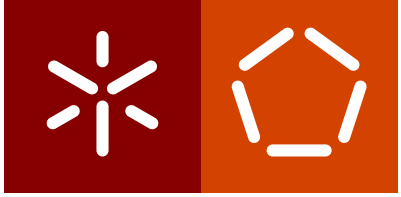
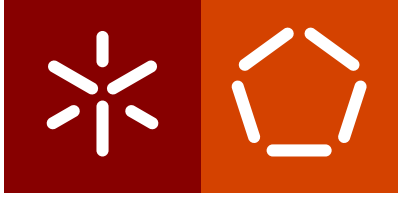


Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Pedro Ferreira Vieira

Mashup de serviços de meteorologia

Julho 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Pedro Ferreira Vieira

Mashup de serviços de meteorologia

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho realizado sob a orientação do Professor
António Nestor Ribeiro

Julho 2021

AGRADECIMENTOS

Gostaria de agradecer a todas as pessoas que contribuíram e me ajudaram não só a realizar esta dissertação como a completar este meu percurso ao longo dos últimos anos.

Agradeço em primeiro lugar ao meu pai e à minha mãe por todo o apoio e sacrifício realizado para me tentarem garantir um futuro melhor. Agradeço ainda mais tendo em conta todas as dificuldades pelas quais passamos, e que me levam de certa forma a dedicar esta dissertação em especial ao meu pai que faleceu no passado ano de 2018, e à minha mãe, que juntamente contribuíram para que eu seja a pessoa que sou hoje.

Agradeço ao professor António Nestor Ribeiro por ter aceite realizar a orientação desta minha dissertação e pelo apoio e ajuda demonstrado ao longo da realização da mesma.

Agradeço a todos os amigos e colegas que me acompanharam e ajudaram ao longo desta etapa, com especial menção ao José Martins, Miguel Quaresma e Simão Barbosa.

Por último, agradeço ainda a todos os restantes familiares e amigos mais próximos por todo o apoio recebido.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

ABSTRACT

Nowadays, there are a lot of information services, many of them making their data available through APIs (some free, others not) so that users can use the data at their own way. Services related to the weather information area are an example of this type of available services, which makes it possible to query weather information (such as temperature, humidity, rain, wind, etc.) in real time from the moment when the information provided by measuring stations (domestic or professional) can be accessed.

The aim of this project is to research and work on aspects of application integration of this kind of information resorting to the development of an application that allows users to access information (obtained from various sources) about, for example, the meteorology of a given location of choice, by combining official sources with other sources, starting with the use of the infrastructure provided by Netatmo as a test bed example.

Given the context of using multiple data sources, it is critical to study and develop an architecture that meets the expectations associated with service-based architectural environment.

In addition, to establish this proof of concept, the system should be scalable and comply with a set of Quality of Service (QoS) parameters established initially, such as the use of message brokers and caching.

KEYWORDS mashups, message brokers, meteorology, scalability, software architectures.

RESUMO

Atualmente existe uma grande quantidade de serviços de informação, muitos deles disponibilizando os seus dados através de APIs (sejam gratuitas ou não) de forma a que os utilizadores possam usar os dados à sua livre vontade. Serviços ligados à área de informações meteorológicas são um exemplo deste tipo de serviços disponíveis, e que tornam assim possível a consulta de dados ligados à meteorologia (como temperatura, humidade, pluviosidade, intensidade do vento, entre outros) em tempo real a partir do momento em que se consegue aceder à informação disponibilizada por estações de medição (domésticas ou profissionais).

Assim, este tipo de sistemas recolhem muita informação e originam um grande volume de dados, que podem ser utilizados e tratados de diversos modos de forma a construir novas aplicações.

Com isto, o objetivo deste projeto passa pela investigação e trabalho em aspetos de integração aplicacional destas informações recorrendo ao desenvolvimento de uma aplicação que permita aos utilizadores obter informações meteorológicas (obtidas a partir de diversas fontes). Um exemplo de uma informação pode ser a meteorologia de um determinado local à escolha. Para conseguir isto, é esperada então uma combinação de fontes oficiais com outras existentes, tendo como ponto inicial a utilização da infraestrutura disponibilizada pela Netatmo como um exemplo de prova de conceito.

Tendo em conta o contexto de utilização de várias fontes de dados, torna-se fundamental estudar e perceber qual a melhor maneira de definir uma arquitetura a implementar que vá de encontro às expectativas a alcançar, sendo esperado um ambiente arquitetural baseado em serviços.

Para além disto, para estabelecer esta prova de conceito é ainda definido como objetivo que a aplicação construída seja escalável e que permita responder com parâmetros de Quality of Service (QoS) definidos inicialmente, tais como a utilização de mecanismos de *message brokers* e de *caching*.

PALAVRAS-CHAVE arquiteturas de *software*, escalabilidade, *mashups*, *message brokers*, meteorologia.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGENS

SIGLAS

AMQP Advanced Message Queuing Protocol. 20

API Application Programming Interface. iv, v, vii, ix, xii, xiii, 1–6, 13, 15, 20, 28, 37, 38, 40, 41, 43, 44, 46, 47, 51–53, 57–65, 67–71, 75, 79–85, 93–95, 97, 99, 100, 108–110, 115, 116

CDN Content Delivery Network. 32

CORS Cross-Origin Resource Sharing. 100

EIPs Enterprise Integration Patterns. 17, 30, 31, 47, 48

FIFO First In, First Out. 18, 24, 36

HTTP Hypertext Transfer Protocol. 12, 17, 43

IDC International Data Corporation. 8

IDE Integrated Development Environment. 8

IoT Internet of Things. 37, 51

IP Internet Protocol. 15, 100, 109

IPC Inter-Process Communication. 9, 11

JAR Java ARchive. 8

JDBC Java Database Connectivity. 21

JMS Java Message Service. viii, 20, 22–24, 26–28, 30

JSON JavaScript Object Notation. 12, 43, 48, 50, 52, 65, 79, 81, 82, 87, 90, 92, 95, 96

JWT JSON Web Token. 95

LFRU Least Frequent Recently Used. 36

LFU Least Frequently Used. 36, 75, 92

LFUDA LFU with Dynamic Aging. 36

LIFO Last In, First Out. 36

LRU Least Recently Used. 36

MQTT Message Queuing Telemetry Transport. 20

MRU Most Recently Used. 36

ORM Object Relational Mapping. 8, 70, 85, 94

QoS Quality of Service. iv, v, 2

RAM Random Access Memory. 32, 36
REST Representational State Transfer. 12, 15, 59, 68, 71, 85, 93, 95, 97, 99
RPC Remote Procedure Call. 49

SLRU Segmented LRU. 36
SMTP Simple Mail Transfer Protocol. 17
STOMP Simple (or Streaming) Text Orientated Messaging Protocol. 20

TLRU Time Aware Least Recent Used. 36
TTL Time to Live. 34, 35, 92

URL Uniform Resource Locator. 79, 82, 100

WAR Web Application Resource. 8

XML Extensible Markup Language. 12
XMPP Extensible Messaging and Presence Protocol. 20

1

INTRODUÇÃO

1.1 ENQUADRAMENTO

Tendo em conta a cada vez maior disponibilização de APIs na área dos serviços de informação e a facilidade de acesso às mesmas, quer seja de forma gratuita ou através de um pagamento, aumentou também o número de aplicações desenvolvidas que utilizam estas mesmas informações nas mais diversas áreas, quer sejam na área da meteorologia, desporto, filmes, ou até mesmo para obter dados estatísticos de uma pandemia como aquela que atualmente vivemos, a COVID-19.

A disponibilização deste tipo de informações permitiu também aumentar o desenvolvimento de projetos que conjuguem dados de variadas fontes numa mesma aplicação, conceito esse que é conhecido como *mashup* [1, 2].

Desta forma, são vários os serviços relacionados com a área meteorológica que desenvolvem e disponibilizam as suas próprias APIs, de forma a que os utilizadores das mesmas consigam aceder a informação meteorológica em tempo real, com origem em estações que efetuam leituras como temperatura, humidade ou pressão atmosférica (efetuadas com um intervalo de tempo definido). Estas estações podem ser profissionais ou domésticas, visto que nos dias de hoje qualquer pessoa pode ter a sua própria estação em casa, comercializada por empresas que utilizam depois as informações obtidas por estes aparelhos nas suas APIs (exemplo da Netatmo[1]).

Assim, a finalidade do projeto desta dissertação consiste na combinação de várias fontes de dados meteorológicos, desenvolvendo uma aplicação que coloque ao dispor informações ligadas a esta área. Sendo assim, a arquitetura aplicacional que se pretende desenvolver deverá ser escalável e robusta, sendo então capaz de fornecer informações meteorológicas que podem ser consumidas por utilizadores finais ou por outras aplicações. A temperatura de um dado local ou as medições efetuadas num dado espaço de tempo definido são exemplos de informações que podem ser fornecidas.

É absolutamente fulcral que a arquitetura planeada seja escalável e que, assim, seja mais facilmente modificável quer seja pela introdução de novas fontes de dados meteorológicos, quer seja pela introdução de novas funcionalidades à arquitetura aplicacional, ou até mesmo para ser capaz de receber e dar resposta a um maior número de utilizadores. Com isto, grande parte deste desafio passa pelo estudo das melhores opções a nível arquitetural de forma a ir ao encontro destas necessidades de escalabilidade. Para além disto, de forma a

1 https://dev.netatmo.com/apidocumentation/

estabelecer esta prova de conceito, é ainda definido que a aplicação para além de ser escalável deve ainda responder com parâmetros de Quality of Service (QoS), sendo assim estabelecida a utilização de mecanismos de *message brokers* para a troca de mensagens entre os serviços, e de mecanismos de *caching* para melhorar o desempenho da aplicação.

Como ponto de partida para o desenvolvimento desta aplicação, é pretendido que se utilize a API fornecida pela Netatmo e pelas suas infraestruturas. É importante perceber também que outras fontes de informação podem ser utilizadas neste contexto e que sejam interessantes de adicionar à arquitetura a desenvolver, servindo assim as várias fontes de dados meteorológicos selecionadas como um exemplo de prova de conceito. A partir do momento em que se encontrem definidas e devidamente estudadas as fontes a utilizar, é fundamental perceber até que ponto se conseguem manipular as informações providenciadas por cada API de forma a conseguir uniformizar as informações definindo um protocolo de resposta que as várias fontes devem seguir para se desenvolver a arquitetura com base nas mesmas, combinando os vários serviços ao dispor do sistema.

Relacionado também com as fontes de dados a utilizar, é importante ainda referir que a grande maioria das APIs têm limites de utilização num contexto de utilização sem pagamento, o que leva também a ter que ser feita uma gestão inteligente do crédito de pedidos que a aplicação pode realizar com cada uma das suas fontes. Isto é também um forte motivo para o estudo e utilização de mecanismos de *caching* na arquitetura a desenvolver, como referido em cima. A introdução destes mecanismos é também valiosa na arquitetura ao ser também esperado que a carga computacional de cada um dos componentes da arquitetura seja apenas a necessária e que não se efetuem processamentos repetidos (principalmente os mais pesados) num curto espaço de tempo.

Sendo esperado um ambiente arquitetural baseado em serviços e com comunicação entre os vários componentes da arquitetura, outro desafio passa por tornar esta mesma arquitetura mais completa, garantindo uma troca de mensagens rápida e segura entre os vários componentes, com procura de garantias de entregas, que permita escalar adaptando-se às necessidades e ao número de mensagens a serem enviadas, e que torne os componentes o mais independentes possíveis dos restantes, até mesmo a nível da ferramentas tecnológicas com que cada um é desenvolvido. Desta forma, é esperado o estudo e a introdução de *message brokers* que devem ter um papel preponderante para conseguir este efeito.

De forma a tornar a aplicação final mais interessante para os utilizadores, é ainda definido que a mesma deve ter um módulo desenvolvido para os utilizadores que se pretendam registar na aplicação. Assim, o objetivo é que um utilizador registado possa selecionar as suas estações favoritas, consiga definir alertas relacionados com as medições das mesmas e, assim, possa posteriormente receber notificações relacionadas com esses mesmos alertas definidos. A ideia passa por demonstrar com alguns tipos de alertas que este componente pode ainda ser estendido de acordo com necessidades futuras e servir também assim como uma prova de conceito.

É pretendido que com a conclusão de um projeto desta dimensão, se consiga perceber no fim do desenvolvimento do mesmo que, para introduzir, por exemplo, uma nova fonte de dados meteorológicos no sistema, seja algo facilmente e rapidamente modificável, sinal de que uma parte dos requisitos foram alcançados. Para além disto, pretende-se com este projeto demonstrar uma integração aplicacional de informações de várias fontes, não apenas de meteorologia, mas em que os conceitos possam assim ser utilizados num outro contexto.

1.2 OBJETIVOS

Sendo assim, tendo em conta o desafio em causa, os principais objetivos para este projeto são:

- Investigação e análise de fontes de dados meteorológicos (APIs).
- Seleção das fontes de dados meteorológicos mais ajustadas.
- Definição dos métodos/requisitos principais que a API do *backend* deve disponibilizar.
- Definição de um protocolo de uniformização das várias fontes de dados.
- Definição de uma arquitetura de *backend* para o sistema, incluindo:
  - Investigação sobre arquiteturas, escalabilidade e arquiteturas orientadas a serviços.
  - Estudo e utilização de mecanismos de *message brokers* e de *caching*.
  - Definição da forma de interação com as APIs das fontes de dados.
  - Adicionar a possibilidade de um utilizador se registar, selecionar as suas estações favoritas, definir alertas e receber notificações.
- Implementação de referência da arquitetura, incluindo:
  - Seleção das tecnologias a utilizar (linguagens de programação, bases de dados, *message brokers*, etc).
  - Desenvolver uma documentação final da API.
- Desenvolvimento do *frontend* da aplicação.

Tendo em conta os objetivos que se pretendem alcançar, torna-se então necessário investigar APIs meteorológicas e, tendo em conta as informações e caraterísticas de cada uma, assim como as limitações a nível de chamadas à API que se pode realizar a cada uma destas, selecionar assim as fontes que podem ser consideradas para utilizar no sistema a desenvolver, considerando as vantagens e desvantagens de cada uma. Este estudo deve ter também em conta a própria API da Netatmo que, apesar de ser tomada como um ponto de partida para o desenvolvimento do sistema, torna-se igualmente importante perceber a mesma.

O objetivo final da arquitetura aplicacional a desenvolver passa por disponibilizar uma API que possa ser utilizada para desenvolver uma interface de apresentação (*frontend*) ou, até, servir para o desenvolvimento de outras aplicações. Com isto, um passo importante deste projeto passa por definir os principais métodos que o *backend* deve disponibilizar. Para tal, é importante perceber o que as outras APIs nesta área apresentam e que outras informações podem ser disponibilizadas que sejam interessantes no contexto de um utilizador final.

Sendo pretendido utilizar as informações de mais do que uma fonte e que o resultado dessa mesma junção (a API a desenvolver) devolva informações das mesmas, torna-se fundamental manipular o formato de resposta de cada uma e uniformizar as mesmas num protocolo de resposta, visto que as fontes não devolvem todas o

mesmo tipo de resposta, o mesmo formato, as medições nas mesmas unidades, etc. Este protocolo a definir pretende facilitar o tratamento das informações, e, também, ter já em vista que no futuro para adicionar uma nova fonte de informação, esta deve então também seguir este formato definido.

Podemos considerar que o principal objetivo deste projeto passa por definir a arquitetura de *backend* da aplicação. Assim, é importante culminar o estudo de vários tópicos (arquiteturas, escalabilidade, *message brokers*, *caching*, etc) com os objetivos referidos anteriormente para definir uma arquitetura que vá de encontro ao desafio proposto, tendo sempre em mente que a mesma deve ser capaz de escalar para modificações futuras serem mais facilmente introduzidas em todo o sistema. Esta arquitetura deverá ter em conta a capacidade de registar utilizadores e oferecer a estes uma série de ações, tal como descrito anteriormente.

Depois de definida a arquitetura, o passo seguinte passa por implementar a mesma. Para tal, torna-se necessário perceber quais as tecnologias a utilizar para desenvolver os componentes do sistema. É pretendido também construir uma documentação detalhada da API disponibilizada, facilitando o desenvolvimento de qualquer aplicação que pretenda fazer uso destas informações.

Por fim, a última etapa passa pelo desenvolvimento do *frontend* da aplicação, sendo necessário definir a melhor forma de apresentar todas as informações disponibilizadas pelo *backend* ao utilizador final, seja este um utilizador registado ou não.

1.3 ESTRUTURA DO DOCUMENTO

No segundo e próximo capítulo desta dissertação é então documentado o estado da arte atual, que cobre assuntos como as *mashups*, arquiteturas de micro-serviços, *message brokers* com comparação de duas possibilidades tecnológicas (ActiveMQ e Kafka), mecanismos de *caching* e aprendizagem sobre a API da Netatmo.

Quanto ao terceiro capítulo, é feita uma interação com a API da Netatmo e com o Kafka, e é ainda realizado um levantamento de informações sobre outras fontes de dados meteorológicos.

No quarto capítulo é identificado o desafio em mãos e são apresentadas algumas caraterísticas e também funcionalidades que se pretendem ver implementadas na aplicação final.

Já no quinto capítulo é explicada toda a evolução e decisões tomadas até culminar numa proposta de arquitetura final para o *backend* da aplicação.

No capítulo seguinte é então descrito o processo de implementação desta arquitetura definida, assim como o desenvolvimento da aplicação de *frontend*, apresentando decisões como as tecnologias ou algoritmos utilizados.

Por fim, o sétimo capítulo consiste numa conclusão sobre todo o trabalho desenvolvido tendo em conta os objetivos definidos, tendo ainda em vista possíveis melhorias para um trabalho futuro.

2

ESTADO DA ARTE

Neste capítulo é documentado o estado da arte atual, cobrindo assuntos relacionados com esta dissertação. Desta forma, *mashups*, arquiteturas de micro-serviços, *message brokers*, mecanismos de *caching* e levantamento de informações sobre a API da Netatmo são temas abordados no decorrer deste capítulo.

2.1 MASHUPS

Tendo em conta que se pretende desenvolver uma aplicação que combine várias fontes de informação, torna-se necessário adquirir conhecimentos sobre o conceito de *mashups*. Desta forma, é importante perceber o que são *mashups* e quais os seus principais tipos. Para além disto, é ainda feito um levantamento de duas *mashups* que podem ser considerados casos de sucesso. Com isto, é assim pretendido que se perceba o contexto do problema no que a este conceito diz respeito.

2.1.1 *O que é uma mashup?*

Uma *mashup* é uma aplicação *web* que utiliza conteúdo de mais do que uma fonte de forma a criar um único novo serviço [1, 2].

Podemos verificar que nos últimos anos as *mashups* criaram um impacto muito forte na *internet*. Podemos justificar o crescimento deste tipo de aplicações com dois grandes motivos [1]:

- Em primeiro lugar, muitas das maiores companhias ligadas ao setor informático, como a *Google* ou a *Amazon*, disponibilizaram os seus dados podendo assim serem usados com outras fontes de dados sem que seja necessário uma grande negociação para obter acesso aos mesmos. Assim como estas empresas fornecem os seus dados (sejam eles gratuitos ou não) muitas outras empresas ou projetos disponibilizam os seus dados, havendo assim um vasto e diferenciado conjunto de dados à disposição dos programadores.

- Em segundo lugar, este rápido crescimento deveu-se também à criação de ferramentas que tornaram o processo de construção de *mashups* mais fácil, até para pessoas sem tantos conhecimentos técnicos na área. Ferramentas como o Yahoo Pipes, Google Mashup Editor, ou o Microsoft Popfly [3] são exemplos de ferramentas que existiam e que facilitavam a criação de *mashups*. Por exemplo, o Yahoo Pipes permitia

combinar *data feeds* populares para criar um *data mashup* através de um editor visual [3], tendo o projeto sido encerrado em 2015.

2.1.2 *Tipos de mashups*

As *mashups* podem ser classificadas pelo seu tipo, sendo que podemos considerar três categorias pelas quais podemos avaliar as mesmas:

- **Consumer mashups** - consistem na utilização de diferentes formas de informações de várias fontes e a combinação das mesmas numa única interface gráfica. São desenhadas para o público em geral [2] ou consumidores [4].
- **Data mashups** - ao contrário das *consumer mashups*, estas consistem na utilização de informações/dados semelhantes de várias fontes e a combinação das mesmas numa única representação [2].
- **Business mashups** (ou *enterprise*) - este tipo de *mashups* são focados na agregação de dados numa única apresentação [2]. Permitem automatizar as atividades críticas entre equipas e sistemas para um ambiente mais produtivo [4].

2.1.3 *Exemplos de mashups*

A aplicação *HousingMaps* é um exemplo de uma *mashup* com sucesso [3], podendo ser considerada como a primeira *mashup* com base no *Google Maps*, tendo sido criada antes ainda da existência da *Google Maps API* [5]. O propósito do site passava por combinar os imóveis listados no *Craigslist* (site americano de anúncios classificados com várias secções) com dados do Google Maps com o intuito de ajudar pessoas a mudarem-se de uma cidade para outra e a procurarem casas [3]. Funcionava para cerca de 30 cidades dos Estados Unidos e ainda Londres. Quando o site foi lançado, tornou-se bastante popular a opinião de que os imóveis seriam assim melhor pesquisados com a ajuda de um mapa ao invés da apresentação de uma listagem dos mesmos. De momento, *HousingMaps* já não se encontra disponível na *web*, tendo em conta que o próprio *Craigslist* já dispõe de uma *map view* para todos os imóveis listados no seu site. A Figura 1 demonstra a interface que esta aplicação apresentava.

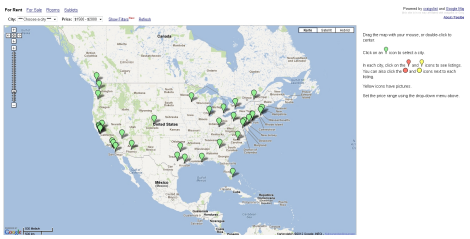


Figura 1: Site *HousingMaps*.[1]

1 Figura retirada de 'HousingMaps' ([5]).

Outra *mashup* bem conhecida e utilizada hoje em dia é o *Trendsmap* [6], sendo a sua interface apresentada na Figura 2. Este *website* combina tópicos populares do *Twitter* e a sua localização com o *Google Maps*, sendo possível desta forma obter os temas mais falados nas várias regiões do globo. O site permite filtragem de temas populares por palavras, *hashtags* ou utilizadores, tendo ainda outras funcionalidades ao dispor dos utilizadores.

Figura 2: Site *Trendsmap*.

Uma conclusão que se pode tirar destes dois exemplos e que reflete a realidade é a de que muitas das *mashups* que são desenvolvidas utilizam mapas, sendo assim verificado que existe uma grande adesão por parte das pessoas a este tipo de disponibilização e consulta de dados.

2.1.4 *Aplicação ao projeto*

No contexto desta dissertação, podemos considerar a *mashup* a desenvolver como sendo do tipo *Data mashup*, tendo em conta que passa pela utilização de dados semelhantes com origem em diferentes fontes (ligadas a meteorologia).

Os desafios a considerar passam pela seleção das melhores fontes de dados, estudo e desenvolvimento da melhor arquitetura para um projeto deste tipo, assim como escolha da melhor forma de apresentar os dados ao utilizador final, visto que um serviço que conjugue várias fontes tem que obrigatoriamente se tornar útil e agradável ao utilizador, caso contrário todo o conceito de construção de uma *mashup* perde o seu sentido.

2.2 MICRO-SERVIÇOS

A utilização de arquiteturas em micro-serviços para o desenvolvimento de aplicações tecnológicas é cada vez maior ao longo dos últimos anos. Este tipo de arquiteturas tornou-se muito popular tendo a comunidade percebido que para escalar mais, para ser mais eficaz na entrega e colocação de *software* em produção, assim

como para conseguir tirar vantagem de coisas como entregas contínuas, é necessário uma abordagem que permita escalar em diferentes eixos independentemente [7]. A International Data Corporation (IDC) previu que em 2021, 80% das aplicações desenvolvidas em plataformas da *cloud* serão em micro-serviços [8].

2.2.1 *Arquitetura monolítica*

A forma mais tradicional, se assim podemos dizer, para o desenvolvimento de software, passa pela utilização de um padrão arquitetural monolítico. Normalmente, podem ser divididas em quatro partes: a interface do utilizador, a lógica de negócio, uma interface de dados e uma base de dados [9], como demonstra a Figura 3.

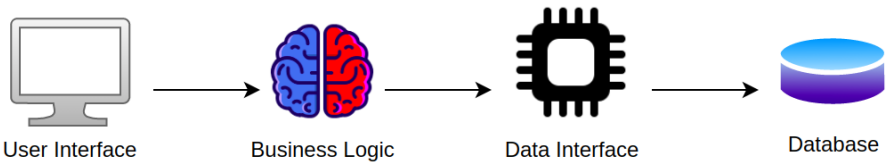


Figura 3: Exemplo de arquitetura monolítica.

Podemos considerar assim que as aplicações monolíticas utilizam uma arquitetura em três camadas:

- **Apresentação** - representa a camada superior da aplicação, responsável pelo contacto com o utilizador. Com este contacto, são assim feitas determinadas consultas ou ações colocadas ao dispor através da interface.
- **Negócio** - é nesta camada que estão as regras de negócio da aplicação, sendo que é nela que são tomadas decisões lógicas e feitos determinados cálculos. Processa os dados entre as outras duas camadas.
- **Dados** - esta camada serve principalmente para obter e guardar informações da base de dados, sendo esta a camada mais baixa, a partir da qual os dados passam para a camada de negócio e em último lugar para o utilizador. São utilizadas tecnologias ORM como o Hibernate (Java) ou o Sequelize (Node.js) para processar informações.

Estas aplicações, apesar de terem a sua própria arquitetura modular e com múltiplas funcionalidades implementadas são, ainda assim, empacotadas e realizado o seu deploy como um monolítico, sendo que o formato da mesma depende das linguagens e frameworks utilizadas (por exemplo, Java com ficheiros JAR ou WAR, aplicações Node.js que são empacotadas como uma hierarquia de diretórios, etc) [10].

Podemos associar a este padrão arquitetural algumas **vantagens** [10, 9]:

- São simples de desenvolver, tendo em conta que os IDEs e outras ferramentas que utilizamos são focadas no desenvolvimento de uma única aplicação.
- Fáceis de testar, através de ferramentas como o Selenium por exemplo.

- O *deploy* deste tipo de aplicações tende a ser mais simples, sendo normalmente apenas necessário copiar os ficheiros necessários para um servidor desejado.
- Os componentes numa aplicação monolítica por norma partilham memória pelo que tende a ser mais rápido que a comunicação entre serviços utilizando Inter-Process Communication (IPC) ou outros mecanismos.
- Para escalar uma aplicação monolítica, é possível escalar a mesma horizontalmente ao replicar a aplicação por vários servidores e utilizar um balanceador de carga para balancear os pedidos pelos servidores disponíveis.

Os problemas na utilização de aplicações monolíticas tendem a surgir quando novas funcionalidades são adicionadas com o passar do tempo e a aplicação torna-se gigantesca. A partir daqui, a aplicação fica demasiado complexa, o desenvolvimento tende a ser mais desorganizado, assim como a utilização de uma metodologia ágil torna-se custosa. Neste ponto, podemos apontar **desvantagens** às aplicações monolíticas [10, 9]:

- Corrigir problemas ou adicionar novas funcionalidades passa a demorar demasiado tempo tendo em conta a dimensão da aplicação e o tempo necessário para perceber a mesma.
- No caso de aplicações muito grandes, o tempo de inicialização pode ser demasiado elevado.
- Nos tempos de hoje, em que as equipas de desenvolvimento de software fazem mudanças em produção várias vezes, uma aplicação monolítica complexa representa um verdadeiro obstáculo, visto que para fazer uma pequena mudança é necessário fazer novamente *deploy* de toda a aplicação para que a mudança tenha efeito.
- É extremamente complicado inserir uma nova linguagem ou framework numa aplicação deste tipo, pois, se temos por exemplo uma aplicação escrita em Node.js e queremos usar Java na mesma, torna-se necessário mudar todo o código para esta nova linguagem, o que para uma aplicação grande e complexa torna-se demasiado exigente. Sendo assim, tende-se a ficar preso às tecnologias escolhidas numa fase inicial de desenvolvimento.
- Como todos os módulos da aplicação são executados por norma no mesmo processo, um erro num módulo pode comprometer toda a aplicação.

2.2.2 *Arquitetura de micro-serviços*

A solução para resolver este tipo de problemas e adotada por várias empresas de renome passa pela utilização de um padrão arquitetural de micro-serviços. A ideia deste padrão passa por dividir uma aplicação por um conjunto de serviços mais pequenos e que comuniquem entre si. Cada serviço pode ser visto como uma aplicação que tem a sua própria arquitetura e que implementa um conjunto de funcionalidades distintas. Atualmente,

cada instância de um micro-serviço representa normalmente uma máquina virtual na *cloud* ou um *container* Docker por exemplo.

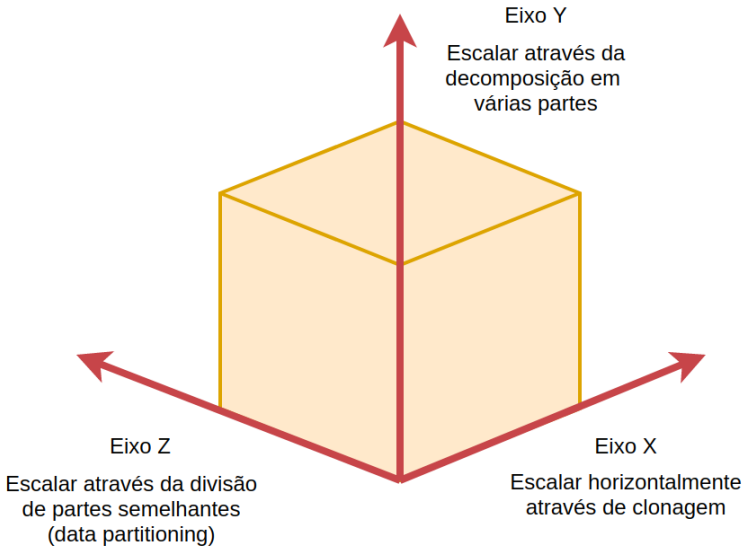


Figura 4: *The Scale Cube*.[2]

De acordo com a Figura 4 que mostra o cubo do escalar de aplicações, a utilização de um padrão arquitetural baseado em micro-serviços corresponde ao escalar de aplicações no eixo Y do cubo, visto que, essencialmente, a ideia passa por **decompor o sistema em várias partes**. Quanto ao eixo do X, enquanto que numa aplicação monolítica corresponde a utilizar várias cópias da aplicação total por trás de um *load balancer*, numa metodologia de micro-serviços, o escalonamento horizontal passa assim por utilizar múltiplas instâncias dos serviços com um *load balancer*, quer seja para garantir disponibilidade do serviço (para o caso de falha de alguma instância), como para dar uma maior capacidade de resposta (*throughput*). Quanto ao eixo Z, este tipo de escalonamento é normalmente utilizado para escalar bases de dados através da divisão das mesmas (*sharding*), sendo os dados repartidos por entre um conjunto de servidores baseado num atributo de cada item. A título de exemplo, dividir uma tabela de utilizadores por dois servidores baseado nas chaves primárias da tabela é um exemplo para esta prática. É necessário referir ainda que a clonagem do eixo X pode ser aqui aplicada a cada partição através da implementação de servidores que sirvam de réplicas/*slaves* [11].

Feita a introdução a este tipo de arquiteturas, podemos assim apresentar algumas **vantagens** associadas à utilização deste padrão [10, 12, 9]:

- Por norma, cada serviço tem o seu esquema de base de dados, o que contribui para um acoplamento fraco (*loose coupling*).
- Este acoplamento fraco permite que os serviços sejam testados individualmente.
- Cada serviço é mais fácil de desenvolver e, comparativamente a um cenário monolítico, é muito mais fácil a compreensão e a manutenção de cada componente.

2 Figura baseada em [Figure 1-3] de 'Microservices: From Design to Deployment' ([10]).

- É possível tomar as decisões mais acertadas para cada serviço, visto que o mesmo é desenvolvido de forma independente e, assim, facilita a escolha das melhores tecnologias a utilizar em cada contexto, desde que o serviço cumpra com o contrato estabelecido para comunicação com os restantes componentes da arquitetura.
- É uma arquitetura otimizada para a substituibilidade, pois sendo os serviços mais pequenos, o custo de substituir um deles por uma melhor implementação ou até mesmo de apagar o mesmo é muito mais simples de gerir.
- O *deploy* de cada serviço é feito de forma independente, não sendo necessário neste contexto reinicializar toda a aplicação para que possíveis mudanças façam efeito.
- É possível escalar cada serviço de forma independente, ou seja, podemos utilizar as instâncias necessárias para cada serviço de forma a corresponder às necessidades, o que também otimiza a gestão de recursos.

Por outro lado, para poder usufruir destas vantagens existe um preço a pagar, pelo que existem **desvantagens** na utilização de micro-serviços como [10, 9]:

- Uma aplicação com micro-serviços é por consequência um sistema distribuído, o que adiciona complexidade à solução escolhida. Decisões como a escolha de mecanismos de interação entre vários componentes (Inter-Process Communication (IPC)) são um exemplo deste tipo de complexidade.
- A implementação de transações que necessitem de efetuar mudanças em vários serviços é muito mais complexa comparativamente a uma solução monolítica em que tudo se encontra na mesma máquina a ser executado.
- Implementar mudanças com o alcance de mais do que um serviço pode ser difícil em certos contextos, sendo preciso estudar e coordenar as mudanças a serem feitas nos vários serviços tendo em conta as alterações e as dependências que existem entre estes componentes.
- Realizar o *deploy* de uma solução em micro-serviços é bem mais complexo, tendo em conta a quantidade de serviços que devem ser implementados e postos a executar.
- No contexto de uma empresa, a utilização deste paradigma deve ser aliada a uma formação ou conhecimento por parte das pessoas que vão desenvolver a aplicação, caso contrário, a introdução deste tipo de arquiteturas e os seus benefícios são, muito provavelmente, discutíveis.
- Ao dividir a aplicação em vários componentes, obrigatoriamente passam a existir mais componentes para manter e monitorizar, o que sem os conhecimentos e ferramentas certas pode levantar problemas no desenvolvimento.

Numa aplicação baseada em micro-serviços um ponto fulcral passa pela escolha de **mecanismos de interação** entre os vários componentes do sistema (IPCs). Enquanto que numa aplicação monolítica os componentes

interagem uns com os outros ao nível da linguagem de programação escolhida ou através da chamada de funções, neste novo padrão por norma temos um sistema distribuído a ser suportado por diversas máquinas, sendo assim necessário perceber a melhor forma de trocarem informações entre os vários serviços.

Podemos caraterizar os tipos de interação em duas dimensões diferentes:

- A interação é síncrona ou assíncrona? Numa interação síncrona é expectável que depois de um pedido efetuado a resposta chegue num curto de espaço de tempo, podendo até o programa bloquear enquanto espera por esta resposta. Já numa interação assíncrona não existe qualquer tipo de bloqueio à espera de uma resposta, e, no caso de ser mesmo necessária uma resposta, esta não necessita de ser enviada imediatamente.
- A interação é de um para um ou de um para muitos? Uma interação de um para um pressupõe que o pedido é processado por uma única instância, enquanto que no caso de um para vários são esperados que vários serviços processem o pedido.

A Tabela 1 apresenta os vários tipos de comunicação entre processos.

	interação síncrona	**interação assíncrona**	
um para um	pedido / resposta	notificação	pedido / resposta assíncrona
um para vários	-	publicar / subscrever	publicar / respostas assíncronas

Tabela 1: Tipos de comunicação entre processos.[3]

No que toca às interações de um para um através de um padrão de 'pedido/resposta' o processo que realizou o pedido espera que a resposta chegue rapidamente, podendo bloquear à espera desta. Quanto à comunicação através de uma 'notificação', o processo envia um pedido para um serviço mas não está à espera de receber uma resposta em troca. Por último, com uma comunicação 'pedido/resposta assíncrona' é esperado que face a um pedido realizado, a resposta seja assíncrona e, com isto, o cliente não precisa de bloquear e já é criado tendo em conta que a resposta pode demorar algum tempo até ser recebida [10].

Já quanto às interações de um para vários, a interação 'publicar/subscrever' pressupõe que o cliente publica uma mensagem de notificação que vai ser consumida por zero ou mais serviços interessados. Por outro lado, através da interação 'publicar/respostas assíncronas' o cliente publica uma mensagem relativa a um pedido e fica depois à espera de respostas por parte dos serviços interessados [10].

Quanto às tecnologias utilizadas para conseguir estas interações, por norma, para uma comunicação síncrona, é muitas vezes utilizado um mecanismo de comunicação HTTP como o REST (Representational State Transfer) [13]. Por outro lado, para uma comunicação assíncrona baseada em mensagens é comum utilizar sistemas de troca de mensagens como o Apache ActiveMQ, Apache Kafka ou o RabbitMQ [10]. Quanto ao formato a utilizar nas mensagens, é normal a escolha pelos formatos que são mais facilmente compreendidos por nós, tal como o JSON ou o XML, apesar de existirem alternativas como Protocol Buffers que utilizam um formato binário e que consegue ser mais eficiente [10].

3 Tabela baseada em [Table 3-1] de 'Microservices: From Design to Deployment' ([10]).

Um componente muito importante numa arquitetura de micro-serviços é o **API Gateway**, pelo que é de extrema importância perceber a sua utilidade.

Enquanto que numa arquitetura monolítica, para um cliente interagir com a aplicação existe apenas um conjunto de *endpoints* expostos e cujo acesso é facilitado, numa arquitetura de micro-serviços temos vários serviços que podem expor um conjunto de pontos de acesso da API (*endpoints*) mais refinados, sendo necessário perceber a melhor forma de aceder aos mesmos.

Vamos supor que pretendemos desenvolver uma aplicação *web* e uma aplicação *mobile* que utilizem as funcionalidades de uma arquitetura de micro-serviços que tem três serviços a expor *endpoints* que devem ser utilizados.

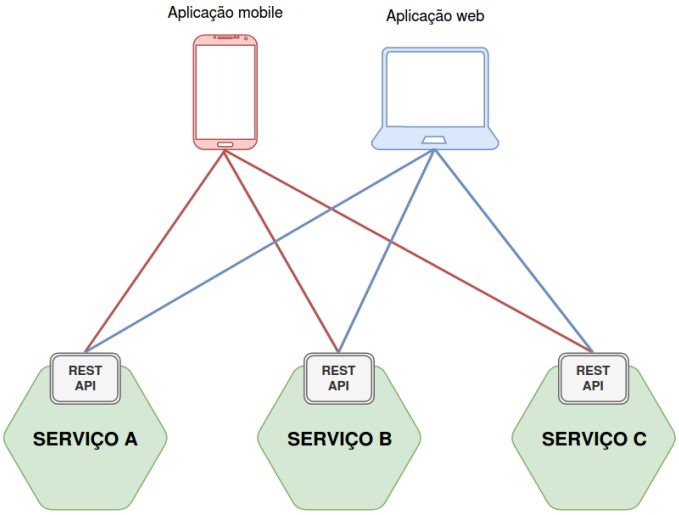


Figura 5: Mapeamento dos *requests* de aplicações *web* e *mobile* por um conjunto de micro-serviços.

Uma primeira abordagem poderia passar por cada um destes micro-serviços ter um endereço público para um *load balancer* de cada um destes serviços e assim cada uma das aplicações (*web* e *mobile*) interagir com os três endereços destes serviços, como é representado na Figura 5.

Ainda assim, podemos apontar um defeito grande desta abordagem. Por exemplo, isto inviabiliza em grande parte uma refatorização dos serviços da arquitetura, visto que ao longo do tempo podemos querer mudar a forma como o sistema está dividido em serviços e, se por exemplo, quisermos dividir um serviço em dois ou mais serviços novos, isso vai obrigar também a fazer alterações no código dos clientes que interagem com este serviço, o que é um grande obstáculo para uma mudança que deveria apenas ter impacto na arquitetura.

É nesta altura que surge o API Gateway como uma solução para este problema, que funciona no fundo como um *backend* para os *frontends*, ao ter do lado do servidor uma agregação de endpoints [12]. O API Gateway é assim um servidor que funciona como um **ponto de entrada único** do sistema, similar ao padrão *Facade* do desenvolvimento orientado a objetos [10]. Para além disto, este componente pode ainda ter outras funções como monitorização, *load balancing*, autenticação, *caching*, entre outros.

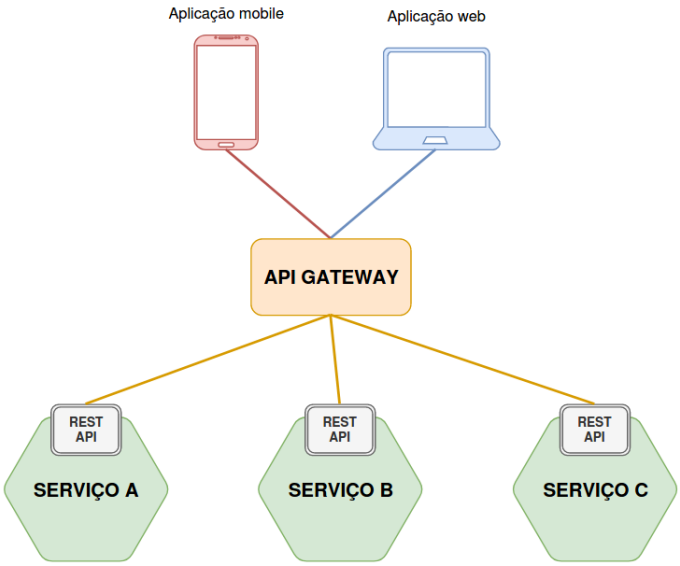


Figura 6: Utilização de um API Gateway numa arquitetura de micro-serviços.

Assim, e como demonstrado na Figura 6, para o caso de desenvolvimento de aplicações *web* e *mobile*, estas apenas necessitam de estar em contacto apenas com um endereço público que serve de entrada na arquitetura de *backend*. Deste modo, estas aplicações ficam muito mais preparadas e suscetíveis a possíveis mudanças a acontecer no futuro ao nível dos *endpoints* e serviços que são utilizados na arquitetura. No caso de serem pretendidas diferenças nos pedidos e respostas feitas pela aplicação *web* e *mobile*, pode também optar-se por ter dois API Gateways desenvolvidos à medida. Desta forma, tem-se assim um componente deste tipo para cada aplicação de *frontend*.

Ao adicionar um componente deste tipo à arquitetura de micro-serviços temos claramente vantagens e desvantagens. Ao nível de **vantagens** podemos identificar [10, 14]:

- O API Gateway consegue encapsular a estrutura interna da aplicação, pois, ao invés do cliente entrar em contacto com vários serviços, este passa a apenas interagir com este componente.
- Permite oferecer uma API otimizada para cada cliente.
- Simplifica o código do cliente, movendo a lógica de chamar vários serviços do lado do cliente para o API Gateway.
- Permite reduzir o número de pedidos/*roundtrips*.

Quanto a **desvantagens**, temos algumas como [10, 14]:

- A mais evidente é que passamos a ter um novo componente que deve ser desenvolvido, instalado e gerido, e que deve estar sempre disponível.
- Existe o risco deste componente se tornar num estrangulamento do sistema.

- Este componente acaba por aumentar o tempo de resposta devido à introdução de um novo interveniente nos pedidos. Ainda assim, para a grande maioria de aplicações, o custo adicional acaba por ser insignificante.

Outro aspeto que deve ainda ser referenciado quando se falam em arquiteturas de micro-serviços é a **descoberta de serviços**. Numa arquitetura de micro-serviços, se um serviço pretende interagir com outro (para consumir uma REST API por exemplo) este deve conhecer um IP e uma porta de uma instância deste serviço. Por norma, é habitual configurar numa aplicação estas localizações de serviços através da definição de um ficheiro de configuração que pode ser atualizado ocasionalmente. No entanto, numa aplicação moderna baseada em micro-serviços que tipicamente é executada em ambientes virtualizados ou *containers*, o número de instâncias dos vários serviços e cada uma das suas localizações são alteradas dinamicamente, devido a falhas, *upgrades* ou à capacidade de automaticamente escalar, pelo que se torna necessário utilizar um mecanismo de descoberta de serviços mais elaborado [15, 16, 10]. Para resolver este problema existem essencialmente dois padrões que são maioritariamente utilizados: *client-side discovery* e *server-side discovery*.

Quanto ao padrão *client-side discovery* (Figura 7), este determina que o cliente é responsável por determinar as localizações das instâncias disponíveis de um serviço. Para conseguir isto, as instâncias de um serviço são registadas no *service registry* (uma base de dados que contêm as instâncias de um serviço disponíveis) quando são iniciadas, e removidas assim que terminadas. Depois, assim que um cliente pretender fazer um pedido a um determinado serviço, este questiona o registo de serviços pela lista de instâncias e, na posse desta listagem, utiliza um algoritmo de *load balancing* para escolher uma das instâncias disponíveis e finalmente realizar o pedido.

Uma vantagem desta abordagem é que podemos considerar que para além do registo de serviços não existem mais componentes neste contexto [10, 16]. Para além disto, o cliente conhece as instâncias dos serviços, pelo que pode assim fazer decisões ao nível do algoritmo de *load balancing* de acordo com o melhor para a aplicação em causa [10]. Por outro lado, uma principal desvantagem é que existe um grande acoplamento entre o cliente e o registo de serviços, e também que se torna necessário implementar código de descoberta de serviços do lado de cliente para as linguagens de programação a utilizar [10, 16].

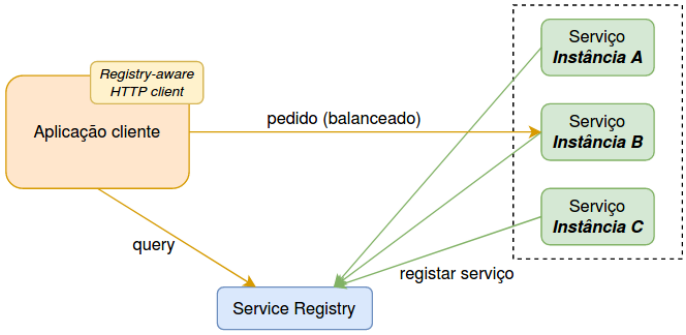


Figura 7: Padrão *client-side discovery*.[4]

4 Figura baseada em [Figure 4-2] de 'Microservices: From Design to Deployment' ([10]) e 'Pattern: Client-side service discovery' ([16]).

O outro padrão, *server-side discovery* (Figura 8), determina que o cliente faz o pedido a um serviço através de um *router* / *load balancer*, que tem uma localização definida e conhecida. De seguida, este *load balancer* é responsável por obter a partir do registo de serviços a lista de instâncias disponíveis e, depois, mapeia o pedido recebido para uma instância disponível do serviço em causa.

Uma grande vantagem deste padrão é que os detalhes da descoberta de serviços passam a ser abstraídos do cliente, que passa apenas a fazer um pedido simples ao *load balancer*, não sendo assim necessário implementar lógica de descoberta para cada linguagem utilizada [10, 15]. Como desvantagens, a não ser que o *load balancer* faça já parte do ambiente arquitetural implementado, trata-se de introduzir outro componente que deve estar disponível e necessita de ser configurado e mantido, para além de que esta solução adiciona mais saltos (*hops*[5]) na rede para a realização de um pedido comparativamente ao padrão anterior [10, 15].

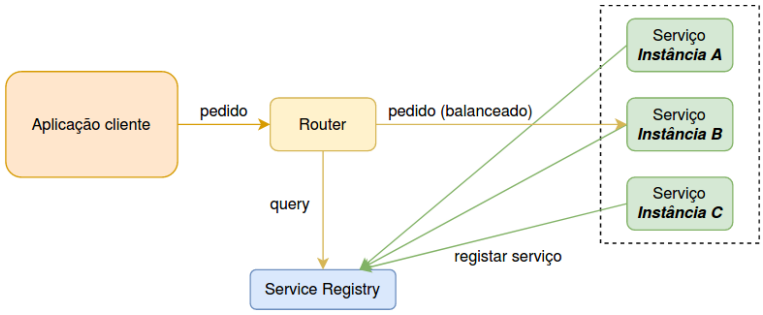


Figura 8: Padrão *server-side discovery*.[6]

2.2.3 *Aplicação ao projeto*

No contexto desta dissertação, podemos assim culminar o estudo destes temas para concluir que a escolha acertada para um projeto com estas caraterísticas passa pelo desenvolvimento de uma arquitetura em micro-serviços, tendo em conta as vantagens e desvantagens que devem ser tidas em conta.

Tendo em consideração que o sistema que se pretende desenvolver deve ser pensado em adaptações futuras para, por exemplo, adicionar novas fontes de dados meteorológicos ou para dar resposta a um maior número de utilizadores, a aplicação claramente beneficia de um arquitetura deste tipo que tem em vista a capacidade de mais facilmente escalar.

Para além disto, e tendo em conta que se pretende desenvolver um *frontend*, é de todo o interesse a adição ao sistema de um API Gateway que sirva de contacto entre o *frontend* e o *backend*, de acordo com necessidades que possam ser levantadas.

5 Numa rede, o número de *hops* indica o número de intermediários pelos quais os dados devem passar desde a origem ao destino de um pedido.

6 Figura baseada em [Figure 4-3] de 'Microservices: From Design to Deployment' ([10]) e 'Pattern: Server-side service discovery' ([15]).

2.3 MESSAGE BROKERS

É esperado que a introdução de *message brokers* permita, por exemplo, adicionar uma maior independência entre os vários componentes do sistema. Assim, para além de perceber o que são *message brokers*, é interessante perceber quais são os principais modelos de trocas de mensagens que existem, assim como abordar dois dos *message brokers* mais conhecidos, Apache ActiveMQ e Apache Kafka, percebendo o funcionamento e caraterísticas de cada um destes. Por último, é feito ainda um levantamento de informação sobre os Enterprise Integration Patterns (EIPs).

2.3.1 *O que é um message broker?*

Muitas vezes, devido à falta de conhecimento e compreensão de como funciona um serviço de *message broker* [7], existem certas coisas que tomamos como garantidas acerca dos mesmos, tais como [17]:

- O sistema nunca vai perder mensagens.
- As mensagens são processadas por ordem.
- Adicionar consumidores vai tornar o sistema mais rápido.
- As mensagens vão ser entregues apenas uma vez.

A verdade é que enquanto algumas destas frases são incorretas, outras apenas são aplicáveis em determinadas circunstâncias e, daí, surge então a necessidade de compreender melhor estas ferramentas num contexto de utilização das mesmas.

Para que duas aplicações possam comunicar uma com a outra, é necessário estabelecer uma interface, que envolve a escolha de um método de transporte ou protocolo (como HTTP ou SMTP), assim como estabelecer um acordo no formato das mensagens a serem trocadas. A partir do momento em que a forma de comunicação é estabelecida, passa então a ser possível a comunicação entre os dois sistemas sem que exista uma preocupação em saber como o outro sistema está implementado, não sendo por isso necessário saber informações como quais as *frameworks* utilizadas ou até a linguagem de programação utilizada, visto poderem até ser alteradas e evoluírem ao longo do tempo, desde que o contrato estabelecido seja mantido. Sendo assim, podemos dizer que os dois sistemas são independentes através dessa interface [17].

Os sistemas de trocas de mensagens (mediadores) introduzem um intermediário entre os dois sistemas em comunicação. Este intermediário permite tornar ainda mais independentes o remetente do recetor (ou recetores) da mensagem. Isto permite ao remetente enviar uma mensagem sem saber onde está o recetor da mesma, para além de remover a necessidade de saber se este está ativo ou se existe mais do que uma instância do mesmo.

Podemos considerar três tipos de modelos de troca de mensagens em sistemas deste tipo, que de seguida são explicados.

7 A partir deste momento, *broker* é tratado como 'mediador'.

Modelo Point-to-Point

Uma boa metáfora para perceber o domínio de mensagens ponto-a-ponto é comparar o mesmo a um serviço de correios que usamos no nosso dia-a-dia. Quando queremos enviar uma encomenda para alguém dirigimo-nos aos correios e entregamos o objeto em questão, recebendo um comprovativo dessa mesma ação. Com isto, a pessoa para quem queremos enviar a encomenda não precisa de estar em casa no momento em que a mesma é enviada, pois acreditamos que a mesma vai ser entregue em algum momento no futuro. Sendo assim, mais tarde, a encomenda acaba por ser recebida tal como esperado.

Com este exemplo, conseguimos perceber que o serviço de correios funciona como um mecanismo de distribuição de encomendas, que separa o ato de envio do recebimento da mensagem e que garante que cada parcela é entregue (apenas uma vez).

Algumas caraterísticas de um domínio ponto-a-ponto são [17, 18]:

- A implementação é feita com recurso a filas (*queues*), que atuam como um *buffer* First In, First Out (FIFO).
- Mais do que um consumidor pode subscrever uma fila.
- Cada mensagem é entregue apenas a um dos consumidores subscritos, como mostra a Figura 9.
- A escolha sobre qual o consumidor a receber uma mensagem é sempre de forma a tentar que a distribuição das mensagens seja o mais justa possível.
- Estas filas têm durabilidade, e, sendo assim, o sistema de troca de mensagens retém as mensagens no caso de inexistência de pelo menos um subscritor da mesma, até que um consumidor realize a subscrição e as receba.

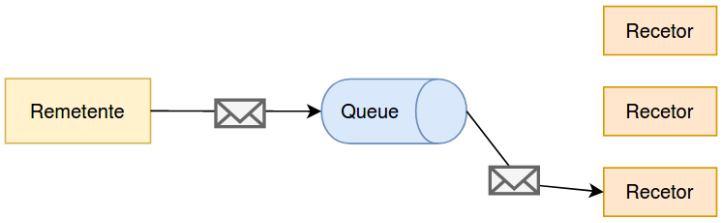


Figura 9: Modelo de mensagens *point-to-point*.

Modelo Publish-Subscribe

Outro exemplo para compreender este domínio de mensagens é comparar o mesmo a uma transmissão em direto que é feita através de uma plataforma como o YouTube ou o Twitch. Enquanto nos ligamos à transmissão, conseguimos ver o que está a acontecer na mesma, assim como as outras pessoas a ver a emissão durante o mesmo momento. Se sairmos da emissão deixamos de ver aquilo que está a acontecer. Se mais tarde nos voltarmos a ligar à transmissão, vamos poder voltar a ver o que se passa na mesma.

Sendo assim, neste exemplo a transmissão de um direto funciona como um mecanismo de *broadcast*. A pessoa que está a fazer a transmissão não tem que se preocupar com as pessoas ou com a quantidade das mesmas que estão a ver a transmissão, isto porque o sistema garante que qualquer pessoa que esteja na emissão vai ver o que está a acontecer.

As caraterísticas essenciais deste domínio de mensagens podem ser [17, 18]:

- É implementado através de tópicos, que são na prática o meio de comunicação.
- Um tópico pode ter múltiplos subscritores.
- Uma mensagem enviada para um tópico é distribuída para todos os subscritores do mesmo, como mostra a Figura 10.
- Os tópicos normalmente não têm durabilidade, ou seja, da mesma forma que no exemplo de cima quando saímos da emissão deixamos de ver o que está a acontecer na mesma, os subscritores de um tópico perdem todas as mensagens que são enviadas quando estes se encontram em baixo. Com isto, podemos concluir que os tópicos garantem que no máximo as entregas são feitas uma vez para cada consumidor (*at-most-once delivery guarantee*). Ainda assim, os tópicos podem também ter durabilidade.
- Este tipo de mecanismo de troca de mensagens é geralmente utilizado em situações nas quais as mensagens tendem a ser relacionadas com uma natureza ligada a informações, onde a perda de algumas mensagens esporadicamente pode não ser muito grave.

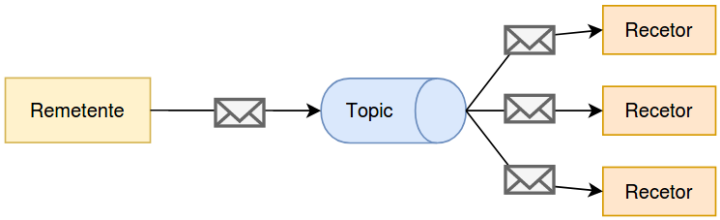


Figura 10: Modelo de mensagens *publish-subscribe*.

Modelos híbridos

Por vezes, certos contextos de problemas exigem uma combinação dos modelos *publish-subscribe* e *point-to-point*. Um exemplo pode ser quando múltiplos sistemas necessitam de uma cópia de uma mensagem e necessitam tanto de durabilidade como persistência de forma a poder prevenir a perda de mensagens [17].

Estes casos exigem uma fila ou um tópico que distribua as mensagens como um tópico, de forma a que cada mensagem seja enviada a um sistema diferente interessado nessas mensagens, mas em que cada sistema pode definir múltiplos consumidores que consomem as mensagens que vão chegando, tal como uma fila (*once-per-interested-party*) [17]. Tal como referido anteriormente, estes destinos híbridos frequentemente requerem durabilidade, garantindo que se um consumidor ficar inativo, as mensagens enviadas durante esse espaço temporal vão ser recebidas assim que o mesmo volte a ficar ativo.

Nas próximas secções são estudadas duas das tecnologias mais utilizadas como mediadores no desenvolvimento de aplicações: Apache ActiveMQ e Apache Kafka.

2.3.2 *Apache ActiveMQ*

ActiveMQ foi desenhado para implementar a especificação Java Message Service (JMS) (cuja vista geral pode ser vista na Figura 11), e, desta forma, tal como todos os sistemas de troca de mensagens que implementam esta especificação, é composto por:

- *Broker* - Componente central responsável pela distribuição das mensagens.
- *Client* - Componente de *software* que troca mensagens com recurso a um *broker*, sendo o mesmo composto por três elementos:
  - Código, que utiliza a API do JMS.
  - A API do JMS, conjunto de interfaces que de acordo com as garantias estabelecidas na especificação JMS permitem interagir com o mediador.
  - A biblioteca do sistema do cliente, que oferece uma implementação da API e comunica com o mediador.

A comunicação entre estes componentes é realizada com recurso a um protocolo da camada de aplicação, denominado *wire protocol*, e que oferece os meios para a interoperabilidade entre aplicações numa rede.

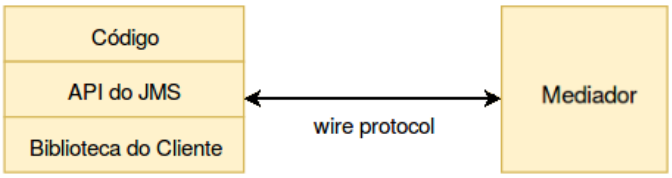


Figura 11: Vista geral de uma especificação JMS.[8]

Quanto a este protocolo para comunicação entre o cliente e o mediador, é deixado de fora da especificação JMS, de forma a que os mediadores que já existam se possam tornar compatíveis com esta especificação. Com isto em conta, o ActiveMQ definiu o seu próprio protocolo, o OpenWire. Ainda assim, ao longo do tempo o suporte para outros protocolos foi adicionado a esta ferramenta, de forma a aumentar a interoperabilidade com outros ambientes e linguagens, tais como os protocolos como AMQP 1.0[9], STOMP[10], XMPP[11] ou MQTT[12] [17].

Quando se define o sistema de mensagens a implementar, é necessário perceber o **equilíbrio** que existe entre o **desempenho e a confiabilidade** na solução a escolher. Quando se interage com dados, como por

8 Figura baseada em [Figure 2-1] de 'Understanding Message Brokers' ([17]).
9 https://activemq.apache.org/amqp
10 https://activemq.apache.org/stomp
11 https://activemq.apache.org/xmpp
12 https://activemq.apache.org/mqtt

exemplo de forma mais comum na interação com bases de dados, é necessário especificar o comportamento a ter se o sistema falhar (por exemplo, por falta de energia ou algo semelhante). Quando falamos de sistemas de troca de mensagens, este é também um aspeto a considerar. Assim, se os dados estiverem em memória (por exemplo em cache) e a máquina falhar, então os dados são perdidos, no entanto, se os dados estivessem guardados num armazenamento não-volátil, como um disco, então os mesmos voltavam a ser acessíveis quando o sistema volta a estar disponível. Se o problema fosse tão simples, então a solução passava por escrever as mensagens num disco (por exemplo) de forma a não perder as mesmas se um mediador ficar indisponível, o problema é que o custo desta decisão é muito alto, visto que escrever em disco é muito mais lento do que escrever na memória. Sabendo isto, cabe a quem desenvolve uma aplicação perceber se compensa a confiabilidade das mensagens face ao impacto que a mesma tem no desempenho do sistema. Podemos afirmar que quanto maior a confiabilidade, menor o desempenho [17].

O ActiveMQ oferece várias possibilidades para **persistir mensagens**, tanto opções baseadas em persistência em disco como KahaDB ou LevelDB, assim como a possibilidade de usar uma base de dados através de Java Database Connectivity (JDBC), sendo que as formas de persistência em disco são as mais utilizadas. As mensagens a persistir recebidas pelo mediador são escritas no fim de um *journal* [13], uma estrutura de dados baseada em arquivos. Esta estrutura contém um histórico de todas as mensagens recebidas assim como informações das mensagens que foram consumidas pelos clientes. De forma a não estar sempre a aumentar o espaço em disco utilizado para este efeito, quando todas as mensagens de um ficheiro do jornal são consumidas, então o mesmo é apagado ou arquivado por um *worker* em segundo plano. Em caso de falha do mediador, o ActiveMQ consegue refazer o jornal com base na informação dos seus ficheiros. Um problema a ter em conta deste mecanismo é que, como as mensagens de todas as filas são escritas nos mesmos ficheiros, isto significa que se uma única mensagem não for consumida, então esse ficheiro não pode ser removido.

Para se perceber o funcionamento do ActiveMQ, pode-se dividir o mesmo em duas partes: a parte responsável por aceitar mensagens dos produtores e a outra responsável pela entrega dessas mensagens aos consumidores.

A Figura 12 demonstra o processo de **produção de mensagens** e inserção das mesmas para uma fila (utilizando um modelo *point-to-point*). Este conjunto de passos é iniciado com o empacotamento da mensagem para o formato OpenWire, sendo então enviado para o mediador (1), que trata de desempacotar o mesmo para um estado interno de representação. O objeto da mensagem é então passado para o adaptador de persistência, que empacota a mensagem utilizando o formato Google Protocol Buffers e a escreve em disco [17]. Assim que a mesma é escrita, o adaptador necessita de uma confirmação de que a mensagem foi de facto persistida (3), sendo esta tipicamente a parte mais lenta de toda esta interação. Assim que o mediador sabe que a mensagem foi guardada, responde com um *acknowledgement* para o cliente (4), estando assim a *thread* que invocou esta operação livre para continuar o seu trabalho. O papel deste *acknowledgement* é importante, visto que, se pretendemos que uma mensagem seja persistida, é de pressupor que seja importante perceber também se a mesma foi aceite pelo mediador numa primeira fase.

13 Por uma maior comodidade, *journal* a partir deste ponto será tratado como 'jornal'.

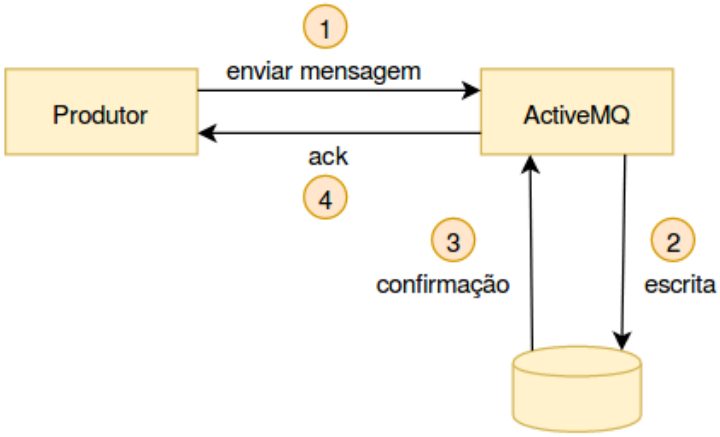


Figura 12: Produção de mensagens em JMS.[14]

A utilização de um único jornal para todas as filas adiciona complexidade à solução, visto que podem existir vários produtores a enviar mensagens para o mediador. Dentro deste, existem várias *threads* que recebem as mensagens e necessitam de as persistir no jornal mas, como não é possível que todas escrevam simultaneamente (visto que as mesmas iriam criar conflitos entre elas), as escritas necessitam de ser colocadas numa fila através de um mecanismo de exclusão mútua - este processo tem o nome de *thread contention*. Cada mensagem deve ser escrita e sincronizada antes de qualquer outra poder ser processada e, esta limitação pode levar a concluir que o tempo de aceitação de uma mensagem é igual ao tempo de escrita em disco a somar ao tempo de espera para que outras *threads* façam as suas escritas [17].

Tendo em conta este problema, o ActiveMQ inclui um *buffer* de escritas para o qual as *threads* que recebem as mensagens escrevam as mesmas enquanto esperam que a operação de escrita anterior seja concluída. Este *buffer* é depois então escrito numa única operação na próxima vez que a mensagem está disponível. Quando completado, as *threads* são notificadas, sendo assim maximizada a utilização da largura de banda do armazenamento.

Outro ponto importante a analisar passa pelas **transações** a efetuar pelo mediador. Na perspetiva dos autores do mediador em questão (o ActiveMQ), é muito mais simples implementar transações ao utilizar apenas um jornal.

14 Figura baseada em [Figure 2-3] de 'Understanding Message Brokers' ([17]).

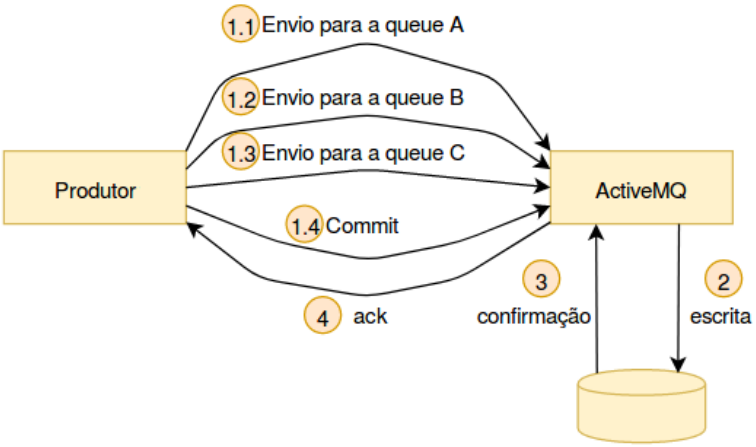


Figura 13: Exemplo de transação em ActiveMQ.[15]

O exemplo mostrado na Figura 13 permite descrever o conjunto de passos numa transação em ActiveMQ. Imaginemos que pretendemos enviar três mensagens para três filas distintas (1.1, 1.2, 1.3), ao usar uma transação obrigatoriamente o conjunto de mensagens tem que ser tratado como uma única operação atómica e, para além disso, a biblioteca do cliente ActiveMQ é capaz de fazer algumas otimizações que permitem aumentar o desempenho da operação de envio. Num processo de transação, ao invés de cada mensagem ter um *acknowledgement*, as três mensagens são enviadas assincronamente, não esperando pela resposta. Estas mensagens são mantidas em memória pelo mediador e, quando a operação é concluída, o produtor faz então *commit* (1.4), que leva o mediador a efetuar uma única (e grande) escrita com uma única operação de sincronização. Este tipo de abordagem permite remover o tempo de espera antes que o próximo envio seja possível no produtor, assim como combinar várias operações de disco pequenas numa maior [17].

Quanto ao **consumo das mensagens** de uma fila por parte dos consumidores, a Figura 14 ilustra este mesmo processo num mediador que respeite a especificação JMS.

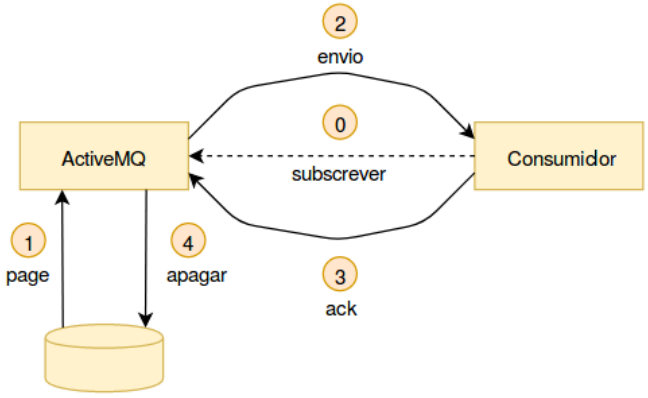


Figura 14: Consumo de mensagens em JMS.[16]

15 Figura baseada em [Figure 2-4] de 'Understanding Message Brokers' ([17]).
16 Figura baseada em [Figure 2-5] de 'Understanding Message Brokers' ([17]).

Quando o ActiveMQ está ciente de um consumidor, pagina as mensagens do disco para a memória para distribuição (1). Essas mensagens são depois enviadas ao consumidor (2) (sendo por vezes enviadas em conjuntos para minimizar a sobrecarga sobre a rede), sendo mantido um rastreamento de quais as mensagens que foram enviadas e para quem [17]. O consumidor quando recebe as mensagens não as processa imediatamente, pois são colocadas numa área em memória denominada *prefetch buffer*. Esta área é utilizada para conseguir uniformizar o fluxo de mensagens para que o mediador possa alimentar o consumidor com mensagens assim que as mesmas ficam disponíveis para envio, enquanto o consumidor processa assim as mensagens de uma forma ordenada, uma de cada vez. Quando finalmente uma mensagem é processada pela lógica da aplicação do consumidor, um *acknowledgement* é enviado de volta para o mediador (3), como comprovativo da mesma ter sido consumida. Quanto a este *ack*, o modo de ordenação numa linha cronológica entre o processamento da mensagem e o seu *acknowledgement* é configurável, sendo esta definição denominada *acknowledgement mode*. O modo de `AUTO_ACKNOWLEDGE` é o mais comum, sendo o comprovativo enviado ao mesmo tempo que a mensagem é consumida; o modo `CLIENT_ACKNOWLEDGE` envia apenas o comprovativo quando o consumidor chama explicitamente o método `acknowledge()` e por último, no modo `DUPS_OK_ACKNOWLEDGE` os *acks* vão sendo acumulados num *buffer* e enviados ao mesmo tempo. Por fim, assim que o mediador recebe este "comprovativo", a mensagem é removida da memória e apagada da *storage* (4).

É pertinente também ter em conta os casos em que existem **vários consumidores** a subscrever uma fila. Nestes casos, o comportamento do mediador por norma é distribuir as mensagens num mecanismo *round-robin* pelos consumidores que têm espaço nos seus *prefetch buffers* [17]. As mensagens são enviadas na ordem em que chegam à fila - a única garantia FIFO oferecida. Quando um consumidor recebe um conjunto de mensagens, mas fica indisponível e, por isso, não envia os *acknowledgements*, essas mensagens vão ser distribuídas para outro consumidor disponível. Sendo assim, isto levanta um ponto importante, que passa por não existir garantia de que uma mensagem não vai ser processada mais que uma vez (mesmo com o uso de transações). Com isto, as filas oferecem uma garantia de entrega *at-least-once*, pelo que pode ser necessário considerar partes de código para tratar a possibilidade de receber mensagens duplicadas (possível solução referida na secção 2.3.4).

Para os casos em que é necessário que algumas mensagens (ou todas) sejam **processadas por ordem**, o ActiveMQ oferece um mecanismo que ajuda a lidar com esta situação, denominado *JMS message groups*. Estes grupos de mensagens são um tipo de mecanismo de partição que permite que os produtores de mensagens categorizem as mesmas segundo grupos, de forma a que sejam processadas de acordo com uma *business key* (que toma o nome de *JMSXGroupID* nas propriedades da mensagem). Sendo assim, quando uma mensagem é processada e a sua *key* pertence a um grupo que ainda não tenha aparecido, esse grupo é delegado a um consumidor de acordo com um mecanismo *round-robin* e, a partir de então, todas as mensagens desse grupo vão ser enviadas para esse mesmo consumidor. No caso de um consumidor ficar indisponível, os grupos que estavam associados ao mesmo passam a ser realocados entre os consumidores disponíveis, e as mensagens para as quais não foram recebidas *acknowledgments* vão ser enviadas novamente, com o intuito de fazer com que todas as mensagens sejam realmente consumidas.

A tecnologia ActiveMQ oferece ainda **alta disponibilidade**. A mesma pode ser obtida através de um esquema *master-slave* com recurso a um armazenamento partilhado [17], como demonstrado na Figura 15. Com isto, é possível instalar dois mediadores (ou até mais) em diferentes servidores com as mensagens a serem persistidas num armazenamento externo. Este armazenamento não pode ser utilizado por mais do que um mediador ao mesmo tempo e por isso outra função associada ao mesmo é funcionar como um mecanismo de *locking* para determinar qual o mediador que tem acesso exclusivo ao mesmo.

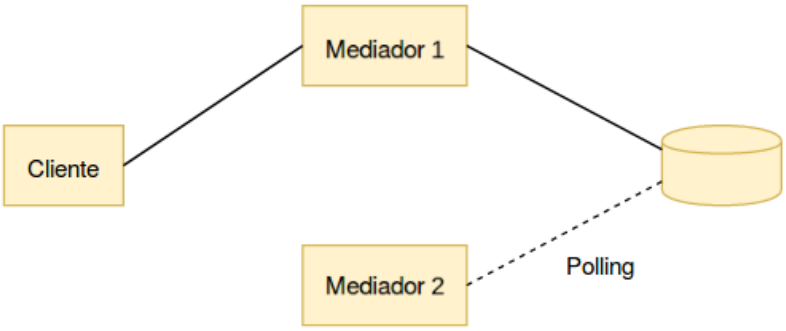


Figura 15: Solução de alta disponibilidade com dois mediadores.[17]

Enquanto que o Mediador 1 toma o papel de *master*, o Mediador 2 conecta-se ao armazenamento e tenta adquirir o *lock*. Não conseguindo, para por momentos e volta a tentar. A partir do momento em que o Mediador 1 falhar, liberta o *lock* e finalmente o segundo mediador consegue adquirir o mesmo, tomando agora ele o papel principal e fazendo com que o sistema continue disponível. Desvantagens que podem ser apresentadas a esta solução de alta disponibilidade passam pela necessidade de ter vários servidores físicos para funcionar como um único mediador lógico, e também a necessidade de que o armazenamento do mediador seja também ele altamente disponível (seja um *file system* ou uma base de dados).

Outro aspeto a ter em conta quando se utilizam ferramentas como esta é ter a noção e a capacidade de análise para perceber qual a melhor arquitetura para **escalar** o sistema. E com isto, um problema muito comum no desempenho de um mediador passa simplesmente por tentar fazer mais do que aquilo que uma única instância pode oferecer. Muitas vezes o problema começa logo numa fase preliminar do desenvolvimento, aquando da decisão de realizar a comunicação de várias aplicações de forma assíncrona via ActiveMQ, e a solução proposta passar pela solução base de um único mediador físico para dar conta de todas as mensagens - abordagem conhecida como *Universal Data Pipeline*. Com isto, a sugestão passa por analisar cada caso, verificando parâmetros como o fluxo das mensagens, o volume de mensagens esperado em cada fila assim como o tamanho das mesmas, se os sistemas comunicam remotamente, etc. A ideia passa por identificar *use cases* individuais de troca de mensagens que possam ser combinados ou separados em diferentes mediadores.

17 Figura baseada em [Figure 2-6] de 'Understanding Message Brokers' ([17]).

2.3.3 *Apache Kafka*

O Kafka surgiu de forma a poder resolver algumas das limitações dos mediadores de mensagens tradicionais, e evitar que seja necessário utilizar diferentes mediadores para diferentes contextos ponto-a-ponto [17], como falado anteriormente. No fundo, o Kafka tenta tornar possível utilizar uma arquitetura de mensagens como definido em *Universal Data Pipeline* [17].

O Apache Kafka é a tecnologia mais popular utilizada para ingerir fluxos de dados em plataformas de processamento [19].

De forma a conseguir concretizar o efeito pretendido, é então preciso que suporte tanto *publish-subscribe* como *point-to-point*, seja bastante rápido e que permita um grande processamento de mensagens num curto espaço de tempo. Para além disto, ao contrário do que acontece com o ActiveMQ, quando o número de consumidores num destino aumenta o desempenho não deve diminuir, deve ser escalável horizontalmente e deve permitir a retenção e repetição de leitura de mensagens [17].

Claro que para concretizar isto, tem um impacto naquilo que é o trabalho tanto da parte do cliente como do mediador. Enquanto que o modelo JMS é centrado na ação do mediador e os clientes apenas precisam de enviar e receber mensagens, o Kafka é mais centrado no cliente, pelo que o mesmo tem que fazer certos trabalhos que normalmente seriam realizados pelo mediador, de forma a que o mesmo seja mais rápido.

O Kafka **unifica** os modelos *point-to-point* e *publish-subscribe* em apenas um tipo de destino - **topic** [18]. Os tópicos do Kafka devem ser considerados como um modelo híbrido, mencionado anteriormente na secção 2.3.1.

Ao contrário do que acontece com o ActiveMQ, cada tópico tem o seu próprio jornal. Sendo assim, quando um produtor envia uma mensagem para o mediador, este anexa a mesma ao jornal respetivo, sendo que os consumidores leem as mensagens a partir do jornal com a utilização de apontadores, que vão continuamente sendo incrementados. De forma periódica, as partes mais antigas do jornal vão sendo apagadas, independentemente das mesmas terem sido ou não consumidas - a responsabilidade da leitura das mesmas pertence ao cliente.

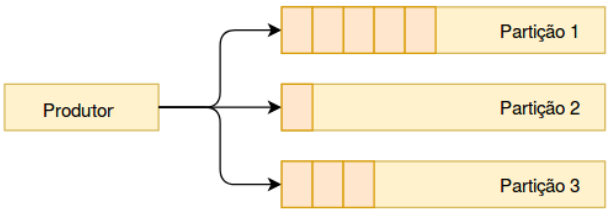


Figura 16: Partições do Kafka.[19]

Um jornal é composto por várias **partições** (demonstrado na Figura 16), existindo uma garantia de ordem em cada partição. Cada partição é como um *log* contínuo que contém um subconjunto de todas as mensagens enviadas para o tópico pelos seus produtores. Quando uma mensagem é enviada para um tópico, é decidida

18 A partir deste momento, *topic* é tratado como 'tópico'.
19 Figura baseada em [Figure 3-1] de 'Understanding Message Broker' ([17]).

qual a partição para a qual a mensagem deve ser enviada. A ideia das partições é fulcral para a capacidade de escalar horizontalmente, sendo que cada tópico criado tem apenas uma partição por padrão.

Para um cliente **consumir mensagens** o mesmo tem um ponteiro denominado *consumer group*, que corresponde a um único consumidor lógico ou grupo de consumidores. Este ponteiro aponta para um *offset* de mensagens numa partição. Neste contexto de consumo de mensagens, a dificuldade passa por quando vários sistemas consomem mensagens através de várias instâncias e *threads*, o que faz com que várias instâncias partilhem o mesmo *consumer group*. Para além disto, tem que se cobrir ainda a possibilidade de vários *consumer groups* consumirem do mesmo tópico ao mesmo tempo.

O caso mais simples é perceber o funcionamento de um tópico com apenas uma partição. Quando um consumidor se conecta com o identificador do seu grupo a um tópico, é-lhe atribuída uma partição e um *offset* da mesma. Esse *offset* pode ser configurado de forma a apontar para a última posição ou para a primeira, que correspondem respetivamente às mensagens mais recentes e às mais antigas, sendo a leitura feita de forma sequencial. Com isto, o cliente pode também andar para trás no *offset* se quiser reprocessar mensagens que já viu. Quando outro consumidor se liga com outro identificador de grupo, ele tem outro ponteiro, que é independente do outro consumidor, como apresentado na Figura 17. Desta forma, um tópico Kafka funciona como uma fila quando apenas tem um consumidor, e como um tópico *publish-subscribe* com mais do que um consumidor. Para além disto, as mensagens são persistidas e podem ser processadas várias vezes [17].

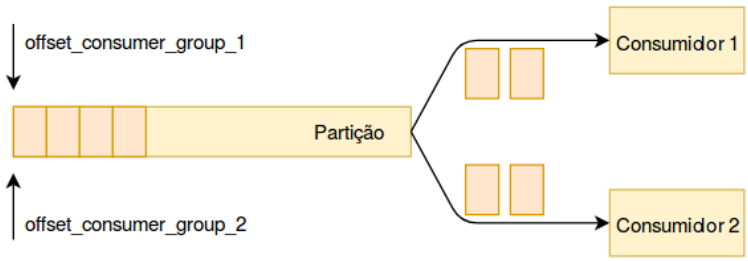


Figura 17: Dois consumidores de diferentes *consumer groups* a ler mensagens através da mesma partição.[20]

Quando mais do que uma instância com o mesmo identificador de grupo se ligam a um tópico com uma partição, a instância que se conecta em último lugar é que passa a ter controlo sobre o ponteiro, e passa a ser esta instância a receber todas as mensagens daí em diante. Aqui pode-se ver uma diferença em relação à especificação JMS, pois nesse modelo as mensagens enviadas numa fila seriam partilhadas uniformemente pelos dois consumidores.

Mas e se pretendermos ter várias instâncias do mesmo grupo para adicionar paralelismo no processamento das mensagens? Uma possível solução passa por usar um único consumidor de todas as mensagens e encarregar o tratamento das mesmas a uma *pool* de *threads*. Esta é mais uma situação em que se verifica um aumento da complexidade do lado do cliente. O problema nesta solução é que se o consumidor for abaixo, então o consumo das mensagens para. A forma recomendada de resolver isto no Kafka é através da utilização de mais partições.

20 Figura baseada em [Figure 3-3] de 'Understanding Message Broker' ([17]).

As **partições** são o mecanismo primário a utilizar para paralelizar o consumo de mensagens e escalar um tópico para além dos limites que um único mediador oferece [17].

No caso em que um tópico tem por exemplo duas partições, mas apenas um consumidor subscrito ao mesmo, o consumidor tem controlo dos apontadores correspondentes ao seu identificador de grupo em cada uma das partições, consumindo mensagens de ambas.

Se outro consumidor subscrever o tópico que pertença ao mesmo grupo, então uma das partições passa a estar sob a alçada do novo consumidor, estando assim um consumidor a ler mensagens de uma única partição do tópico. Sendo assim, se pretendemos paralelizar o processamento de mensagens por *X threads* necessitamos então de *X* partições [17].

Com isto, podemos reconhecer que a complexidade é inferior no Kafka em comparação com JMS na distribuição de mensagens, visto que o mediador só tem que se preocupar em alimentar de forma sequencial para um consumidor conforme o mesmo vai requisitando. Ainda assim, os requisitos para paralelizar o processamento e a repetição de mensagens mantêm-se, só que passam para o lado do cliente.

No que diz respeito à parte do **envio de mensagens** para o Kafka, e no caso de um tópico ter mais que uma partição, a decisão de saber qual a partição para a qual a mensagem é enviada é da responsabilidade do produtor da mensagem [17, 20]. Quando a mensagem é enviada utilizando a API Kafka Producer, esta é entregue a um método de particionamento que dada a mensagem e o estado do *cluster* retorna o identificador da partição para a qual deverá ser enviada a mensagem. Esta estratégia por norma funciona bem na maioria dos casos. Ainda assim, pode ser necessário estabelecer um estratégia de particionamento própria, até porque o Kafka não disponibiliza um mecanismo para transmitir metadados, apenas uma chave e um valor (*key-value*).

Ainda assim, um ponto importante sobre o envio de mensagens para o *Kafka*, é que a operação de envio não é feita de imediato. O que acontece é que a mensagem é inserida num *buffer* para cada partição ativa e transmitida para o mediador com recurso a uma *thread* em segundo plano dentro da biblioteca Kafka do cliente. Um problema associado a isto é que, enquanto que isto torna a operação muito rápida, uma aplicação "ingénua" pode perder mensagens se o processo for abaixo. Claro que isto pode ser contornado, mas sempre com um custo no desempenho, colocando o tamanho do *buffer* a zero e ao enviar a mensagem obrigar a *thread* a esperar que a operação seja concluída.

Enquanto que no JMS é definido um modo de *acknowledgment*, no Kafka, tal como referido anteriormente, as mensagens não são apagadas quando são consumidas, e a responsabilidade de saber o que fazer em caso de falha cabe ao código do lado do consumidor. Neste caso, o ponteiro é o *offset* no *log*, cuja posição corresponde à próxima mensagem a ser processada em resposta a `poll()`. A parte crítica neste processo é a **altura em que esse *offset* é incrementado**. Em Kafka, esta ação de *commit* é algo que pode ser chamado pelo cliente periodicamente. Para isto, duas propriedades definem este comportamento: `enable.auto.commit` e `auto.commit.interval.ms`. Por exemplo, se a primeira propriedade tiver o valor `true` (como tem por defeito) e a segunda o valor de `5000`, então a cada 5 segundos o consumidor vai fazer *commit* do seu *offset* ao Kafka, ou cada vez que os dados forem obtidos do tópico em questão vai enviar o seu *offset* mais recente [21].

Imaginemos que o consumidor está a processar uma mensagem e enquanto a processa recebe mais mensagens, o *offset* é submetido e o consumidor falha. Quando voltar ele vai consumir mensagens tendo em conta o *offset* mais recente, mas como temos certeza que não perdemos mensagens e que o *offset* da nova mensagem não é posterior ao da mensagem processada? O que se pode fazer é que o *commit* seja realizado de forma manual após o processamento das mesmas. Sendo assim, o valor de `enable.auto.commit` é colocado a `false` e, assim, se o consumidor fizer *commit* após o processamento da mensagem e ficar indisponível enquanto processa uma mensagem ele vai voltar a consumir a partir do mesmo *offset*, não perdendo assim qualquer mensagem [21].

Outro ponto importante é que o Kafka é **não-transacional**, ou seja, não é possível enviar mensagens para vários tópicos dentro da mesma operação atómica ou até reverter automaticamente uma mensagem que falhou. Se existe a possibilidade que o *offset* de um consumidor seja incrementado antes da mensagem ser processada, então o consumidor não consegue saber se o seu grupo de consumidores perdeu mensagens quando ocorre a atribuição de uma partição. Uma abordagem passa por fazer um "rebobinar" do *offset* para uma posição anterior, o que ainda assim faz com que haja uma grande possibilidade que algumas mensagens anteriormente processadas voltem a ser processadas. Outra abordagem é assumir que simplesmente perder algumas mensagens ou processar mensagens mais do que uma vez é aceitável, casos em que mensagens individuais não têm um grande impacto no sistema, o que não é aceitável por exemplo em casos de pagamentos ou transações monetárias.

A abordagem do Kafka para ser possível obter uma **alta disponibilidade** é significativamente diferente do ActiveMQ. Este mediador é desenhado à volta de *clusters* horizontalmente escaláveis nos quais cada instância aceita e distribui mensagens ao mesmo tempo [17].

Cada *cluster* Kafka é composto por várias instâncias do mediador a serem executadas em diferentes servidores, sendo que cada nodo deve ter o seu próprio armazenamento dedicado. Os mediadores estão ligados a um *cluster* de servidores ZooKeeper que é utilizado para coordenar o papel de cada mediador (o próprio ZooKeeper é um sistema distribuído e altamente disponível).

Quando um tópico Kafka é criado no *cluster* são definidas as seguintes propriedades: o seu número de partições (como falado anteriormente) e o fator de replicação, sendo este valor o número de instâncias no *cluster* que devem conter *logs* desta partição. Com a utilização do ZooKeeper, as partições são então distribuídas de forma justa pelos mediadores do *cluster* - tarefa desempenhada por uma instância que tem o papel de supervisor. Este supervisor define ainda em tempo de execução para cada partição de um tópico, quais os mediadores que funcionam como líder (*master*) ou *slaves*. Este mediador líder é responsável por receber e distribuir mensagens. Neste contexto, quando as mensagens são enviadas para uma partição de um tópico, são replicadas para todos os *slaves*, sendo que cada nodo que contem os *logs* para a partição são réplicas. Cada servidor pode atuar como um líder para algumas partições e *follower* (*slave*) de outras partições ao mesmo tempo [22].

Parte relevante aqui é que se o líder ficar indisponível, qualquer mediador que esteja atualizado ou em sincronização para aquela partição pode tomar este mesmo papel de maior relevância. Uma configuração muito utilizada na criação de um tópico é definir o fator de replicação com o valor 3 (1 líder e 2 *followers* para cada

partição), definindo a propriedade `min.insync.replicas` a `2`, de forma que o *cluster* fique tolerável à falha de um mediador sem que isso tenha impacto nos clientes [17].

Em jeito de conclusão neste tópico, é possível obter melhor desempenho através da utilização de *clusters* Kafka comparativamente a um único mediador, escalando de forma horizontal as partições de um tópico por várias máquinas [17].

2.3.4 *Enterprise Integration Patterns*

Enterprise Integration Patterns (EIPs) são soluções testadas para problemas de conceção específicos encontrados ao longo dos anos no desenvolvimento e integração de aplicações informáticas. Um ponto importante de referir é que estes padrões são independentes das tecnologias, ou seja, não são específicos de uma linguagem de programação ou até mesmo de um sistema operativo [23].

Um dos padrões denominado ***Idempotent Receiver*** (do tipo *Messaging Endpoints*) ajuda a resolver o problema dos consumidores receberem a mesma mensagem mais do que uma vez. Isto acontece porque quando uma aplicação envia uma mensagem apenas uma vez, a aplicação recetora pode ainda assim receber a mesma mais do que uma vez [24], como se observa no exemplo da Figura 18.

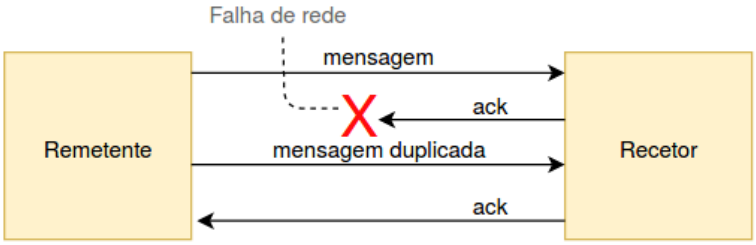


Figura 18: Envio de mensagem duplicada devido a falha no envio de *acknowledgment*.[21]

Sendo assim, com este padrão a ideia passa por conceber um recetor de mensagens no qual a receção mais do que uma vez de uma mensagem seja segura. Para que tal seja possível são sugeridas duas alternativas de abordagem ao problema [24]:

- O recetor pode manter um histórico de mensagens que já recebeu. Isto pode ser conseguido com cada mensagem a ter um identificador único, que ajuda a detetar quais as mensagens com o mesmo conteúdo. Muitos sistemas de troca de mensagens, como as ferramentas compatíveis com JMS, automaticamente atribuem uma identificação a cada mensagem, sem a aplicação ter que se preocupar com o mesmo. Sendo assim, o recetor deve manter um *buffer* no qual mantém os identificadores das mensagens já recebidas.

- Outra abordagem passa por definir uma semântica para as mensagens de forma a que o reenvio de uma mensagem não tenha impacto no sistema. Por exemplo, em vez de definir uma mensagens como

21 Figura baseada em [página 469] de 'Enterprise Integration Patterns' ([24]).

"Adicionar 5€ à conta com o id 100", podemos mudar a mensagem para "Definir 15€ como o saldo da conta com o id 100", em que ambas as mensagens têm o mesmo resultado se o saldo atual da conta for de 10€. A segunda mensagem é segura porque receber a mesma duas vezes não ia ter qualquer efeito. Claro que este exemplo serve apenas para se perceber a ideia, pois ignora situações de concorrência em que podia receber uma mensagem do género "Definir 25€ como o saldo da conta com o id 100" entre a mensagem original e a duplicada.

Outro padrão relevante é o ***Message Channel*** (do tipo *Messaging System*), que descreve um canal lógico que é usado para conectar aplicações, como apresentado na Figura 19. Uma aplicação escreve mensagens no canal e a outra (ou outras) lê(em) essa mensagem do canal [24]. Filas e tópico como falados anteriormente são exemplos de canais de mensagens.

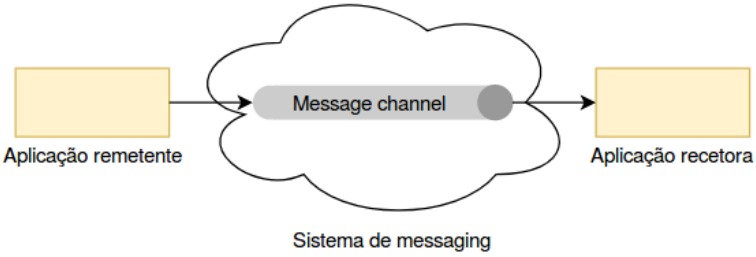


Figura 19: *Message Channel pattern*.[22]

Os próprios modelos de *point-to-point* e *publish-subscribe* são *patterns* englobadas no EIPs, denominadas ***Point-to-Point Channel*** e ***Publish-Subscribe Channel*** respetivamente, ambas da categoria *Messaging Channels*.

2.4 CACHING

Caching é o termo atribuído ao processo de armazenamento de dados em *cache* [25]. Este mecanismo pode servir para melhorar o tempo de carregamento de páginas e serve também para reduzir a carga em servidores e bases de dados [26]. Por norma, a *cache* é uma área de armazenamento de dados extremamente rápida com um tamanho de memória não muito grande (a maioria das vezes) e bem definido.

Com a introdução de um componente deste tipo, a ideia passa por verificar se os dados requisitados já se encontram primeiramente em *cache*. Se os dados estiverem em *cache*, estes são obtidos a partir da mesma (resultantes de um pedido anterior) e devolvidos, caso contrário, é então necessário obter os dados em causa recorrendo à base de dados. Isto leva também a perceber o significado de *cache hit* e *cache miss*. *Cache hit* é quando um cliente faz um pedido de conteúdo à *cache* e esta tem esses dados guardados, já quando a *cache* não tem os dados ocorre um *cache miss* [27].

Podemos identificar algumas vantagens na utilização destes mecanismos [28]:

22 Figura baseada em [página 77] de 'Enterprise Integration Patterns' ([24]).

- Aumento da responsividade, ao permitir que o conteúdo seja obtido mais rapidamente. *Caches* que se encontram perto do utilizador (como o *browser*) podem até tornar a obtenção de informações em *cache* quase instantâneas.
- Redução de utilização de rede, visto que o conteúdo pode ser guardado temporariamente entre o consumidor final e a origem do mesmo.
- Disponibilidade de informações durante interrupções de rede, visto que em certas abordagens o *caching* torna possível fornecer dados aos utilizadores mesmo quando os servidores de origem dos dados se encontram em baixo.

2.4.1 *Abordagens de caching*

Ainda assim, este tema tem muito por onde se explorar e muitas opções a diferentes níveis de abordagens [25, 26]:

- ***Client caching*** - a *cache* é colocada do lado do cliente, tal como no próprio sistema operativo, no *browser* que utilizamos (tal como o Chrome ou o Firefox) ou através da utilização de servidores que funcionem como um cliente, como um *Reverse-Proxy*.
- ***CDN caching*** - Content Delivery Network (CDN) encontram-se entre os clientes e os servidores, e são também utilizados como *cache*. Assim, um CDN guarda conteúdos tais como páginas, imagens ou vídeos em servidores *proxy* que se encontram mais perto dos clientes do que dos servidores requisitados, sendo assim possível devolver conteúdos mais rapidamente.
- ***Web server caching*** - Os servidores web podem também fazer *cache* de pedidos realizados, para que possam assim fornecer respostas mais tarde sem ter que interagir com os servidores aplicacionais.
- ***Database caching*** - Passa por ajustar as configurações de bases de dados para obter uma melhor performance para certos *use cases*.
- ***Application caching*** - Neste tipo de abordagem, a *cache* é colocada entre a aplicação e a base de dados. São por norma utilizados sistemas de *cache in-memory* tal como o Redis ou o Memcached que funcionam como armazenamentos de pares chave-valor. Sendo assim, os dados são mantidos na RAM que como é sabido é uma memória muito mais rápida que aceder a uma base de dados normal, em que os dados são guardados em disco. Por outro lado, a RAM é muito mais limitada (em espaço) que um disco, pelo que devem ser tidos em conta algoritmos de *cache invalidation*.

2.4.2 *Caching aplicacional*

Sendo que a abordagem de *application caching* é a que pode ser mais útil a um projeto deste tipo, é importante referir que existem dois padrões para realizar *caching* de dados nesta abordagem [25, 26]:

- Fazer *caching* de *queries* da base de dados, ou seja, quando se realiza um pedido à base de dados, faz-se um *hash* da *query* e guarda-se o resultado em *cache* para que num novo pedido possa esse resultado ser utilizado. Assim, cada vez que se correr a *query* é primeiro verificado se a chave já se encontra em *cache*. Se a mesma lá estiver devolve-se a mesma a partir daí, caso contrário obtém-se a resposta recorrendo à base de dados e guarda-se em *cache* para o futuro. O principal obstáculo deste padrão passa por perceber quando os dados guardados em *cache* são inválidos e devem ser removidos, pois, por exemplo, se existir alguma mudança na base de dados ao nível de uma coluna de uma tabela é necessário então remover da *cache* todas as informações relacionadas com essa coluna alterada.

- Fazer *caching* de objetos, que consiste em armazenar os dados como um objeto, tal como se faz ao nível do código da aplicação. A aplicação pode montar um *dataset* a partir da base de dados numa instância de uma classe ou numa estrutura de dados, e então guardar a mesma em *cache*. O obstáculo é semelhante ao do padrão anterior, visto que é necessário remover um objeto da *cache* se os dados subjacentes forem alterados. Exemplos de implementação deste padrão passam por armazenar sessões de utilizadores ou páginas web totalmente renderizadas.

2.4.3 *Estratégias de atualização da cache*

Existem várias estratégias para determinar o momento em que se fazem atualizações no sistema de *cache*, sendo necessário analisar as várias opções para determinar a que melhor se adapta a cada caso prático.

Cache Aside é a estratégia mais comum, em que a *cache* se encontra ao lado da base de dados. A aplicação pede primeiro os dados à *cache*, se a informação existir é devolvida diretamente (*cache hit*), caso contrário (*cache miss*) a aplicação vai requisitar os dados à base de dados e escrever os mesmos em *cache* para que possam ser utilizados numa próxima vez [29]. Este processo é esquematizado na Figura 20.

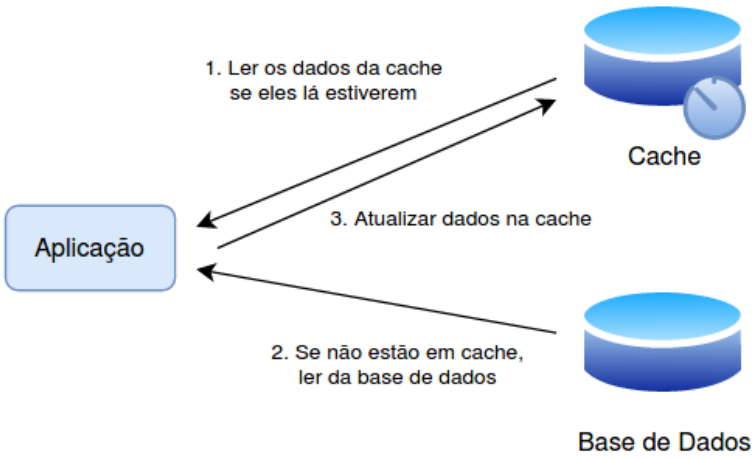


Figura 20: Estratégia *Cache Aside*.[23]

23 Figura baseada em 'Things You Should Know About Database Caching' ([29]).

Esta estratégia permite que a aplicação possa funcionar mesmo após uma falha no sistema de *cache* (sendo claro que o desempenho vai sofrer com isso), e permite, devido a um *lazy loading*, que apenas os dados requisitados sejam armazenados em *cache* [25]. Como desvantagens, no caso dos dados não estarem presentes na *cache*, isso implica que sejam feitas 3 viagens na rede, o que pode causar algum atraso, por outro lado, os dados podem tornar-se desatualizados se forem atualizados na base de dados, o que pode ser resolvido por exemplo ao adicionar um Time to Live (TTL) [26].

A estratégia ***Read Through*** coloca a *cache* entre a aplicação e a base de dados, sendo que a aplicação apenas requisita dados à *cache*, não interagindo diretamente com a base de dados. No caso dos dados não estarem em *cache*, esta é que é a responsável por obter os dados da base de dados, atualizar-se a si mesma e devolver as informações à aplicação [29]. Estes procedimentos são apresentados na Figura 21.

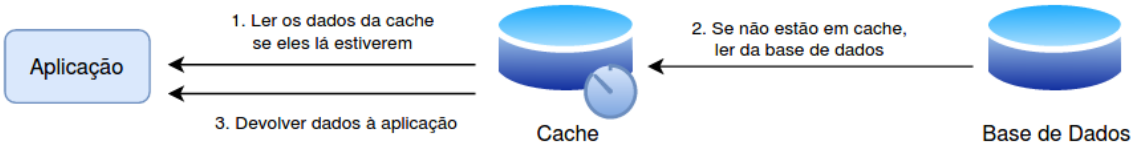


Figura 21: Estratégia *Read Through*.[24]

Ao nível das vantagens e desvantagens desta estratégia, esta é similar à estratégia anterior, apesar de, ao contrário da anterior, o modelo de dados na *cache read-through* não pode ser diferente da utilizada na base de dados [25].

Relacionado com estas duas estratégias anteriores aparece o padrão ***Write Around***, que normalmente combina com a estratégia *Cache Aside* ou *Read Through*, sendo que a aplicação escreve diretamente na base de dados e apenas os dados que são lidos vão para *cache* [29].

Outra opção passa pela estratégia ***Write Through***, em que a *cache* encontra-se também entre a aplicação e a base de dados. A aplicação utiliza a *cache* como a principal fonte de dados tanto para leituras como para escritas, sendo a *cache* responsável por ler e escrever na base de dados. O comportamento da aplicação passa por adicionar ou atualizar uma entrada na *cache*, o que leva esta a escrever de forma assíncrona na base de dados e depois devolver a entrada à aplicação [25]. Esta estratégia é esquematizada na Figura 22.

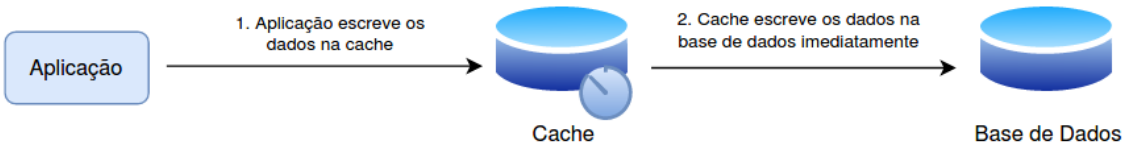


Figura 22: Estratégia *Write Through*.[25]

Como principal vantagem podemos dizer que os dados que se encontram em *cache* nunca estão desatualizados. Por outro lado, esta estratégia é mais demorada no geral devido à escrita de dados em primeiro lugar

24 Figura baseada em 'Things You Should Know About Database Caching' ([29]).
25 Figura baseada em 'Things You Should Know About Database Caching' ([29]).

na *cache* e depois de forma síncrona na base de dados. Outra desvantagem passa por a maioria dos dados presentes na *cache* poderem nunca vir a ser lidos, o que pode ser minimizado com a utilização novamente de um TTL ou expiração [25, 28].

Outra estratégia é denominada de ***Write Back*** ou ***Write Behind***, que é uma abordagem parecida à do *Write Through* em que a aplicação escreve dados na *cache*. A diferença está no *delay* na escrita da *cache* para a base de dados, que passa a ser realizada de forma assíncrona, o que leva a aumentar o desempenho das escritas [29, 25]. Esta estratégia é apresentada em baixo na Figura 23.

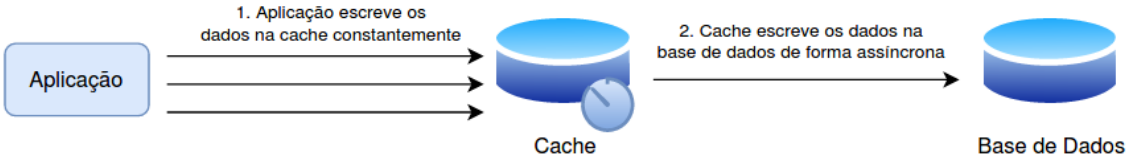


Figura 23: Estratégia *Write Back*.[26]

A principal vantagem deste padrão é, tal como referido, aumentar o desempenho das escritas da aplicação. Outra vantagem é que a aplicação está protegida em caso de uma falha na base de dados, pois os dados em fila podem ser lidos novamente ou recolocados em fila [28]. Por outro lado, um principal problema que podemos identificar passa pela possibilidade de perda de dados no caso da *cache* falhar antes de escrever na base de dados. Outra desvantagem evidente é que implementar este padrão é obrigatoriamente mais complexo que o padrão *Cache Aside* ou *Write Through*.

O último padrão é conhecido por ***Refresh Ahead***, que consiste em configurar a *cache* de maneira a que atualize automaticamente conteúdo acedido na *cache* antes da sua expiração [25].

A principal vantagem passa por reduzir a latência se a *cache* conseguir prever que dados podem vir a ser necessários no futuro. A principal desvantagem (para além de uma maior dificuldade de implementação) está relaciona com este mesmo assunto pois, se as previsões não forem acertadas isto vai levar a um desempenho inferior comparativamente a uma solução sem *Refresh Ahead* [25].

2.4.4 *Políticas de despejo*

Outro ponto que é importante realçar na utilização de mecanismos de *caching* passa pela escolha de uma política de despejo (*eviction policy*), que é responsável por garantir que o tamanho da *cache* não ultrapasse o seu limite máximo definido. Ora, para conseguir este efeito, elementos que se encontram em *cache* têm assim que ser removidos de acordo com a política escolhida [28].

Com isto, são de seguida apresentadas algumas políticas das mais conhecidas e utilizadas que tentam resolver este problema:

26 Figura baseada em 'Things You Should Know About Database Caching' ([29]).

- Least Recently Used (LRU), que consiste em eliminar os elementos que não foram utilizados recentemente. Tem a vantagem de ser uma abordagem perto do algoritmo ideal mas, por outro lado, apenas tem em conta metade do problema se assim podemos dizer, visto que a maior parte das vezes é mais importante o número de vezes que um elemento foi acedido do que quando foi acedido [28].

- Least Frequently Used (LFU), que consiste em eliminar os elementos que foram menos vezes usados em primeiro lugar. A diferença deste algoritmo em relação ao LRU é que, enquanto o LRU guarda o quão recentemente um elemento foi acedido, o LFU guarda o número de vezes que o elemento foi acedido [28].

- Most Recently Used (MRU), que consiste, ao contrário do algoritmo LRU, em eliminar os elementos mais usados recentemente em primeiro lugar. Este algoritmo é normalmente utilizado em contextos em que quanto mais antigo um elemento em *cache* for, maior a probabilidade de ser acedido [28].

- First In, First Out (FIFO), que pode ser considerado semelhante ao algoritmo MRU mas em que este segue estritamente a ordem de inserção dos elementos em memória, ao passo que o MRU não toma em conta a ordem de inserção [28].

- Last In, First Out (LIFO) que pode ser considerada a versão oposta do algoritmo FIFO, pois a *cache* elimina primeiro os elementos inseridos em memória mais recentemente.

De referir que para além das políticas aqui referidas existem ainda muitas outras tais como Time Aware Least Recent Used (TLRU), Segmented LRU (SLRU), Least Frequent Recently Used (LFRU) ou LFU with Dynamic Aging (LFUDA), que podem ser consideradas variantes de algumas apresentadas em cima.

2.4.5 *Comparação entre duas tecnologias: Redis e Memcached*

Passadas as explicações sobre os mecanismos de *caching*, importa então perceber um pouco sobre as tecnologias mais utilizadas habitualmente neste contexto. Desta forma, duas das ferramentas mais utilizadas para o efeito são o Redis e o Memcached.

Devido às caraterísticas destas tecnologias, quando se pensa em utilizar mecanismos de *caching* para melhorar o desempenho de uma aplicação, é normal comparar as duas para perceber qual a melhor opção para o contexto da aplicação a desenvolver.

Ambas as ferramentas servem como um armazenamento em memória de pares chave-valor, apesar do Redis ser descrito com mais precisão como um armazenamento de estruturas de dados [30]. Para além disto, ambas mantêm os dados em RAM, o que contribui para serem tão rápidos. Assim, tanto um como outro suportam respostas em tempos inferiores a milissegundos, lendo mais rapidamente que as bases de dados normalmente em disco [31]. Tanto o Redis como o Memcached podem ser considerados fáceis de utilizar e tendem a necessitar de pouco código para fazer a sua integração numa aplicação, isto também porque ambas têm vários clientes *open-source* disponibilizados, incluindo linguagens como Python, Java, JavaScript, etc [31]. Por último, ambas

permitem a distribuição dos dados por vários nodos, o que permite uma maior capacidade de escalar quando assim o for necessário [31].

Depois do que ambas têm em comum, é interessante perceber algumas caraterísticas de cada uma destas tecnologias.

Quanto ao **Memcached**, a sua gestão da memória interna é mais eficiente em casos mais simples, isto porque consome menos recursos de memória para metadados, sendo *strings* o único tipo de dados suportado por esta ferramenta [30]. Uma vantagem que pode ser apontada ao Memcached comparativamente ao Redis é a sua capacidade para escalar, pois sendo o Memcached *multithreaded*, isto significa que pode tratar mais operações ao fazer escalar a capacidade do computador / adicionar mais recursos [31, 30]. Sendo assim, o Redis apenas permite escalar horizontalmente sem a perda de dados com recurso à utilização de *clusters*. Por fim, uma desvantagem do Memcached é que este não suporta replicação, ao contrário do Redis [31, 30].

Quanto ao **Redis**, este permite armazenar chaves e valores com um tamanho de até 512 MB, o que conclui que permite ter numa só entrada da *cache* 1 GB de dados [30]. Uma caraterística diferenciadora do Redis é que este suporta mensagens *publish-subscribe* que pode ser utilizado para o desenvolvimento de aplicações. Por outro lado, o Redis suporta transações, o que permite executar grupos de comandos como uma operação atómica e isolada [31]. Por último, e para além da capacidade de replicação já mencionada anteriormente, o Redis permite ainda a criação de *snapshots* ao guardar os dados em disco a certa altura, o que pode ser até utilizado para a recuperação de dados em caso de falha [31].

2.5 NETATMO API

Tendo como ponto de partida a utilização da API da Netatmo[27], é realizada de seguida uma contextualização da infraestrutura desta empresa, assim como um estudo sobre as informações que a API disponibiliza na área deste problema e também os limites estipulados na utilização da mesma. Por fim, é feito um levantamento de informação sobre uma aplicação que utiliza as informações desta empresa.

2.5.1 *Infraestrutura da Netatmo*

A Netatmo é uma empresa francesa especializada em dispositivos conectados, relacionados com o conceito de *Internet of Things (IoT)*. A empresa comercializa vários produtos de forma a tornar as nossas casas um pouco mais inteligentes, como estações meteorológicas, termostatos, câmaras *indoor* e *outdoor*, entre outros.

Desta forma, a infraestrutura que a empresa possui será utilizada como ponto de partida para este projeto, tendo em conta a rede de estações meteorológicas que possui e cujos dados disponibilizados por esta rede podem ser obtidos através da API que a empresa disponibiliza. Com isto, é possível utilizar esta API para desenvolver aplicações através de uma conta de *developer*. A rede criada por esta companhia deve-se ao facto

27 https://dev.netatmo.com/apidocumentation/

de cada pessoa poder assim ter uma estação Netatmo em casa (como a que pode ser vista na Figura 24), que trata de recolher dados e os partilhar na rede em que está inserida, podendo mais tarde serem consultados.

Figura 24: Exemplo de *Netatmo Weather Station*.

2.5.2 *Dados fornecidos pela API*

No que diz respeito a dados meteorológicos, é importante perceber dois dos *endpoints* fornecidos pela API da Netatmo com maior probabilidade de serem utilizados neste projeto: `Getpublicdata` e `Getmeasure`.

Getpublicdata

Permite obter dados meteorológicos publicamente partilhados dos dispositivos exteriores para uma área predefinida.

A utilização deste *endpoint* permite obter para uma dada região dados como temperatura, pressão atmosférica, humidade, força ou ângulo do vento, força ou ângulo de rajadas, chuva, etc. Para além disto, são ainda obtidos detalhes sobre a localização das estações meteorológicas.

O processo para a utilização deste e outros métodos inclui uma autenticação, como é indicado na Figura 25.

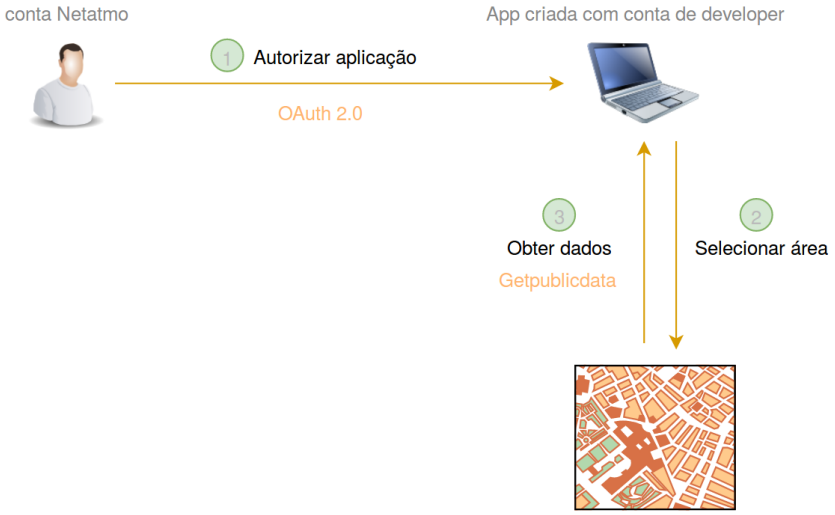


Figura 25: Processo de utilização do *endpoint* Getpublicdata.

Em primeiro lugar, é necessário realizar uma autenticação com a conta de utilizador da Netatmo de forma a autorizar a aplicação a consultar este tipo de dados.

De seguida, utilizando este *endpoint* é necessário saber qual a área sobre a qual se pretende obter os dados meteorológicos. Ao invés de dizer um ponto específico do globo com recurso a uma coordenada de latitude e longitude, neste método são utilizados dois valores para a latitude e dois valores para a longitude, com o intuito de selecionar uma região [32]:

- ***lat_ne*** - valor da latitude no canto nordeste da área pretendida.
- ***lon_ne*** - valor da longitude no canto nordeste da área pretendida.
- ***lat_sw*** - valor da latitude no canto sudoeste da área pretendida.
- ***lon_sw*** - valor da longitude no canto sudoeste da área pretendida.
- Restrições:
  - -85 $\leq$ ***lat_ne*** e ***lat_sw*** $\leq$ 85
  - -180 $\leq$ ***lon_ne*** e ***lon_sw*** $\leq$ 180
  - ***lat_ne*** > ***lat_sw***
  - ***lon_ne*** > ***lon_sw***

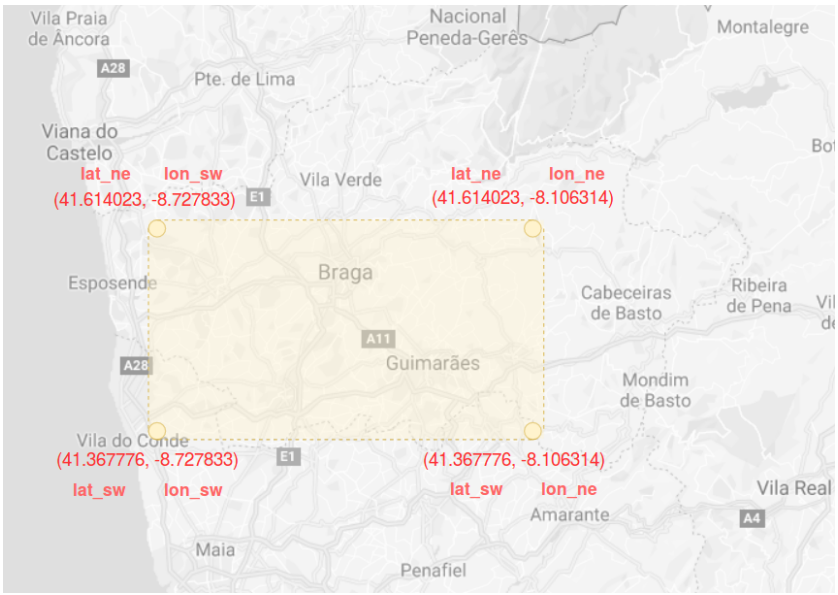


Figura 26: Exemplo de região definida através dos parâmetros *lat_ne*, *lat_sw*, *lon_ne* e *lon_sw*.

Através da Figura 26 podemos perceber em que consiste a utilização de uma região e não de um ponto em específico. Desta forma, é de esperar que o *endpoint* devolva medições relativas a várias estações que possam existir nesta região, assim como obter informações sobre a localização das mesmas.

Voltando ao processo de utilização deste "método", para além destes 4 valores pedidos pela API (e excetuando claro o `access_token`), podem ainda ser passados valores para dois campos opcionais:

- ***required_data*** - permite filtrar estações tendo em conta as medições pretendidas. Por exemplo, se pretendermos dados sobre chuva (*rain*), apenas serão devolvidos dados de estações que contêm pluviómetros.
- ***filter*** - quando colocado a verdadeiro (*true*) exclui estações com medições anormais.

Com isto, é possível então solicitar o pedido e verificar a resposta devolvida pelo *endpoint*, tendo em conta os dados que a mesma retorna. Os dados devolvidos são um conjunto de valores das várias estações que fazem parte dos critérios pedidos. Para cada uma das estações são fornecidos os dados:

- ***_id*** - id da estação.
- ***place*** - informações sobre a localização da estação: latitude e longitude, altitude e *timezone*.
- ***measures*** - últimas medições feitas pela estação. A estrutura desta propriedade pode ser exemplificada como:

```
measures: {
    <module_mac_address>: {
        res: {
            <timestamp>: ( °C, % )
        },
        type: ("temperature", "humidity")
    }
}
```

Getmeasure

Este *endpoint* permite obter dados de um dispositivo ou módulo específico (estações meteorológicas ou termostatos apenas).

Alguns dos parâmetros relevantes deste método, excluindo o necessário `device_id` e a possibilidade de especificar um módulo específico de uma estação através de `module_id`, são enunciados e explicados de seguida.

- ***date_begin*** e ***date_end*** - permitem especificar os *timestamps* da primeira e última medições efetuadas.
- ***scale*** - parâmetro obrigatório e que determina o intervalo entre duas medições (por exemplo: todas as medições, de 30 em 30 minutos, de 1 em 1 hora, entre outros).
- ***limit*** - limite de medições. O valor predefinido e máximo é de 1024.
- ***type*** - é também um parâmetro obrigatório e permite especificar as medições nas quais se tem interesse, como por exemplo chuva ou humidade. Os dados que podem ser pedidos dependem de *scale*.

2.5.3 *Limite de chamadas*

A Netatmo faz questão de referir que os limites de chamadas impostos são escaláveis conforme o número de clientes que a aplicação tem, justificando estes limites não como uma forma de limitar o trabalho dos programadores, mas sim como sendo uma forma de segurança contra usos maliciosos da API.

Quando se utiliza a API Netatmo Smart Home (onde se incluem métodos como Getmeasure) podem-se atingir dois tipos de limites [33]:

- ***Application limit*** - limitação global da aplicação. Se um utilizador fizer muitos pedidos através da aplicação e a mesma atingir o seu limite global, isto vai fazer com que os outros utilizadores não possam utilizar a aplicação (isto é, efetuar pedidos).
- ***User limit*** - este limite existe para prevenir que um utilizador impeça a aplicação de fazer pedidos à API Netatmo.

2.5.4 *Exemplo de utilização das informações Netatmo*

São algumas as aplicações que podemos encontrar na Play Store (Android) ou na App Store (iOS) que utilizam informações Netatmo de forma a criar um produto atrativo para o utilizador final.

Um exemplo destas aplicações é a *Report for Netatmo* [34], ainda que disponível apenas na App Store e que seja necessário ter obrigatoriamente uma estação Netatmo para poder utilizar a aplicação. Esta aplicação oferece a apresentação de informações da comunidade Netatmo.

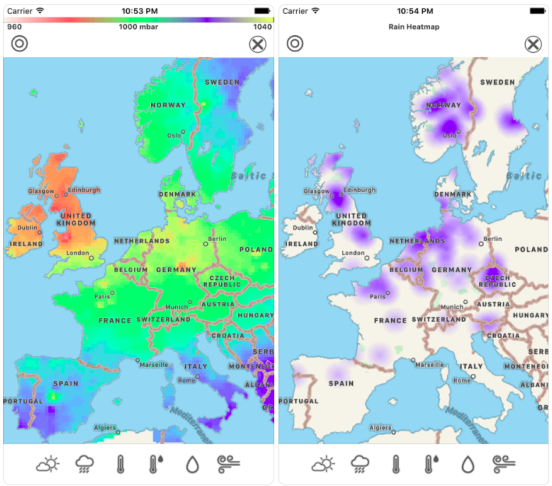


Figura 27: *Screenshots* da aplicação *Report for Netatmo* com apresentação de valores de pressão atmosférica (esquerda) e chuva (direita) em mapa.[28]

28 Figuras retiradas de 'Report for Netatmo' ([34]).

Permite consultar informações em mapa, medições como a pressão atmosférica, chuva, temperatura ou até velocidade do vento. As *screenshots* da aplicação permitem perceber uma abordagem muito interessante para a apresentação de informações num mapa, com a utilização de um esquema de cores que permite perceber o contraste entre as várias zonas do mesmo, como é verificável na Figura 27 na demonstração da interface que apresenta os dados relativos a valores de pressão atmosférica, indicando também um guia (no topo) para perceber o contexto das cores relativamente ao valor das medições. Acaba por ser uma abordagem interessante e que se torna atrativa para o utilizador da aplicação.

Para além disto, dados adicionais são calculados para a estação do utilizador em causa, tal como alterações registadas por hora, lua, ou ciclos solares [34].

Permite a consulta de diagramas dos últimos quatro dias relativa a informações como pressão atmosférica, temperatura, chuva, humidade relativa e específica, direção ou velocidade do vento e rajadas. Como funcionalidades adicionais, esta aplicação oferece ainda a hipótese de adicionar páginas *web* a uma lista de favoritos para adicionar previsões ou outros serviços, assim como ter ainda suporte em inglês para a tecnologia VoiceOver (um leitor de tela compatível com monitores braille) da Apple. A Figura 28 ajuda a perceber as informações disponibilizadas pela aplicação.

Figura 28: *Screenshot* da aplicação *Report for Netatmo* com apresentação de várias informações meteorológicas.[29]

29 Figura retirada de 'Report for Netatmo' ([34]).

3

ACESSO E UTILIZAÇÃO DE FONTES DE DADOS METEOROLÓGICOS

Neste capítulo é feita uma interação com a API da Netatmo e com o Kafka, sendo ainda também realizado um levantamento de informações sobre outras fontes de dados meteorológicos que forneçam dados interessantes para um contexto de aplicação como a que se pretende desenvolver.

3.1 INTERAÇÃO COM API DA NETATMO E KAFKA

Para adquirir conhecimento sobre a API disponibilizada pela Netatmo e que será utilizada no projeto, começou-se por desenvolver código Python que utilize os *endpoints* `Getpublicdata` e `Getmeasure`.

A escolha pela utilização da linguagem Python deve-se a uma facilidade de utilização na realização de pedidos HTTP (biblioteca `requests`), agilizar o processo de interação com dados JSON, assim como permitir a manipulação dos mesmos com recurso a dicionários da própria linguagem, como é referido um pouco mais à frente neste documento.

3.1.1 *Atualização do token*

Um ponto importante para trabalhar com esta API passa pela necessidade de ter que **refrescar o *token*** que necessita de ser passado como parâmetro em todas as chamadas aos *endpoints* disponibilizados (parâmetro `refresh_token`). Pela utilização realizada é possível verificar que este *token* tem uma validade de três horas. Desta forma, o código implementado o que faz é que quando tenta utilizar um *endpoint* da API e recebe como resposta um erro devido ao *token* ter expirado, invoca o método `refresh_token()` que trata de atualizar o valor do mesmo e, depois então, consegue finalmente obter a resposta pretendida fazendo um pedido à API. A implementação deste método, e de forma a também perceber um pouco a interação que é feita com a API em causa, é apresentado na Listagem 3.1.

```
def refresh_token():
    payload = {
        'grant_type': 'refresh_token',
        'client_id': config.CLIENT_ID,
        'client_secret': config.CLIENT_SECRET,
        'refresh_token': config.REFRESH_TOKEN
```

```
}
try:
    response = requests.post(config.REFRESH_TOKEN_URL,data=payload)
    response.raise_for_status()

    access_token = response.json()["access_token"]
    refresh_token = response.json()["refresh_token"]
    expires_in = response.json()["expires_in"]
    expires_in = datetime.now() + timedelta(seconds=int(expires_in))

    print("Access token:", access_token)
    print("Refresh token:", refresh_token)
    print("Your token expires in:", expires_in)

    config.ACCESS_TOKEN = access_token
except requests.exceptions.HTTPError as error:
    print(error.response.status_code, error.response.text)
```

Listagem 3.1: Método utilizado para refrescar o *token* da API da Netatmo.

3.1.2 *Alteração do formato de resposta*

Outro dos motivos para escrever código que interaja com a API ao invés de simplesmente fazer os pedidos à mesma onde assim for necessário, é que torna-se possível **manipular o formato das respostas** consoante for mais vantajoso para o sistema a desenvolver.

No que toca à utilização do *endpoint* `Getpublicdata`, são apresentados na Figura 29 o formato original de informações de uma estação meteorológica fornecido pela Netatmo, assim como o formato fornecido através do código desenvolvido.

Tendo em conta a necessidade de mais tarde estes dados possivelmente serem processados por um código aplicacional e que trate de os inserir numa apresentação através de um mapa, a alteração do formato de resposta tem isso mesmo em conta. Sendo assim, são feitas simplificações e remoções de certas informações, ao passo que outras são apresentadas de uma forma mais percetível.

Primeiramente, a propriedade `_id` dá origem a um `station_id` que serve de identificador da estação. No que diz respeito às informações sobre a localização desta estação, são mantidos apenas os valores de longitude e latitude (com o intuito de marcar a mesma num mapa), a altitude e o *timezone* da mesma. Como é fácil de perceber no formato original de resposta, uma estação Netatmo pode ter mais do que um *device*, o que permite adicionar à estação a capacidade de efetuar mais medições (como vento ou chuva). Com isto, no formato original as medições (`measures`) são apresentadas por *device* (através do seu *mac address*), algo alterado no formato manipulado, de forma a apresentar as medições pela área das mesmas, como temperatura ou pressão atmosférica. Sendo assim, dentro de cada área de medição, passam a conter informações sobre o *mac address* do *device* responsável pelas medições (`module`), os valores das próprias medições e a in-

formação de quando foi realizada a última atualização destes mesmos valores (`last_update`), sendo esta informação convertida de um formato *timestamp* para um formato de data e hora mais amigável para interpretação. Para alterar esta parte relativa à apresentação de medições, é possível verificar na versão original que as medições relativas a pressão atmosférica, temperatura e humidade são apresentadas de forma diferente dos valores relativos a chuva e vento, o que leva a um tratamento diferente para cada uma destas formas. Enquanto que no que diz respeito aos valores de chuva e vento a manipulação dos dados é mais simples, no caso da temperatura e da humidade (medições do *device* `01:00:00:00:00:001`) as medições propriamente ditas são apresentadas com recurso a um objeto `res` que indica o *timestamp* do momento em que foi efetuada a última atualização destes valores e de um *array* que apresenta os mesmos. Para perceber o que representam estes valores, é necessário comparar a ordem com que são apresentadas neste *array* com o *array* apresentado por `type`.

Figura 29: Comparação entre o formato de resposta fornecido pela Netatmo (esquerda) e manipulado através de código desenvolvido (direita) para o *endpoint* Getpublicdata.

Já no que diz respeito à manipulação do *endpoint* `Getmeasure` utilizado para consultar o histórico de medições de uma estação Netatmo, são apresentados os dois formatos (original e manipulado) na Figura 30.

Neste método, existem menos modificações a necessitar de uma explicação. Em primeiro, o início de um período de medições (`beg_time`) é convertido também de um *timestamp* para uma representação mais percetível (`start_time`). No que diz respeito aos valores das medições, espaçados por uma determinada escala (neste caso, 86400 segundos que representam um espaço temporal de um dia), os mesmos são originalmente representados com recurso a *arrays* com um único valor dentro de outro *array*, algo pouco simpático. Sendo assim, passa a ser utilizado um *array* com a *key* `values`, que contem objetos que representam as várias medições. Cada um destes objetos tem o valor da respetiva medição, mas foi também optado por apresentar o tempo em que a mesma foi medida, na perspetiva de poder ajudar a reduzir a complexidade de cálculos numa outra máquina para por exemplo apresentar um histórico de medições num gráfico ou algo semelhante.

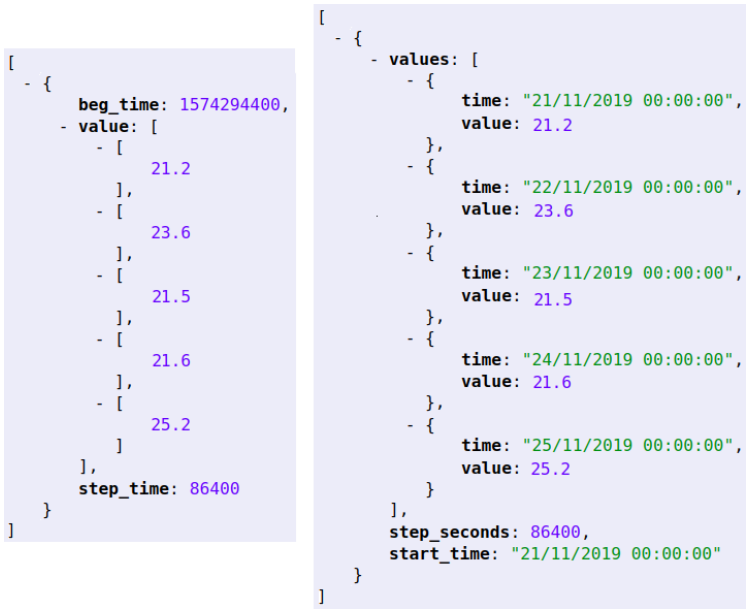


Figura 30: Comparação entre o formato de resposta fornecido pela Netatmo (esquerda) e manipulado através de código desenvolvido (direita) para o *endpoint* Getmeasure.

3.1.3 *Integração com Kafka*

De forma a ter um primeiro contacto com as tecnologias de *message brokers*, procedeu-se à integração do código desenvolvido (referente à interação com a API da Netatmo) com o Kafka. A escolha pelo Kafka recai por uma maior simplicidade para iniciar o processo de aprendizagem assim como a existência da biblioteca `kafka-python` na linguagem Python já utilizada, o que facilita a comunicação e manipulação de informações utilizando as mesmas ferramentas.

Desta forma, a ideia passava por construir duas aplicações de teste:

- Uma aplicação *cliente* que disponibiliza *endpoints* tanto para obter informações meteorológicas relativas a uma dada área (*endpoint* `Getpublicdata` referido na Secção 2.5.2) assim como para obter um histórico de medições de uma estação (*endpoint* `Getmeasure` referido na Secção 2.5.2).
- Uma aplicação *servidor* que trata de receber os pedidos, interage com a API da Netatmo, manipula o formato de resposta como explicado anteriormente e envia a resposta para a aplicação *cliente*.

Sendo que as duas aplicações comunicam através do Kafka. O esquema apresentado na Figura 31 permite ter uma perceção do que era pretendido atingir com o desenvolvimento das duas aplicações.

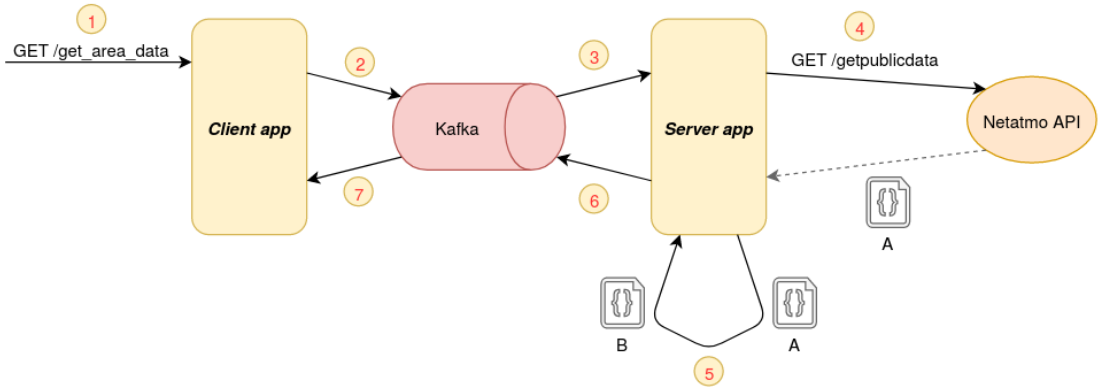


Figura 31: Etapas da comunicação entre a aplicação *cliente* e *servidor*.

Sendo assim, o processo é desencadeado com um pedido GET feito por um utilizador (neste caso do *endpoint* `get_area_data`) (1). A aplicação *cliente* envia uma mensagem para o Kafka onde é indicado o método que foi pedido e os respetivos parâmetros (2), com o intuito da aplicação *servidor* a receber (3). A partir daqui, a aplicação *servidor* verifica que o método requisitado é o `get_area_data` e sendo assim, utiliza o método `Getpublicdata` da Netatmo com os parâmetros pedidos para receber a resposta devolvida pela API (4). Como relatado anteriormente, este formato de resposta é modificado para um formato mais adequado (5) e enviado para o Kafka (6), de modo a que a aplicação *cliente* consiga receber a resposta (7) que deve depois devolver ao cliente.

A dificuldade deste processo com integração do Kafka passa por, como explicado no processo anterior, a aplicação *cliente* ter que fazer um pedido à aplicação *servidor* e esperar pela resposta que o utilizador pretende receber. Desta forma, ambas as aplicações são **produtoras e consumidoras** de mensagens do Kafka, ou seja, as mensagens seguem um fluxo bidirecional, o que aumenta a complexidade do problema.

A solução para esta dificuldade passa pela utilização do padrão ***Request-Reply***, pertencente ao conjunto de EIPs, que sugere uma forma de abordar este problema. Este padrão é apresentado na Figura 32.

> *When two applications communicate via Messaging, the communication is one-way. The applications may want a two-way conversation.* [24]

Este padrão pressupõe a existência de dois participantes: um *requester* que envia um pedido e espera pela resposta, e um *replier* que recebe o pedido e responde ao mesmo. Assim, a sugestão passa pelo envio de um par de mensagens *request-reply*, cada um no seu próprio canal.

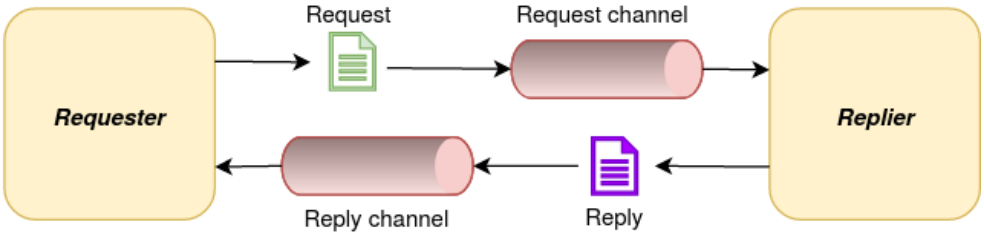


Figura 32: *Request-Reply pattern*.[1]

O canal para o envio da mensagem referente ao pedido pode ser *point-to-point* ou *publish-subscribe*, dependendo do pedido ser enviado por *broadcast* para todos os interessados ou ser apenas processado por um único consumidor (como sabemos, o Kafka junta de certa forma este dois modelos de mensagens num modelo híbrido, como referido na Secção 2.3.3). Quanto ao canal para o envio das respostas, é praticamente sempre *point-to-point*, até porque normalmente faz pouco sentido enviar uma resposta em *broadcast*, mas sim enviar a mesma apenas para quem realizou o pedido.

Assim, são criados dois tópicos Kafka que vão fazer o papel destes dois canais: `request-topic` e `response-topic`. A criação dos mesmos é realizada com os comandos apresentados na Listagem 3.2.

```
$ ./kafka-topics.sh --create --bootstrap-server localhost:9092 \
    --replication-factor 1 --partitions 1 --topic request-topic
$ ./kafka-topics.sh --create --bootstrap-server localhost:9092 \
    --replication-factor 1 --partitions 1 --topic response-topic
```

Listagem 3.2: Criação dos tópicos request-topic e response-topic.

Ainda assim, o *requester* quando recebe uma mensagem necessita de saber qual o pedido ao qual aquela resposta corresponde. Para isto, é tido em conta o padrão ***Correlation Identifier*** [24] (também do conjunto de EIPs). Desta forma, a cada pedido é atribuído um identificador, de forma a que a resposta também possua esse mesmo identificador e seja possível perceber a origem do pedido realizado.

Desta forma, a mensagem de pedido enviada da aplicação *cliente* para o `request_topic` tem um formato JSON no seu `value`, no qual é identificado o método requisitado e quais os parâmetros pedidos, ao passo que é enviado nos `headers` da mensagem o valor do `correlation_id` que identifica a mensagem e ainda o nome do `response_topic`, para onde a mensagem de resposta deve ser enviada (esta abordagem é melhor pois o nome do tópico de resposta pode mais tarde ser alterado). A Listagem 3.3 mostra como esta mensagem é formada.

1 Figura baseada em [página 148] de 'Enterprise Integration Patterns' ([24]).

```
correlationId = generate_correlationId()
producer.send(
    request_topic,
    value = {
        'method': 'get_area_data',
        'parameters': {
            'lat_ne': lat_ne,
            'lon_ne': lon_ne,
            'lat_sw': lat_sw,
            'lon_sw': lon_sw,
            'filter': filter,
            'required_data': required_data
        }
    },
    headers = [
        ('correlationId', str.encode(correlationId)),
        ('responseTo', str.encode(response_topic))
    ]
)
```

Listagem 3.3: Formação da mensagem enviada pela aplicação *cliente* para o tópico request-topic.

Desta forma, a aplicação *servidor* quando recebe esta mensagem consegue obter o `correlationId` e o tópico para onde enviar a resposta (`responseTo`) depois de obter a resposta para o método e parâmetros pretendidos, como demonstrado na Listagem 3.4.

```
correlationId = message.headers[0][1].decode()
responseTo = message.headers[1][1].decode()
    ...
res = netatmo.getAreaData_aux(lat_ne, lon_ne, lat_sw, lon_sw, filter,
    required_data)
    ...
producer.send(
    responseTo,
    value = res,
    headers= [('correlationId', str.encode(correlationId))]
)
```

Listagem 3.4: Tratamento por parte da aplicação *servidor* do pedido recebido.

Quando se realiza um Remote Procedure Call (RPC), a *thread* da aplicação que efetuou o pedido deve bloquear enquanto espera pela resposta. No entanto, com o padrão *Request-Reply* são sugeridas duas abordagens para receber a resposta [24]. Uma solução passa por ter uma *thread* que envia a mensagem de pedido e bloqueia à espera da resposta, como um *polling consumer*, para depois processar a mesma. Esta abordagem implementada numa primeira fase faz pouco sentido tendo em conta a utilização de mecanismos nos quais se

pretende operações assíncronas. A segunda abordagem sugere a utilização de uma *thread* que envia a mensagem e define um *callback* para a resposta; entretanto uma *thread* está à escuta de novas respostas e, quando tem uma nova mensagem, chama o respetivo *callback*, recupera o contexto e processa a resposta. Esta abordagem faz mais sentido tendo em conta o contexto do problema mas, ainda assim, não faz grande sentido neste caso colocar simplesmente o pedido GET do utilizador "pendurado" enquanto a resposta está a ser preparada, para além de que vão sendo acumulados cada vez mais pedidos no sistema. Sendo assim a abordagem tomada passa por quando um pedido GET é realizado à aplicação *cliente*, esta envia a mensagem do pedido para o tópico correspondente e retorna para o utilizador ao mesmo tempo um JSON com o identificador de correlação do pedido do mesmo, por exemplo:

```
{
    'correlation_id': 481213372609
}
```

Com isto, a aplicação *cliente* mantém uma *thread* dedicada a receber mensagens de respostas provenientes do Kafka e trata de mapear as mesmas num dicionário *chave-valor* no qual a chave utilizada é o *correlation identifier*. Assim, para o cliente aceder à resposta que pretendia obter, deve aceder ao *endpoint* `/get_reply` que ou envia a resposta e remove a mesma do dicionário interno (devido a questões de armazenamento), ou devolve uma resposta indicando que ainda não tem uma resposta para aquele pedido ou que não existe nenhum pedido com aquele identificador. A obtenção da devida resposta devia então ser obtida através do método:

```
GET /get_reply?correlation_id=481213372609
```

Neste ponto, uma melhoria que podia ser realizada nesta implementação passava por persistir o dicionário, de forma a que se a aplicação por algum motivo falhar ou ficar *offline* mantenha o estado de pedidos e respostas que até então mantinha. Outra melhoria passaria por limpar este mesmo dicionário periodicamente, de forma a remover respostas que não foram "levantadas" pelo utilizador que as requisitou.

A forma escolhida para implementar ambas as melhorias passou pela introdução do `Redis`. Sendo assim, em vez de utilizar um dicionário de `Python` é utilizado o `Redis` para guardar os pares *chave-valor*, tendo sempre atenção com o *encoding* e *decoding* necessários de realizar para guardar e ler os valores de JSON na base de dados. A Listagem 3.5 mostra como isto pode ser implementado.

```
# guardar par chave-valor
redis.set(correlationId, json.dumps(message.value).encode())

# obter valor de uma chave
r = redis.get(correlationId)
response = r.decode()
```

Listagem 3.5: Utilização do Redis para guardar e obter pares *chave-valor*.

Quanto à parte do dicionário ser limpo de forma periódica, sem a introdução do `Redis` possivelmente tornar-se-ia necessária a implementação de tarefas em *background* para executar essa operação ou mesmo ter que

guardar valores de datas relativas a cada pedido para depois esta tarefa conseguir perceber quais as respostas com mais do que determinado tempo que poderiam ser removidas. Assim, com a introdução do `Redis`, e tendo em conta que o mesmo efetua armazenamento de chaves com durabilidade opcional, a solução escolhida passa por colocar um tempo de expiração nas chaves das respostas armazenadas (de por exemplo 15 minutos), utilizando para isso o parâmetro `ex` do método `set()` da biblioteca `redis` que permite estabelecer uma *flag* de expiração de *X* segundos numa chave. Com isto, ao fim desses segundos, a chave deixa de existir na base de dados. Sendo assim, para guardar uma resposta de um pedido representado pelo seu identificador de correlação, o código passa a ser como demonstrado na Listagem 3.6.

```
redis.set(
    correlationId,
    json.dumps(message.value).encode(),
    ex=config.KEYS_EXPIRE_TIME_SECS
)
```

Listagem 3.6: Utilização do Redis com a definição de expiração em chaves.

3.2 OUTRAS FONTES DE DADOS METEOROLÓGICOS

Tendo em conta o âmbito do projeto, torna-se interessante perceber que tipo de APIs existem que possam providenciar dados semelhantes sobre informações meteorológicas. É importante realçar que as fontes e respetivas informações sobre as mesmas apresentadas de seguida têm em conta um contexto de pesquisa de APIs que providenciem informações de forma gratuita, ainda que isto implique a estipulação de limites na utilização das mesmas.

3.2.1 *OpenWeatherMap*

A *OpenWeatherMap*[2] é uma das fontes de informações meteorológicas mais conhecidas e utilizadas, sendo esta uma empresa da área de IoT.

O acesso gratuito à API disponibilizada por esta companhia permite consultar informações meteorológicas atuais e permite por exemplo consultar previsões meteorológicas para 5 dias, apesar de isto ser algo que não é fornecido pela API da Netatmo que é tida como ponto de partida. Ainda assim, informações interessantes mas não disponibilizadas na versão gratuita são informações relativas a datas anteriores, que permitem consultar as informações meteorológicas num intervalo de datas à escolha. É relevante ainda referir que sendo assim, a versão gratuita permite fazer 60 chamadas por minuto à API (num máximo de 1 milhão de chamadas por mês).

O *endpoint* que permite obter as informações atuais de um determinado local permite a pesquisa através do nome da cidade ou do código postal juntamente com o código que representa o país, para além da utilização de coordenadas como exemplificado na Listagem 3.7. Para além do *token* obrigatório em qualquer chamada

2 https://openweathermap.org/

da API, e inevitavelmente os valores de latitude e longitude, o valor "metric" indica que o sistema métrico deve ser o utilizado para mostrar as unidades de medição (por exemplo, para apresentar os valores da temperatura em Celsius), e o valor "pt" em `lang` indica qual a tradução a fazer para o estado do tempo, neste caso, para português.

```
params = {
    'lat': 41,
    'lon': -8,
    'lang': 'pt',
    'units': 'metric',
    'APPID': '<OpenWeatherMap_TOKEN>'
}
r = requests.get('http://api.openweathermap.org/data/2.5/weather', params=params)
```

Listagem 3.7: Exemplo de interação com a API da OpenWeatherMap.

A resposta em formato JSON permite obter valores como a temperatura real e a perceção que as pessoas têm da mesma (`main.temp` e `main.feels_like` respetivamente), pressão atmosférica (`main.pressure`), humidade (`main.humidity`), velocidade e direção do vento (`wind.speed` e `wind.deg`) ou, por exemplo, a nebulosidade (`clouds.all`).

Enquanto que o excerto apresentado em cima apenas apresenta um *dataset* de informações, pode ser ainda utilizado outro *endpoint* (`http://api.openweathermap.org/data/2.5/find`) que permite retornar os dados de locais dentro de um círculo cujo seu centro é definido pelos valores de latitude e longitude (`lat` e `lon`). O número de locais que se pode obter é definido pelo parâmetro `cnt`, que tem como 50 o seu valor máximo.

3.2.2 *Weatherbit*

Outra API desta área de informações é a fornecida pela *Weatherbit*[3]. A versão gratuita para utilização das informações providenciadas por esta empresa permite apenas realizar 500 chamadas no total de um dia, o que é um número consideravelmente baixo. Esta versão gratuita permite consultar informações de medições atuais e, tal como a fonte anterior, permite a consulta de previsões para os próximos dias (neste caso, para 16 dias). Outra possibilidade é a consulta de dados históricos no espaço temporal de um mês, o que seria interessante não fossem as limitações de chamadas dos *endpoints* relacionados, que um pouco à frente são explicadas.

Para consultar as informações meteorológicas atuais de um dado local, as mesmas podem ser obtidas através do *endpoint* `http://api.weatherbit.io/v2.0/current`, passando para além do *token* necessário (`key`) os valores de latitude e longitude (`lat` and `lon`) para os quais se quer obter informações (sendo também possível realizar a procura por cidade ou código postal). A resposta recebida (também em formato JSON), para além de informações como temperatura, pressão atmosférica ou velocidade do vento, per-

3 https://www.weatherbit.io/

mite obter valores tais como a direção do vento em formato verbal (por exemplo “west-southwest”), o ponto de condensação da água, a humidade relativa ou até mesmo as horas do nascer e pôr do sol.

Quanto à consulta de valores históricos, como referido anteriormente, a versão gratuita permite a consulta de valores num espaço temporal de um mês. Existe a opção de realizar a consulta de informações com espaço entre medições de um dia (`https://api.weatherbit.io/v2.0/history/daily`) ou uma hora (`https://api.weatherbit.io/v2.0/history/hourly`). O problema é que esta mesma versão gratuita não permite que em ambos os *endpoints* se receba informações de medições de mais do que um dia, sendo necessário assim dividir um pedido em vários pedidos para consultas de mais do que um dia. Sendo assim, podemos concluir que torna-se pouco viável a utilização destes *endpoints*, tendo em conta que para a consulta dos valores históricos de um mês o mesmo pedido teria que ser dividido por exemplo em 30 pedidos, o que para um limite de 500 pedidos num único dia é demasiado consumidor.

3.2.3 *meteostat*

Por último, é também apresentada a API da *meteostat*[4], que é mais centrada na consulta de valores históricos. Ao contrário das fontes apresentadas anteriormente, esta API não faz qualquer menção a quaisquer planos de pagamentos, pelo que podemos considerar que a mesma pode ser utilizada de forma gratuita, sendo que a limitação de utilização desta é de 200 pedidos por cada hora, e sendo que não é garantida a disponibilidade do serviço. Ao contrário do que é normal nestas fontes de informação, esta API não fornece nenhum *endpoint* para consultar os dados atuais para um dado local.

Um dos *endpoints* disponibilizados permite descobrir quais as estações que existem para um determinado local (`https://api.meteostat.net/v1/stations/nearby`), passando a latitude e longitude desse mesmo local, para além do *token* para utilização da API. Infelizmente, com alguma utilização deste método, pode-se perceber que as fontes de dados utilizadas por esta API pode não ser suficiente, tendo em conta que com uma pesquisa por estações perto das coordenadas da cidade de Braga apenas são devolvidas cinco estações, incluindo algumas com localização no Porto, Viana do Castelo ou até mesmo Vigo.

Com base nas estações encontradas, podemos obter informações sobre as mesmas através do método `https://api.meteostat.net/v1/stations/meta`, passando para além do *token* qual a estação para a qual queremos obter dados. Isto permite obter informações como as coordenadas da estação, a elevação, a região da mesma ou até o seu *timezone*.

Tendo em conta uma dada estação, podem então ser usados três métodos que permitem obter valores históricos de medições, com intervalos entre medições de uma hora, um dia ou um mês. Ainda assim, com base em alguma utilização destes *endpoints* foi possível perceber que os mesmo podem ser um pouco instáveis, pois em certos casos dependentes da estação escolhida, os valores com intervalo de uma hora funcionam mas os restantes não, assim como por vezes alguns valores obtidos têm o seu valor a `null`. Os valores que esta API fornece são por exemplo temperatura, temperatura mínima e máxima, pressão atmosférica, precipitação e velocidade ou direção do vento.

4 https://api.meteostat.net

4

IDENTIFICAÇÃO DO PROBLEMA

Tendo sido estudados todos estes tópicos nos Capítulos 2 e 3, importa então agora contextualizar o desafio e perceber as caraterísticas da aplicação que se pretende desenvolver.

É sabido que o objetivo final passa por ter uma aplicação *web* que permita aos utilizadores da mesma obter informações meteorológicas. Para tal acontecer, torna-se necessário perceber a forma como o *backend* do sistema deve ser definido de forma a ir ao encontro das expectativas e tendo em conta que deve ser uma solução escalável. Para além disto e como já foi explicado anteriormente, esta mesma solução arquitetural deve ser baseada em serviços (daí o estudo de arquiteturas baseadas em micro-serviços na Secção 2.2.2), deve fazer uso de *message brokers* e deve incluir mecanismos de *caching* (assuntos abordados nas Secções 2.3 e 2.4). Por fim, é ainda importante culminar o desenvolvimento do sistema com a construção de um *frontend* para o mesmo, que deve permitir a consulta e a realização de todas as ações que o *backend* facultar.

No que diz respeito às informações meteorológicas, é esperado que a aplicação permita aos utilizadores a consulta de estações através de um mapa e que vá ao encontro de conseguir fornecer informações meteorológicas sobre um local ou uma zona à escolha do utilizador, sendo que para caso de estudo e aplicação de conceitos, as informações meteorológicas a implementar vão-se apenas focar no território português. Para além disto, é ainda expectável que seja possível consultar um histórico de medições que permita ver a evolução das medições efetuadas ao longo do tempo, e que para além disto possam ainda ser adicionadas outras informações que possam ser consideradas interessantes e vantajosas.

Quanto à introdução de utilizadores registados na aplicação, em primeiro lugar é importante referir que este contexto foi apenas pensado para ser enquadrado neste projeto mais tarde, visto ter surgido como uma sugestão numa fase mais adiantada do mesmo. Feita a sugestão, o desafio passou por perceber que tipo de informações e que tipo de funcionalidades faria sentido adicionar à aplicação de forma a que se torne interessante a um utilizador registar-se na aplicação. Sendo assim, propõe-se que os utilizadores registados na aplicação possam:

- Ter um conjunto de estações meteorológicas que possam ser marcadas como favoritas.
- Criar alertas relacionados com as estações meteorológicas.
- Receber notificações que resultam dos alertas definidos.

Com isto, cada utilizador pode ter um máximo de cinco estações marcadas como favoritas e deve ser capaz de fazer a gestão dessa mesma lista. Um utilizador pode criar alertas e ser capaz de gerir os mesmos, podendo

apenas ter alertas de estações favoritas, ou seja, se o utilizador tiver um alerta criado e decidir remover a estação associada ao alerta da lista das suas estações favoritas, esse mesmo alerta vai deixar de existir. Quanto às notificações, essas são uma consequência dos alertas, pois o alerta no fundo define que o utilizador pretende receber notificações sobre esse mesmo alerta em relação a uma estação meteorológica.

No que diz respeito aos alertas das estações meteorológicas, os mesmos vão apenas dizer respeito a dois tipos de medições meteorológicas: temperatura e humidade. As notificações relacionadas com temperatura poderiam ser interessantes num contexto, por exemplo, de monitorização de regas, enquanto que as de humidade poderiam acrescentar valor, por exemplo, num contexto de monitorização de vinhas. Ainda assim, a aplicação deve ser desenvolvida de forma a que possam ser adicionados novos tipos de medições num futuro para além destes dois *use cases*. Quanto aos tipos de alertas que podem ser definidos, foram pensados para ser implementados de momento quatro tipos de alertas:

- **Alertas por valor** - o utilizador recebe uma notificação sempre que a última medição da estação for igual ou superior a um valor definido por si. Ou seja, a ideia é criar um alerta para uma dada estação, uma medição (temperatura ou humidade) e um valor, por exemplo, 15 ºC de temperatura, e a partir de então quando uma nova medição dessa estação for igual ou superior a esse valor é automaticamente enviada para o utilizador uma notificação. Exemplo:
  - Alerta: medição = temperatura, valor = 15 ºC
  - Medições:
    - 01/01/2020 12:00 = 14 ºC, sem envio de notificação
    - 01/01/2020 13:00 = 15.6 ºC, envio de notificação
- **Alertas por percentagem** - o utilizador recebe uma notificação sempre que a última medição da estação tiver um crescimento percentual igual ou superior a um valor definido por si, em comparação com a medição imediatamente anterior. Sendo assim, a ideia é criar um alerta que para uma determinada estação, uma medição e um valor percentual, por exemplo, 15% em relação a temperatura, e a partir de então quando de uma medição anterior para a medição seguinte (mais recente) a subida percentual for igual ou superior a esse valor definido é automaticamente enviada para o utilizador uma notificação. Exemplo:
  - Alerta: medição = temperatura, valor = 13%
  - Medições:
    - 01/01/2020 12:00 = 14 ºC
    - 01/01/2020 13:00 = 15.6 ºC, sem envio de notificação (crescimento percentual de 11%)
    - 01/01/2020 14:00 = 18.3 ºC, envio de notificação (crescimento percentual de 17%)
- **Alertas de medições consecutivas** - o utilizador recebe uma notificação quando as últimas *X* medições têm todas um valor igual ou superior a um valor definido por si. Ou seja, o objetivo é que para uma dada estação, uma medição e um valor, por exemplo, 15 ºC de temperatura, e a partir de então quando duas

ou mais medições consecutivas tiverem um registo de valores igual ou superior ao definido é enviada automaticamente uma notificação. Exemplo:

  - Alerta: medição = temperatura, valor = 15 ºC
  - Medições:
    * 01/01/2020 12:00 = 14 ºC
    * 01/01/2020 13:00 = 15.6 ºC, sem envio de notificação
    * 01/01/2020 14:00 = 18.3 ºC, envio de notificação (2 medições consecutivas)
    * 01/01/2020 15:00 = 16.7 ºC, envio de notificação (3 medições consecutivas)
    * 01/01/2020 16:00 = 14.4 ºC, sem envio de notificação

- **Alertas de médias temporais** - o utilizador recebe uma notificação sempre que a média de medições da estação nas últimas *X* horas for superior à média de medições das *X* horas imediatamente anteriores. Assim, para uma determinada estação e uma medição, é esperado que, por exemplo, num contexto de períodos de 5 horas, sempre que as medições das últimas 5 horas tiverem uma média superior às medições das 5 horas imediatamente anteriores é então enviada uma notificação automaticamente para o utilizador. Exemplo:

  - Alerta: medição = temperatura
  - Medições:
    * média de 5 horas (01/01/2020 01:00-02:00-03:00-04:00-05:00) = 14 ºC
    * média de 5 horas (01/01/2020 06:00-07:00-08:00-09:00-10:00) = 13.5 ºC, sem notificação
    * média de 5 horas (01/01/2020 11:00-12:00-13:00-14:00-15:00) = 14.5 ºC, envio de notificação

5

CONCEÇÃO DA ARQUITETURA

Neste capítulo o objetivo passa por descrever o conjunto de decisões e escolhas que levaram a uma proposta de solução final para a arquitetura de *backend* da aplicação que se pretende desenvolver. Assim, são apresentadas inicialmente as fontes de dados escolhidas para servirem como um ponto de partida para o sistema. De seguida, são então propostas diferentes soluções para este problema, que demonstram a evolução da solução ao longo do tempo e o porquê dessas mudanças serem efetuadas até culminar numa proposta final.

5.1 FONTES DE DADOS

No que toca às fontes de dados, a primeira fonte que deve ser analisada é a Netatmo, tendo até em conta que a mesma surge no início deste projeto como sendo o ponto de partida. Ainda assim, a mesma deveria ser analisada para perceber se faria sentido adicioná-la a uma implementação deste projeto. Como tal, esta fonte e a sua API foi devidamente analisada no Capítulo 2, sendo feito um estudo pormenorizado da mesma na Secção 2.5, e comprovado até que a mesma deve ser utilizada neste projeto até pela interação feita com ela, como descrito na Secção 3.1 do Capítulo 3.

Posto isto, é importante perceber quais as medições meteorológicas a selecionar daquelas que a sua API devolve. Tendo em conta as medições meteorológicas que normalmente mais são consultadas e que podem ter mais interesse, foi assim decidido que devem ser utilizadas as medições fornecidas pela Netatmo apresentadas na Tabela 2 para cada uma das suas estações (isto é, se uma estação for capaz de obter todas estas medições selecionadas, caso contrário devolve apenas as que tem em comum com o conjunto de medições definido).

Medição meteorológica (métrica)	Propriedade na API	Unidade
Temperatura	`temperature`	ºC (graus Celsius)
Pressão atmosférica	`pressure`	hPa (hectopascal) = mbar (millibar)
Humidade	`humidity`	%
Força / Velocidade do vento	`wind_strength`	km/h
Ângulo do vento	`wind_angle`	º (graus)
Força / Velocidade de rajadas	`gust_strength`	km/h
Ângulo de rajadas	`gust_angle`	º (graus)
Chuvas	`rain_live`	mm (milímetros de chuva)

Tabela 2: Medições selecionadas para utilização na fonte Netatmo.

Para além desta fonte, torna-se necessário selecionar outras que vão de encontro às caraterísticas pretendidas para o desenvolvimento de uma *mashup* de serviços meteorológicos. Sendo assim, foi feita uma análise de 3 fontes de dados que poderiam interessar na Secção 3.2. Destas 3, podemos colocar fora de questão a possibilidade de utilizar a API da meteostat, tendo em conta que a mesma não permite consultar medições atuais e também os problemas detetados e mencionados anteriormente, como por exemplo procurar estações perto de coordenadas (latitude e longitude) situadas na cidade de Braga e o resultado devolver algumas estações que se encontram nas cidades do Porto, Viana do Castelo ou Vigo. Com isto, importa agora fazer um balanço em relação às fontes OpenWeatherMap e Weatherbit. A principal vantagem que pode ser apontada à API da Weatherbit em relação à da OpenWeatherMap é que esta permite consultar dados históricos no espaço temporal de um mês. No entanto, esta fonte apenas permite fazer 500 chamadas por dia, o que é consideravelmente pouco, ao passo que a API da OpenWeatherMap permite realizar 60 chamadas por minuto (num máximo de 1 milhão de chamadas por mês), o que é bastante superior à Weatherbit. Desta forma, e tendo em conta que a API da OpenWeatherMap é bastante interessante e muito utilizada e conhecida, assim como apresenta uma maior comodidade na gestão das chamadas que se podem realizar, é escolhida esta fonte para utilizar neste projeto, fazendo ainda assim uma avaliação positiva da API da Weatherbit que pode ser interessante de introduzir no sistema num trabalho futuro.

Tomada esta decisão, é então necessário escolher as medições meteorológicas a utilizar desta fonte de dados meteorológicos, assim como foi apresentado anteriormente para a Netatmo.

Medição meteorológica (métrica)	Propriedade na API	Unidade
Temperatura	`temperature`	ºC (graus Celsius)
Pressão atmosférica	`pressure`	hPa (hectopascal) = mbar (millibar)
Humidade	`humidity`	%
Força / Velocidade do vento	`wind_speed`	m/s
Ângulo do vento	`wind_angle`	º (graus)
Nuvens	`clouds`	%

Tabela 3: Medições selecionadas para utilização na fonte OpenWeatherMap.

As escolhas são apresentadas na Tabela 3. Importa realçar que obviamente as APIs de fontes de dados meteorológicos não oferecem todas o mesmo tipo de medições ou informações, pelo que por exemplo a Netatmo oferece informações como a velocidade de rajadas, o ângulo de rajadas, ou a chuva, informações essas que a API da OpenWeatherMap não permite obter. Por outro lado, a OpenWeatherMap oferece informações sobre as nuvens, informação essa que não é oferecida pela Netatmo. Com isto, o que se pretende concluir é que ao fazer uma *mashup* destas e de outras fontes a ideia passará por oferecer uma união dos conjuntos de medições que cada uma das fontes oferece, tentando devolver o máximo de informações desse conjunto total de medições sempre que possível.

5.2 MÉTODOS PRINCIPAIS DA API

De forma a poder planear uma arquitetura para o *backend* da aplicação que se pretende desenvolver, é preciso primeiro delinear as principais informações que a mesma deve fornecer, ou seja, quais os principais métodos que a API da arquitetura final deve efetivamente ser capaz de oferecer, e que leva a perceber quais as informações que a arquitetura tem que ter em sua posse ou que deve poder aceder em qualquer altura.

Como ponto de partida para a definição da arquitetura e, no que diz respeito a informações meteorológicas que é o principal ponto atrativo deste projeto, é assim definido que a REST API do *backend* deve implementar o conjunto de métodos apresentados de seguida, tendo em conta que as fontes de dados meteorológicos selecionadas funcionam com base em estações meteorológicas colocadas em variadas localizações.

- `/get_area_data` - a ideia deste método passa por seguir a abordagem do método da Netatmo `Getpublidata`, ou seja, através dos valores da latitude do canto sudoeste, longitude do canto sudoeste, latitude do canto nordeste e longitude do canto nordeste, que juntas definem uma área no mapa, obter para essa mesma área as medições mais recentes das estações que se encontram nela.

- `/get_historical_measures` - o objetivo deste método é devolver para uma determinada estação e um intervalo de datas definido um historial de medições inseridas nesse mesmo intervalo para uma determinada medição à escolha (por exemplo temperatura ou humidade), podendo ainda definir uma escala entre as medições, por exemplo 1 hora de diferença entre duas medições, 3 horas, 1 dia, etc. Este método deve permitir a definição do intervalo de datas de duas formas: com a passagem como parâmetro de duas datas ou com a utilização de um intervalo definido, por exemplo `1day` para definir o intervalo das últimas 24 horas ou `1week` que define a última semana, que são consultas mais gerais e provavelmente mais procuradas.

- `/get_area_average_measures` - seguindo a lógica da escolha de uma área através de quatro valores, como apresentado no método `/get_area_data`, este método devolve para a área definida os valores médios das medições mais recentes das estações que se encontram nesta área. Por exemplo, se numa área existirem 3 estações em que as suas medições mais recentes são as seguintes:
  - Estação A: temperatura = 10 ºC, velocidade do vento = 2 km/h
  - Estação B: temperatura = 8 ºC, velocidade do vento = 2.3 km/h
  - Estação C: temperatura = 11 ºC, velocidade do vento = 1.8 km/h

  Então a resposta para esta área vai indicar que os valores médios das medições da mesma são para a temperatura de 9.67 ºC e para a velocidade do vento 2.03 km/h.

- `/get_historical_average_measures` - o objetivo deste método é devolver para uma determinada estação e um intervalo de datas definido os valores médios das medições efetuadas pela estação nesse intervalo de tempo. Tal como o método `/get_historical_measures`, este *endpoint* deve permitir a definição do intervalo de datas das duas formas explicadas.

Com a definição destes métodos que devem ser desenvolvidos, pode ser concluído que a aplicação deve ser capaz de aceder a dados sobre as medições mais recentes das estações de cada uma das fontes, o que permite ter dados para implementar os métodos `/get_area_data` e `/get_area_average_measures`, e deve ser capaz de aceder a medições históricas dessas mesmas estações, permitindo assim implementar os métodos `/get_historical_data` e `/get_historical_average_measures`.

5.3 PRIMEIRA ABORDAGEM

A primeira abordagem arquitetural seria de certa forma uma continuidade do que foi desenvolvido e explicado na Secção 3.1, em que é feita uma interação da API da Netatmo com o Kafka, com troca de mensagens entre dois componentes, e onde é feita uma manipulação do formato das respostas que a API da Netatmo fornece.

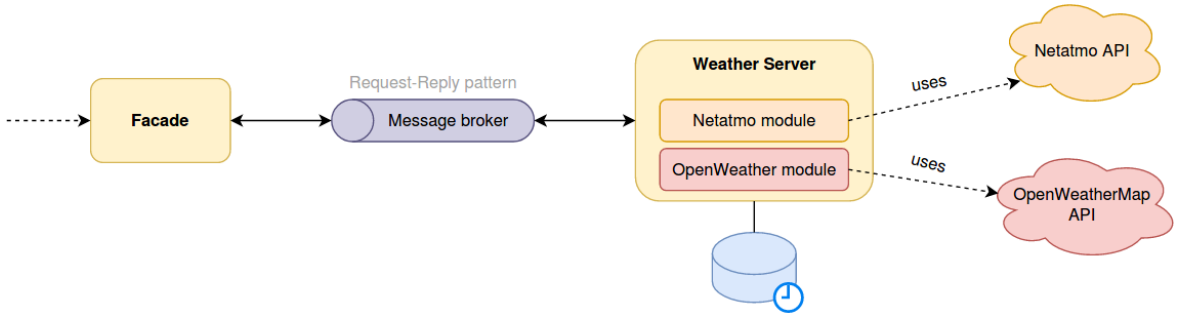


Figura 33: Primeira abordagem para a arquitetura do sistema.

Assim, e como mostra a Figura 33, o objetivo passaria por ter uma arquitetura com dois diferentes componentes aplicacionais e um *broker*.

O primeiro funcionaria como um `Facade`, que expõe os métodos que a arquitetura pretende expor e seria o componente através do qual os clientes interagiam com o sistema. Este seria capaz de ser uma primeira barreira da arquitetura, no sentido em que pedidos que por exemplo não pertencem ao conjunto de métodos expostos ou que não possuem os parâmetros exigidos seriam logo excluídos e não sobrecarregariam o outro componente. Os pedidos que fossem válidos, seriam assim enviados através do *broker* para o `Weather Server` que responderia aos mesmos de forma assíncrona para o `Facade` utilizando o padrão *Request-Reply*. O `Facade`, tal como explicado na Secção 3.1, teria que manter um dicionário com os *correlation identifiers* para que consiga, ao receber as respostas, perceber a que pedidos estas correspondem.

Quanto ao `Weather Server`, este teria um módulo para interagir com a API da Netatmo e outro módulo para interagir com a API da OpenWeatherMap (e outros módulos assim que fossem adicionadas novas fontes de dados). Desta forma, a ideia seria ao receber um pedido do método `/get_area_data` por exemplo, o módulo da Netatmo seria responsável por obter essas informações com a utilização da API da Netatmo e retirar as informações necessárias e o módulo do OpenWeather o mesmo para esta fonte, podendo depois ser feita uma união dos resultados das várias fontes. No caso do método `/get_historical_measures` por exemplo, seria necessário perceber qual a fonte a que estação requisitada pertence e enviar esse mesmo

pedido para o respetivo módulo. Para além disto, o `Weather Server` poderia ainda possuir um sistema de *caching* para guardar alguns pedidos e não necessitar de os calcular novamente e, acima de tudo, até evitar fazer novas chamadas às APIs que são limitadas no número de pedidos que se podem efetuar devido à versão gratuita utilizada das mesmas.

Este cenário de arquitetura poderia resultar, não fosse a fonte OpenWeatherMap na sua versão gratuita não fornecer informações sobre históricos de medições (como referido anteriormente), e, desta forma, não seria possível com esta solução dar respostas aos pedidos dos métodos `/get_historical_measures` e `/get_historical_average_measures` com estações da OpenWeatherMap, mas sim apenas da Netatmo. Desta forma, é necessário reformular a abordagem, como é explicado de seguida.

5.4 SEGUNDA ABORDAGEM

De forma a contornar o problema das consultas históricas na fonte OpenWeatherMap, uma segunda abordagem passou por seguir maioritariamente a arquitetura anterior, mas inserir um componente que realizasse uma ligação entre o módulo OpenWeather do `Weather Server` e a API da OpenWeatherMap - o componente `OpenWeather service` como se pode ver na Figura 34.

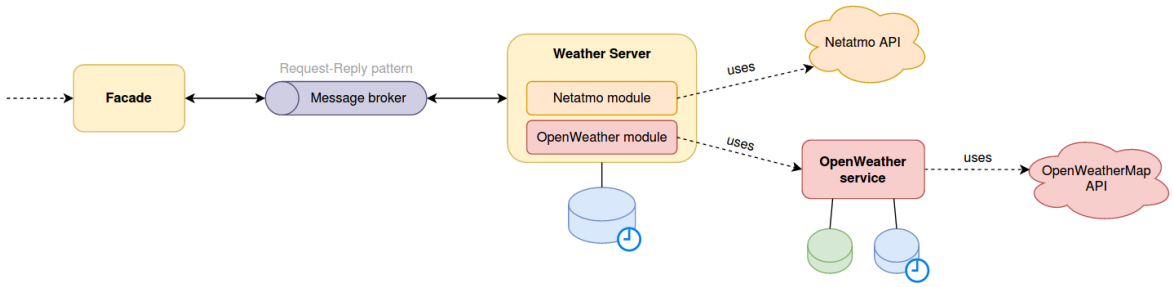


Figura 34: Segunda abordagem para a arquitetura do sistema.

O objetivo deste `OpenWeather service` para contornar o problema das consultas históricas seria então este mesmo componente passar a ter um histórico de medições das estações da OpenWeatherMap em sua posse. Ou seja, a ideia passa por obter regularmente (de hora a hora) as medições mais recentes de todas as estações da OpenWeatherMap e conseguir construir uma base de dados com essas mesmas informações que, com o passar do tempo, faz com que este serviço tenha o seu próprio histórico de medições destas estações e permite também não ser necessário contactar a API da OpenWeatherMap para fazer qualquer pedido a não ser para atualizar a sua base de dados periodicamente com as informações mais recentes. Assim, este `OpenWeather service` deve expor uma API com os métodos que a arquitetura necessita de fornecer aos utilizadores (`/get_area_data`, `/get_historical_measures`, `/get_area_average_measures` e `/get_historical_average_measures`) de forma a que pedidos recebidos pelo `Weather Server` que envolvam as estações OpenWeatherMap sejam possíveis de tratar pelo seu módulo OpenWeather que passa a consumir da API deste novo serviço ao invés de utilizar diretamente a API da OpenWeatherMap. As entidades da camada de negócio deste componente seriam

as apresentadas no diagrama de classes da Figura 35, em que uma estação possui um conjunto de medições (com as devidas medições selecionadas desta fonte e uma data em que a mesma foi obtida). Para além disto, importante realçar que este serviço para além da sua própria base de dados poderia ainda ter um sistema de *caching* que permitisse aumentar a sua *performance* e consequentemente de todo o sistema ao guardar certos pedidos que poderiam ser utilizados novamente sem ter que aceder à base de dados.

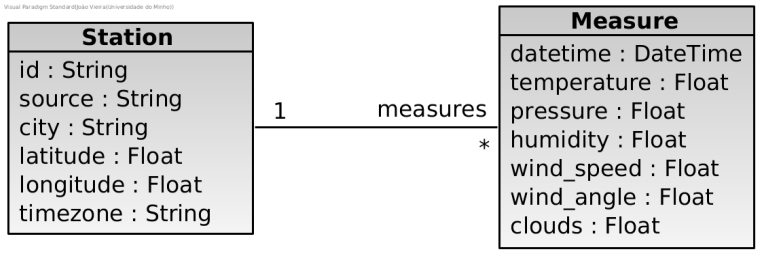


Figura 35: Modelo lógico do componente OpenWeather service.

Ainda assim, esta abordagem (e também a anterior) têm várias limitações que podem ser apontadas e necessitam de ser corrigidas para se obter uma melhor solução:

- O problema do limite de chamadas às APIs, pois enquanto que do lado da fonte OpenWeatherMap este problema é resolvido com a introdução do novo componente `OpenWeather service`, no que toca à Netatmo este continua a ser um problema, até porque na abordagem atual, qualquer informação das estações da Netatmo que seja necessária (e não esteja em *cache*) necessita de utilizar a API da Netatmo para poder satisfazer esse mesmo pedido, o que pode fazer com que a qualquer momento ou numa altura de maior interesse na aplicação se acabe por esgotar as chamadas possíveis e fique impossível obter novas informações desta fonte.

- Relacionado ainda com o ponto anterior, tendo em conta que uma estação da Netatmo pode ter vários módulos e cada um ser responsável por um conjunto de medições, a consulta de valores históricos na API da Netatmo é apenas possível para um módulo de cada vez. O que isto quer dizer é que se uma estação tiver três módulos torna-se necessário fazer pelo menos três diferentes consultas históricas sobre a estação para conseguir obter uma listagem histórica de todas as medições que a estação recolheu (que contam como três chamadas à API). Em cima disto, para fazer apenas os três pedidos seria ainda necessário saber exatamente quais os três módulos que a estação possui e quais as medições pelas quais cada um deles é responsável. Assim, podemos concluir que para a implementação do método `/get_historical_measures` seria necessário não só saber a estação em causa mas também qual o módulo da mesma que possui a medição em que estamos interessados. Já para o caso do método `/get_historical_average_measures`, para satisfazer um pedido do utilizador pode levar a uma série de diferentes chamadas à API desta mesma fonte. Concluindo, a consulta de informações históricas da fonte da Netatmo é não só bastante trabalhosa e cheia de detalhes, como também para implementar a mesma pode levar a esgotar ainda mais facilmente o *plafond* de chamadas à API disponíveis.

- A interação entre o `Facade` e o `Weather Server`, pois num contexto de aplicação de novas aprendizagens pode fazer sentido esta troca de mensagens assíncrona entre estes dois componentes, que pressupõe que um pedido seja realizado por um cliente ao `Facade` que devolve inicialmente um *correlation identifier* que identifica esse mesmo pedido, que será depois tratado pelo `Weather Server` e enviada a resposta de volta para o `Facade`, que guarda o mesmo no seu dicionário e que seria levantado posteriormente pelo cliente quando este utilizar o método `/get_reply` para obter as informações requisitadas de uma forma assíncrona. No entanto, tendo em conta que o objetivo final desta arquitetura passa também por criar uma API que possa ser consumida por um *frontend* (*browser*, aplicação *mobile*, etc), esta abordagem torna-se insuficiente, visto que não faz sentido desenvolver um *frontend* que quando quiser pedir informações ao *backend* necessita de fazer um pedido inicial e depois ir fazendo *polling* periodicamente em curtos espaços de tempo para ver quando a resposta está pronta. Desta forma, a abordagem tem que obrigatoriamente ser outra para que um cliente obtenha a sua resposta de forma "imediata".

5.5 TERCEIRA ABORDAGEM

Tendo em conta os problemas mencionados em cima das abordagens anteriores, torna-se assim necessário pensar em alterações que devam ser feitas na arquitetura que os possam solucionar. A abordagem para a nova arquitetura é apresentada na Figura 36.

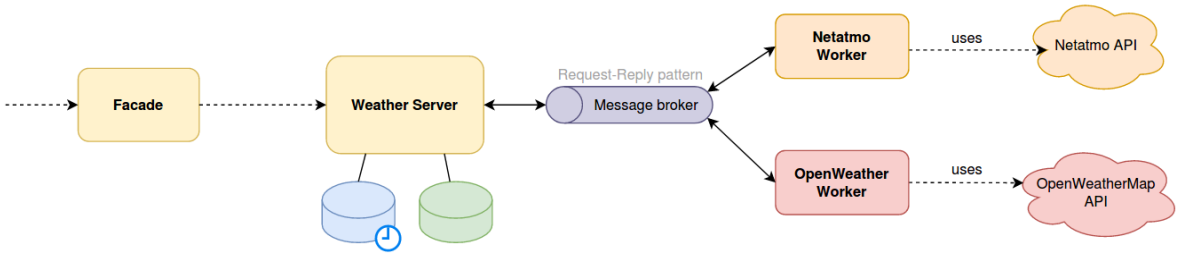


Figura 36: Terceira abordagem para a arquitetura do sistema.

Desta forma, importa perceber como esta arquitetura contorna os problemas descritos.

- Para resolver os dois primeiros problemas relacionados com o limite das chamadas e com a dificuldade das consultas históricas da API da Netatmo, a solução inicial passaria num primeiro plano por adicionar um serviço para esta mesma fonte como mencionado anteriormente para a API da OpenWeatherMap e ir assim construindo uma base de dados.
  Ainda assim, isto faria com que para solucionar um pedido do método `/get_area_data` por exemplo fosse necessário o `Weather Server` fazer um pedido do equivalente deste método para a fonte OpenWeatherMap ao componente `OpenWeather service` e o mesmo pedido para a fonte Netatmo a um possível novo componente `Netatmo service`, sendo depois necessário conjugar os dois resultados, para além da carga que é exercida na rede da arquitetura com tantas interações entre

o `Weather Server` e estes dois serviços que seriam necessárias. Assim, cada serviço de uma fonte teria de ter a sua própria base de dados que deveria ser consultada para dar resposta aos pedidos recebidos.

De forma a melhorar este cenário, a ideia passa por em vez de ter uma base de dados para cada serviço passar a ter uma base de dados no `Weather Server` que passa a ter a informação de todas as fontes disponíveis da consulta periódica das medições mais recentes, que é o que pode ser visto na Figura 36. Com isto, e para libertar o peso de ter que ser o `Weather Server` a fazer essas consultas para cada uma das fontes e até seguindo a ideologia de uma arquitetura em micro-serviços pretendida, opta-se pela solução de ter um *worker* para cada fonte de dados meteorológicos, `Netatmo Worker` e `OpenWeather Worker`, que interagem com o `Weather Server` através de uma comunicação assíncrona com recurso ao padrão *Request-Reply* através de um *broker*.

- Já para resolver o terceiro problema mencionado anteriormente, necessariamente o `Facade` continua a servir do ponto de entrada no sistema e de tratar de fazer filtragem de pedidos, mas o contacto com o `Weather Server` não pode obrigatoriamente ser feito de forma assíncrona. Assim, o `Facade` faz *proxy* dos pedidos válidos para o `Weather Server` que com base nas informações que tem em base de dados ou no seu sistema de *cache* responde "imediatamente".

Os *workers* são responsáveis por obter os dados das medições mais recentes das estações de cada uma das suas fontes e, sendo assim, interagem com a API da respetiva fonte de dados. Quanto ao `Weather Server`, este servidor é responsável por fazer pedidos aos *workers* das fontes disponíveis, ir construindo e evoluindo sempre a sua base de dados, cujas informações podem ser obtidas através dos *endpoints* que expõe. As entidades da camada de negócio deste servidor seriam as apresentadas em baixo no diagrama de classes da Figura 35, cujas medições da classe `Measure` resultam da união dos conjuntos de medições selecionadas de cada fonte de dados (neste caso, Netatmo e OpenWeatherMap), e que pode facilmente ser alterado com a adição de novas fontes que adicionem novas medições meteorológicas.

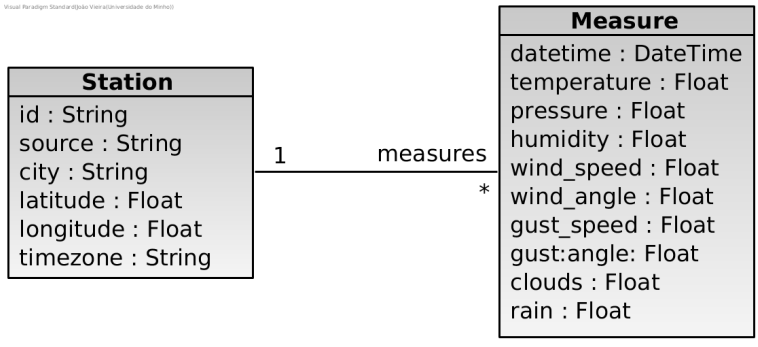


Figura 37: Modelo lógico do servidor Weather Server.

Para perceber melhor a interação que é feita entre o `Weather Server` e os dois *workers*, `Netatmo Worker` e `OpenWeather Worker`, a mesma é apresentada na Figura 38.

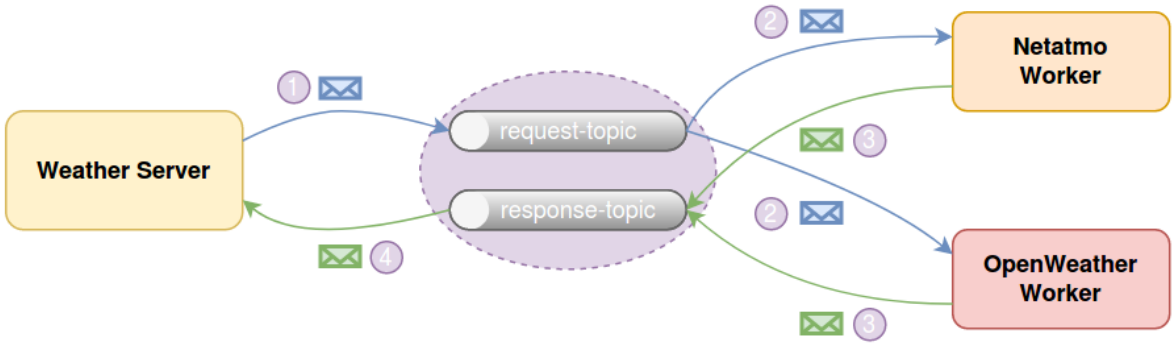


Figura 38: Padrão *Request-Reply* entre Weather Server e os *workers*.

A ideia para implementar o padrão *Request-Reply* para a comunicação entre estes componentes passa por criar dois tópicos no *broker*, o tópico `request-topic` e o tópico `response-topic`.

Assim, o `Weather Server` tem a tarefa de requisitar periodicamente (de hora a hora) as medições mais recentes das várias fontes de dados, enviando uma mensagem para o tópico `request-topic` cujo corpo da mesma indica que pretende essas mesmas medições mais recentes e qual o número de estações que pretende que sejam recebidas por cada mensagem (1), sendo ainda enviado no cabeçalho da mensagem qual o *correlation identifier* deste pedido e qual o tópico para o qual a resposta (ou respostas) ao mesmo deve ser enviada. Este pedido enviado por parte do `Weather Server` deve ser realizado ao minuto *XX*h30min de cada hora, devido a observações realizadas às respostas fornecidas pela API tanto da Netatmo como da OpenWeatherMap. Depois, como ambos os *workers* são subscritores deste tópico, ambos vão receber a mensagem (2) e vão tratar de interagir cada um com a API da sua fonte de dados para recolher as medições mais recentes das estações. Estas estações são separadas em várias mensagens consoante o número de estações por mensagens definido pelo `Weather Server` e enviadas para o tópico `response-topic` (3), enviando no cabeçalho de cada mensagem o *correlation identifier* do pedido. De seguida, sendo o `Weather Server` um subscritor do tópico `response-topic`, este vai receber estas mensagens dos *workers* de forma assíncrona e consegue atualizar a sua base de dados com as medições das estações mais recentes (4), sendo importante verificar que o *correlation identifier* de cada mensagem é válido.

Para além disto, esta abordagem permite que a adição de novas fontes de dados no futuro seja conseguida facilmente, através de alterações de pequenos detalhes/configurações no `Weather Server` e da criação de um novo *worker* para essa mesma fontes que trate também de ser capaz de receber os pedidos do *broker* e dar resposta aos mesmos.

Para perceber melhor a troca de mensagens, um exemplo do conteúdo da mensagem enviada periodicamente do `Weather Server` para os *workers* disponíveis é apresentado na Listagem 5.1, sendo utilizado um formato JSON que é fácil de ler e perceber.

```
{
    'method': 'get_last_measures',
    'arguments': {
        'stations_per_message': 10
    }
}
```

Listagem 5.1: Exemplo de conteúdo da mensagem enviada periodicamente pelo Weather Server.

Quanto às respostas que são enviadas pelos *workers*, são estas que contribuem para existir uma uniformização das várias fontes de dados. Um exemplo de uma destas mensagens enviadas pelo `OpenWeather Worker` é apresentada na Listagem 5.2.

```
[
    {
        'station_id': '123',
        'source': 'openweather',
        'location': {
            'latitude': 42.123,
            'longitude': -8.123,
            'city': 'Braga',
            'timezone': 'Europe/Lisbon'
        },
        'measures': {
            'temperature': 25,
            'pressure': 1010,
            'humidity': 91,
            'wind_speed': 2.34,
            'wind_angle': 55,
            'clouds': 40
        },
        'last_update': 1577836800
    },
    ...
]
```

Listagem 5.2: Exemplo de conteúdo das respostas enviadas pelos workers.

Desta forma, todos os *workers* devem seguir este formato de mensagens para conseguir uniformizar as informações de todas as fontes. Quanto ao corpo da mensagem em si, para cada estação deve ser indicado a identificação da mesma (`station_id`), a fonte de dados (`source`) que neste caso é da OpenWeatherMap (`openweather`), e deve ser indicado também o *timestamp* relativo à data em que esta medição foi efetuada pela estação (`last_update`). Para além disto, devem ser enviadas informações relativas à localização da própria estação: as coordenadas de latitude e longitude, o nome da localidade onde se encontra (uma cidade, uma rua, etc) e o *timezone* da mesma, que pode num futuro servir para adaptar as datas apresentadas

aos utilizadores de acordo com o local em que os mesmos se encontram por exemplo. Por fim, os campos pertencentes ao conjunto de medições de cada estação (`measures`) é que pode variar de estação em estação e de fonte para fonte, sendo que são apenas enviadas as medições que a estação em causa suporta. Estas medições devem ser todas elas convertidas pelos *workers* para as unidades apresentadas na Tabela 4, pois não faz sentido por exemplo um *worker* enviar a temperatura em graus Celsius e ter outro *worker* a enviar a mesma em graus Fahrenheit. Com isto, consegue-se assim que o `Weather Server` receba mensagens que podem todas elas ser processadas da mesma forma, devido à uniformidade definida e em que as medições obrigatoriamente pertencem a um conjunto resultante da união das várias fontes que se estão a utilizar.

Medição meteorológica (métrica)	Propriedade na API	Unidade
Temperatura	`temperature`	ºC (graus Celsius)
Pressão atmosférica	`pressure`	hPa (hectopascal) = mbar (millibar)
Humidade	`humidity`	%
Força / Velocidade do vento	`wind_speed`	km/h
Ângulo do vento	`wind_angle`	º (graus)
Força / Velocidade de rajadas	`gust_speed`	hm/h
Ângulo de rajadas	`gust_angle`	º (graus)
Nuvens	`clouds`	%
Chuva	`rain`	mm (milímetros de chuva)

Tabela 4: Unidades que os *workers* devem adotar para as medições das estações.

5.6 ADIÇÃO DE NOVOS MÉTODOS METEOROLÓGICOS À API

Sabendo agora que nesta abordagem arquitetural temos a nossa própria base de dados atualizada com a qual não necessitamos de ter limitações nos acessos feitos à mesma, podemos pensar então em novos métodos que podem ser úteis tanto num contexto informativo como num contexto de utilização num *frontend*. Assim, são adicionados novos métodos à API do componente `Weather Server` e consequentemente à API que o *backend* expõe, sem contar com os métodos `/get_area_data`, `/get_historical_measures`, `/get_area_average_measures` e `/get_historical_average_measures` já mencionados anteriormente:

- `/get_station_data` - a ideia deste método passa por obter as informações e as últimas medições de uma estação.
- `/get_closest_stations` - já o objetivo deste método passa por conseguir obter através do mesmo a informação e as últimas medições das estações que se encontram mais perto de uma determinada localização. A partir da localização, o objetivo passa por encontrar as estações que se encontram num raio de 4 km. Caso não exista nenhuma estação nesse mesmo raio, devolve-se a estação que se encontra mais perto. A ideia passa por poder utilizar este método de duas formas:
  - Fornecendo a latitude e longitude de um local.

  - Fornecendo um "*search term*" de um determinado local (Braga, Maximinos, Barcelos, por exemplo) e fazer com que a implementação deste método consiga converter esta pesquisa num conjunto de coordenadas (latitude e longitude).

5.7 MÉTODOS A FORNECER AOS UTILIZADORES

Tendo neste ponto já uma abordagem bem definida, que permite ir ao encontro das necessidades de consultas meteorológicas pensadas para a aplicação que se pretende desenvolver, importa agora introduzir as necessidades no que diz respeito às informações e ações que devem estar ao dispor dos utilizadores da aplicação.

Assim, importa definir os métodos que a REST API do *backend* da aplicação deve expor de acordo com as funcionalidades pensadas (como mencionado no Capítulo 4) e que envolvem estações meteorológicas, avisos, notificações, etc. Sendo assim, os *endpoints* implementados devem ser:

- `/register` - o objetivo do método passa por permitir que um novo utilizador se registe na aplicação, através de um *e-mail*, *username* e uma *password*.
- `/login` - método que permite realizar a autenticação de um utilizador através do *e-mail* e *password* do mesmo.
- `/get_notifications` - permite devolver a lista de notificações do utilizador.
- `/remove_notification` - permite eliminar uma notificação, sendo importante averiguar claro que o utilizador que está a efetuar a ação é o utilizador a quem a notificação é atribuída.
- `/get_favourite_stations` - permite devolver a lista das estações que foram marcadas como favoritas pelo utilizador.
- `/add_station_to_favourites` - método com o objetivo de oferecer ao utilizador a possibilidade de adicionar uma estação à lista das suas estações favoritas. No caso de já ter atingido um máximo de 5 estações favoritas (como mencionado anteriormente), o utilizador necessita primeiro de remover alguma.
- `/remove_station_from_favourites` - permite remover uma estação da lista de estações favoritas do utilizador. Por consequência, este método remove também os alertas relacionados com a esta estação.
- `/get_alerts` - permite obter a lista de alertas que o utilizador tem definidos.
- `/create_alert` - permite ao utilizador criar um novo alerta, para poder receber notificações relacionadas com o mesmo. Importante referir que a estação a que o alerta se refere tem que obrigatoriamente pertencer à lista de estações favoritas do utilizador.

- `/remove_alert` - por fim, o objetivo deste método é permitir ao utilizador remover um alerta que tenha sido definido anteriormente. Tal como o método `/remove_notification` (e todos os outros obviamente), é importante perceber que o utilizador que está a efetuar a ação é o mesmo a quem o alerta pertence.

5.8 MODIFICAÇÕES DA ARQUITETURA PARA INTRODUÇÃO DE NOVAS FUNCIONALIDADES

Neste altura, torna-se então necessário modificar a arquitetura apresentada na Secção 5.5 para incluir estas novas funcionalidades relacionadas com as consultas e ações que devem ser colocadas à disponibilidade dos utilizadores.

Desta forma, seguindo a ideia de uma arquitetura baseada em micro-serviços leva a não sobrecarregar componentes que já existem na arquitetura, sendo a opção considerada mais acertada a introdução de um novo componente, denominado `Client Server`, como pode ser visto na Figura 39.

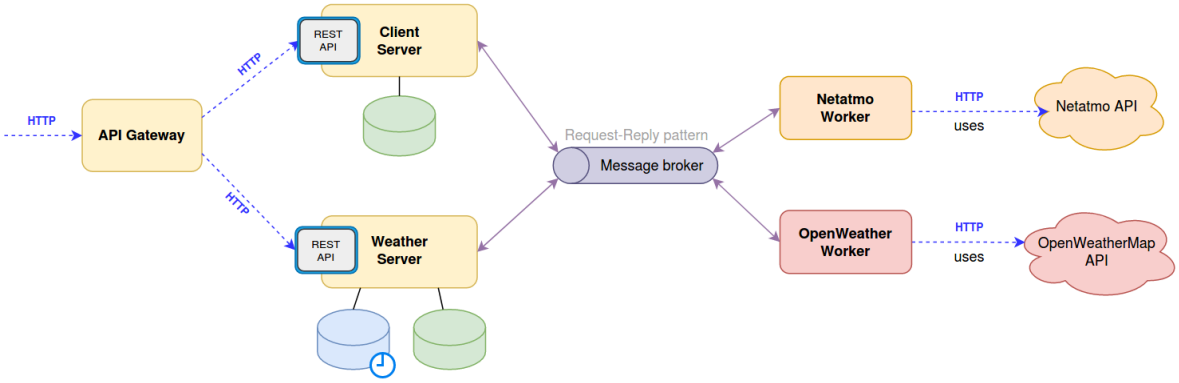


Figura 39: Abordagem final para a arquitetura do sistema com a introdução das funcionalidades oferecidas aos utilizadores.

Quando pensamos que o novo componente `Client Server` deve ser capaz de criar notificações de acordo com os alertas criados pelos utilizadores, percebemos que o mesmo deve ser capaz de consultar informações meteorológicas das estações das várias fontes. Assim, poderiam existir numa primeira fase de pensamento algumas soluções rápidas para este problema:

- o `Client Server` partilhar a mesma base de dados com o `Weather Server`, tendo assim fácil acesso às informações das estações.
- o `Client Server` ter a sua própria base de dados para guardar os utilizadores, avisos e notificações, mas ter também acesso direto à base de dados do `Weather Server`.
- o `Weather Server` expor um conjunto de métodos necessários para o `Client Server` consultar através da API deste componente.

No entanto, é necessário avaliar cada uma destas soluções. Em primeiro lugar, todas elas de certa forma vão criar uma carga maior no `Weather Server`, quer seja na execução do mesmo ou a aumentar o número de consultas na sua base de dados. A primeira hipótese apresentada para além do peso colocado em cima da base de dados conjunta, pode também apresentar alguns problemas na implementação do mesmo ao utilizar certas tecnologias ORM que pretendem que a base de dados seja totalmente controlada pela mesma, enquanto que neste caso duas aplicações gostariam de controlar a base de dados, para além de que alterações na base de dados levaria a alterações em ambos os componentes. Quanto à segunda hipótese, para além de uma dificuldade acrescida no `Client Server` ao ter que interagir com duas bases de dados, temos também os problemas mencionados para o primeiro caso de certa forma. Por último, a terceira hipótese tem a desvantagem de aumentar o tráfego na rede da arquitetura e de tirar algum tempo de processamento de pedidos dos utilizadores ao `Weather Server` visto que a sua API passa também a ser utilizada pelo `Client Server` (e possivelmente numa carga que pode ser considerada excessiva). Sendo assim, é melhor pensar noutra solução para este problema.

A solução escolhida passa por ter o novo componente `Client Server` a ter na sua própria base de dados as informações das estações e das suas medições. No entanto, de forma a não ter de certa forma a base de dados do `Weather Server` replicada na base de dados do `Client Server`, a ideia passa por ter apenas as últimas *N* medições de todas as estações, de forma a com elas poder processar as informações necessárias e criar as notificações dos utilizadores de acordo com os alertas definidos. Importante referir que apesar de não ser apresentado na Figura 39, também pode ser uma opção colocar ambas as bases de dados numa mesma base de dados, mas claro, com esquemas diferentes.

Relacionado com este tema das estações meteorológicas, é preciso então saber como pode o `Client Server` ter acesso às medições mais recentes das estações, de forma a evoluir a sua base de dados. Assim, a solução encontrada passa por envolver o `Client Server` no padrão *Request-Reply* através do qual os componentes `Weather Server` e os dois *workers* comunicavam, como explicado na Secção 5.5. Desta forma, a nova solução para a comunicação entre estes componentes passa a acontecer como demonstrado na Figura 40.

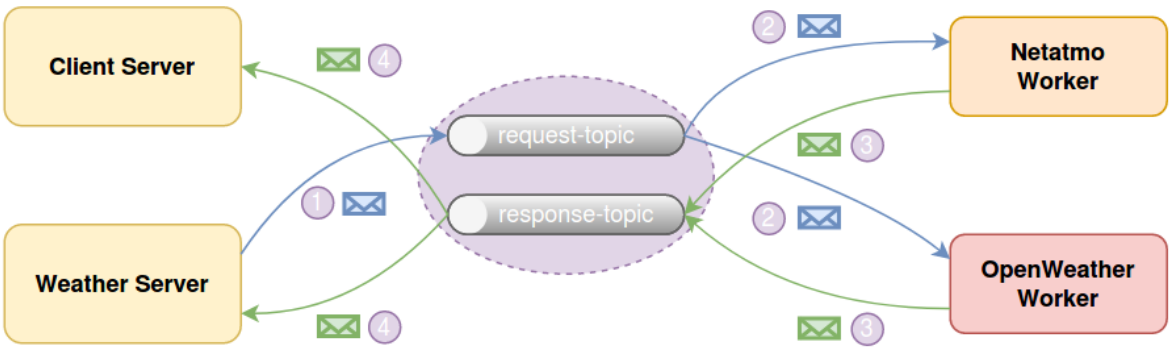


Figura 40: Padrão *Request-Reply* envolvendo o Client Server, o Weather Server e os workers.

Assim, continua a ser o `Weather Server` que pede periodicamente aos *workers* as medições mais recentes (1), mas, o tópico `response-topic` que recebe essas mesmas respostas passa a ter um novo

subscritor, o `Client Server`, que passa então assim a ter acesso a estas informações e consegue depois processar as mesmas da maneira pretendida (4).

Para se ter uma maior compreensão deste `Client Server`, é relevante falar sobre as entidades da camada de negócio deste servidor, sendo apresentado na Figura 41 o respetivo diagrama de classes.

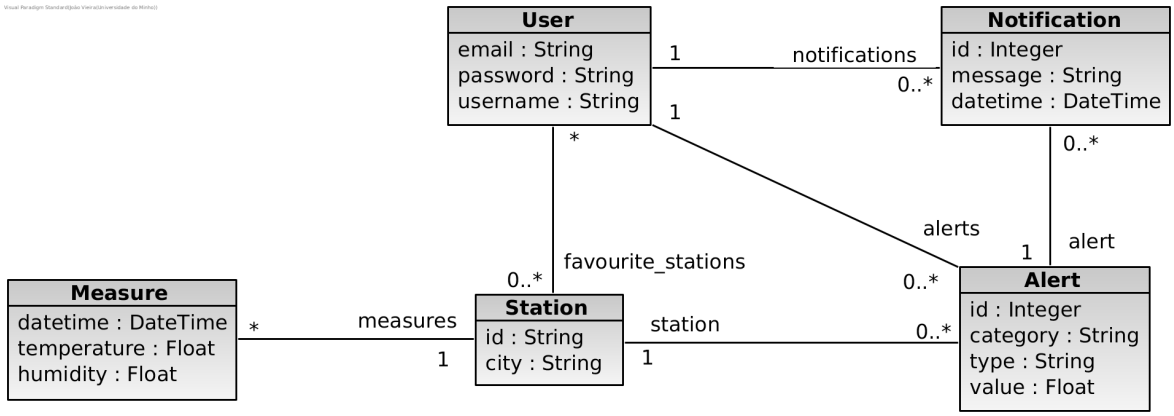


Figura 41: Modelo lógico do servidor Client Server.

Assim, segue-se a lógica utilizada no `Weather Server` em que uma estação (`Station`) tem um conjunto do medições (`Measure`), sendo que a única alteração é que aqui apenas são guardados os valores de temperatura e humidade, como mencionado anteriormente. Quanto às restantes classes, a classe `User` guarda os utilizadores da aplicação através de um *e-mail*, *username* e uma *password*, utilizador esse que tem um conjunto de estações favoritas, um conjunto de alertas definidos por si relativo a essas estações, e um conjunto de notificações resultante desses mesmos alertas. Um alerta (`Alert`) para além de um identificador e da estação a que está associado, tem uma categoria (`category`) devido aos vários tipos de alertas possíveis de registar, um tipo (`type`) que define se o alerta é relativo a temperatura ou humidade, e um valor (`value`) que é utilizado como referência para este alerta. Por último, uma notificação (`Notification`) para além do seu identificador e do alerta a que está associada, tem a mensagem que descreve a mesma (`message`) e uma data relativa à altura em que a mesma foi criada (`datetime`).

Para terminar esta secção, e voltando à arquitetura da Figura 39, é de extrema importância realçar que o componente `Facade` apresentado em abordagens anteriores, passa agora a dar a vez a um `API Gateway`. Tendo em conta que este tipo de componente numa arquitetura baseada em micro-serviços foi discutido na Secção 2.2.2, passa a ser um componente que inquestionavelmente trás valor à solução proposta. Sendo que nesta solução temos dois componentes que expõem cada um deles uma REST API que deve ser exposta aos utilizadores da aplicação (`Weather Server` e `Client Server`), a introdução de um `API Gateway` permite fazer a agregação dos *endpoints*, sendo o único ponto de entrada no sistema (similar ao padrão *Facade*) e que pode ainda exercer outras funcionalidades. Isto funciona para expor os métodos dos dois componentes referidos como pode servir mais *endpoints* consoante a adição de novos componentes em que também seja necessário expor APIs. Como outras vantagens, e tal como referido na Secção 2.2.2, a introdução deste novo

componente permite encapsular a estrutura interna da arquitetura e simplifica o código do cliente ao mover a lógica de interação com vários serviços do lado do cliente para o `API Gateway`.

5.9 INTRODUÇÃO DE MECANISMOS DE CACHING

Tendo sido definida uma proposta de arquitetura final para o *backend* da aplicação a desenvolver, importa perceber a forma como são introduzidos mecanismos de *caching* na mesma.

Tendo em conta que não faz sentido utilizar mecanismos de *caching* de cada uma das fontes nos *workers*, deve ser questionado então em que outros componentes estes mecanismos adicionam vantagens com a sua introdução. Como o `API Gateway` vai encaminhar os pedidos válidos recebidos tanto para o `Weather Server` como para o `Client Server`, em princípio são estes os componentes que podem ganhar um maior desempenho com a introdução de uma *cache*. Ainda assim, e tendo em conta que a aplicação pensada para ser desenvolvida deve funcionar tanto para utilizadores registados como para não registados, facilmente percebemos que a maior carga de pedidos vai ser atribuída garantidamente ao `Weather Server`, visto que o `Client Server` apenas receberá pedidos de utilizadores registados e até porque é o próprio `Weather Server` o responsável por dar resposta às questões sobre as estações meteorológicas e as suas medições, que é o grande interesse de uma aplicação deste tipo para os utilizadores, quer sejam registados ou não. Com isto, surge então a decisão de introduzir mecanismos de *caching* no componente `Weather Server`, como mostrado anteriormente na Figura 39.

Tomada esta decisão, importa então perceber qual a melhor forma para este componente ter um sistema de *caching*. Tendo em conta que os *endpoints* que este componente expõe apresentam parâmetros que muito dificilmente são repetidos ou utilizados muitas vezes num curto espaço de tempo, tal como coordenadas de latitudes e longitudes ou a utilização de datas com horas e minutos, percebe-se que a introdução de estratégias de *caching* nestes métodos não é do todo facilitada.

Sendo assim, a solução proposta passa por introduzir algoritmos de *caching* mais "pensados" e refinados para cada método, de forma a que faça sentido adicionar este tipo de mecanismos e com que o `Weather Server` ganhe com o mesmo.

A ideia passa por utilizar um sistema de *cache in-memory*, como o Redis ou o Memcached, e armazenar pares chave-valor, ou seja, fazer *caching* aplicacional (ver Secção 2.4.2). Dentro deste paradigma, o objetivo passa por de algum modo fazer *caching* de queries feitas à base de dados. Uma diferença é que neste caso depois de pedidos os dados da base de dados podem ainda ser feitos processamentos ou cálculos e serem os resultados finais a serem guardados em *cache*, e não apenas as respostas às *queries* feitas à base de dados. Outra diferença é que normalmente é feito um *hash* da *query* que serve como a chave para o par a guardar em *cache*, enquanto que nesta solução não vai ser realizado esse mesmo *hash* mas vai ser sim definido um tipo de chaves que podem ser utilizadas, como mais à frente é explicado.

Em termos de estratégia de *caching* o objetivo passa por ter a *cache* ao lado da base de dados como na estratégia *Cache Aside* explicada na Secção 2.4.3, pois a ideia passa por ter o `Weather Server` a pedir

primeiro os dados à *cache* e, se a informação existir é devolvida diretamente, caso contrário, o `Weather Server` vai requisitar os dados à base de dados e escrever os mesmos em *cache* para que possam ser utilizados numa próxima vez. Importante referir que a aplicação que neste caso é o `Weather Server` é que escreve diretamente na base de dados e só depois os dados lidos é que vão para *cache*, como define o padrão *Write Around*.

Assim, os algoritmos pensados para cada um dos métodos que se pretende expor são os seguintes:

- `/get_station_data`
  - Guardar em cache: o objetivo é guardar em *cache* a resposta a um pedido processado com a chave "`station_data_<station_id>`". Tendo em conta que o processo de atualização das medições é realizado ao minuto 30 de cada hora, o objetivo passa também por colocar esta chave a ser expirada por volta do próximo minuto *XX*h30min, altura a partir da qual a informação em *cache* deixa de ser válida.
  - Obter da cache: quando é feito um pedido deste tipo para uma determinada estação verifica-se primeiro se esta chave já se encontra em *cache*.
- `/get_area_data`
  - Guardar em cache: o objetivo é guardar uma chave do tipo "`area_data_<lat_ne>_<lon_ne>_<lat_sw>_<lon_sw>`" que permite perceber qual a área que foi requisitada e colocar a data de expiração também no próximo minuto *XX*h30min.
  - Obter da cache: inicialmente, uma primeira abordagem passaria por quando recebido um pedido deste método, verificar se das chaves que estão em *cache* existe alguma em que a diferença entre o canto sudoeste requisitado e o que se encontra em *cache* é inferior a 1 km (por exemplo) e fazer a mesma verificação para o canto nordeste. Ainda assim, este algoritmo não é ideal, pois tanto uma área pode ser grande como pequena e a distância tida em conta (1 km) é sempre a mesma, que pode não ser ajustado para uma área muito grande e pode também ser grande demais para uma área pequena. Sendo assim, a segunda abordagem e abordagem final passa por, utilizando o cálculo da área das áreas em *cache* e da área requisitada, calcular a intersecção da área requisitada e da área em *cache* e, se a área da intersecção da área requisitada e uma das áreas em *cache* for de pelo menos 80% (valor que pode ser ajustado), então é utilizada a resposta já em *cache*. Para conseguir isto e realizar estes cálculos, é tido em conta que as áreas se encontram num gráfico de eixos *XY* (latitude e longitude), o que permite assim verificar a intersecção das áreas.
- `/get_closest_stations`
  - Guardar em cache: não faz sentido guardar as procuras feitas com base numa latitude e numa longitude, visto que dificilmente as mesmas vão ser repetidas num curto espaço de tempo. Sendo assim, apenas são guardadas as pesquisas feitas com um *search term* através de uma chave "`closest_<search_term>`" (por exemplo "closest_braga"). A data de expiração é também o próximo minuto *XX*h30min.

  - Obter da cache: quando um pedido é feito com o *search term* "Braga" por exemplo, a *string* é convertida em minúsculas e são removidos espaços no início e no fim que possam existir, e procura-se se já existe em *cache* a chave "closest_braga".

- `/get_historical_measures`
  - Guardar em cache: apenas faz sentido guardar em *cache* os pedidos que são feitos com recurso a intervalos de tempo definidos, ou seja, utilizando um parâmetro `interval` vamos assim supor ("`1day`" para as medições das últimas 24 horas por exemplo), visto que os pedidos feitos com intervalos de datas bem definidas dificilmente serão repetidas num curto espaço de tempo. Sendo assim é guardada em *cache* uma chave do tipo "`historical_<station_id>_<interval>_<type>_<scale>_<parser>`" sendo
    `type` por exemplo temperatura, `scale` a escala entre duas medições e o *parser* o tipo de formato de apresentação dos resultados deste método. Também a chave deste método expira no próximo minuto *XX*h30min.
  - Obter da cache: quando um pedido é realizado com recurso ao parâmetro `interval` é verificado se já existe em *cache* a respetiva chave.

- `/get_area_average_measures`
  É utilizado basicamente o mesmo algoritmo que o apresentado para o método `/get_area_data` mas aqui a chave passa a ser do tipo "`area_average_<lat_ne>_<lon_ne>_<lat_sw>_<lon_sw>`".

- `/get_historical_average_measures`
  - Guardar em cache: assim como o *endpoint* `/get_historical_measures`, apenas faz sentido guardar em *cache* os pedidos efetuados com um parâmetro `interval`. Sendo assim é guardada em *cache* uma chave do tipo "`historical_average_<station_id>_<interval>`". No que toca ao tempo de expiração da chave, este é o único método que pode ser considerado não ser sempre o próximo minuto *XX*h30min. A ideia passa por perceber que em intervalos maiores o tempo de expiração pode ser maior pois, por exemplo, para um intervalo de uma semana em que são esperadas ter 168 medições para uma estação (24 * 7), não é apenas mais uma medição que vai alterar de forma significativa uma média. Sendo assim, podem ser definidas datas de expiração para os intervalos da seguinte forma:
    * 3hours - próximo minuto *XX*h30min.
    * 12hours - próximo minuto *XX*h30min + 1 hora.
    * 1day - próximo minuto *XX*h30min + 1 hora.
    * 3days - próximo minuto *XX*h30min + 3 horas.
    * 1week - próximo minuto *XX*h30min + 3 horas.
    * 1month - próximo minuto *XX*h30min + 5 horas.

Tenta-se assim ter um compromisso entre não haver a necessidade de remover logo certas chaves da *cache* visto que os valores não vão ser muito alterados, mas ainda assim não exagerar no tempo que se acrescenta às datas de expiração.

– Obter da cache: quando um pedido é realizado com recurso ao parâmetro `interval` é verificado se já existe em *cache* a respetiva chave.

Por fim, e para encerrar esta secção das decisões ao nível dos mecanismos de *caching*, é importante ter em conta que o espaço de uma máquina, e consequentemente o espaço de um sistema de *caching*, não é infinito. Desta forma, é importante para além de adicionar um limite de espaço ao sistema de *caching*, considerar ainda quais as ações deste mesmo sistema quando a sua memória se encontra totalmente ocupada e ainda assim queremos adicionar novas informações à *cache*. Para isso, é importante então escolher uma política de despejo, algo estudado na Secção 2.4.3. A opção para este sistema de *caching* do `Weather Server` considerada mais acertada é a utilização do algoritmo Least Frequently Used, que consiste em eliminar os elementos menos usados da *cache* em primeiro lugar. Complementarmente, podemos ainda considerar que existe um algoritmo de despejo que toma em conta o fator tempo, que é a implementação de tempos de expiração das chaves, visto que a maioria destas expiram em menos de uma hora a partir do momento em que foram adicionadas.

5.10 DISCUSSÃO

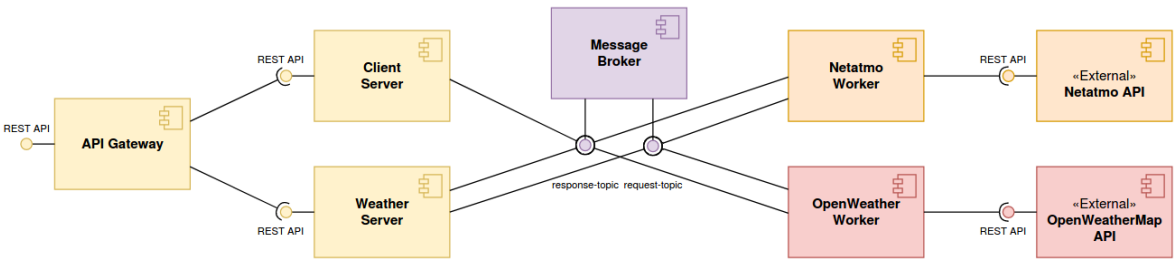


Figura 42: Representação da arquitetura final.[1]

Tendo então a arquitetura final a implementar definida (Figura 39), e cuja representação da mesma através de um diagrama de componentes pode ser vista na Figura 42 , podemos assim fazer um balanço da mesma.

Em primeiro lugar, temos em teoria uma arquitetura que vai ao encontro dos requisitos e que permitirá ir ao encontro das funcionalidades esperadas com o desenvolvimento da mesma.

Para além disto, consegue-se assim fazer a introdução de *message brokers* utilizados na comunicação entre os componentes. São ainda introduzidos também os mecanismos de *caching* no `Weather Server`, que se espera que ofereçam um maior desempenho não só a este componente como ao sistema em geral.

Ao seguir uma arquitetura baseada em micro-serviços e face a todas estas caraterísticas, podemos também considerar que temos uma solução que vai ao encontro da escalabilidade pretendida, e que pode evoluir

1 Neste diagrama é utilizado o estereótipo «`External`» para representar um sistema externo (as APIs da Netatmo e da OpenWeatherMap).

ao longo do tempo e conforme as necessidades a partir deste ponto. Sendo assim, podem por exemplo ser adicionados novos componentes ao sistema para permitir adicionar novas funcionalidades ou podem ainda ser modificados os componentes atuais mais facilmente, uma vantagem de uma abordagem em micro-serviços face às típicas arquiteturas monolíticas.

Para além disto, em caso de necessidade devido a um grande número de utilizadores e em que uma possível monitorização indique que se torne necessário a introdução de mais máquinas, esta arquitetura permite aumentar o número de instâncias individualmente, ao adicionar um maior número de instâncias ao `Weather Server` e `Client Server`, fazendo balanceamento de pedidos entre as várias instâncias de cada um, ou até introduzir um *cluster* para cada um dos *workers* de forma a garantir disponibilidade em caso de falha, tendo desta forma *workers* em espera, prontos para assumir em caso de falha. Assim, uma definição arquitetural mais elaborada para uma aplicação deste tipo pode ser vista na Figura 43.

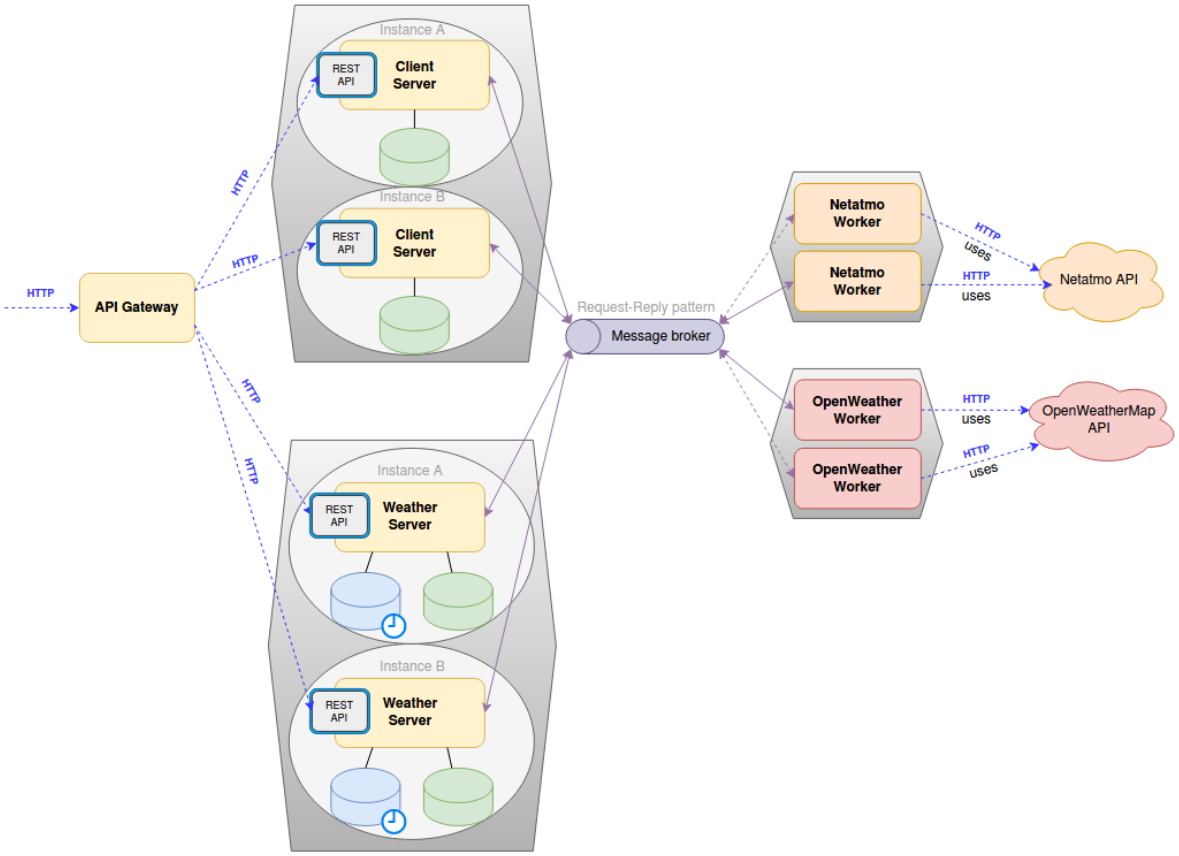


Figura 43: Possibilidade de inclusão de mais instâncias nos componentes da arquitetura.

6

DESENVOLVIMENTO

Esta capítulo serve para descrever a implementação da arquitetura final apresentada no Capítulo 5, sendo apresentadas decisões técnicas, opções sobre as ferramentas e tecnologias a utilizar, os desafios e os principais detalhes de todos os componentes desta arquitetura. Para além disto, é ainda explicado por fim o desenvolvimento da aplicação de *frontend*.

6.1 COMPONENTE - MESSAGE BROKER

Um primeiro passo na implementação desta arquitetura passa pela decisão do *message broker* a utilizar e pela configuração do mesmo, visto que estas decisões vão ter impacto também na implementação dos componentes que interagem com o mesmo, ou seja, o `Weather Server`, o `Client Server` e os dois *workers*, `OpenWeather Worker` e `Netatmo Worker`.

Sendo assim, foi previamente feito um estudo na Secção 2.3 sobre duas das tecnologias mais utilizadas como mediadores no desenvolvimento de aplicações, o Apache ActiveMQ e o Apache Kafka. Face às qualidades e caraterísticas de cada um, é importante perceber que cada um tem as suas, e cada um pode ser a melhor opção para certos contextos, pelo que o mais importante passa por analisar as necessidades que o sistema que pretendemos implementar tem. Desta forma, percebemos que como descrito no Capítulo 5, o objetivo passa pela criação de dois tópicos, `request-topic` e `response-topic`, que na prática são dois tópicos que seguem o modelo *Publish-Subscribe*. No `request-topic`, o produtor de mensagens é o `Weather Server`, enquanto que os consumidores dessas mesmas mensagens e assim subscritores do tópico são o `OpenWeather Worker` e o `Netatmo Worker`. Já no caso do `response-topic`, os produtores de mensagens são estes *workers*, enquanto que os consumidores das mensagens enviadas por estes componentes são o `Weather Server` e o `Client Server`. Por estas necessidades, basicamente seria necessário um *broker* com a capacidade de criar tópicos *Publish-Subscribe*, pelo que ambas as tecnologias resultariam. Com isto, importa então ter em conta outras caraterísticas que ajudem na decisão. Tal como relatado na Secção 2.3.3, o Kafka surge com o intuito de resolver algumas limitações dos *brokers* tradicionais e tenta tornar possível uma arquitetura baseada no padrão *Universal Data Pipeline* [17], o que facilita a implementação de arquiteturas em muitos contextos. Para além disto, o Kafka é bastante rápido e permite um grande processamento de mensagens num curto espaço de tempo, o que é bastante interessante no contexto

da carga de mensagens enviadas por parte dos *workers* na tarefa periódica de atualização das medições mais recentes das estações. Para além disto, e tal como referido anteriormente, é possível paralelizar o consumo de mensagens através do mecanismo de partições que pode ser bastante útil na redução do processamento de todas as mensagens sobre as estações meteorológicas por parte do `Weather Server` e do `Client Server`, para além de que são estas mesmas partições que permitem ao *broker* escalar horizontalmente em caso de necessidade. Como constatado anteriormente, claro que para ser tão rápido o Kafka passa algumas responsabilidades que seriam de um *broker* tradicional para o lado do cliente, em que certos detalhes devem ser tidos em conta na implementação de outros componentes como será relatado mais à frente. Ainda assim, mesmo considerando e supondo que a perda de mensagens no Kafka possa ser mais provável de acontecer comparativamente ao ActiveMQ, devemos ter em conta novamente que o Kafka oferece um grande *throughput* e, mesmo em caso de perca esporádica de alguma mensagem, face ao contexto da aplicação que estamos a desenvolver, pode não ser considerado grave, visto que as mensagens são de um caráter informativo e em que não é uma mensagem das medições mais recentes de algumas estações que tem um grande impacto na aplicação. Sendo assim, e também face à interação já feita com o Kafka como explicado na Secção 3.1, opta-se pela utilização desta ferramenta como *message broker* na arquitetura a desenvolver.

Quanto às configurações do mesmo, e tal como mencionado na Secção 2.3.3, para executarmos o Kafka, necessitamos primeiramente de iniciar a ferramenta ZooKeeper. A sequência de comandos é apresentada na Listagem 6.1, sendo que, inicialmente, o desenvolvimento é tido em conta com apenas um servidor Kafka devido à maior carga de mensagens se concentrar em apenas alguns minutos de hora em hora. Ainda assim, face à escalabilidade pretendida, isto pode ser alterável no futuro consoante novas necessidades.

```
$ ./zookeeper-server-start.sh config/zookeeper.properties
$ ./kafka-server-start.sh config/server.properties
```

Listagem 6.1: Comandos de inicialização do Kafka.

Já para criar os dois tópicos, `request-topic` e `response-topic`, são utilizados os dois comandos apresentados na Listagem 6.2. Importa realçar em ambos a configuração do parâmetro `--partitions` que, enquanto no `request-topic` é utilizado o valor *default* devido à pouca quantidade de mensagens que o tópico vai ter, no tópico `response-topic` são utilizadas duas partições com o intuito de que os componentes `Weather Server` e `Client Server` possam paralelizar o consumo e processamento das mensagens sobre as estações meteorológicas e possam assim fazer o processo de atualização de dados num melhor tempo.

```
$ ./kafka-topics.sh --create --bootstrap-server localhost:9092 \
     --replication-factor 1 --partitions 1 --topic request-topic
$ ./kafka-topics.sh --create --bootstrap-server localhost:9092 \
     --replication-factor 1 --partitions 2 --topic response-topic
```

Listagem 6.2: Comandos de criação dos tópicos request-topic e response-topic.

6.2 COMPONENTE - OPENWEATHER WORKER

Este componente, tal como referido anteriormente, deve ser capaz de receber os pedidos periódicos do `Weather Server` e assim enviar as medições mais recentes das estações da fonte OpenWeatherMap.

Sendo assim, este componente tem que receber e produzir mensagens através do *message broker* Kafka, deve interagir com a API da OpenWeatherMap e deve ser capaz de extrair as informações necessárias das respostas fornecidas por esta API.

A implementação deste componente é feita na linguagem Python, devido à facilidade que a mesma oferece para manipulação de formatos JSON através dos dicionários da própria linguagem, e permite ainda uma maior facilidade de comunicação com o Kafka através da utilização da biblioteca `kafka-python`[1].

Em primeiro lugar, deve ser delineada uma estratégia para ter acesso a todos os identificadores das estações meteorológicas desta fonte no território português e, depois, conseguir acesso a informações e às medições mais recente de cada uma dessas estações, que será o que o *worker* terá que fazer a cada hora.

Assim, é possível obter uma lista de todas as estações desta fonte através da utilização do URL `http://bulk.openweathermap.org/sample/city.list.json.gz`, que dá acesso a um ficheiro JSON com uma lista de estações meteorológicas, em que cada estação tem o formato da Listagem 6.3.

```
{
    'id': 1234567,
    'name': 'Braga',
    'state': '',
    'country': 'PT',
    'coord': {
        'lon': 42.12345,
        'lat': -8.12345
    }
}
```

Listagem 6.3: Informações sobre uma estação devolvida pela listagem de estações da OpenWeatherMap.

Desta forma, a ideia passou por primeiro processar esta lista e convertê-la num ficheiro `openweather_stations.json` em que da lista inicial são apenas selecionadas estações localizadas em Portugal e selecionando de cada estação apenas o seu identificador, a localidade e as suas coordenadas, que resulta numa lista JSON em que cada estação é um objeto com um conteúdo como apresentado na Listagem 6.4.

```
{
    'station_id': 1234567,
    'city': 'Braga',
    'lat': -8.12345,
    'lon': 42.12345
}
```

Listagem 6.4: Informações selecionadas sobre uma estação da OpenWeatherMap.

1 https://kafka-python.readthedocs.io/en/master/

Depois, e tendo acesso aos identificadores de todas as 3231 estações localizadas em Portugal, isto permite utilizar o *endpoint* `http://api.openweathermap.org/data/2.5/group` que através do parâmetro `id` deixa passar até 20 identificadores de estações e obter as últimas medições de cada uma delas. A informação devolvida pela API para cada estação é mostrada na Figura 44. Sendo assim, de todos estes detalhes devolvidos são obtidos os valores da temperatura (`main.temp`), da pressão atmosférica (`main.pressure`), da humidade (`main.humidity`), da velocidade do vento que necessita de ser convertida para km/h (`wind.speed`), do ângulo do vento (`wind.deg`), das nuvens (`clouds.all`), do *timestamp* em que esta medição foi realizada (`dt`) e é ainda utilizado o campo `sys.timezone` para obter o *timezone* da estação. Informações úteis mas que já temos em nossa posse são as coordenadas da estação, o seu identificador e a localidade.


```
{
  - coord: {
        lon: -8.12345,
        lat: 42.12345
    },
  - sys: {
        country: "PT",
        timezone: 3600,
        sunrise: 1601618400,
        sunset: 1601661600
    },
  - weather: [
      - {
            id: 803,
            main: "Clouds",
            description: "nuvens quebradas",
            icon: "04n"
        }
    ],
  - main: {
        temp: 16.43,
        feels_like: 13.52,
        temp_min: 15.54,
        temp_max: 17.12,
        pressure: 1011,
        humidity: 100
    },
    visibility: 6000,
  - wind: {
        speed: 6.5,
        deg: 240
    },
  - clouds: {
        all: 80
    },
    dt: 1601640600,
    id: 1234567,
    name: "Braga"
}
```


Figura 44: Informação de uma estação meteorológica devolvida pela API da OpenWeatherMap.

Com isto, sempre que o consumidor deste componente receber um pedido de atualização das medições mais recentes, percorre então as 3231 estações desta fonte e utiliza o *endpoint* mencionado em cima para pedir as informações pretendidas em blocos de 20 estações para usar da melhor maneira possível os limites da API. Depois, processa as respostas e seleciona as informações como descrito e, tendo então a informação atualizada de todas as estações, divide as mesmas de acordo com o número definido pelo `Weather Server` no seu pedido e envia cada um destes conjuntos de estações numa mensagem para o Kafka.

Quanto à utilização da biblioteca `kafka-python`, esta permite definir um consumidor e um produtor de mensagens que são utilizados. O consumidor é definido através de um objeto `KafkaConsumer` que pode por exemplo ser definido como apresentado na Listagem 6.5.

```
self.consumer = KafkaConsumer(
    'request-topic',
    bootstrap_servers='localhost:9092',
```

```
    auto_offset_reset='latest',
    group_id='openweather_worker',
    value_deserializer=self.value_deserializer
)
```

Listagem 6.5: Exemplo de definição de um consumidor Kafka utilizando a biblioteca kafka-python.

Neste consumidor é definido o campo `group_id` que permite definir o *consumer group* do consumidor e o campo `value_deserializer` que define a forma como as mensagens recebidas são desserializadas, que neste caso é um método que permite verificar se a mensagem recebida é um objeto JSON depois de fazer *decoding* de `utf-8`, para por exemplo evitar o processamento de mensagens inválidas. Outras propriedades que este consumidor permite definir, e que foram mencionadas na Secção 2.3, são os campos `enable_auto_commit` que por omissão toma o valor de `True` e que define que a operação de *commit* do *offset* do consumidor seja realizada periodicamente em *background*, e o campo `auto_commit_interval_ms` que define os milissegundos entre dois *commits*, que toma o valor de 5000 milissegundos por defeito.

Quanto ao produtor de mensagens, este pode ser definido como apresentado na Listagem 6.6, em que o `value_serializer` permite fazer o inverso do que foi mencionado em cima, ou seja, serializa a mensagem no formato JSON e codifica a mesma em `utf-8`.

```
self.producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)
```

Listagem 6.6: Exemplo de definição de um produtor Kafka utilizando a biblioteca kafka-python.

Por fim, para enviar as mensagens com as informações das estações, é utilizado o método exposto na Listagem 6.7 que utiliza este mesmo produtor e envia ainda no cabeçalho o *correlation identifier* do pedido recebido.

```
def send_message(self, value):
    self.producer.send(
        self.responseTo,
        value = value,
        headers = [('correlationId', str.encode(self.correlationId))]
    )
```

Listagem 6.7: Método utilizado para enviar as informações das estações para o Kafka.

6.3 COMPONENTE - NETATMO WORKER

A implementação deste *worker* segue a mesma ideia do `OpenWeather Worker`, tendo em conta que os objetivos deste são os mesmos com a diferença deste interagir com a API da Netatmo.

Desta forma, é necessário então perceber qual a estratégia a seguir para ter acesso a todos os identificadores das estações meteorológicas desta fonte e conseguir aceder às informações das medições mais recentes das mesmas.

A primeira dificuldade passa logo por não existir qualquer listagem ou algum URL que permita ter acesso aos identificadores das estações que se encontram em Portugal. Para além disto, mesmo tendo acesso a uma possível listagem, iria ser praticamente impossível ter acesso às medições meteorológicas mais recentes de todas as estações, tendo em conta que o único *endpoint* que poderia ser utilizado para o efeito seria o `Getmeasure` mas seria necessário também ter acesso a todos os módulos que a estação tem e saber ainda quais as medições que cada módulo fica responsável por recolher, o que, mesmo que não fosse impossível, iria exigir várias chamadas à API para recolher informação de apenas uma estação, que poderia levar a esgotar o *plafond* do limite de chamadas facilmente. Tendo em conta este problema, foi ainda tentado contactar a equipa de desenvolvimento da API da Netatmo para tentar perceber qual a melhor estratégia a seguir, mas infelizmente não conseguiram adiantar grande ajuda visto que não podia ser conseguido mais nada através desta abordagem.

Sendo assim, foi necessário delinear outra abordagem para este problema. A solução passou ainda por, podendo usar o *endpoint* `Getpublicdata` que devolve informações das estações que se encontram numa dada área e as suas medições mais recentes, dividir então o território português em várias áreas e conseguir obter as informações das estações que se encontram inseridas em todas elas.

Para concretizar isto, a primeira tarefa passou por dividir o território continental português em várias áreas, tendo sido desenvolvido um algoritmo que com base nas latitudes e longitudes do canto nordeste e sudoeste de uma área que apanhe todo o território desejado, e definido um número de linhas e colunas, divide essa mesma área em *linhas * colunas* áreas, escrevendo as coordenadas `lat_sw`, `lon_sw`, `lat_ne` e `lon_ne` de todas as áreas num ficheiro JSON `portugal_areas.json`. Foi utilizada a biblioteca `shapely`[2] para ajudar a concretizar este algoritmo. As coordenadas iniciais da área que cobre todo o território continental definidas são apresentadas na Listagem 6.8.

```
lat_sw = 36.812906
lon_sw = -9.682215
lat_ne = 42.142322
lon_ne = -6.056727
```

Listagem 6.8: Coordenadas iniciais que cobrem o território continental português.

Esta área processada tendo em conta o número 5 definido para o número de linhas e 18 para o número de colunas resulta num total de 90 áreas, em que cada área tem então uma informação como exibida na Listagem 6.9.

```
{
    'lat_sw': 38.293299,
    'lon_sw': -9.682215,
    'lat_ne': 38.589378,
```

2 https://shapely.readthedocs.io/en/stable/

```
    'lon_ne': -8.957117
}
```

Listagem 6.9: Conteúdo de uma área calculada através da divisão do território continental português.

Estas coordenadas representadas em cima de uma mapa permitem então averiguar as 90 áreas definidas, demonstradas na Figura 45.

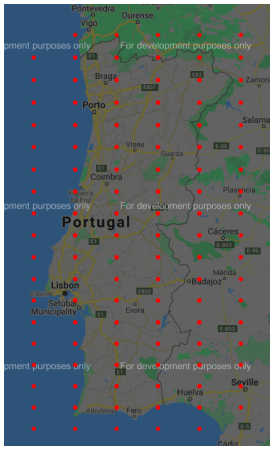


Figura 45: Divisão do território continental português em 90 áreas.

Depois disto, foi feito um trabalho manual de perceber quais são as áreas que não apanham qualquer parte do território português (pois encontram-se em Espanha ou no mar), de forma a não gastar chamadas desnecessárias à API. Para além disto, foram ainda melhoradas as coordenadas de algumas áreas para abranger um pouco mais de espaço e não serem necessárias outras áreas a ocuparem apenas um curto espaço de território português. As áreas finais são apresentadas na Figura 46, estando as coordenadas das que foram melhoradas marcadas a amarelo.

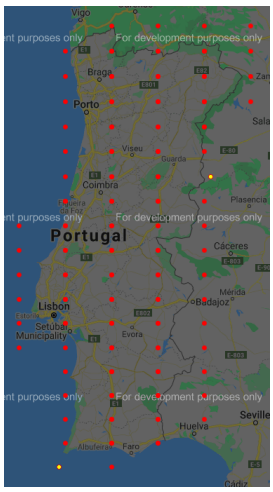


Figura 46: Divisão do território continental com remoção de áreas desnecessárias e melhoria de outras.

Assim, conseguimos obter 63 áreas às quais foram ainda adicionadas manualmente mais duas áreas, que cobrem o arquipélago da Madeira e dos Açores, apresentadas na Listagem 6.10.

```
{
    'lat_sw': 32.316004,
    'lon_sw': -17.398456,
    'lat_ne': 33.143154,
    'lon_ne': -16.244892
},
{
    'lat_sw': 36.714455,
    'lon_sw': -31.654732,
    'lat_ne': 39.907424,
    'lon_ne': -24.661934
}
```

Listagem 6.10: Áreas que cobrem os arquipélagos da Madeira e dos Açores.

Sendo assim, sempre que este *worker* receber um pedido de atualização das medições mais recentes, percorre então estas 65 áreas e para cada uma obtém as estações que se encontram nas mesmas e as respetivas medições destas através do *endpoint* `https://api.netatmo.com/api/getpublicdata`. Da resposta para cada estação obtida (que pode ser vista na Figura 29) são depois obtidas as informações necessárias fazendo também uma filtragem das estações que se localizam em Portugal (visto que algumas áreas apanham território português e espanhol). Como nesta API cada módulo tem uma data que corresponde ao momento da última medição, foi decidido utilizar o *timestamp* da medição de temperatura, de forma a seguir o formato de uniformização de fontes de dados definido. Para além do cuidado com o refrescar do *token* já mencionado na Secção 3.1, um outro cuidado tido em conta foi depois de obter os dados de todas as áreas, passar essa lista por um método que permite remover as estações duplicadas. A necessidade deste método é comprovada quando, por exemplo, no total das áreas são obtidas normalmente mais de 1500 estações, que depois de passadas por este filtro resultam apenas num número abaixo das 500, pois uma estação pode aparecer no resultado obtido para várias áreas.

Se existe um problema que se pode apontar a esta abordagem é que não podemos garantir que vamos sempre obter as medições mais recentes das mesmas estações ao longo do tempo, visto que o número final de estações obtidas varia facilmente. Isto pode levar assim a que algumas estações depois na base de dados a construir por exemplo pelo `Weather Server` tenham alguns intervalos em que não foi possível obter as medições das mesmas. Ainda assim, temos que considerar que isto é algo que não pode ser ultrapassado pois não depende desta implementação mas sim do *endpoint* `Getpublicdata` da Netatmo que não nos garante que para uma área vão sempre ser devolvidas as mesmas estações, caso contrário isto não aconteceria.

6.4 COMPONENTE - WEATHER SERVER

Para implementar este servidor importa ter em atenção as necessidades mais relevantes a que o mesmo deve ser capaz de responder:

- Enviar aos *workers* os pedidos das medições mais recentes das estações ao minuto 30 de cada hora.
- Receber assincronamente as respostas enviadas pelos *workers*, processar as mesmas, e com isso ser capaz de atualizar a sua base de dados.
- Fornecer uma REST API com os seis métodos definidos anteriormente no Capítulo 5.
- Fazer *caching* de resultados para poderem posteriormente ser utilizados e implementar os algoritmos pensados para cada *endpoint* relacionados com estes mesmos mecanismos.

Tendo em conta que temos uma base de dados relacional neste componente (a Figura 37 apresenta o diagrama de classes da mesma), seria uma vantagem utilizar uma *framework* que fosse capaz de fornecer um Object Relational Mapping (ORM) para interagir com a mesma, permitindo por exemplo abstrair o sistema de base de dados e até alterar este sistema de MySQL para PostreSQL facilmente no futuro se assim fosse pretendido [35]. Para além disto, necessitamos de uma *framework* que permita desenvolver uma REST API. Existem várias opções disponíveis, tais como o desenvolvimento em Node.js com a utilização de Express para o desenvolvimento da API e a utilização do ORM Sequelize, ou o desenvolvimento em Java com a utilização de Spring. Ainda assim, face ao desenvolvimento de outros componentes em Python, uma fácil ligação a tecnologias como o Kafka ou o Redis, e uma experiência anterior na utilização destas tecnologias, a decisão passou por utilizar a *framework* Django[3] para implementar este componente. Relativamente à questão do ORM falado, o Django permite que depois de criados os modelos de dados pretendidos para a aplicação, esta *framework* automaticamente fornece uma API de abstração da base de dados que permite criar, obter, atualizar e eliminar objetos [36]. A base de dados utilizada é do tipo MySQL.

A representação das classes apresentadas na Figura 37 é feita através do código apresentado na Figura 47. Sendo assim são criados dois modelos, `Station` e `Measure`, sendo que a relação entre as estações e as medições é feita no modelo `Measure`, ao referir a *foreign key* `station` e definindo que em caso de remoção de uma estação as suas medições devem então também ser removidas (`cascade`). Para além disto, todos os campos de medições em `Measure` podem tomar um valor nulo, visto que nem todas as estações são capazes de obter todos os tipos de medições definidas. É ainda definido que a ordenação do modelo `Measure` deve ser feita pela data das medições (pode ser visto em `class Meta`), sendo que a ordenação é feita de forma decrescente, das medições mais recentes para as mais antigas.

3 https://www.djangoproject.com/

```python
class Station(models.Model):
    id = models.CharField(max_length=128, primary_key=True)
    source = models.CharField(max_length=128)
    city = models.CharField(max_length=128)
    latitude = models.FloatField()
    longitude = models.FloatField()
    timezone = models.CharField(max_length=128)
```

```python
class Measure(models.Model):
    datetime = models.DateTimeField()
    temperature = models.FloatField(null=True)
    pressure = models.FloatField(null=True)
    humidity = models.FloatField(null=True)
    wind_speed = models.FloatField(null=True)
    wind_angle = models.FloatField(null=True)
    gust_speed = models.FloatField(null=True)
    gust_angle = models.FloatField(null=True)
    clouds = models.FloatField(null=True)
    rain = models.FloatField(null=True)
    station = models.ForeignKey(
        'Station',
        on_delete=models.CASCADE
    )
    class Meta:
        ordering = ('-datetime',)
```

Figura 47: Representação das classes da base de dados do Weather Server na *framework* Django.

Quanto ao sistema de *caching* a utilizar, a decisão recaiu pela utilização do Redis, devido a uma fácil integração com a linguagem Python e assim com a *framework* Django, para além de experiências positivas com esta ferramenta anteriormente, assim como ser possivelmente o sistema de *caching in-memory* mais famoso e utilizado.

Passando a aspetos mais concretos da implementação, uma das tarefas deste serviço passa então por enviar um pedido aos *workers* ao minuto 30 de cada hora. Assim, para realizar esta tarefa é utilizada uma *task* Celery[4] e feita a integração desta ferramenta com o Django, utilizando o Redis como *broker* do Celery. Desta forma, e para além de outras configurações necessárias e que não é necessário referir, é definida uma *task* para pedir as medições mais recentes como apresentado na Listagem 6.11.

```
CELERY_BEAT_SCHEDULE = {
    'request_last_measures': {
        'task': 'weather.requester.request_last_measures',
        'schedule': crontab(minute=30),
    },
}
```

Listagem 6.11: Task Celery que permite requisitar as informações das estações ao minuto 30 de cada hora.

Assim, através do `crontab` utilizado é definido que ao minuto 30 de cada hora é utilizado o método `request_last_measures()`. Este método inicia um produtor de mensagens Kafka e inicia uma conexão ao Redis usando a biblioteca `redis`[5], como apresentado na Listagem 6.12.

```
producer = KafkaProducer(
    bootstrap_servers=settings.KAFKA_SERVER,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)
redis = Redis(
    host=settings.REDIS_HOST,
    port=settings.REDIS_PORT,
```

4 https://docs.celeryproject.org/en/stable/django/first-steps-with-django.html
5 https://pypi.org/project/redis/

```
    db=0
)
```

Listagem 6.12: Definição do produtor de mensagens para requisitar informações sobre as estações e definição de uma conexão ao Redis.

Enquanto que a ligação ao Kafka já foi explicada, a ligação ao Redis passa por aproveitar a necessidade de utilizar este sistema e assim ter uma lista em *cache* com os *correlation identifiers* das mensagens enviadas, de forma a que este componente consiga assim ter um fácil acesso aos identificadores de mensagens que enviou e, ao receber as respostas, consiga averiguar que de facto as mensagens são em resposta aos pedidos que o mesmo enviou e que são válidos. O *array* guardado no Redis para este efeito tem a chave `correlation_ids` e guarda os últimos 10 *correlation identifiers* enviados, ou seja, os identificadores enviados aproximadamente nas últimas 10 horas.

Quanto à mensagem enviada, esta segue o formato definido na Listagem 5.1 do Capítulo 5, sendo definido o número 10 no ficheiro de configuração para o número de estações que se pretendem receber por cada resposta. Nota ainda para o envio no cabeçalho da mensagem de informações sobre o *correlation identifier* e o tópico para o qual as respostas devem ser enviadas, como pode ser visto nas linhas 11 e 12 da Listagem 6.13.

```
1  correlationId = generate_correlationId()
2  producer.send(
3      settings.KAFKA_PRODUCER_TOPIC,
4      value = {
5          'method': 'get_last_measures',
6          'arguments': {
7              'stations_per_message': settings.STATIONS_PER_MESSAGE
8          }
9      },
10     headers = [
11         ('correlationId', str.encode(correlationId)),
12         ('responseTo', str.encode(settings.KAFKA_CONSUMER_TOPIC))
13     ]
14 )
```

Listagem 6.13: Código utilizado para enviar o pedido periódico do Weather Server para os workers.

Já para consumir as mensagens recebidas sobre as estações, a estratégia passa por ter essa tarefa a correr em *background*. Desta forma, é criada uma classe `ConsumerThread` que estende a classe `Thread`, que tem o intuito de processar as mensagens e atualizar a base de dados. Assim, na construção da classe é inicialmente criado um consumidor de mensagens Kafka e criada também uma ligação ao Redis, como demonstrado já em excertos anteriores. Ao processar cada uma das mensagens, é primeiro percebido se a mesma é válida, verificando se o formato da mensagem é válido (ou seja, JSON, com o auxílio do `value_deserializer` definido), se contém um *correlation identifier*, e se esse mesmo identificador é válido (com o auxílio do *array* em memória no Redis). Estas verificações são realizadas através do código apresentado na Listagem 6.14.

```
for message in self.consumer:
    if not isinstance(message.value, str):
        if message.headers and message.headers[0][1]:
            correlationId = message.headers[0][1].decode()
            if correlationId in literal_eval(self.redis.get(settings.
                CORRELATION_IDS_KEY).decode()):
                ...
```

Listagem 6.14: Excerto de código utilizado para verificar e validar uma mensagem.

Quanto ao processamento da mensagem propriamente dito, para cada estação percebe-se se a mesma já existe na base de dados, caso contrário é criado um objeto `Station` (Listagem 6.15) para poder depois ser guardado, verificando que se tem todos os dados necessários para criar o mesmo.

```
station = Station(
    id = data['station_id'],
    source = data['source'],
    city = data['location']['city'],
    latitude = data['location']['latitude'],
    longitude = data['location']['longitude'],
    timezone = data['location']['timezone']
)
```

Listagem 6.15: Criação de um objeto Station.

Se o objeto não existir na base de dados, é então criado um objeto `Measure` (Listagem 6.16) com a *foreign key* a apontar para esta mesma nova estação e guardados ambos os objetos na base de dados. No caso de já existir, um cuidado que é tido em conta é verificar se a data desta medição é mais recente que a última medição que temos da estação. Isto para além de ser uma medida correta de verificação, tem também o objetivo de tornar o recetor das mensagens mais preparado para o caso de receber mensagens repetidas, podendo até ser relacionado de alguma forma com o padrão *Idempotent Receiver* falado na Secção 2.3.4. Sendo assim, se esta nova medição for mais recente é então guardado este objeto `Measure` na base de dados. Para além disto, outro cuidado é verificar se o nome do local onde a estação se encontra foi alterado, pois foi reparado que algumas estações da Netatmo podem mudar esse mesmo detalhe ao longo do tempo, existindo assim a possibilidade de fazer *update* dos objetos `Station`.

```
measure = Measure(
    datetime = datetime.fromtimestamp(float(data['last_update'])),
    station = station
)
for type in settings.MEASURES_TYPES_SUPPORTED:
    setattr(measure, type, data['measures'][type] if type in data['measures']
        else None)
```

Listagem 6.16: Criação de um objeto Measure.

Tal como foi explicado anteriormente, o tópico `response-topic` foi criado com duas partições. Assim, para paralelizar o processamento das mensagens recebidas são criadas em tempo de execução duas instâncias desta classe `ConsumerThread`, ficando assim cada uma das *threads* responsável por ler mensagens recebidas e esperando assim ter um processamento mais rápido de todas as mensagens recebidas por este componente.

Passando para a implementação dos *endpoints* que o componente tem que expor, podemos apontar detalhes da implementação de alguns deles.

get_area_data

Este *endpoint* tem que devolver informações sobre as estações que se encontram numa determinada área. Desta forma, uma primeira consulta passa por obter as estações que se enquadram na pesquisa feita através da interação com a base de dados. Assim, para obter a lista de estações que se encontram dentro de uma área requisitada é utilizado o código apresentado na Listagem 6.17, utilizando a camada de abstração fornecida pelo Django.

```
stations = Station.objects.filter(Q(latitude__gte=lat_sw) & Q(latitude__lte=
    lat_ne) & Q(longitude__gte=lon_sw) & Q(longitude__lte=lon_ne))
```

Listagem 6.17: Código utilizado para obter as estações inseridas numa determinada área.

Sendo assim, são depois obtidas as estações mais recentes de cada uma destas estações. Quanto à resposta devolvida, a mesma passa por uma lista de objetos em que cada objeto é relacionado com uma estação e em que o formato de cada um destes segue o formato apresentado na Figura 48.


```
{
    station_id: "1234567",
    source: "openweather",
  - location: {
        longitude: -8.12345,
        latitude: 42.12345,
        timezone: "Europe/Lisbon",
        city: "Vizela"
    },
  - measures: {
        temperature: 14.54,
        pressure: 1022,
        humidity: 87,
        wind_speed: 2.25,
        wind_angle: 144,
        gust_speed: null,
        gust_angle: null,
        clouds: 100,
        rain: null
    },
    last_update: "05/10/2020 22:11:40"
},
```


Figura 48: Exemplo de resposta do *endpoint* /get_station_data.

get_historical_measures

Um dos passos iniciais deste *endpoint* passa por obter a data inicial e a final para a procura das medições que se pretende realizar para uma determinada estação. Se o utilizador já passar esses dois parâmetros o problema está resolvido, caso contrário, ao utilizar um dos intervalos definidos (`1day` por exemplo), converte-se primeiro esse intervalo em duas datas.

A partir deste ponto, são feitos então processamentos para apresentar estas informações da melhor forma e apenas da medição pedida, e até tendo em conta a escala entre duas medições definidas, disponibilizando até dois *parsers* diferentes neste *endpoint* que permitem dois formatos de apresentação: um em que cada medição apresenta a data da medição e outra que devolve as medições num *array*. Os dois formatos podem ser consultados na Figura 49 e é importante notar que o formato JSON apresenta uma lista de objetos devido a poderem existir "buracos" no histórico de medições de uma estação que não permitem fazer uma sequência total das mesmas.

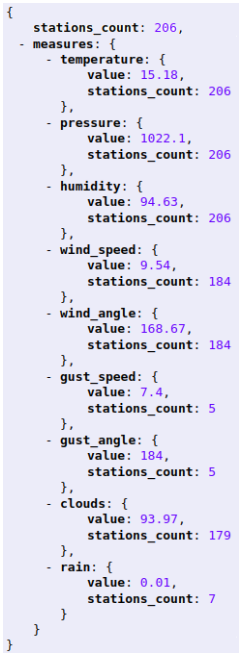

```
[
    {
        values: [
            {
                value: 13.38,
                time: "05/10/2020 11:14:38"
            },
            {
                value: 14.28,
                time: "05/10/2020 12:14:30"
            },
            {
                value: 14.18,
                time: "05/10/2020 13:14:30"
            },
            {
                value: 15.09,
                time: "05/10/2020 14:14:43"
            }
        ],
        start_time: "05/10/2020 11:14:38",
        step_seconds: 3600
    }
]
```

```
[
    {
        values: [
            13.38,
            14.28,
            14.18,
            15.09
        ],
        start_time: "05/10/2020 11:14:38",
        step_seconds: 3600
    }
]
```


Figura 49: Exemplos de respostas do *endpoint* /get_historical_measures.

get_area_average_measures

Este *endpoint* segue a mesma lógica do método `/get_area_data` na obtenção de estações que se encontram dentro da área definida e depois na seleção das medições mais recentes dessas mesmas estações. A partir desse ponto, o algoritmo trata de fazer as médias de cada medição tendo em conta o número de estações que devolvem informações sobre cada uma das medições que o sistema suporta. Um exemplo de resposta deste *endpoint* é apresentado na Figura 50.

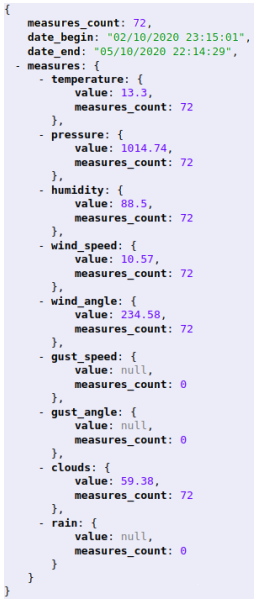


```
{
   stations_count: 206,
 - measures: {
    - temperature: {
         value: 15.18,
         stations_count: 206
      },
    - pressure: {
         value: 1022.1,
         stations_count: 206
      },
    - humidity: {
         value: 94.63,
         stations_count: 206
      },
    - wind_speed: {
         value: 9.54,
         stations_count: 184
      },
    - wind_angle: {
         value: 168.67,
         stations_count: 184
      },
    - gust_speed: {
         value: 7.4,
         stations_count: 5
      },
    - gust_angle: {
         value: 184,
         stations_count: 5
      },
    - clouds: {
         value: 93.97,
         stations_count: 179
      },
    - rain: {
         value: 0.01,
         stations_count: 7
      }
   }
}
```


Figura 50: Exemplo de resposta do *endpoint* /get_area_average_measures.

get_historical_average_measures

Já o *endpoint* `/get_historical_average_measures` segue a mesma lógica do método `/get_historical_measures` na obtenção de todas as medições de uma estação entre duas datas. A partir desse ponto, o algoritmo é responsável por fazer as médias de cada medição tendo em conta o número e as medições realizadas. Um exemplo de resposta deste *endpoint* pode ser visto na Figura 51.



```
{
   measures_count: 72,
   date_begin: "02/10/2020 23:15:01",
   date_end: "05/10/2020 22:14:29",
 - measures: {
    - temperature: {
         value: 13.3,
         measures_count: 72
      },
    - pressure: {
         value: 1014.74,
         measures_count: 72
      },
    - humidity: {
         value: 88.5,
         measures_count: 72
      },
    - wind_speed: {
         value: 10.57,
         measures_count: 72
      },
    - wind_angle: {
         value: 234.58,
         measures_count: 72
      },
    - gust_speed: {
         value: null,
         measures_count: 0
      },
    - gust_angle: {
         value: null,
         measures_count: 0
      },
    - clouds: {
         value: 59.38,
         measures_count: 72
      },
    - rain: {
         value: null,
         measures_count: 0
      }
   }
}
```


Figura 51: Exemplo de resposta do *endpoint* /get_historical_average_measures.

Passando para a configuração dos mecanismos de *caching*, é importante primeiro explicar as configurações definidas para o Redis. Assim, uma delas que é essencial referir é a capacidade máxima de memória atribuída a este sistema de *cache in-memory*. Sendo assim são atribuídos 4 GB a esta configuração, definindo para a propriedade `maxmemory` o valor `4gb`. Outra configuração que segue o que já foi discutido no Capítulo anterior passa pela utilização de uma política de despejo, sendo pretendido utilizar o algoritmo Least Frequently Used (LFU). Assim, `maxmemory-policy` é o parâmetro que configura este detalhe. Desta forma, de entre as opções disponibilizadas pelo Redis, foi disponibilizada a partir da sua versão 4.0 a opção `volatile-lfu` que remove as chaves utilizadas com menos frequência que são configuradas com expiração TTL [37]. Assim, tendo em conta que todas as chaves que se pretendem utilizar no Redis para efeito de *caching* têm um tempo de expiração definido esta parece ser a opção mais acertada e a escolhida para a configuração desta ferramenta.

Sendo assim, têm então que ser implementados os algoritmos apresentados na Secção 2.4 do Capítulo anterior para cada um dos *endpoints* que este componente expõe. Desta forma, a ideia passa por no início da implementação de cada um dos métodos ser verificado se o resultado pretendido já se encontra em *cache*. Caso contrário, no fim desta implementação trata-se então de guardar as respostas em *cache* antes de devolver o resultado ao utilizador. A título de exemplo, o método `store_station_data_in_cache()` apresentado na Listagem 6.18 permite guardar em *cache* uma resposta do método `/get_station_data`.

```
1 def store_station_data_in_cache(station_id, data):
2     key = 'station_data_' + station_id
3     redis.set(key, json.dumps(data).encode())
4     expire_datetime = calculate_expiry_datetime()
5     redis.expireat(key, expire_datetime)
6     print('[get_station_data] response stored in cache, expire_date: ' +
          expire_datetime.strftime(settings.DATE_OUTPUT_FORMAT))
```

Listagem 6.18: Método utilizado para guardar respostas do *endpoint* /get_station_data em *cache*.

Desta forma, é guardado o par chave-valor no Redis fazendo *dump* do dicionário do formato Python para um formato JSON seguido de um *encode* (linha 3). O método `calculate_expiry_datetime()` é utilizado para calcular o *datetime* em que a chave deve expirar (linha 4), sendo depois utilizado este *datetime* na chamada `redis.expireat()` que define então no Redis o momento em que a chave deve deixar de ser válida (linha 5). Importante realçar que se optou por não expirar as chaves exatamente no próximo minuto *XX*h30min mas sim um pouco depois devido à operação de atualização que se inicia nesse mesmo momento.

Sobre uns dos algoritmos mais complexos dos mecanismos de *caching* definidos, para o *endpoint* `/get_area_data` e `/get_area_average_measures`, são primeiro pesquisadas as áreas guardadas em *cache* para ver se alguma destas satisfaz as necessidades de resposta. Desta forma, faz-se uma iteração das chaves definidas para este método utilizando o método `scan_iter()` fornecido pela biblioteca `redis`, utilizando um padrão de pesquisa (que faça *match* com o tipo de chaves que se procuram), como apresentado na Listagem 6.19.

```
for area_key in redis.scan_iter(match=key_string + '*'):
    ...
```

Listagem 6.19: Utilização do método scan_iter() da biblioteca redis.

Neste caso, `key_string` toma o valor de "`area_data_`" ou "`area_average_`" dependendo do *endpoint* em causa, visto que ambos partilham o método em que este código é utilizado. Depois de ter a área em *cache* e a área pedida representadas por um objeto `Polygon` da biblioteca `shapely`, é então necessário perceber se a intersecção entre as duas áreas é de pelo menos 80%, de forma a saber se esta resposta armazenada pode ser utilizada. Um excerto do algoritmo utilizado para conseguir isto é apresentado na Listagem 6.20.

```
base_area = max(cache_polygon.area, request_polygon.area)
intersection_area = (cache_polygon.intersection(request_polygon)).area
if intersection_area:
    if min(base_area, intersection_area) / max(base_area, intersection_area) >=
        settings.MIN_INTERSECTION_AREA:
        data = redis.get(area_key)
        print('[' + endpoint_name + '] response is in cache')
        return json.loads(data.decode())
```

Listagem 6.20: Excerto de código utilizado para verificar se uma área em *cache* pode ser utilizada como resposta.

6.5 COMPONENTE - CLIENT SERVER

O outro servidor que se necessita de implementar é então o componente `Client Server`. Analisando as principais necessidades a que este serviço necessita de dar resposta percebemos que tem aspetos em comum com o componente anterior, o `Weather Server`:

- Receber assincronamente as respostas enviadas pelos *workers*, processar as mesmas, e com isso ser capaz de atualizar a sua base de dados e criar notificações para os utilizadores.
- Fornecer uma REST API com os 10 métodos definidos na Secção 5.7 do Capítulo anterior.

Podemos logo identificar que para implementar este componente precisamos de utilizar ferramentas que permitam expor uma API e que permita ter uma ligação a uma base de dados que permita até consultar os dados que dão resposta a esses mesmos *endpoints*. Inicialmente, o desenvolvimento deste componente iria ser feito com recurso à *framework* Flask[6] para desenvolver e expor os métodos da API, e utilizar também a biblioteca Flask-SQLAlchemy que funciona como uma extensão ao Flask e que adiciona suporte para SQLAlchemy à aplicação [38], permitindo interagir com uma base de dados MySQL neste caso (funcionando na prática como

6 https://flask.palletsprojects.com/en/1.1.x/

um ORM). Isto poderia funcionar mas, no entanto, torna-se necessário ter uma *thread* em *background* que precisa de estar a receber mensagens do Kafka e assim interagir e modificar a base de dados. Assim, torna-se necessário existirem duas *threads* com acesso à base de dados: a *thread* principal que expõe e dá respostas sobre os *endpoints* da API e esta *thread* em *background*. Ora, o problema é que a biblioteca SQLAlchemy não é *thread-safe* ou necessita de mais detalhes e outros cuidados para ser utilizada neste contexto, pelo que foi decidido que a implementação deste componente seria então feita com recurso à *framework* Django, tal como o `Weather Server`.

A representação das classes apresentadas na Figura 41 é feita através do código apresentado na Figura 52.

```
class User(models.Model):
    email = models.CharField(max_length=128, primary_key=True)
    password = models.CharField(max_length=128)
    username = models.CharField(max_length=64)
    favourite_stations = models.ManyToManyField(
        'Station',
        related_name='users'
    )

class Notification(models.Model):
    message = models.CharField(max_length=256)
    datetime = models.DateTimeField()
    user = models.ForeignKey(
        'User',
        on_delete=models.CASCADE
    )
    alert = models.ForeignKey(
        'Alert',
        on_delete=models.CASCADE
    )
    class Meta:
        ordering = ('-datetime',)

class Station(models.Model):
    id = models.CharField(max_length=128, primary_key=True)
    city = models.CharField(max_length=128)
```

```
class Alert(models.Model):
    category = models.CharField(max_length=64)
    type = models.CharField(max_length=64)
    value = models.FloatField(null=True)
    user = models.ForeignKey(
        'User',
        on_delete=models.CASCADE
    )
    station = models.ForeignKey(
        'Station',
        on_delete=models.CASCADE
    )

class Measure(models.Model):
    ''' Specific measure from a station
    '''
    datetime = models.DateTimeField()
    temperature = models.FloatField(null=True)
    humidity = models.FloatField(null=True)
    station = models.ForeignKey(
        'Station',
        on_delete=models.CASCADE
    )
    class Meta:
        ordering = ('-datetime',)
```

Figura 52: Representação das classes da base de dados do Client Server na *framework* Django.

Importa realçar neste código alguns detalhes:

- A relação estabelecida entre os modelos `User` e `Station`, existindo uma relação de muitos para muitos, que pode ser vista em `User` através da utilização de `models.ManyToManyField`.
- Um alerta contem uma *foreign key* para o utilizador a que está associado e outra para a estação a que diz respeito.
- Uma notificação está associada da mesma forma ao utilizador e ao alerta a que diz respeito.
- Cada medição, tal como seguido no componente `Weather Server`, está associada à sua estação meteorológica através de uma *foreign key*. Para além disto, mantém-se a ordenação dos objetos deste modelo pelo seu `datetime`, do mais recente para o mais antigo.

Quanto à implementação dos *endpoints* devem ser analisadas várias decisões tomadas.

Começando pelo início do processo de utilização da aplicação por parte de um utilizador registado, a ação de efetuar um registo, um detalhe importante é a decisão de calcular o *hash* das *passwords* e ser este resultado a ser guardado na base de dados. Esta decisão vai ao encontro de poder tornar a aplicação mais segura e, no

caso de existir um *leak* de informações da base de dados, não tornar o acesso às *passwords* dos utilizadores algo direto.

Relacionado com o *endpoint* `/login`, este método do tipo POST exposto pela REST API devolve um objeto JSON que indica o *e-mail* do utilizador, o *username* do mesmo e um *token*. O aspeto mais importante aqui é precisamente este *token*, em que a ideia é que seja passado depois em todos os *endpoints* (claro, excetuando este e o `/register`) que este componente expõe e que permita assim identificar o utilizador e perceber também a autoridade que o mesmo tem para efetuar certas operações e consultas. A decisão é que este *token* seja passado através dos *headers* dos pedidos realizados no campo "`Authentication`". Esta decisão para além de parecer a mais acertada deve-se também ao facto de nos pedidos do tipo GET poderem existir problemas ao utilizar o campo *body* do *request* para enviar o *token*, o que contribuiu ainda mais para tomar esta opção. Assim, um pedido a este componente utilizando a linha de comandos pode ser realizado como demonstrado na Listagem 6.21.

```
curl -H 'Authentication: <jwt_token>' http://localhost:8000/client/
    get_favourite_stations
```

Listagem 6.21: Exemplo de pedido ao servidor Client Server.

Para concretizar este *token* é então utilizada uma abordagem de criação de um JSON Web Token (JWT) utilizando a biblioteca `PyJWT`[7]. Assim, neste conceito JWT, a ideia passa por conseguir encriptar um objeto JSON utilizando para isso um algoritmo `HS256` e um segredo definido, e ao utilizar esse *token* para os utilizadores fazerem os seus pedidos conseguir descodificar esse mesmo *token* novamente para perceber quem é o utilizador em questão.

Passando para o consumo de mensagens do Kafka, é utilizada como no componente `Weather Server` uma classe `ThreadConsumer`, paralelizando este processo de consumo através de duas instâncias desta classe tendo em conta as duas partições criadas para o tópico `response-topic`. Durante o consumo das mensagens é também validado o formato JSON das respostas recebidas e se a mesma contem um *correlation identifier*, apesar de neste componente não ser verificada a validade deste identificador, não tendo acesso ao Redis com que o `Weather Server` interage. Uma solução possível seria criar um tópico Kafka que permitisse a troca de mensagens entre o `Weather Server` e o `Client Server` e, assim, sempre que o `Weather Server` fizesse um pedido aos *workers* enviaria o *correlation identifier* desse pedido ao `Client Server`, de forma a que este possa verificar a validade das respostas recebidas. Ainda assim, optou-se por não tomar este caminho, tendo em conta uma dificuldade acrescida e não ser considerado um detalhe assim tão importante de momento.

Tal como planeado e falado no Capítulo 5, este componente `Client Server` apenas mantém na base de dados algumas das últimas medições de cada estação. Sendo assim, uma das decisões tomadas passou por guardar na base de dados apenas as medições das estações que se encontram aproximadamente nas últimas 10 horas.

7 https://pyjwt.readthedocs.io/en/latest/

Durante o processamento das informações de uma estação são também criadas estações, medições, comparadas as datas com a última medição na base de dados, etc, como relatado que é realizado no `Weather Server`. Com isto, se a estação já existir previamente na base de dados e a última medição armazenada for anterior à nova medição, faz então sentido tomar estas novas informações como válidas e é então iniciado o processo de criação de notificações face aos alertas que existem a que a estação em causa está associada.

Tendo em conta os 4 tipos de alertas explicados no Capítulo 4, podemos então perceber um pouco da implementação de cada um destes.

Alerta por valor

Um alerta deste tipo é da categoria "`value`". Para verificar se é necessário criar uma notificação tendo em conta a medição mais recente da estação e o próprio alerta, apenas se torna necessário verificar se o valor da nova medição é igual ou superior ao valor definido no alerta. Um exemplo de notificação que este alerta torna possível de receber é apresentado na Figura 53, sendo este objeto JSON um exemplo de um objeto que o *endpoint* `/get_notifications` devolve.


```
{
    alert_category: "value",
    id: 1000,
    station_id: "1234567",
    message: "A estação 1234567 atingiu um valor de 16.26 ºC na medição efetuada em 01/10/2020 12:14:40",
    datetime: "01/10/2020 12:30:10"
},
```


Figura 53: Exemplo de notificação resultante de um alerta por valor.

Alerta por percentagem

Quanto ao alerta por percentagem, da categoria "`percentage`", a implementação do mesmo necessita de ter acesso à última medição desta estação que está guardada na base de dados e da nova medição, de forma a calcular o valor do crescimento percentual. Depois, torna-se apenas necessário perceber se o valor desta percentagem é igual ou superior ao valor definido no alerta. Um exemplo de notificação que resulta deste alerta é apresentado na Figura 54.


```
{
    alert_category: "percentage",
    id: 1001,
    station_id: "1234567",
    message: "A estação 1234567 teve um crescimento percentual de 33% (16.26 ºC) na medição efetuada em 01/10/2020 12:14:40 em relação à medição anterior (12.31 ºC)",
    datetime: "01/10/2020 12:30:10"
},
```


Figura 54: Exemplo de notificação resultante de um alerta por percentagem.

Alerta de medições consecutivas

Passando para o alerta de medições consecutivas, este pertence à categoria "`consecutive`", sendo que a implementação deste verifica se o valor da nova medição é igual ou superior ao valor definido no alerta. A confirmar-se esta situação, são então obtidas todas as medições da estação que se encontram na base

de dados das aproximadamente últimas 10 horas. Assim, verifica-se quantas medições consecutivas dessas medições ordenadas da mais recente para a mais antiga são também iguais ou superiores ao valor pretendido, começando com um contador com o valor 1 devido à nova medição. Se o contador tomar um valor superior a 1 é então criada uma notificação. Um exemplo de notificação que resulta deste alerta é apresentado na Figura 55.


```
{
    alert_category: "consecutive",
    id: 1002,
    station_id: "1234567",
    message: "A estação 1234567 tem as suas últimas 5 medições com um valor igual ou superior a 14.22 ºC",
    datetime: "01/10/2020 12:30:10"
},
```


Figura 55: Exemplo de notificação resultante de um alerta de medições consecutivas.

Alerta de médias temporais

Sobre o alerta de médias temporais, o mesmo pertence à categoria "`average`", sendo que a implementação deste tem que verificar se a média de medições de um bloco das aproximadamente últimas 5 horas é superior ao bloco de 5 horas anteriores, sendo este valor de 5 horas definido tendo em conta a intenção de termos na base de dados aproximadamente as medições das últimas 10 horas de cada estação. Por fim, é calculada a média de medições de cada um dos blocos e, em caso das medições mais recentes terem uma média superior às médias anteriores é então criada a notificação. Um exemplo de notificação que resulta deste alerta é apresentado na Figura 56.


```
{
    alert_category: "average",
    id: 1003,
    station_id: "1234567",
    message: "A estação 1234567 registou nas últimas 5 horas uma média (18.21 ºC) superior às 5 horas anteriores (14.56 ºC)",
    datetime: "01/10/2020 12:30:10"
}
```


Figura 56: Exemplo de notificação resultante de um alerta de médias temporais.

Estes alertas e notificações servem assim como um exemplo daquilo que pode ser conseguido com as informações recebidas, sendo evidente que com estes mesmos dados podem ser criados no futuro outros tipos de alertas que acrescentem valor à aplicação.

6.6 COMPONENTE - API GATEWAY

Por fim, falta então descrever as decisões tecnológicas e a implementação do componente `API Gateway`. Assim, a principal função pretendida para este componente passa por rotear os pedidos recebidos pelos dois servidores que expõe uma REST API, o `Weather Server` e o `Client Server`. Este componente permitirá assim fazer uma agregação dos *endpoints* pretendidos e servirá como o único ponto de entrada no sistema, conseguindo encapsular a estrutura interna da arquitetura.

A partir deste ponto, vamos supor que os dois servidores `Weather Server` e `Client Server` são executados nos endereços `http://localhost:8001/weather` e `http://localhost:8002/client`, sendo que para aceder ao *endpoint* `/get_station_data` do `Weather Server` se deve aceder por exemplo a `http://localhost:8001/weather/get_station_data?station_id=1234567` e para obter as notificações de um utilizador através do `Client Server` se deve aceder a `http://localhost:8002/client/get_notifications`.

A implementação deste componente foi realizada com recurso a Express Gateway[8], que utiliza tecnologias como Node.js e Express.

Com isto, as principais funcionalidades do *gateway* são definidas e descritas no ficheiro `gateway.config.yml` [39]. O conteúdo deste ficheiro é apresentado na Listagem 6.22.

```
http:
    port: 8000
admin:
    port: 9876
    host: localhost
apiEndpoints:
    weather:
        host: localhost
        paths: [
            '/get_station_data',
            ...
        ]
    client:
        host: localhost
        paths: [
            '/register',
            ...
        ]
serviceEndpoints:
    weatherServer:
        url: 'http://localhost:8001'
    clientServer:
        url: 'http://localhost:8002'
policies:
    - basic-auth
    - cors
    - expression
    - key-auth
    - log
    - proxy
    - rate-limit
    - rewrite
pipelines:
```

8 https://www.express-gateway.io/

```
34      weatherPipeline:
35          apiEndpoints:
36              - weather
37          policies:
38              - cors:
39              - rewrite:
40                  - condition:
41                      name: regexpmatch
42                      match: ^/(.*)$
43                    action:
44                      rewrite: /weather/$1
45              - log:
46                  - action:
47                      message: ${req.ip} ${req.method} ${req.originalUrl} ${res.
                            statusCode}
48              - proxy:
49                  - action:
50                      serviceEndpoint: weatherServer
51                      changeOrigin: true
52      clientPipeline:
53          ...
```

Listagem 6.22: Configuração do ficheiro gateway.config.yml do API Gateway.

Assim, com a Listagem 6.22 é possível explicar as configurações mais relevantes realizadas neste componente.

Começando pelo início, a secção `http` (linhas 1-2) serve para configurar a porta onde o serviço vai ficar à espera de receber os pedidos dos utilizadores.

Quanto à secção `admin` (linhas 3-5), esta representa um *endpoint* interno para a REST API de administrador [39], apesar de isto não ter sido algo explorado.

Passando para a secção `apiEndpoints` (linhas 6-18), esta permite definir os *endpoints* que queremos expor. Assim, são criadas duas secções para cada um dos serviços que temos na arquitetura e que são depois referenciados mais à frente na secção `pipelines`. A secção `weather` (linhas 7-12) é relativa ao `Weather Server` enquanto que `client` (linhas 13-18) é relativa ao `Client Server`. Assim, em ambos é definido que o `host` toma o valor de `localhost`, servindo o parâmetro `host` para definir o *hostname* onde vão ser aceites os pedidos. Neste caso, se este componente for colocado numa máquina remota (supomos que com o endereço 10.10.10.10), se pretendermos receber pedidos de fora este parâmetro não deve ser configurado com `localhost`, `10.10.10.10` poderia ser uma opção, outra opção seria por exemplo `'*'` que permite pedidos de qualquer domínio. Quanto a `paths`, são aqui definidos os nomes dos *endpoints* que queremos expor, ou seja, os 6 do `Weather Server` e os 10 do `Client Server`, sendo que na listagem é colocado "..." em ambos os *arrays* para reduzir o tamanho da mesma.

O Express Gateway recebe pedidos da API nos `apiEndpoints`, processa os mesmos e depois faz *proxy* para os micro-serviços [39]. Assim, a secção `serviceEndpoints` (linhas 19-23) serve para especificar

os URLs destes serviços. Desta forma, são definidos os dois serviços e a localização dos mesmos. Importante referir que enquanto aqui é utilizado o parâmetro `url` para definir a localização de um serviço, podemos utilizar `urls` se existirem várias instâncias de um serviço a correr, como apresentado na Listagem 6.23, sendo então realizado *load balancing* entre as várias opções disponíveis para cada um.

```
serviceEndpoints:
    weatherServer:
        urls:
            - 'http://localhost:8001'
            - 'http://localhost:8002'
            - 'http://localhost:8003'
    clientServer:
        urls:
            - 'http://localhost:8004'
            - 'http://localhost:8005'
```

Listagem 6.23: Utilização de *load balancing* na configuração do ficheiro gateway.config.yml do API Gateway.

Depois, a secção `policies` (linhas 24-32) define uma *white list* de políticas ativadas, sendo que as políticas que se pretendem utilizar no Express Gateway devem ser declaradas aqui [39].

Por último, resta então explicar a secção `pipelines` (linhas 33-53), que especifica as principais operações do Express Gateway, amarrando todas as entidades declaradas nas secções anteriores [39]. São criadas duas *pipelines*, `weatherPipeline` e `clientPipeline`, sendo que não foi colocado o conteúdo desta última na listagem por ser muito idêntico ao `weatherPipeline`. Sendo assim, e concentrando-nos no `weatherPipeline`, este utiliza os *endpoints* definidos anteriormente de `weather` na secção `apiEndpoints` (linhas 35-36). A partir daqui, são definidas as políticas que devem ser utilizadas:

- `cors` define que seja permitida a utilização de CORS (linha 38).
- `rewrite` é uma política utilizada para reescrever os URLs recebidos (linhas 39-44). Isto torna-se necessário pois necessitamos de converter um pedido do tipo `/get_station_data?station_id=1234567` em `/weather/get_station_data?station_id=1234567` que deve ser recebido pelo `Weather Server` devido à implementação realizada neste componente e à API que o mesmo expõe. Assim, isto é conseguido aqui com recurso a expressões regulares, como se pode perceber pelos campos `match` e `rewrite`, que se encontram em `condition` e `action` respetivamente.
- `log`, que é a política utilizada para costumizar as mensagens de *outputs* de *logging* que este componente deve realizar (linhas 45-47). Assim, consegue-se com isto que o *output* devolva informações como o endereço IP que realizou o pedido, o método utilizado (GET, POST, etc), o URL original do pedido ou o *status code*. Um exemplo de um *output* conseguido através desta configuração é então:

  ```
  2020-10-01T12:00:00.000Z [EG:log-policy] info: ::ffff:10.20.30.40 GET
  /get_station_data?station_id=1234567 200
  ```

- `proxy`, que é a política que permite encaminhar um pedido para o *endpoint* de um serviço [39] (linhas 48-51). Assim, é definido que os pedidos devem ser respondidos pelo `Weather Server` tal como pretendido.

Com esta implementação, supondo que este `API Gateway` corre na porta 8000 do *localhost* e que o `Weather Server` corre na porta 8001 da mesma máquina, torna-se possível que um pedido recebido pelo `API Gateway` na forma `http://localhost:8000/get_station_data?station_id=1234567` seja encaminhado para o `Weather Server` que recebe o pedido na forma `http://localhost:8001/weather/get_station_data?station_id=1234567`.

Com isto, podemos concluir que temos um `API Gateway` que não só dá resposta às exigências do momento como pode mais facilmente ser alterado para necessidades futuras, muito pelas ferramentas utilizadas que são específicas para os comportamentos possíveis que um API Gateway deve ter, daí a utilização do Express Gateway.

6.7 DESENVOLVIMENTO DO FRONTEND

Estando a arquitetura de *backend* implementada e pronta a ser utilizada, passou-se então para o desenvolvimento de um *frontend* que sirva como validação do trabalho realizado e que permita a consulta de todas as informações que o sistema tem para oferecer, assim como todas as ações a que um utilizador registado tem acesso a partir do *browser* de um computador ou de um *smartphone* que utilizamos. Desta forma, podemos considerar este *frontend* como um *add-on* que complementa o trabalho desenvolvido neste projeto.

Sendo assim, foram delineados requisitos funcionais e não funcionais que se pretendem alcançar para o desenvolvimento desta aplicação:

- A aplicação *frontend* deve ser capaz de utilizar todos os métodos fornecidos pelo `Weather Server` e `Client Server` que são expostos no *backend* através do `API Gateway`.
- A aplicação *frontend* deve poder ser utilizada com e sem registo efetuado pelo utilizador, devendo apenas certos conteúdos serem restritos e colocados ao dispor dos utilizadores registados.
- A aplicação *frontend* deve ter em conta a responsividade cada vez mais utilizada nas páginas *web* de forma a adaptar-se às dimensões das janelas e ecrãs dos nossos dispositivos do quotidiano.
- O código da aplicação *frontend* deve ser facilmente modificável para por exemplo adicionar novas fontes de dados meteorológicos, novos tipos de medições, novos tipos de alertas ou adicionar novas funcionalidades.

Começando pela escolha de tecnologias, o *frontend* foi desenvolvido com recurso a React que é uma biblioteca JavaScript para construir interfaces para o utilizador [40]. A escolha pelo React deve-se ao facto de ser uma das tecnologias mais utilizadas nos dias de hoje para o desenvolvimento de aplicações *web*, existir um

grande conjunto de bibliotecas e *frameworks* que podem ser utilizadas no desenvolvimento da aplicação e que fornecem componentes já feitos, existir um grande suporte *online* para a mesma, permitir a reutilização de componentes construídos e também por já ter sido uma ferramenta utilizada com sucesso num passado recente em outros projetos, diminuindo assim também o tempo que poderia ser necessário para compreender as principais caraterísticas e funcionalidades de outras opções disponíveis, como o Angular ou o Vue.js. Para além disto, foi utilizada a biblioteca Semantic UI React[9] que oferece vários componentes utilizados ao longo da aplicação como `Segment`, `Form`, `List`, `Modal`, entre muitos outros.

A principal ideia para a aplicação seria ter na página principal um mapa que permitisse ao utilizador selecionar uma determinada área à sua escolha e ver quais as estações que se encontram nessa mesma área. A página principal da aplicação *frontend* pode ser vista na Figura 57.

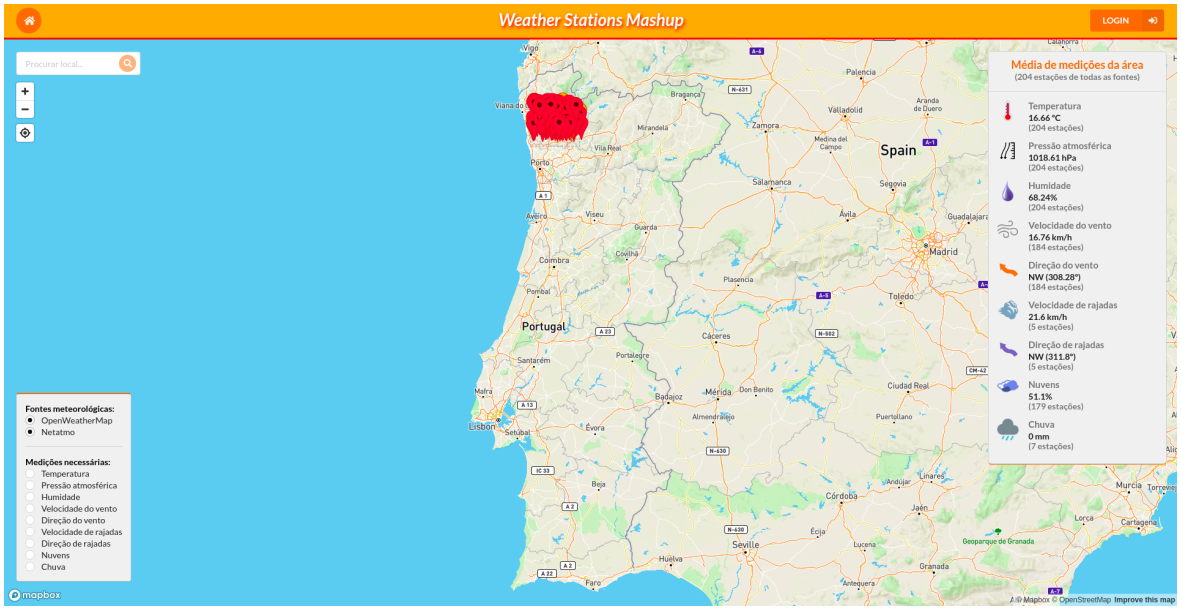


Figura 57: Página principal da aplicação *frontend*.

Para implementar o mapa em si foi utilizado o Mapbox assim como algumas extensões para implementar as funcionalidades pretendidas.

De início, são carregadas as estações de uma área definida, sendo que o utilizador pode definir uma área de acordo com a sua vontade desenhando um retângulo no mapa. Para obter as estações de uma área do mapa é então utilizado o *endpoint* `/get_area_data` da arquitetura implementada. A ideia foi ter uma cor para cada fonte de dados meteorológicos, apresentando duas cores, uma para as estações OpenWeatherMap (vermelho) e outra para as estações Netatmo (laranja). Ao selecionar uma estação, são apresentadas sobrepostas sobre o mapa as informações sobre as últimas medições da estação e é possível ainda aceder a uma página que possibilita a consulta de mais informações, como mostra a Figura 58.

9 https://react.semantic-ui.com/

Figura 58: Informações apresentadas através da seleção de uma estação meteorológica.

Esta página apresenta ainda outras funcionalidades à disposição do utilizador. No canto inferior esquerdo da Figura 57, é apresentado um componente que permite realizar uma filtragem das estações que queremos ver apresentadas no mapa. Assim, podemos selecionar as fontes de dados meteorológicos que queremos ver no mapa (por exemplo, se só quisermos ver estações da Netatmo), assim como também permite filtrar as estações pelas medições que as mesmas apresentam, por exemplo, permitindo consultar estações cujas últimas medições contêm velocidade de rajadas e chuva. Por omissão, ambas as fontes estão selecionadas e nenhuma das medições está selecionada de forma a mostrar todas as estações.

No canto superior esquerdo, o botão de localização permite saber onde se encontra o utilizador, marcar esse mesmo ponto no mapa e conseguir apresentar as estações que estão perto desse mesmo local, utilizando o *endpoint* `/get_closest_stations`. Também se permite ao utilizador procurar por um determinado local (cidade, freguesia, etc) e apresentar no mapa as estações perto desse local, fazendo uso novamente do *endpoint* `/get_closest_stations`.

Quanto ao lado direito do ecrã, o componente apresentado permite apresentar a média dos valores das últimas medições das estações presentes na área do mapa selecionada, fazendo uso do *endpoint* `/get_area_average_measures`.

Voltando à Figura 58, é possível então aceder a mais informações sobre a estação selecionada numa nova página. Um dos componentes desta página permite novamente consultar as últimas medições desta estação, como pode ser visto na Figura 59. Aqui a ideia passa por apresentar estas informações de uma forma mais adornada e dinâmica, em que são tidos em conta detalhes como rodar as setas de acordo com o ângulo nas medições de ângulo do vento e ângulo de rajadas por exemplo.

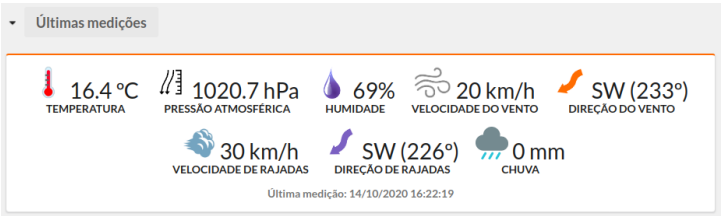


Figura 59: Consulta das últimas medições de uma estação.

Outro componente disponibilizado encontra-se apresentado na Figura 60, que permite consultar um histórico de medições da estação, apresentando um gráfico com essas mesmas informações. Por omissão este gráfico apresenta as medições da estação nas últimas 24 horas relativas à temperatura, sendo que a pesquisa do utilizador pode ser redefinida utilizando as ferramentas apresentadas. Assim, a procura pode ser feita para qualquer tipo de medição que a estação suporte, podendo a pesquisa ser feita tanto com base num intervalo de datas à escolha como por intervalos de tempo definidos, sendo também possível escolher o intervalo entre medições (que por omissão é de uma hora). Caraterísticas interessantes deste gráfico é que o mesmo permite por exemplo a consulta da medição de um ponto em específico, permite realizar *zoom* ao gráfico e mover para a esquerda e direita do mesmo, o que para um grande intervalo de tempo pode ser bastante útil. Este componente utiliza o *endpoint* `/get_historical_measures` para obter as informações que necessita de apresentar através do gráfico.

Figura 60: Consulta do histórico de medições de uma estação.

É ainda apresentado um componente que faculta a obtenção da média das medições efetuadas pela estação, utilizando o *endpoint* `/get_historical_average_measures`. Inicialmente são também apresentadas as informações relativas às últimas 24 horas, podendo isto ser definido novamente tanto através de um intervalo entre datas como com intervalos de tempo já definidos. O componente em causa é apresentado na Figura 61.

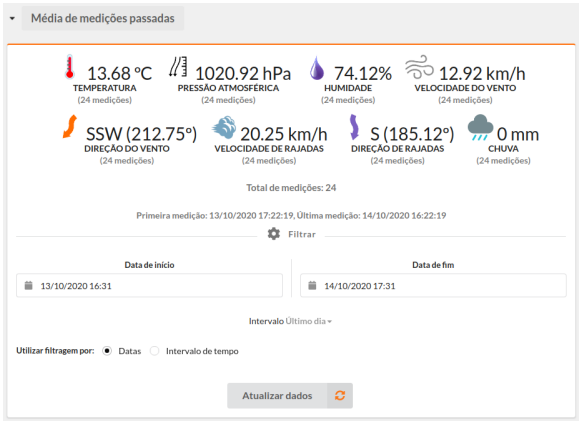


Figura 61: Consulta da média de medições passadas de uma estação.

Quanto a funcionalidades que devem ser colocadas ao dispor dos utilizadores registados na aplicação, a ideia passou por certas páginas e algumas partes de outras serem apenas acedidas por este tipo de utilizadores e serem restritas aos restantes, sendo abordadas de seguida algumas dessas funcionalidades.

Adicionar estação à lista de estações favoritas

Uma das diferenças é que é dada a possibilidade ao utilizador registado de adicionar a estação selecionada à lista das suas estações favoritas. Uma estação favorita passa a ser representada por uma estrela no mapa da página principal, como mostrado na Figura 62.

Figura 62: Apresentação das estações favoritas no mapa.

Criação de alertas

No caso da estação ser uma das favoritas do utilizador, é oferecida a possibilidade de criar um alerta sobre a mesma. O formulário que deve ser preenchido para criar um alerta é apresentado na Figura 63.

Figura 63: Criação de um alerta relativo a determinada estação.

Consulta dos alertas definidos

É também possibilitado a um utilizador registado a consulta das informações sobre os alertas definidos pelo mesmo, como apresentado na Figura 64. Através dos dois botões fornecidos do lado direito de cada alerta é permitido aceder a mais informações sobre a estação ou então apagar o alerta em causa, surgindo um aviso de confirmação da operação.

Figura 64: Apresentação dos alertas definidos pelo utilizador.

Consulta das notificações

O utilizador registado deve ainda poder aceder às suas notificações relacionadas com os alertas definidos pelo mesmo (Figura 65). Em cada notificação existe um botão que permite apagar a mesma.

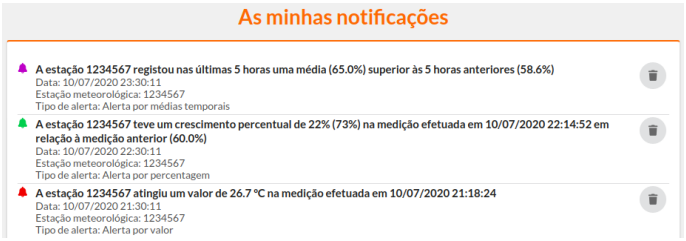


Figura 65: Apresentação das notificações do utilizador.

De forma a corresponder aos objetivos delineados para esta aplicação, o desenvolvimento das páginas da mesma teve especial atenção para a responsividade pretendida face à dimensão do ecrã ou do *browser* em que a mesma está a ser apresentada. Para tal, a ideia geral passou por diminuir o tamanho de certos componentes consoante a largura do ecrã é reduzida. Por outro lado, na página principal, face aos componentes que são apresentados tanto do lado esquerdo como direito do mapa (como pode ser visto na Figura 57), a decisão passou por tornar alguns componentes em botões a partir de uma certa largura de forma a continuar a ter uma página bonita e operacional, sendo que estes botões dão origem a um *modal* que permite mostrar esses mesmos componentes se assim for desejado. Na Figura 66, podemos ver que o componente que trata da filtragem de estações (canto inferior esquerdo) e o componente que mostra a média das últimas medições de

uma área requisitada (canto superior direito) são representadas por um botão que permite mostrar o conteúdo dos mesmos quando desejado.

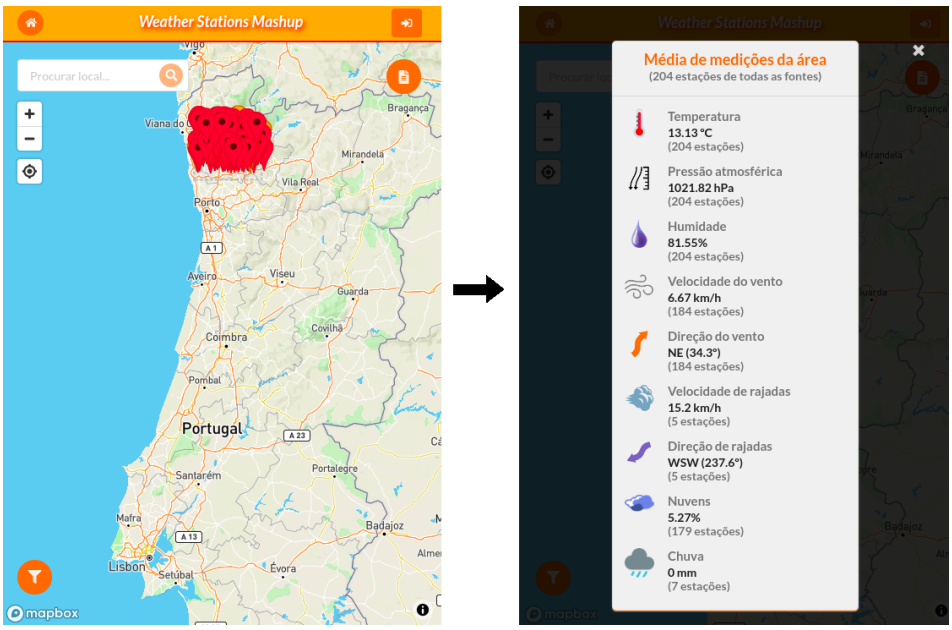


Figura 66: Responsividade da aplicação *frontend*.

Outro dos cuidados tido em conta no desenvolvimento deste *frontend* é relativo à expiração do *token* de um utilizador autenticado. Assim, se qualquer pedido relativo aos utilizadores registados na aplicação receber uma resposta de que o *token* foi expirado é automaticamente feito o *logout* do utilizador na aplicação, sendo referido ao mesmo que a sua sessão expirou. Outro cuidado passou por tentar ao máximo que uma resposta negativa recebida de qualquer *endpoint* não inviabilize a apresentação das páginas corretamente aos utilizadores, apesar de não conterem toda a informação requisitada.

Por fim, ao ser esperado que a aplicação fosse facilmente modificável, um dos cuidados passou por criar um ficheiro `settings.js` que contem variáveis definidas que são utilizadas nos vários componentes da aplicação, sendo que não será necessário modificar os componentes para fazer algumas alterações. Por exemplo, é definido neste ficheiro que existem duas fontes de dados meteorológicos, onde se define o nome da fonte e as imagens dos marcadores no mapa da página principal, através do código apresentado na Listagem 6.24.

```
export const sources_map = {
    openweather: {
        text: 'OpenWeatherMap',
        marker: require('./images/marker_img.svg'),
        favourite_marker: require('./images/star_img.svg')
    },
    netatmo: { ... }
}
```

Listagem 6.24: Definição das fontes de dados suportadas.

Outro exemplo é definir uma variável que estabelece todos os tipos de medições suportadas, sendo assim fácil adicionar um novo tipo de medição ou modificar alguma se pretendido (Listagem 6.25). Assim, para cada tipo de medição é definido o nome a apresentar da mesma, as unidades dos valores apresentados, as cores utilizadas no gráfico de medições históricas (Figura 60) para o tipo de medição, a imagem atribuída à mesma e ainda um campo que identifica se os valores da mesma são relativos a um ângulo em graus, como o ângulo do vento, em que se rodam as imagens e apresenta-se também o ângulo de acordo com a rosa dos ventos.

```
export const types_map = {
    temperature: {
        name: 'Temperatura',
        unit: ' ºC',
        color_line: '#e6e600',
        color_fill: '#ffffb3',
        image: require('./images/temperature.png'),
        angle: false
    },
    ...
}
```

Listagem 6.25: Definição dos tipos de medições suportadas.

6.8 DOCUMENTAÇÃO DA API E DEPLOYMENT DO BACKEND

Ao concluir este capítulo referente ao desenvolvimento deste projeto, é importante referir que face ao objetivo de desenvolver uma documentação final da API exposta pela arquitetura de *backend*, esta encontra-se em https://gist.github.com/JoaoVieira97/9cae2ca458b44b1a644b9e5c0c3b8fd0.

Para além disto, no Anexo A deste documento encontra-se descrito o processo de *deploy* da arquitetura de *backend* com recurso a Docker[10] e Docker Compose[11], com o intuito de facilitar o processo de *setup* e execução do sistema.

10 https://www.docker.com/
11 https://docs.docker.com/compose/

7

CONCLUSÃO

Em jeito de conclusão, importa fazer um balanço sobre o trabalho desenvolvido ao longo deste projeto. Assim, é necessário perceber até que ponto o trabalho desenvolvido vai ao encontro da pretendida combinação de várias fontes de dados meteorológicos, desenvolvendo uma aplicação que coloque ao dispor do utilizador informações ligadas a essa mesma área. Importa também realçar que a meteorologia funciona assim como um *case study* para arquiteturas escaláveis e orientadas a serviços (micro-serviços). Desta forma, espera-se que os conceitos estudados e aplicados nesta dissertação possam ser utilizados noutros contextos.

Tendo em conta os objetivos definidos na Secção 1.2, foi realizado o levantamento de várias fontes de dados que podem ser utilizadas num contexto de desenvolvimento desta *mashup*, das quais foram selecionadas a Netatmo e a OpenWeatherMap. A escolha da OpenWeatherMap deve-se ao conteúdo fornecido por esta fonte e a um maior número de chamadas gratuitas à API face a outras alternativas. Ainda assim, foi considerado que a Weatherbit é uma fonte de dados que também pode ser interessante introduzir futuramente no sistema.

Após isto, era também importante perceber quais seriam as principais funcionalidades que a aplicação deveria disponibilizar, de forma a definir um primeiro esboço de alguns métodos que a API do *backend* deveria fornecer. Assim, e tendo em conta que as fontes de dados meteorológicos selecionadas funcionam com base em estações meteorológicas localizadas em variados locais, foram definidos os quatro *endpoints* que num cenário inicial deveriam ser implementados no que diz respeito ao fornecimento de informações meteorológicas: get_area_data, get_historical_measures, get_area_average_measures e get_historical_average_measures.

A conceção de uma arquitetura de *backend* que vá ao encontro dos objetivos delineados era de extrema importância neste projeto, tendo sido a mesma definida com vista na escalabilidade pretendida numa arquitetura de micro-serviços. Para além disto, foi determinada a melhor forma de introduzir *message brokers* e mecanismos de *caching* nesta arquitetura, assim como a forma de interagir com cada uma das APIs das fontes de dados selecionadas através de um *worker* para cada fonte que deve ser responsável por obter as medições mais recentes das estações da sua fonte. Um dos maiores desafios passava por conseguir contornar o limite de chamadas que as APIs oferecem nas suas versões gratuitas, o que foi conseguido com sucesso e teve impacto nas decisões sobre a arquitetura final, como descrito nos capítulos anteriores. Uma melhoria que poderia ser introduzida nesta arquitetura passaria pela adição de um mecanismo de descoberta de serviços utilizada nas arquiteturas de micro-serviços (como discutido na Secção 2.2.2), visto que cada vez mais as aplicações são executados em ambientes virtualizados ou *containers* que escalam dinamicamente, e em que não é apropriada a definição de valores estáticos para os IPs das instâncias em execução para comunicação entre os vários

serviços. Assim, a solução poderia passar por utilizar o padrão *server-side discovery*, em que os detalhes da descoberta de serviços são abstraídos do cliente, permitindo por exemplo não ter que efetuar alterações na aplicação de *frontend*, ao contrário da abordagem *client-side discovery*. Para além disto, um problema identificado anteriormente é o facto do componente Client Server não conseguir validar os identificadores de correlação recebidos nas mensagens de resposta periódicas sobre as estações, isto porque é o componente Weather Server que envia o pedido aos *workers* das fontes de dados e o componente Client Server apenas é um consumidor de mensagens do tópico response-topic do Kafka. Poderiam ser implementadas duas abordagens para tentar colmatar o problema. A primeira foi já mencionada no Capítulo 6, que passaria por criar um novo tópico Kafka que servisse para o componente Weather Server enviar para o componente Client Server os identificadores dos pedidos que o mesmo realiza aos *workers*. Outra solução mais complexa poderia passar por ter o componente Client Server a efetuar também os pedidos das medições mais recentes das estações mas, para evitar processamentos repetidos e continuar a não ultrapassar os limites impostos pelas APIs das fontes deveriam ser introduzidos mecanismos de *caching* nos *workers* das várias fontes.

Relacionado com as fontes de dados, por forma a utilizar as fontes selecionadas e adicionar futuramente outras fontes, era importante definir um protocolo de uniformização das mesmas. Assim, foi definido que os *workers* de cada fonte de dados devem seguir um formato de resposta definido, com detalhes bem determinados como, por exemplo, as unidades escolhidas a que todos os *workers* devem seguir para cada tipo de medição suportada. Esta abordagem é bastante importante, pois permite que os componentes que recebem estas mensagens (neste caso, os componentes Weather Server e Client Server) possam processar todas elas da mesma forma, independentemente do *worker* e da fonte de dados em causa. Para além disto, isto facilita a introdução de novas fontes no sistema, pois seguindo a abordagem definida a principal tarefa passa por introduzir um *worker* capaz de obter as informações da respetiva fonte de dados e que siga o formato de uniformização de respostas definido.

Tendo esta arquitetura definida, passou-se então para a implementação da mesma. Nesta implementação foi determinada a utilização do Kafka como *message broker*, do Redis como sistema de *caching in-memory*, bases de dados MySQL, implementação dos componentes Weather Server e Client Server com recurso à *framework* Django da linguagem Python, linguagem essa utilizada também nos *workers*. Quanto ao API Gateway, utilizou-se o Express Gateway como ferramenta para o desenvolvimento do mesmo. Quanto à API final que esta arquitetura de *backend* expõe através deste API Gateway, a documentação foi realizada tal como planeado.

Por último, foi desenvolvido com sucesso um *frontend* que conseguisse validar o trabalho realizado e permitisse a consulta das informações e operações que o sistema tem para oferecer, tendo a implementação do mesmo sido realizada com recurso a React. Este *frontend* conseguiu assim utilizar todos os *endpoints* expostos pela arquitetura de *backend* e fornecer informações e operações quer a utilizadores registados quer a não registados. Assim, comprovou-se através de uma interface gráfica a utilidade e o correto funcionamento do sistema desenvolvido.

Posteriormente, foi ainda pensado e desenvolvido um conjunto de ficheiros e operações que permitem fazer *deploy* de toda a arquitetura de *backend* com recurso a Docker e Docker Compose, facilitando o processo de *setup* e execução do sistema em apenas uma máquina ou até em várias máquinas remotas.

7.1 TRABALHO FUTURO

No desenvolvimento de um projeto com estas caraterísticas existirá sempre espaço para melhorias.

Assim, para além dos aspetos já mencionados anteriormente, uma melhoria passaria por explorar mais as funcionalidades do API Gateway. Apesar do principal objetivo de um API Gateway nesta arquitetura passar por encaminhar os pedidos recebidos para os componentes Weather Server e Client Server, poderiam ser exploradas outras funcionalidades como uma possível monitorização do mesmo, por exemplo, fazer recolha de métricas como pedidos respondidos com sucesso/insucesso, informações sobre a utilização de cada um dos *endpoints* expostos, etc.

Outra tarefa futura passaria por abordar com maior detalhe um possível *deploy* da arquitetura tendo em vista uma maior escalabilidade horizontal e uma maior disponibilidade de serviços, ou seja, tentar que não existam pontos únicos de falha pois, por exemplo, se o componente Weather Server estiver apenas a executar uma única instância e a mesma falhar, isto faz com que este serviço fique indisponível. Para além disto, aumentando o número de instâncias deste serviço (a título de exemplo) permitiria uma maior capacidade de resposta caso a carga de pedidos assim o justificasse e permitiria assim fazer *load balancing* dos pedidos pelas instâncias disponíveis. No caso dos *workers*, seria importante ter mais uma instância de cada *worker* para o caso do principal falhar e poder existir assim uma instância que assumisse o seu lugar. O mesmo pode ser analisado para o próprio API Gateway, tendo até em conta que é o ponto de entrada no sistema e é fulcral que o mesmo esteja sempre ativo. Uma ferramenta que poderia ser interessante de introduzir com vista a este mesmo objetivo seria o Kubernetes, tendo em conta que de momento os vários componentes já se encontram devidamente divididos em *containers*. O Kubernetes seria útil para conseguir ter várias instâncias de cada componente, tornar a aplicação resistente à falha de algum deles com a definição de um número de réplicas ou a substituição imediata por outra instância, facilitar no problema da descoberta de serviços referido em cima, entre outras vantagens.

Por último, poderia ser explorada a capacidade de escalar horizontalmente as partições de um tópico Kafka por várias máquinas através da utilização de clusters de mediadores ao invés de utilizar um único mediador, o que poderia permitir um melhor desempenho. Ainda assim, no contexto de troca de mensagens utilizado na arquitetura, a carga de mensagens é maior num curto espaço de tempo periodicamente, ou seja, no momento em que os *workers* são solicitados para a obtenção das medições mais recentes das estações da sua fonte.

Num contexto mais geral e numa perspetiva de um cenário ideal para aplicação final, a mesma poderia ainda ter mais fontes de dados meteorológicos, o que levaria a mudanças no *backend* e a apenas alguns ajustes de detalhes no *frontend*, e poderia também oferecer a possibilidade de mudança das unidades dos tipos de medições suportadas aos utilizadores.

BIBLIOGRAFIA

[1] Darlene Fichter. What Is a Mashup? In Nicole C. Engard, editor, *Library Mashups: Exploring New Ways to Deliver Library Data*, chapter 1. Information Today, 2009.

[2] Sunilkumar Peenikal. *Mashups and the Enterprise*, 9 2009.

[3] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12(5):44–52, 2008. doi: 10.1109/MIC.2008.114.

[4] The Art of Service. Types of Mashups, 11 2009. URL https://theartofservice.com/types-of-mashups.html. Acedido em 02/11/2019.

[5] HousingMaps, n.d. URL http://www.housingmaps.com/. Acedido em 02/11/2019.

[6] TrendsMap, n.d. URL https://www.trendsmap.com/. Acedido em 02/11/2019.

[7] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015. doi: 10.1109/MS.2015.11.

[8] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018. doi: 10.1109/MS.2018.2141030.

[9] Joydip Kanjilal. Pros and cons of monolithic vs. microservices architecture, 1 2020. URL https://searchapparchitecture.techtarget.com/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture. Acedido em 31/08/2020.

[10] Chris Richardson and Floyd Smith. *Microservices: From Design to Deployment*. Nginx, 2016.

[11] Chris Richardson. The Scale Cube, n.d. URL https://microservices.io/articles/scalecube.html. Acedido em 31/08/2020.

[12] Sam Newman. *Building Microservices*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2015.

[13] Roy Thomas Fielding. Representational State Transfer (REST). In *Architectural Styles and the Design of Network-based Software Architectures*, chapter 5. University of California, Irvine, 2000. doi: 10.5555/932295.

[14] Chris Richardson. Pattern: API Gateway / Backends for Frontends, n.d. URL https://microservices.io/patterns/apigateway.html. Acedido em 01/09/2020.

[15] Chris Richardson. Pattern: Server-side service discovery, n.d. URL https://microservices.io/patterns/server-side-discovery.html. Acedido em 23/09/2020.

[16] Chris Richardson. Pattern: Client-side service discovery, n.d. URL https://microservices.io/patterns/client-side-discovery.html. Acedido em 23/09/2020.

[17] Jakub Korab. *Understanding Message Brokers*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2017.

[18] Mark Richards, Richard Monson-Haefel, and David A. Chappell. Messaging Models. In Mike Loukides and Julie Steele, editors, *Java Message Service*, chapter 2. O'Reilly Media, 2 edition, 2009.

[19] Paul Le Noac'h, Alexandru Costan, and Luc Bougé. A performance evaluation of Apache Kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4803–4806. IEEE, 2017. doi: 10.1109/BigData.2017.8258548.

[20] Khin Me Me Thein. Apache Kafka: Next Generation Distributed Messaging System. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[21] Danny Kay. Understanding the 'enable.auto.commit' Kafka Consumer property, 7 2018. URL https://medium.com/@danieljameskay/understanding-the-enable-auto-commit-kafka-consumer-property-12fa0ade7b65. Acedido em 02/11/2019.

[22] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, 2015. doi: 10.14778/2824032.2824063.

[23] Wei Ling Chen. Understanding Enterprise Integration Patterns, 3 2009. URL https://dzone.com/articles/enterprise-integration. Acedido em 02/11/2019.

[24] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley Professional, Boston, Massachusetts, EUA, 2003.

[25] Vivek Kumar Singh. What is Caching ?, 5 2019. URL https://medium.com/system-design-blog/what-is-caching-1492abb92143. Acedido em 18/09/2020.

[26] The System Design Primer, n.d. URL https://github.com/donnemartin/system-design-primer#cache. Acedido em 18/09/2020.

[27] Cloudflare. What is Caching? | How is a Website Cached?, n.d. URL https://www.cloudflare.com/learning/cdn/what-is-caching/. Acedido em 18/09/2020.

[28] Kousik Nath. All things caching- use cases, benefits, strategies, choosing a caching technology, exploring some popular products, 12 2018. URL https://medium.com/datadriveninvestor/all-things-caching-use-cases-benefits-strategies-choosing-a-caching-technology-exploring-fa6c1f2e93aa. Acedido em 18/09/2020.

[29] Bluzelle. Things You Should Know About Database Caching, 3 2019. URL https://blog.bluzelle.com/things-you-should-know-about-database-caching-2e8451656c2d. Acedido em 18/09/2020.

[30] Pankaj Kumar. Memcached vs Redis, Which One to Choose?, 5 2019. URL https://medium.com/@pankaj.itdeveloper/memcached-vs-redis-which-one-to-choose-d5177482dc42. Acedido em 21/09/2020.

[31] AWS. Comparing Redis and Memcached, n.d. URL https://aws.amazon.com/pt/elasticache/redis-vs-memcached/. Acedido em 21/09/2020.

[32] Netatmo. Netatmo connect, n.d. URL https://dev.netatmo.com/apidocumentation/. Acedido em 06/01/2020.

[33] Netatmo. Netatmo connect - Rate limits, n.d. URL https://dev.netatmo.com/guideline/#rate-limits. Acedido em 06/01/2020.

[34] Report for Netatmo, n.d. URL https://apps.apple.com/pt/app/report-for-netatmo/id1201207614. Acedido em 08/01/2020.

[35] Mario Hoyos. What is an ORM and Why You Should Use it, 12 2018. URL https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a. Acedido em 05/10/2020.

[36] Django. Making queries, n.d. URL https://docs.djangoproject.com/en/3.1/topics/db/queries/. Acedido em 05/10/2020.

[37] Google LLC. Configurações do Redis, n.d. URL https://cloud.google.com/memorystore/docs/redis/redis-configs?hl=pt. Acedido em 05/10/2020.

[38] Flask-SQLAlchemy, n.d. URL https://flask-sqlalchemy.palletsprojects.com/en/2.x/. Acedido em 06/10/2020.

[39] Express Gateway | A Microservices and Serverless API Gateway Built on Express.JS, n.d. URL https://www.express-gateway.io/. Acedido em 07/10/2020.

[40] React – A JavaScript library for building user interfaces, n.d. URL https://reactjs.org/. Acedido em 14/10/2020.

[41] What is Docker?, n.d. URL https://opensource.com/resources/what-docker. Acedido em 16/10/2020.

A

GUIÃO PARA O DEPLOYMENT DA ARQUITETURA DE BACKEND

Durante o desenvolvimento do *backend* da aplicação foi utilizado um *setup* em que todos os componentes correm localmente e em portas diferentes da mesma máquina, estando todas as ferramentas necessárias instaladas e configuradas na máquina local.

Ainda assim, para permitir um melhor *deploy* da arquitetura tanto numa só máquina como em várias máquinas diferentes foram utilizadas as ferramentas Docker e Docker Compose para agilizar este mesmo processo. O Docker é uma ferramenta que permite criar, fazer *deploy* e correr aplicações utilizando *containers*, que permitem empacotar uma aplicação com todas as partes necessárias como bibliotecas e outras dependências num só pacote [41]. Quanto ao Docker Compose, este permite correr vários *containers* como apenas um serviço, o que é bastante útil para serviços mais complexos que necessitam de vários *containers* e, assim, através de um ficheiro conseguimos iniciar os mesmos sem ser preciso correr cada um separadamente.

Começando pelo `OpenWeather Worker`, em primeiro lugar é necessário definir o *token* da API OpenWeatherMap no ficheiro `config.py`, que será utilizado para interagir com esta API.

```
OPENWEATHER_KEY = '<openweather_token>'
```

Neste mesmo ficheiro é ainda definido que o servidor Kafka com que o componente deve interagir encontra-se por *default* no *localhost*, mas que poderá ser alterado através de uma variável ambiente `KAFKA_SERVER`.

```
KAFKA_SERVER = os.getenv('KAFKA_SERVER', 'localhost:9092')
```

Tendo em conta que o Kafka deve ser algo executado à parte, visto que vários componentes interagem com o mesmo, este componente precisa então apenas de executar um *container*, sendo então utilizado um Dockerfile para fazer *deploy* do mesmo. Assim, este ficheiro é simples de definir, tal como se pode ver na Listagem A.1. É utilizado o Python na sua versão 3.8, instalam-se as dependências deste mesmo componente e, depois, pode-se então executar o mesmo.

```
FROM python:3.8
WORKDIR /code
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python","main.py"]
```

Listagem A.1: Dockerfile do componente OpenWeather Worker.

Depois, devemos criar a imagem deste componente através do seguinte comando.

```
$ docker build -t netatmo_worker .
```

E precisamos apenas de depois correr esta mesma imagem. No comando apresentado de seguida é tido em conta que o Kafka está a ser executado na mesma máquina, daí também a utilização de `--net=host`. No caso de ser pretendido que este componente interaja com um servidor Kafka que esteja a ser executado numa máquina remota basta então utilizar a variável ambiente `KAFKA_SERVER` com o valor desejado.

```
$ docker run --net=host -e KAFKA_SERVER='localhost:9092' \
   -it netatmo_worker
```

Passando para o outro *worker*, `Netatmo Worker`, este tem exatamente o mesmo processo de *deploy* com o mesmo Dockerfile da Listagem A.1 e os mesmos comandos, sendo que a única diferença é mesmo relativa aos *tokens* da API da Netatmo que devem ser definidos no ficheiro `config.py` (Listagem A.2).

```
NETATMO_CLIENT_ID = '<netatmo_client_id>'
NETATMO_CLIENT_SECRET = '<netatmo_client_secret>'
NETATMO_ACCESS_TOKEN = '<netatmo_access_token>'
NETATMO_REFRESH_TOKEN = '<netatmo_refresh_token>
```

Listagem A.2: Definição dos *tokens* da API da Netatmo no ficheiro config.py.

Quanto ao componente `Client Server`, este já apresenta uma complexidade superior aos dois *workers* mencionados anteriormente. No ficheiro `settings.py` deste componente é definido que o valor por *default* do *host* da base de dados MySQL é o *localhost*, e que a localização do servidor Kafka é também o *localhost* mas na porta 9092, sendo que ambos podem ser alterados através de variáveis ambiente `MYSQL_HOST` e `KAFKA_SERVER`.

Foi definido um ficheiro Dockerfile para a aplicação Django deste componente, sendo o conteúdo do mesmo apresentado na Listagem A.3. Especial referência à segunda linha, pois atribuindo um valor a `PYTHONBUFFERED` estipula que os *outputs* do Django por exemplo são enviados diretamente para o terminal sem serem primeiro colocados em *buffer*, podendo assim ver os mesmos em "tempo real".

```
FROM python:3.8
ENV PYTHONBUFFERED 1
WORKDIR /code
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

Listagem A.3: Dockerfile da aplicação Django do componente Weather Server.

Para além disto, visto que este serviço precisa de ter uma base de dados MySQL, foi utilizado o Docker Compose para orquestrar ambos os *containers*. O conteúdo do ficheiro `docker-compose.yml` é apresentado na Figura 67.

```
1  version: '3.5'
2
3  services:
4      client_db:
5          image: mysql
6          container_name: client_db
7          environment:
8              - MYSQL_DATABASE=client_server_db
9              - MYSQL_USER=client_server_application
10             - MYSQL_PASSWORD=client_password
11             - MYSQL_ROOT_PASSWORD=client_root_password
12         command: --default-authentication-plugin=mysql_native_password --mysqlx=0
13         ports:
14             - "3308:3306"
15
16     client_server:
17         container_name: client
18         build: .
19         environment:
20             KAFKA_SERVER: "${KAFKA_SERVER}"
21             MYSQL_HOST: "${MYSQL_HOST}"
22         command: "sh -c '
23             sleep 40
24             && python manage.py makemigrations client
25             && python manage.py migrate client
26             && python manage.py makemigrations
27             && python manage.py migrate
28             && python manage.py loaddata client_server_dump.json
29             && python manage.py runserver 0.0.0.0:${PORT} --noreload
30         '"
31         ports:
32             - "${PORT}:${PORT}"
33         depends_on:
34             - client_db
```

Figura 67: Conteúdo do ficheiro docker-compose.yml do Client Server.

Desta forma, é definida primeiramente a base de dados (linhas 4-14), utilizando a imagem `mysql` do Docker e definindo os vários detalhes da mesma. Nas linhas 13 e 14 define-se que a porta 3306 deste *container* pode ser acedida na porta 3308 do *localhost*. Esta decisão é tomada porque pode ser conveniente aceder à base de dados até para fazer um *backup* da mesma, e a porta 3308 é escolhida para o caso de já existir na máquina uma instância do MySQL a correr na porta 3306.

Passando para o *container* `client_server` deste ficheiro (linhas 16-34), o `build` define que deve ser utilizado o Dockerfile mencionado em cima para montar a imagem (linha 18). São também declaradas as variáveis ambiente `KAFKA_SERVER` e `MYSQL_HOST` que serão mencionadas novamente de seguida (linhas 19-21). Quanto à secção `commands` (linhas 22-30), a mesma é utilizada para especificar o conjunto de comandos que este *container* deve executar, sendo que inicialmente o mesmo para por 40 segundos como uma forma de garantir que o *container* da base de dados já está pronto. Depois, os restantes comandos especificam as migrações que devem ser feitas, um possível *load* de dados para a base de dados que pode ser feito e, por

fim, é colocado o servidor a executar numa porta definida. Relativamente a esta porta, a mesma é exposta para o *localhost* e também será mencionada de seguida. Por fim, e como esperado, o *depends_on* (linhas 33-34) especifica que este *container* depende do *container* da base de dados.

Para facilitar a utilização deste ficheiro, foi depois criado um *script* `run.sh` que permite perguntar ao utilizador pelos valores que as variáveis `KAFKA_SERVER`, `MYSQL_HOST` e `PORT` devem tomar nas linhas 20, 21, 29 e 32 do ficheiro `docker-compose.yml` e que inicia logo de seguida a execução do `Client Server`. Um exemplo de execução deste *script* pode ser visto na Figura 68.

```
$ ./run.sh
PORT TO EXPOSE THE SERVER: 8002
KAFKA (SERVER:PORT) TO CONNECT: 10.10.10.10:9092
MYSQL HOST TO CONNECT: client_db
```

Figura 68: Execução do Client Server através do *script* run.sh.

Claro que, ainda assim, é importante referir que isto apenas torna mais simples a execução do Docker Compose e que a opção de definir o *host* da base de dados deve ainda assim ser a de utilizar a do *container* criado `client_db`, caso contrário estará a ser executado um *container* MySQL que não estará a ser utilizado e seria mais benéfico procurar outras opções. Serve assim apenas a título de exemplo esta definição, visto que a abordagem tomada passa por ter tanto a aplicação Django como a base de dados a executar na mesma máquina.

Passando para o `Weather Server`, este é o componente cuja complexidade do *deploy* pode ser considerada a mais elevada. No ficheiro `settings.py` deste componente segue-se a mesma abordagem relatada anteriormente relativa à existência de variáveis ambiente, sendo definido que por *default* o *host* em que se encontra a base de dados MySQL e o Redis é o *localhost*, e que a localização do servidor Kafka é também o *localhost* mas na porta 9092, sendo que todos estes valores podem ser alterados através das variáveis ambiente `MYSQL_HOST`, `REDIS_HOST` e `KAFKA_SERVER` respetivamente.

Depois, foi definido novamente um ficheiro Dockerfile para a aplicação Django deste componente, sendo o conteúdo do mesmo igual ao ficheiro Dockerfile para o `Client Server` apresentado na Listagem A.3.

A maior complexidade do *deploy* deste componente passa por o mesmo necessitar de uma base de dados MySQL e de uma instância do Redis para guardar os pares chave-valor como pretendido, sendo novamente utilizado o Docker Compose para orquestrar os *containers* necessários, estando o conteúdo do ficheiro `docker-compose.yml` correspondente apresentado na Figura 69.

```
1  version: '3.5'
2
3  services:
4      weather_db:
5          image: mysql
6          container_name: weather_db
7          environment:
8              - MYSQL_DATABASE=weather_server_db
9              - MYSQL_USER=weather_server_application
10             - MYSQL_PASSWORD=weather_password
11             - MYSQL_ROOT_PASSWORD=weather_root_password
12         command: --default-authentication-plugin=mysql_native_password --mysqlx=0
13         ports:
14             - "3307:3306"
15
16     redis:
17         image: redis
18         container_name: redis
19         restart: always
20         ports:
21             - "6380:6379"
22         command: redis-server --maxmemory 4gb --maxmemory-policy volatile-lfu
23
24     weather_server:
25         container_name: weather
26         build: .
27         environment:
28             KAFKA_SERVER: "${KAFKA_SERVER}"
29             MYSQL_HOST: "${MYSQL_HOST}"
30             REDIS_HOST: "${REDIS_HOST}"
31         command: "sh -c '
32             sleep 40
33             && python manage.py makemigrations weather
34             && python manage.py migrate weather
35             && python manage.py makemigrations
36             && python manage.py migrate
37             && python manage.py loaddata weather_server_dump.json
38             && ${CELERY_COMMAND}
39             && python manage.py runserver 0.0.0.0:${PORT} --noreload
40         '"
41         ports:
42             - "${PORT}:${PORT}"
43         depends_on:
44             - weather_db
45             - redis
```

Figura 69: Conteúdo do ficheiro docker-compose.yml do Weather Server.

Sendo assim, é definida em primeiro lugar a base de dados deste componente (linhas 4-14) de forma similar ao que foi realizado para o `Client Server`, sendo ainda assim definido que a porta 3306 do *container* pode ser acedida através da porta 3307 do *localhost*, escolhendo assim uma porta diferente da escolhida no componente anterior para o caso de se pretender fazer *deploy* de ambos na mesma máquina.

Depois, face à necessidade da presença do Redis neste componente, é definido um *container* para este mesmo efeito utilizando a imagem `redis` do Docker (linhas 16-22). Muito importante referir que no comando que determina a execução do Redis (linha 22) são então passadas as configurações discutidas anteriormente para o sistema de caching, definindo `maxmemory` e `maxmemory-policy`.

Por último, o *container* `weather_server` (linhas 24-45) é bastante semelhante ao *container* `client_server` apresentada na Figura 67, sendo adicionada uma nova dependência relativa ao Redis na secção `depends_on` (linhas 43-45). Para além disto é acrescentada na secção `environment` a variável `REDIS_HOST`, que tal como o `KAFKA_SERVER`, `MYSQL_SERVER` e `PORT` (linhas 39 e 42) podem ser definidas através através de um *script* `run.sh`, criado para facilitar a utilização deste ficheiro `docker-compose.yml`. Um exemplo de execução deste *script* pode ser visto em baixo na Figura 70.

```
$ ./run.sh
PORT TO EXPOSE THE SERVER: 8001
KAFKA (SERVER:PORT) TO CONNECT: 10.10.10.10:9092
MYSQL HOST TO CONNECT: weather_db
REDIS HOST TO CONNECT: redis
SERVER SHOULD ONY LISTEN TO KAFKA MESSAGES? (Y/N): Y
```

Figura 70: Execução do Weather Server através do *script* run.sh.

A quinta pergunta deste *script* foi pensada para o caso de se fazer *deploy* de várias instâncias do `Weather Server` e, assim, ser possível definir que as instâncias podem apenas receber as mensagens do Kafka sabendo que existe outra instância que trata de efetuar os pedidos de hora em hora. Assim, esta pergunta está relacionada com a linha 38 da Figura 69 pois, se o servidor apenas receber mensagens do Kafka o comando executado é apenas um *print* que indica isso mesmo, caso contrário, é então necessário nessa mesma linha executar o comando Celery responsável pela *task* periódica do pedido das medições mais recentes das estações meteorológicas.

Novamente, é importante referir que utilizando este *script* deve ser feito uso dos *containers* `weather_db` e `redis` criados, caso contrário estarão a ser executados *containers* sem serem utilizados, pelo que o melhor para tal seria utilizar outra abordagem.

Um detalhe importante tanto nas Figuras 68 e 69, é a utilização da localização `10.10.10.10:9092` para o servidor Kafka. Isto pode servir meramente como um exemplo para um servidor Kafka que possa estar disponível remotamente, como pode servir para um servidor Kafka que esteja a correr na mesma máquina que o Docker Compose. Assim, se no *localhost* estiver a ser executado o Kafka é necessário criar um *alias* para o *localhost*, de forma a que o Docker Compose consiga interagir com o mesmo. Este *alias* pode ser criado através dos seguintes comandos.

```
$ sudo ifconfig lo:0 10.10.10.10
$ sudo service networking restart
```

Por último, falta então mencionar o deploy do `API Gateway`. Este componente da arquitetura necessita então de criar um único *container*, sendo utilizado um Dockerfile para fazer o seu *deploy*, com o seu conteúdo apresentado na Listagem A.4.

```
FROM node:12
WORKDIR /usr/src/app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 8000
CMD ["npm", "start"]
```

Listagem A.4: Dockerfile do componente API Gateway.

Assim, é utilizada a imagem `node` do Docker, são instaladas as dependências desta aplicação Node.js, é exposta a porta 8000 onde o componente deverá ser contactado e é por fim colocado o `API Gateway` em execução. A imagem deste componente pode ser criada através do seguinte comando.

```
$ docker build -t api_gateway .
```

Sendo então necessário depois apenas colocar a mesma imagem em execução, como pode ser visto em baixo. Importante realçar que a configuração `--net=host` é opcional e dependente das necessidades do *deploy* feito pois, por exemplo, pode tornar-se necessária para aceder a instâncias do `Weather Server` e `Client Server` se os mesmos estiverem a ser executados na mesma máquina. Quanto à configuração `-p`, a mesma permite mapear a porta 8000 do *container* à porta 8000 do *host*.

```
$ docker run --net=host -p 8000:8000 -it api_gateway
```