e com as respetivas métricas (como o tamanho da cache ou o número de *hits* e *misses*). Por *default*, os objetos de cache criados possuem um tamanho máximo determinado pelo valor da constante *DEFAULT_MAX_SIZE*, sendo que quando o tamanho dela chega a esse tamanho máximo, os dados mais antigos são excluídos.
  - ***ParametersEncapsulation***: classe que permite fazer o encapsulamento dos parâmetros sobre os quais se pretende armazenar a resposta dada pelo serviço externo na cache, isto

é, quando é feito um pedido da qualidade do ar ao serviço externo, com uma determinada latitude, longidute, tipo de resposta (*current*, *history* ou *forecast*) e possivelmente um número de horas, o resultado deste pedido pode ser armazenado na cache com um identificador representado pelo encapsulamento destes parâmetros.

- **Ligadas às mensagens criadas pelo serviço**:
  - ***Message***: classe que permite encapsular a resposta dada pela *API* a um determinado pedido feito. Desta forma, ela tem um campo de sucesso, um de detalhes (para mensagens de sucesso ou erro), um da qualidade de ar (quando é feito um pedido da atual) e de uma lista de várias medições da qualidade do ar (quando é feito um pedido sobre os dados do passado ou futuro).
  - ***MessageErrorDetails***: enumerável com as várias mensagens de erro que podem ser devolvidas pela mensagem enviada pela *API*.

Na figura 2.4 é possível verificar a constituição e relações entre cada uma destas classes.

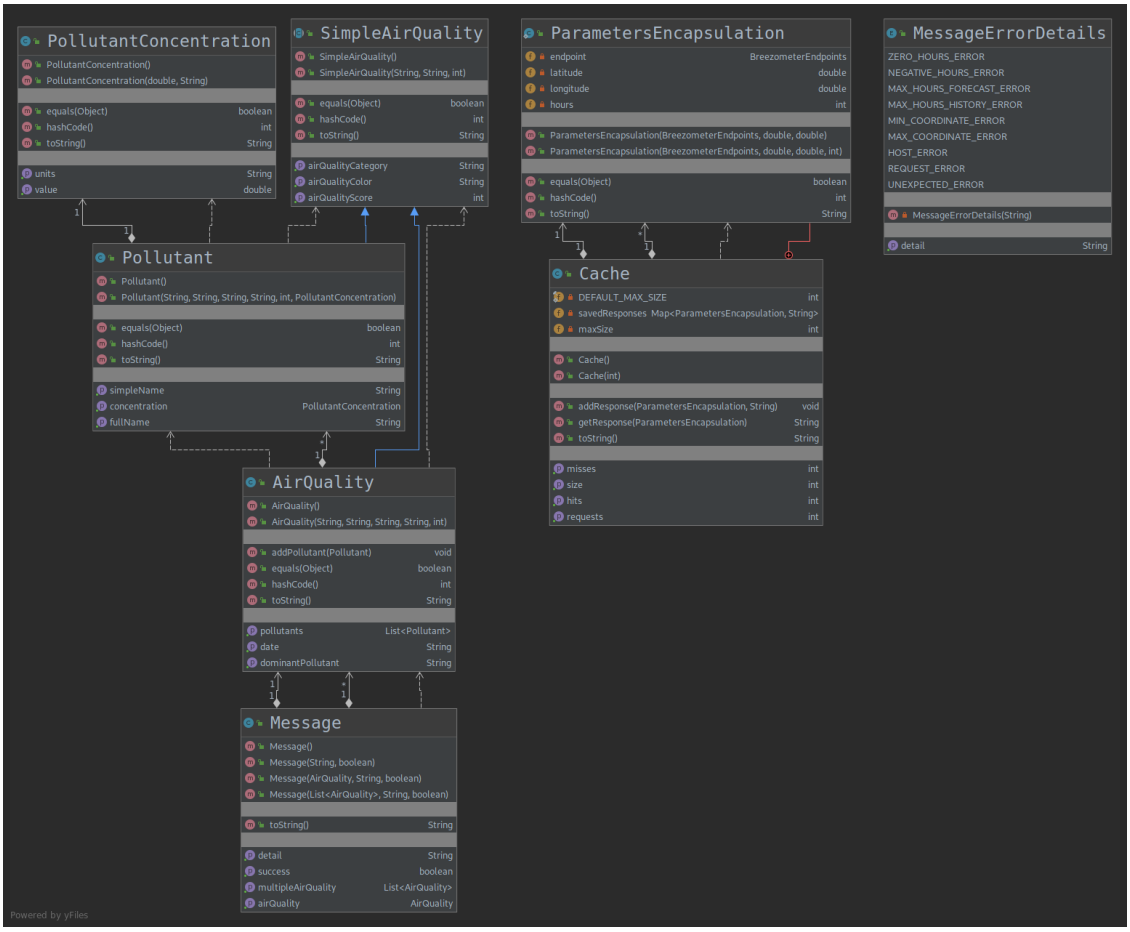


**Figure 2.4:** *Diagrama das classes do package* ***model****.*

### 2.2.3 Package serializers

De forma a poder fazer a "tradução" entre os dados recebidos do serviço externo e algumas das classes do *model*, foram criados alguns *deserializers* para esse efeito, como é possível verificar

no diagrama da figura 2.5. Para isso, foi usada a livraria *Jackson*.

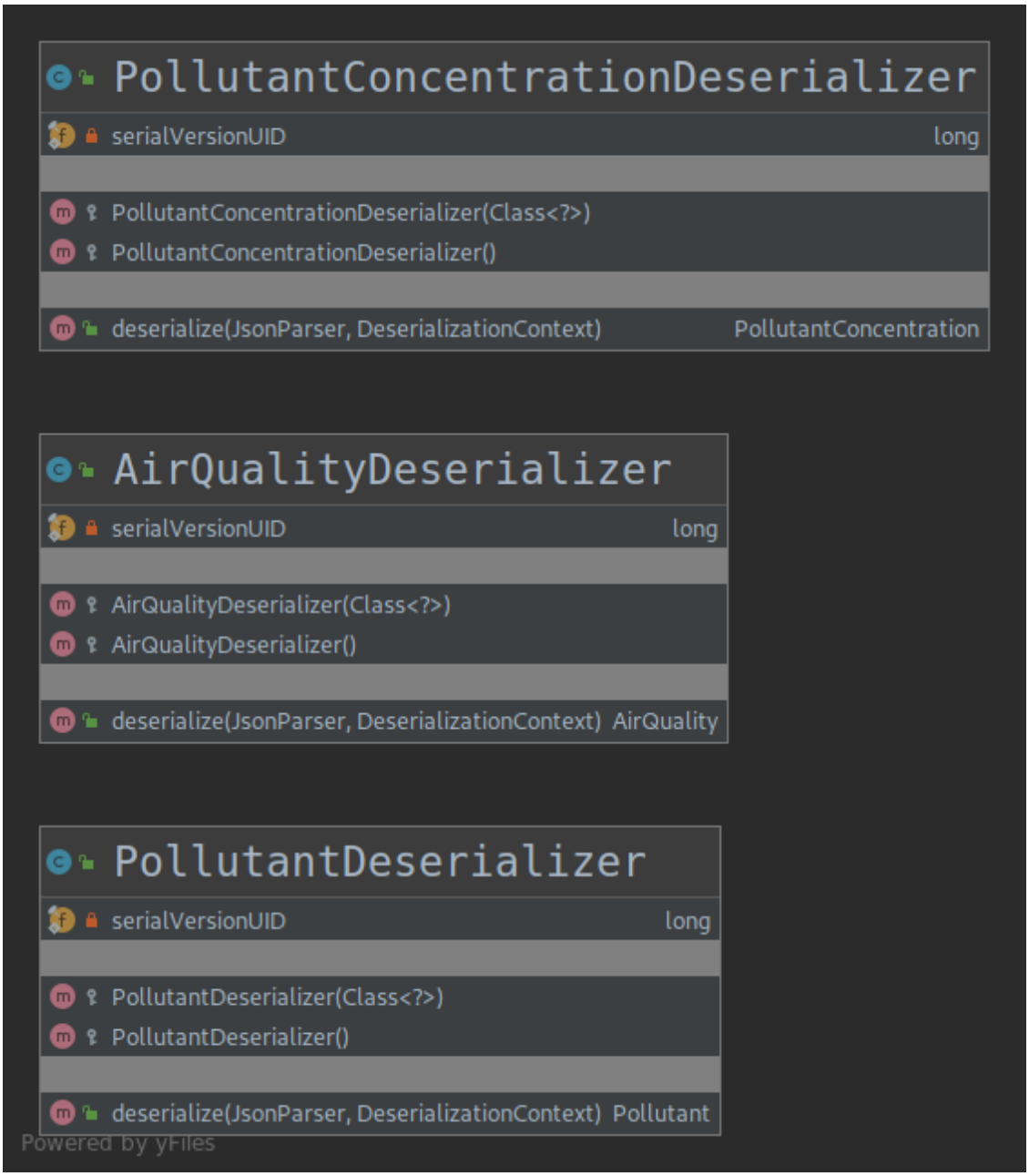


**Figure 2.5:** *Diagrama das classes do package* ***serializers****.*

### 2.2.4 Package service

De maneira a obter os dados necessários sobre a qualidade do ar, foi necessário criar um elo de ligação entre o *back-end* criado e o serviço externo usado (*BreezoMeter*), pelo que neste *package* se encontram as classes que o permitem fazer. Quando a *API* recebe um determinado pedido (excluindo o das estatísticas da cache), o método da classe ***ConditionsController*** responsável por tratar do pedido "chama" o método *requestApi* da classe ***ConditionsController*** deste *package*

com o respetivo pedido, sendo que esta ultima faz um pedido ao serviço externo usando a classe ***HttpBasic*** (disponibilizada pelo professor da disciplina numa das aulas práticas). A resposta do serviço externo é de seguida processada pelos *deserializers* já descritos anteriormente e o objeto ou lista de objetos da classe ***AirQuality*** são retornados ao método do *controller* que tinha feito o pedido, encapsulados sob a forma dum objeto da classe ***Message***. Na figura 2.6 é possível encontrar a estrutura interna das classes deste *package*.

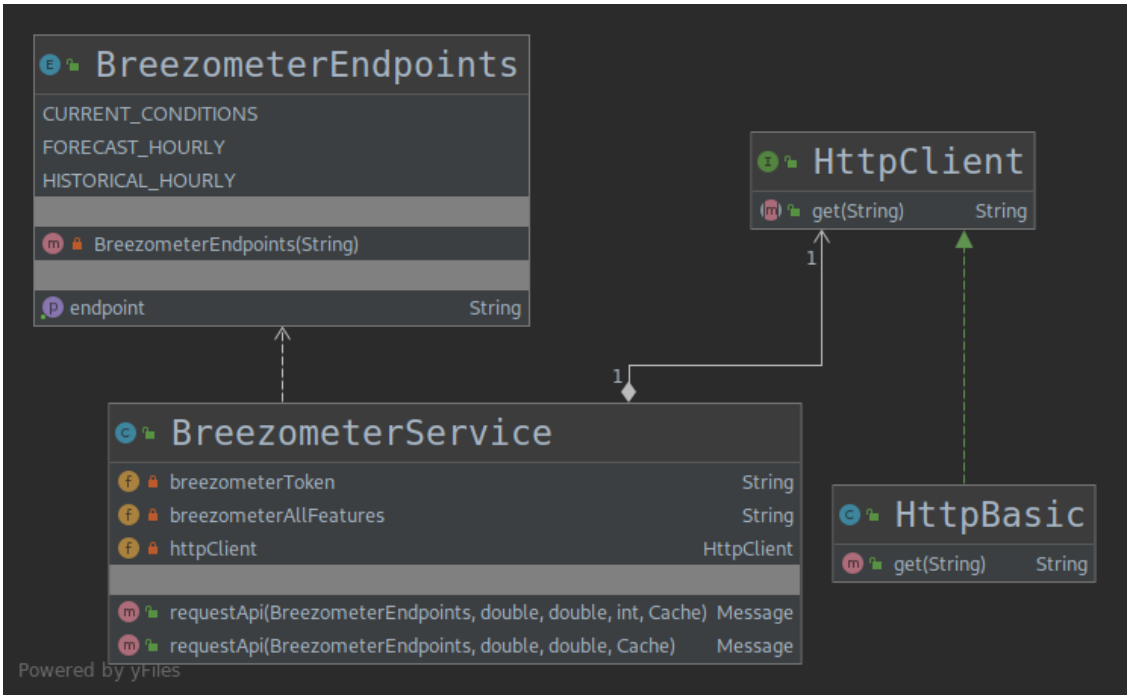


**Figure 2.6:** *Diagrama das classes do package* ***service***.

## 2.3 API para desenvolvedores

Como já explicado múltiplas vezes ao longo deste relatório, o serviço criado permite obter informações sobre a qualidade do ar presente, passada ou futura, sendo que para isso utiliza um serviço externo para obter as informações necessárias. Assim, a *API* criada possui 4 *endpoints*:

- ***/current***:
  - Permite obter a qualidade do ar atual.
  - Parâmetros:
    - ***lat***: número decimal representante da latitude do lugar pretendido.
    - ***lon***: número decimal representante da longitude do lugar pretendido.
- ***/forecast***:
  - Permite obter a previsão da qualidade do ar até 95 horas futuras.
  - Parâmetros:
    - ***lat***: número decimal representante da latitude do lugar pretendido.

    * ***lon***: número decimal representante da longitude do lugar pretendido.
    * ***hours***: número inteiro representante da número de horas pretendidas.

- */history*:
  - Permite obter o histórico da qualidade do ar até 168 horas passadas.
  - Parâmetros:
    * ***lat***: número decimal representante da latitude do lugar pretendido.
    * ***lon***: número decimal representante da longitude do lugar pretendido.
    * ***hours***: número inteiro representante da número de horas pretendidas.

- */cache*:
  - Permite obter as estatísticas da cache usada (*requests*, *hits*, *misses* e *size*).

Para informação mais detalhada sobre cada endpoint, pode ser acedida a página `http://localhost:8080/swagger-ui.html#/` (se a *API* for executada em ***localhost*** e na porta **8080**), criada com a ajuda do *Swagger* (figura 2.7).

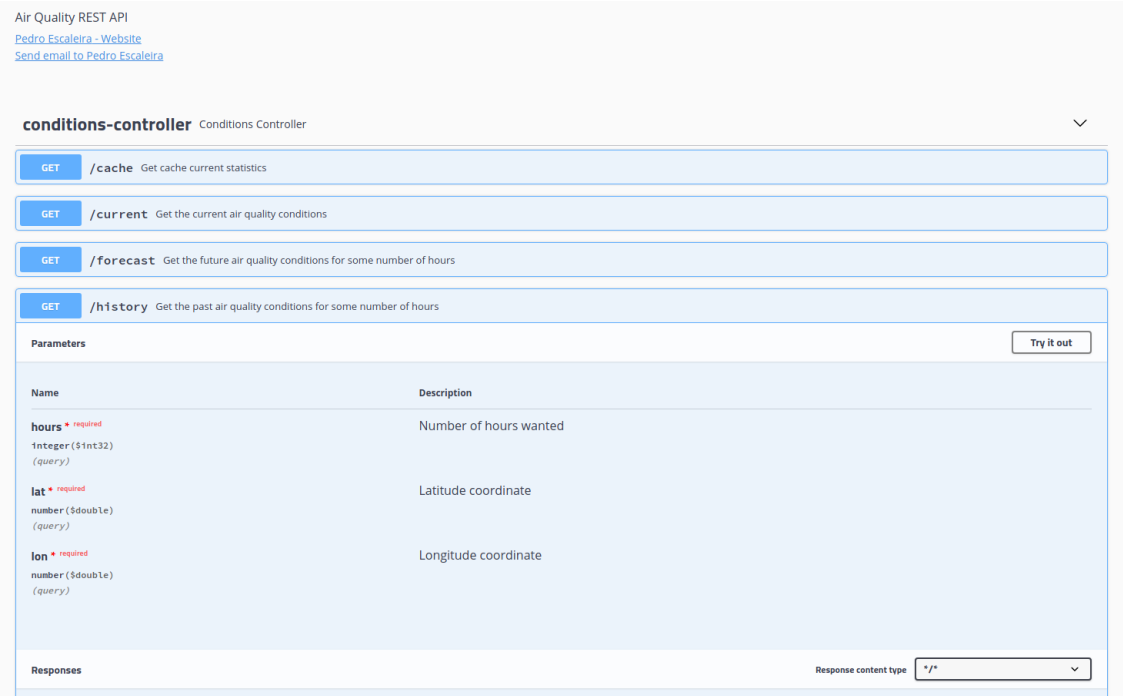


**Figure 2.7:** *Página da documentação da API criada com a ajuda do Swagger.*

# 3. Garantia de qualidade

## 3.1 Estratégia geral usada para os testes

Duma forma geral, a estratégia usada para fazer os testes foi usar *TDD*, *Test Driven Development*, e *TLD*, *Test Last Development*. Foi dado um especial destaque ao uso da primeira, como tentativa de mudar o hábito de fazer testes só depois de fazer todo o código, e aprender uma técnica de desenvolvimento de código aconselhada e que permite criar código fácilmente mantido. Contudo, foi impossível mudar completamente, e num só projeto, este *mindset*. A ultima, apesar de não tão recomendada como a primeira, permite fazer código mais rápidamente e mais simples, numa primeira fase, mas alterações futuras são mais difíceis de serem feitas.

Quando usado *TDD*, usualmente a classe sob a qual era pretendido serem feitos testes unitários foi criada primeiro, com toda a estrutura suposta e, posteriormente, foram criados os respetivos testes. Quando completada a criação destes, todo o código da respetiva classe era analisado, de forma a criar as *features* supostas para que os testes passassem.

Para código mais complexo, foi usado *TLD*, já que seria extremamente difícil de criar testes para algo que não se sabia inicialmente o comportamento. Muitas vezes, o código inicial deste tipo de classes foi feito mais como protótipo, de seguida criados testes para o comportamento que era esperado quando tudo funciona-se devidamente e por ultimo, melhorado o código desses protótipos de forma a criar classes com um comportamento adequado para o produto final.

É de sublinhar que não foi usada nenhuma abordagem *BDD*.

Quanto a ferramentas usadas, para além do *JUnit*, foi usado *Mockito*, de forma a simular o comportamento de certos integrantes, e *MockMvc* do *Spring Boot*, de forma a simular os pedidos à *API* criada e a obter as respetivas respostas.

## 3.2 Testes unitários e de integração