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**.

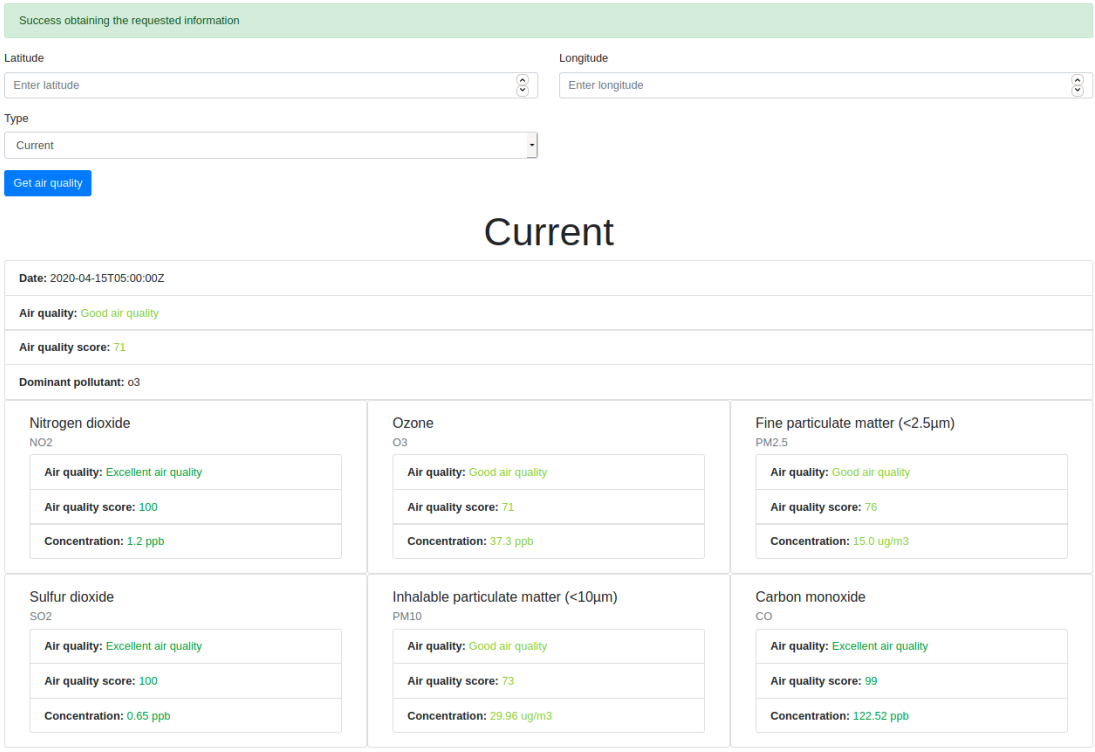

Figure 2.1: *Print da interface quando feito um pedido da qualidade do ar atual.*

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

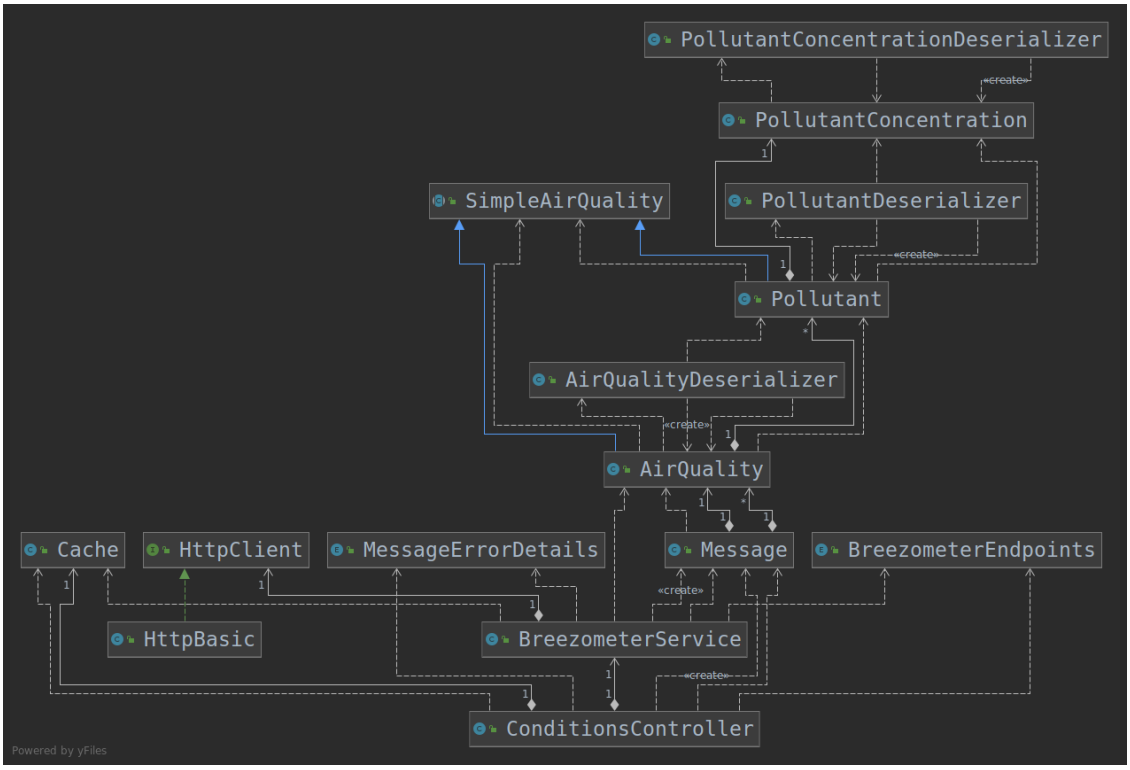

Figure 2.2: *Diagrama de classes simples do projeto.*

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.

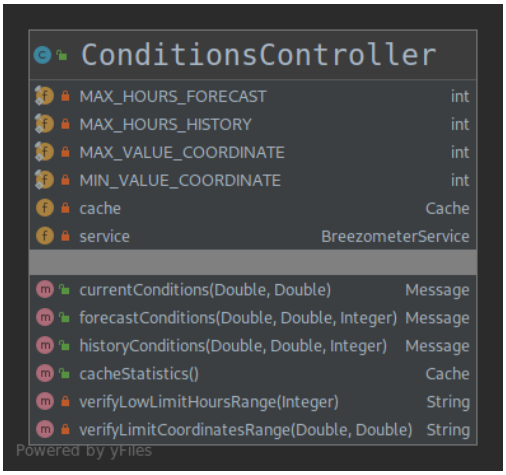

Figure 2.3: *Diagrama das classes do package* ***controller****.*

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.

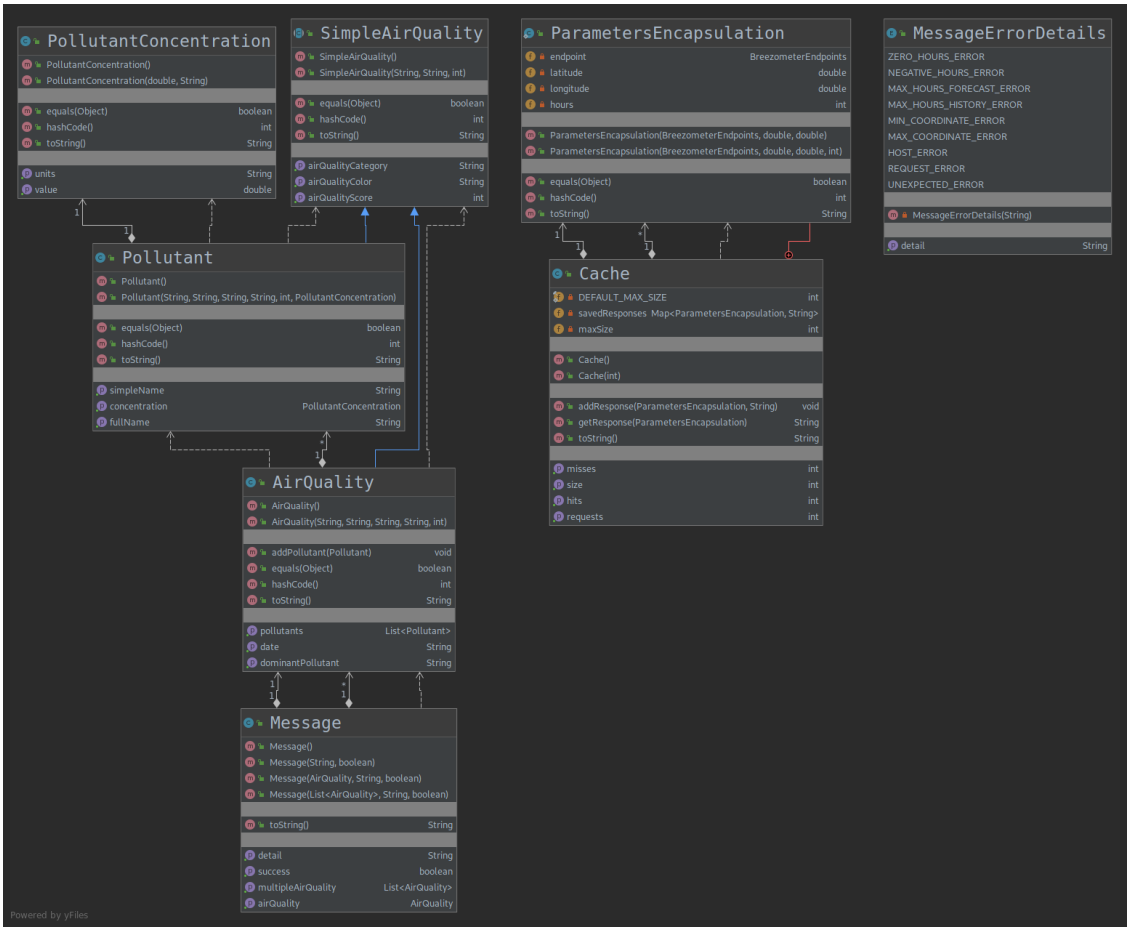

Figure 2.4: *Diagrama das classes do package* ***model****.*

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*.

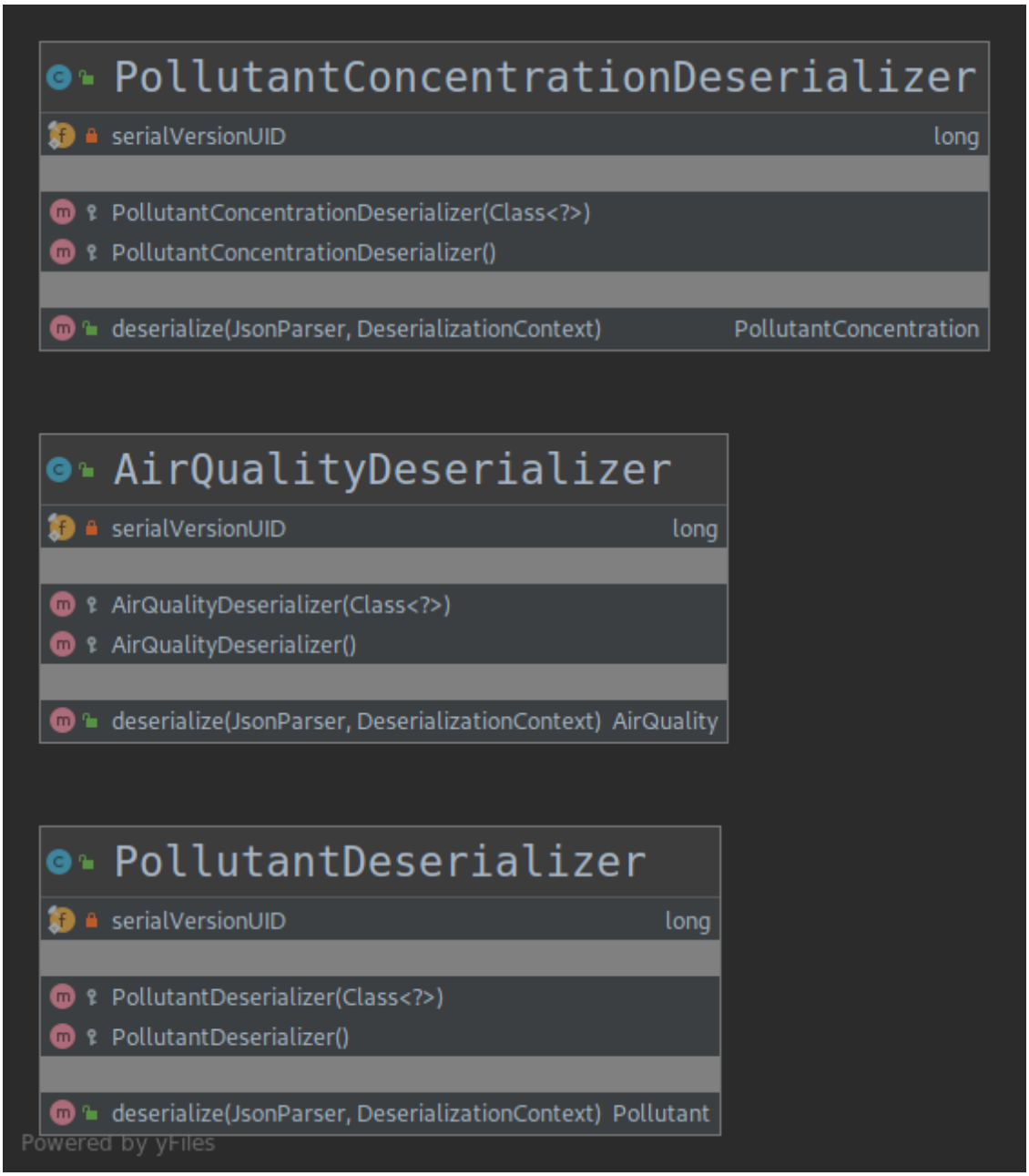

Figure 2.5: *Diagrama das classes do package* ***serializers****.*

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*.

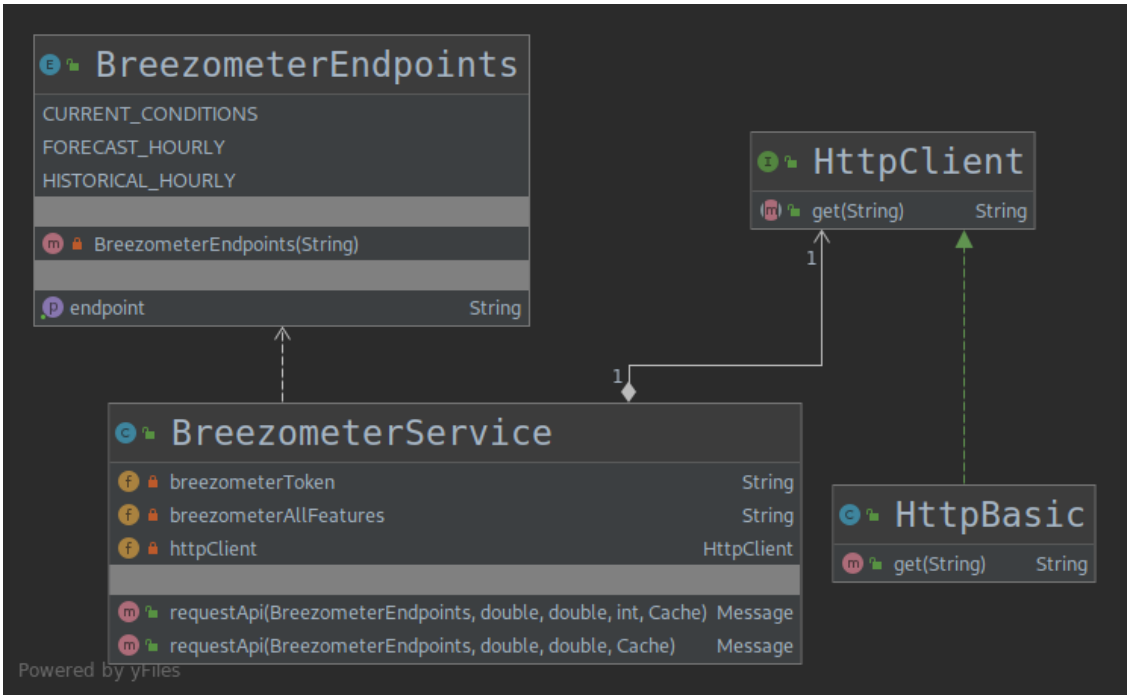

Figure 2.6: *Diagrama das classes do package* ***service***.

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).

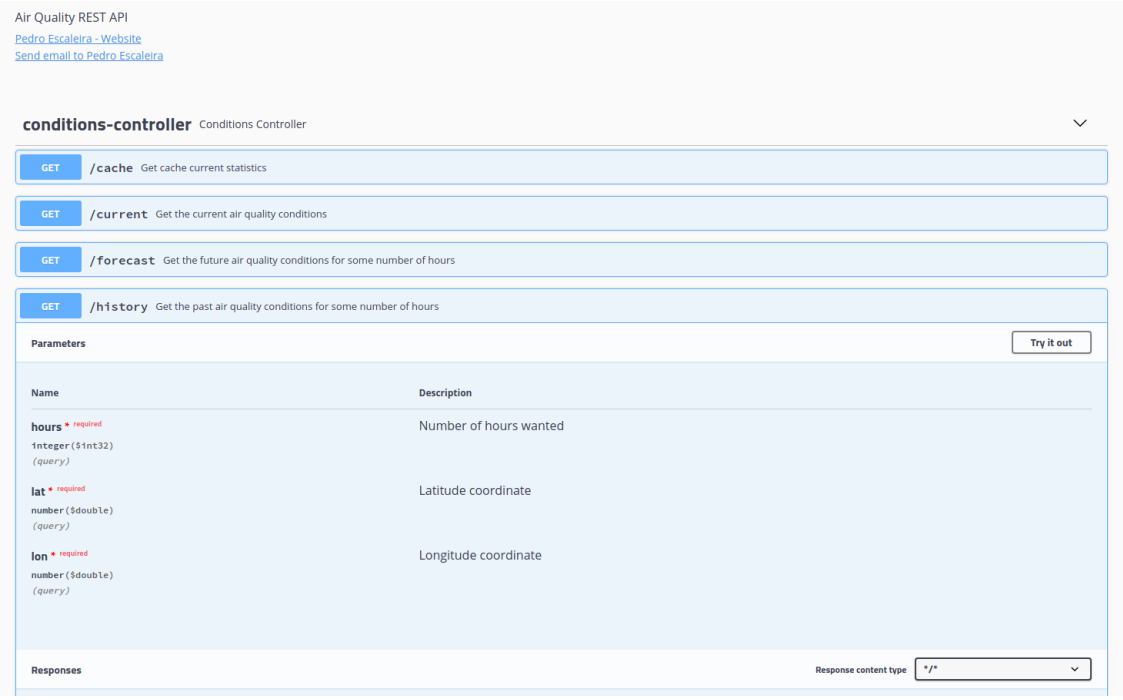

Figure 2.7: *Página da documentação da API criada com a ajuda do Swagger.*

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

Numa primeira fase, foram criados testes unitários para cada os modelos principais e para os *deserializers*. Posteriormente, foram feitos testes de integração quer ao serviço responsável por fazer a ligação ao serviço externo, quer ao controlador. Nas subsecções seguintes é explicado, duma forma geral, os testes feitos de cada tipo.

3.2.1 Testes unitários

Modelos

Quanto aos modelos, foram criados testes unitários para as classes ***AirQuality***, ***Cache***, ***Pollutant*** e ***PollutantConcentration***, já que são os modelos que estão na base do negócio. Não foram criados testes para os restantes pelo simples motivo de que não apresentavam nenhum comportamento de risco, isto é, não têm nenhuma especificação que possa provocar erros no projeto,

já que são apenas *wrappers*, um das possíveis mensagens de erro e outro duma mensagem em si. Para as classes, foram feitos testes sobre o comportamento do método *equals*, já que este foi modificado em todas as classes e em algumas, foi alterado o código auto-gerado pelo *IDE*. Claro que noutras o código gerado pelo *IDE* não foi alterado, mas poderia ser no futuro dado a utilização que foi dada a estas classes, pelo que se decidiu que era o melhor a ser feito de forma a controlar possíveis problemas. Para além disso, sendo que a classe ***AirQuality*** possuía um método que permitia adicionar novos poluentes em momentos distintos, isto é, não se tratava dum simples *addPollutants*, onde seria dada a lista de poluentes toda, foram também feitos testes de forma a controlar o comportamento da adição de novos poluentes ou da não adição dos mesmos (exemplo dum destes testes na figura 3.1, onde é verificada se a ordem com que cada poluente é adicionado à qualidade do ar não afeta que duas qualidades do ar sejam diferentes).

```
96      @Test
97      void testEqualsWithPollutantsWithDifferentOrder() {
98          AirQuality expectedAirQuality = new AirQuality(dominantPollutantTest, date, colorTest, categoryTest,
99                  scoreTest);
100
101         this.airQuality.addPollutant(new Pollutant( simpleName: "simple_name_test", fullName: "full_name_test",
102                 airQualityColor: "color_test", airQualityCategory: "category_test", airQualityScore: 13,
103                 new PollutantConcentration( value: 6.9, units: "unit_test")));
104         expectedAirQuality.addPollutant(new Pollutant( simpleName: "simple_name_test", fullName: "full_name_test",
105                 airQualityColor: "color_test", airQualityCategory: "category_test", airQualityScore: 13,
106                 new PollutantConcentration( value: 6.9, units: "unit_test")));
107
108         assertThat(airQuality, is(expectedAirQuality));
109     }
```

Figure 3.1: *Exemplo dum código de teste da adição de poluentes a um objeto da classe* ***AirQuality****.*

Já para a classe ***Cache***, sendo que esta apresentava um comportamento muito distinto das outras, foram criados testes diferentes. Principalmente testes que verificaram o tamanho da cache após excedido o tamanho máximo desta (exemplo do código em 3.2[1]), testes das condições iniciais e testes dos valores de cada uma das métricas de estatística.

```
114
115     @Test
116     void testSizeMoreThanMax() {
117         cache.addResponse(
118                 new Cache.ParametersEncapsulation(BreezometerEndpoints.HISTORICAL_HOURLY, latitude: 3, longitude: 3, hours: 3),
119                 response: "response3");
120         cache.addResponse(
121                 new Cache.ParametersEncapsulation(BreezometerEndpoints.HISTORICAL_HOURLY, latitude: 4, longitude: 4, hours: 4),
122                 response: "response4");
123
124         assertThat(cache.getSize(), is( value: 3));
125     }
126
```

Figure 3.2: *Exemplo do código de teste do tamanho máximo da cache.*

Deserializers

Sendo que foi alterado o comportamento original dos deserializers usados pelo *Spring Boot*, as classes onde essa alteração foi feita tiveram de ser testadas. Desta forma, criaram-se testes unitários para as classes ***AirQualityDeserializer***, ***PollutantDeserializer*** e ***PollutantConcentrationDeserializer***. Para testar o comportamento de cada uma destas, foi feito um teste

[1] no inicio de cada teste da cache são adicionados dois novos elementos, pelo que, ao todo, foram adicionados 4 elementos

similar em cada uma, em que foi dado uma *string json* no mesmo formato das respostas obtidas nos *requests* ao serviço externo e verificou-se se o objeto da respetiva classe criado era o suposto dado os parâmetros da *string json*. Um exemplo dum destes testes pode ser encontrado na figura 3.3, onde é criada uma *string* com a formatação da concentração dum poluente que usualmente é recebida do serviço externo, sendo posteriormente esta "fornecida" aod respetivo *deserializer* e por fim é verificado se o objeto retornado por este é o suposto.

```
class PollutantConcentrationDeserializerTest {

    @Test
    void testDeserialize() throws JsonProcessingException {

        String expectedUnits = "ppb";
        double expectedValue = 41.46;

        String json = "{\n" +
                "                    \"value\": " + expectedValue + ",\n" +
                "                    \"units\": \"" + expectedUnits + "\"\n" +
                "                }";

        PollutantConcentration expectedPollutantConcentration = new PollutantConcentration(expectedValue, expectedUnits);
        PollutantConcentration obtainedPollutantConcentration = new ObjectMapper().readValue(json,
                PollutantConcentration.class);

        assertThat(obtainedPollutantConcentration, is(expectedPollutantConcentration));
    }
}
```

Figure 3.3: *Exemplo do código de teste do deserializer* ***PollutantConcentrationDeserializer****.*

3.2.2 Testes de integração

Serviço

Sendo que a classe ***BreezometerService*** se liga a serviços externos e usa uma grande quantidade de componentes criados, foi necessário fazer alguns testes de integração para verificar se todas as partes funcionavam devidamente em conjunto. Para isso, foi dado uso ao *MockBean* do *Spring Boot*, de forma a simular o comportamento da interface ***HttpClient*** e para, desta maneira, emular os resultados obtidos quando são feitos pedidos ao serviço externo. Um exemplo da configuração duma destas representações é demonstrada na imagem da figura 3.4, onde é obtida uma *string json* com os parâmetros pretendidos através da utilização do método *jsonAirQualityOnePollutantData* da classe ***JsonSamples*** e se indica que, quando é feito um pedido ao à *API* do *BreezoMeter* dos dados da qualidade do ar atuais, para a latitude 10 e longitude 20, esta string deve ser retornada.

```
        BuildBreezometerLink linkBuilder = new BuildBreezometerLink(breezometerToken, breezometerAllFeatures);
        String json;

        // for current conditions test
        json = JsonSamples.jsonAirQualityOnePollutantData(expectedScore[0], expectedColor[0], expectedCategory[0],
                expectedPollutant[0], expectedDate[0], expectedSimpleName, expectedFullName, expectedPollutantScore,
                expectedPollutantColor, expectedPollutantCategory, expectedValue, expectedUnits);
        json = "{\n" +
                "    \"metadata\": null,\n" +
                "    \"data\": " + json + ",\n" +
                "    \"error\": null\n" +
                "}";
        when(httpClient.get(linkBuilder.createLinkString(BreezometerEndpoints.CURRENT_CONDITIONS,  latitude: 10,  longitude: 20)))
                .thenReturn(json);
```

Figure 3.4: *Exemplo duma simulação feita sob a interface* ***HttpClient****, de forma a que esta dê uma dada resposta quando for feito um pedido especifico.*

Os testes desta classe passaram essencialmente por testar se quando feito um dado pedido, foi retornado o objeto da classe ***Message*** suposto. Um exemplo dum destes testes pode ser consultado na imagem da figura 3.5, onde é feito um pedido ao serviço ***BreezometerService*** e é verificado se este retorna a mensagem suposta.

```
109     @Test
110     void testCurrentConditionsRequest() throws ParseException, IOException, URISyntaxException {
111
112         // get resultant data from the message returned
113         AirQuality returnedAirQuality = breezometerService.requestApi(BreezometerEndpoints.CURRENT_CONDITIONS,  latitude: 10,
114                 longitude: 20, cache).getAirQuality();
115
116         // create a air quality object with two pollutants
117         AirQuality expectedAirQuality = new AirQuality(expectedPollutant[0], expectedDate[0], expectedColor[0],
118                 expectedCategory[0], expectedScore[0]);
119         expectedAirQuality.addPollutant(new Pollutant(expectedSimpleName[0], expectedFullName[0],
120                 expectedPollutantColor[0], expectedPollutantCategory[0], expectedPollutantScore[0],
121                 new PollutantConcentration(expectedValue[0], expectedUnits[0])));
122         expectedAirQuality.addPollutant(new Pollutant(expectedSimpleName[1], expectedFullName[1],
123                 expectedPollutantColor[1], expectedPollutantCategory[1], expectedPollutantScore[1],
124                 new PollutantConcentration(expectedValue[1], expectedUnits[1])));
125
126         // test
127         assertThat(returnedAirQuality, is(expectedAirQuality));
128     }
```

Figure 3.5: *Exemplo dum teste para verificação se quando feito um dado pedido ao serviço externo, o* ***BreezometerService*** *retorna o pretendido.*

Controlador

Pelo facto de na classe ***ConditionsController*** se encontrarem todos os *endpoints* e as especificações dos trabalhos a serem feitos quando é recebido um dado pedido na *API*, foi necessário fazer uma grande quantidade de testes de integração sob esta classe, já que é ela que desencadeia a utilização de todas as outras classes criadas no projeto. Mais uma vez, foi usada a anotação *MockBean* para permitir simular o comportamento da interface ***HttpClient***.

Posto isto, foram feitos testes não só para verificar se os vários métodos desta classe retornavam a mensagem da classe ***Message*** devida, como também foram testadas as respostas dadas em *json*, para confirmar que os resultados produzidos tinham a formatação esperada. Foram também induzidos erros quer nas respostas dadas pelo serviço externo, quer erros de execução inesperados, para confirmar se a *API* tinha a capacidade de retornar as devidas mensagens de erro ou sucesso.

Para além de testes mais voltados para a interação com o serviço externo, foi também testado o serviço de cache usado, de forma a confirmar que quando feito um determinado conjunto de *requests* à *API*, os valores devolvidos quando pedidas as estatísticas da cache batiam certo.

Na figura 3.6 é demonstrado um dos testes feitos, neste caso para confirmação de que o *json* retornado quando feito um pedido das condições atuais possuía os valores e parâmetros corretos.

3.3 Testes funcionais

```
@Test
void testCurrentConditionsJsonObject() throws Exception {

    RequestBuilder request = get( urlTemplate: "/current").contentType(MediaType.APPLICATION_JSON)
            .param( name: "lat",  ...values: "10").param( name: "lon",  ...values: "20");

    mvc.perform(request).andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.airQuality.dominantPollutant", is(expectedPollutant[0])))
            .andExpect(jsonPath( expression: "$.airQuality.pollutants", hasSize(2)))
            .andExpect(jsonPath( expression: "$.airQuality.pollutants[*].simpleName",
                    containsInAnyOrder(expectedSimpleName)))
            .andExpect(jsonPath( expression: "$.airQuality.pollutants[*].concentration.value",
                    containsInAnyOrder(expectedValue[0], expectedValue[1])))
            .andExpect(jsonPath( expression: "$", hasKey("detail")))
            .andExpect(jsonPath( expression: "$.multipleAirQuality", nullValue()))
            .andExpect(jsonPath( expression: "$.success", is( value: true)));
}
```

Figure 3.6: *Exemplo dum teste feito à classe* ***ConditionsController****, de forma a verificar se a resposta json era a esperada para o pedido feito.*