Politécnico do Porto
Instituto Superior de Engenharia do Porto

Developing applications with Ballerina

Tiago Nora

LETI
Licenciatura em Engenharia de Telecomunicações e Informática

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

Junho, 2023

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Projeto/Estágio, do 3º ano, da Licenciatura em Engenharia de Telecomunicações e Informática.

Candidato: Tiago Nora, Nº 1201050, 1201050@isep.ipp.pt

Orientação Científica: Isabel Azevedo, ifp@isep.ipp.pt

Organização: GILT - Games, Interaction and Learning Technologies

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

Junho, 2023

(Opcional) Poderá usar esta secção para dedicar o trabalho a alguém...

Agradecimentos

(Opcional) Agradecimentos que sejam devidos...

Resumo

Os microsserviços são uma abordagem que atualmente tem cada vez mais ganho popularidade e relevância. Tem ganho este prestigio e adoção devido as suas caraterísticas comparando com outras arquiteturas.

Hoje em dia, esta arquitetura é muito utilizada devido ao crescente número de utilizadores, aumento de tráfego necessário e diferentes localizações desses mesmos utilizadores.

Esta abordagem é vantajosa dado que cada um dos serviços pode ser implementado e testado independentemente, permitindo assim o desenvolvimento paralelo, facilidade na manutenção do sistema e adição de novas funcionalidades.

O uso de microsserviços auxiliado com o uso de ferramentas de gestão de containers facilitam a implementação do serviço e promovem caraterísticas já existentes como, por exemplo, a escalabilidade e elasticidade.

Apesar das vantagens, esta arquitetura apresenta desafios, tais como, na comunicação entre serviços e à medida que são adicionados mais serviços a complexidade do sistema aumenta.

No presente documento será explorado a linguagem em estudo, Ballerina, como são implementados os serviços e as várias tecnologias associadas ao desenvolvimento de um sistema em que é usado microsserviços.

Palavras-Chave: microsserviços, Ballerina, serviços web, deteção de linguagem

Abstract

Microservices are an approach that is currently gaining more and more popularity and relevance. It has gained this prestige and adoption due to its characteristics when compared to other architectures.

Today, this architecture is widely used due to the growing number of users, the increase in traffic required, and the different locations of those users.

This approach is advantageous because each of the services can be implemented and tested independently, thus allowing parallel development, easy system maintenance, and the addition of new features.

The use of microservices aided with the use of container management tools facilitates service implementation and promotes existing features such as scalability and elasticity.

Despite the advantages, this architecture presents challenges, such as in the communication between services and as more services are added the complexity of the system increases.

This paper will explore the language under study, Ballerina, how the services are implemented and the various technologies associated with the development of a system in which microservices are used.

Keywords: microservices, Ballerina, web services, language detection

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Lista de Acrónimos e Siglas

API	*Application Programming Interface*
CQRS	*Command Query Responsibility Segregation*
CRUD	*Create, Read, Update and Detroy*
CSV	*Comma-Separated Values*
DDD	*Domain-Driven Design*
gRPC	*Google Remote Procedure Call*
IDE	*Integrated Development Environment*
IMAP	*Internet Message Access Protocol*
JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
MariaDB	*Maria Database*
MySQL	*My Structured Query Language*
OOP	*Object Oriented Programming*
RabbitMQ	*Rabbit Message Queue*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
SOAP	*Simple Object Access Protocol*
SOAPUI	*Simple Object Access Protocol User Interface*
TCP	Transmission Control Protocol
UML	*Unified Modeling Language*
VSC	*Visual Studio Code*
WSO2	*Web Services Oxygenated 2*

XML *Extensible Markup Language*

YAML *YAML ain't markup language*

Capítulo 1

Introdução

Neste capitulo será apresentado o problema a ser resolvido, a descrição do projeto e os seus objetivos, além disso, será apresentado as atividades desenvolvidas em etapas ao longo do projeto e por fim será descrito como o documento foi dividido.

1.1 Contextualização

A abordagem da arquitetura de microsserviços no desenvolvimento de *software* tem cada vez ganho, mais distinção nos últimos anos. Surge pelos problemas registados pelas empresas na construção de um sistema que seja escalável e flexível relativamente aos requisitos de hoje em dia. A construção de um sistema numa só aplicação dificulta a implementação desses mesmos aspetos.

Cada serviço é responsável pela execução de uma tarefa específica ou um grupo específico de tarefas, agrupados conforme as funcionalidades. Visto que, que cada serviço pode funcionar de uma forma independente em relação aos outros serviços, estes podem ser escalado individualmente, fazendo com que, o desenvolvimento e manutenção do sistema seja mais ágil e flexível.

A arquitetura de microsserviços facilita a integração com outros sistemas e serviços, mas, por outro lado, é preciso garantir que a comunicação entre eles seja bem-sucedido, isso inclui a sua segurança, disponibilidade e escalabilidade, caraterísticas importantes para uma comunicação eficiente.

As novas linguagens de programação como *Ballerina* fornecem mecanismos mais simplificados e eficientes na implementação de microsserviços e disponibilização suporte nativos à maioria dos protocolos de comunicação (*HTTP*, *RPC*, *Internet Message Access Protocol (IMAP)*, entre outros), que permitem aos desenvolvedores resolver muitos dos problemas referidos em cima. Além disso, recentemente chegou ao top 100 das linguagens de programação mais populares, mais em específico coloca-se no lugar 87º do *ranking*, ou seja, indica que cada vez mais, esta linguagem está a ganhar mais popularidade no tema em estudo [1].

1.2 Descrição do Projeto

É importante mencionar que este trabalho é de natureza académica e visa explorar as possibilidades da linguagem de programação para um público amplo, incluindo aqueles que não possuem ampla experiência no desenvolvimento de aplicações com múltiplos serviços. Além disso, todo o código desenvolvido e o presente documento será disponibilizado publicamente no *Github*, com a licença X (também chamada licença MIT).

1.2.1 Objetivos

O trabalho passa por desenvolver uma aplicação com a linguagem de programação *Ballerina* com múltiplos serviços (*Sandwich*, *Ingredient*, *Review*, *Shop*, *Order*, *Customer*) para suprir as necessidades de uma loja de sanduíches com funcionalidades relacionadas com o negócio, como, por exemplo, criação de sanduíches, listagem dos ingredientes disponíveis, realizar uma encomenda, com algum suporte multilíngua. Uma arquitetura baseada em microsserviços será adotada recorrendo também a eventos para comunicação entre serviços. Todo o trabalho ficará disponível num repositório público como contributo para divulgação da linguagem, das suas características, potenciais dificuldades, tal como, problemas que condicionem a sua adoção.

O objetivo deste trabalho é desenvolver todas as funcionalidades que suportem as operações necessárias para o funcionamento correto do sistema na totalidade com duas restrições: a linguagem de programação tem de ser *Ballerina* e a aplicação deve usar todas as características próprias da linguagem para a implementação de microsserviços. Dado a implementação das funcionalidades anteriormente mencionadas, devem ser feitos, os testes necessários, testes unitários como testes de desempenho para garantir a boa implementação das regras de negócio.

1.3 Calendarização

1.4 Organização do Relatório

Este relatório apresenta-se dividido nos seguintes capítulos:

- **Introdução**: Introdução ao problema e descrição do mesmo
- **Introdução teórica**: Introdução aos tipos de arquitetura, comunicações entre serviços e estudo sobre as tecnologias
- ***Ballerina***: Estudo sobre a linguagem de programação *Ballerina*
- **Deteção de linguagem**: Estudo sobre aplicações de deteção de linguagem
- **Implementação**: Apresentação da implementação da aplicação com vários serviços
- **Conclusão**: Conclui a apresentação do tema, apresenta trabalho futuro e descrição dos resultados obtidos

Capítulo 2

Estado da Arte

Neste capitulo é apresentado várias arquiteturas de *software*, incluindo a comunicação entre serviços e as tecnologias utilizadas para suportar essa abordagem. Será feita uma reflexão no que diz respeito as suas qualidades, importância, imperfeições e caraterísticas.

2.1 Arquiteturas de software

A arquitetura de *software* é um termo usado para descrever as caraterísticas, estrutura e comportamento dos componentes de um sistema. As quatro principais arquiteturas de *software* são: arquitetura monolítica, arquitetura orientada a serviços, arquitetura de microsserviços e arquitetura orientada a eventos. A arquitetura monolítica é a mais antiga, passando pela arquitetura orientada a serviços e acabando pelas duas restantes, respetivamente em ordem cronológica. Cada uma das arquiteturas segue uma abordagem diferente, apesar que, a arquitetura de microsserviços e arquitetura orientada a eventos podem coexistir na mesma aplicação.

2.1.1 Arquitetura Monolítica

Na arquitetura monolítica, o código e as suas funcionalidades estão contidas e consolidadas numa única aplicação [2]. Este método de desenvolvimento de *software* é indicado para pequenos projetos, dado que, torna-se complicado escalar e manter em grandes projetos, devido as constantes mudanças dos mesmos e outros fatores. A arquitetura monolítica tem várias vantagens, tais como, um desenvolvimento inicial

rápido, custos de desenvolvimento mais baixos e testes de funcionalidade apenas num local. Além disso, a comunicação entre módulos é facilitada e ocupa menos espaço. Contudo, as suas desvantagens incluem manutenção difícil das funcionalidades, escalabilidade comprometida, dificuldade em efetuar alterações ao código, existência de um único ponto de erro, e pouca flexibilidade [3].

A figura 2.1 é um exemplo de uma aplicação construida com a arquitetura monolítica apresenta todos os seus componentes no mesmo espaço.

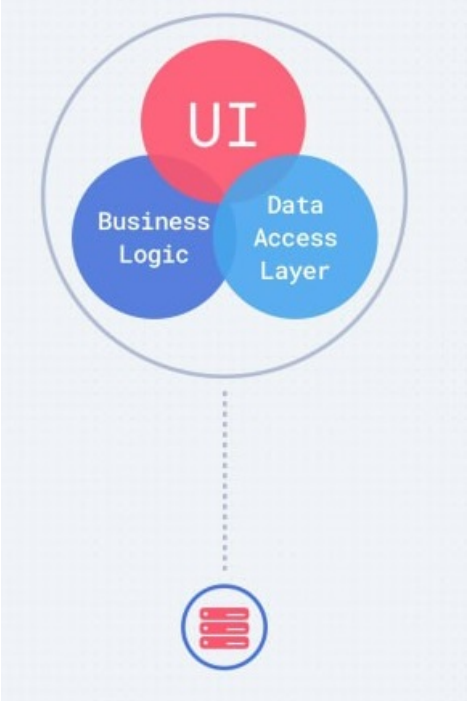

Figura 2.1: Exemplo da arquitetura monolítica [4].

2.1.2 Arquitetura orientada a serviços

A arquitetura orientada a serviços é uma evolução da arquitetura monolítica, por dividir a aplicação em serviços reutilizáveis que podem ser excetuados e implementados independentemente uns dos outros. Sendo que, estes podem estar conectados entre si, recorrendo a protocolos de comunicação. A escalabilidade é facilitada pela arquitetura, ao possibilitar o baixo acoplamento entre os componentes, facilitando a manutenção, reduz os custos e a complexidade [5].

2.1.3 Arquitetura de microsserviços

No caso, da arquitetura de microsserviços, este divide o sistema em pequenas aplicações, independentes umas das outras, ou seja, cada uma pode ser implementada e gerida individualmente, a essas aplicações dá-se o nome de microsserviços. Hoje em dia, esta arquitetura é muito utilizada devido ao crescente número de utilizadores,

aumento de tráfego necessário e diferentes localizações desses mesmos utilizadores [6].

As mais importantes caraterísticas de uma arquitetura de microsserviços são: a escalabilidade e a elasticidade. Sendo que, escalabilidade refere-se à capacidade de um sistema conseguir ajustar se ao crescimento no número de utilizadores ou no volume de dados sem comprometer o desempenho, ou a disponibilidade. A escalabilidade pode ser alcançada sem interromper o normal funcionamento do serviço mediante processos de escalabilidade horizontal, adicionar mais nós, ou vertical, aumentar a capacidade de processamento ou armazenamento de um, ou mais nós. Por outro lado, a elasticidade refere-se à capacidade de um sistema conseguir ajustar-se com as variações no tráfego ou carga de forma automática, aumentando ou diminuindo a alocação de recursos para algum serviço conforme os requisitos no momento. Normalmente esta gestão de alocar automaticamente recursos é feita por meio de contentores [7] [8].

Normalmente esta arquitetura está associada a padrões de *software* como, por exemplo *Command Query Responsibility Segregation* (CQRS) (Segregação de Responsabilidade de Comando e Consulta), onde as responsabilidades da aplicação são divididos em dois blocos, o bloco das operações que modificam dados e o bloco que apresenta os dados da aplicação [9] e *Database Per Service* (base de dados por serviço), em que cada aplicação contém uma base de dados, com os dados estritamente necessários para o seu correto funcionamento e funciona como um tipo de *backup*, dado a redundância de dados [10].

A arquitetura de microsserviços apresenta flexibilidade em relação a modificações, dado o seu baixo acoplamento entre módulos e previne o colapso total do sistema no caso de erro ou falha. No entanto, isso exige um trabalho extra na implementação da comunicação entre serviços, o que pode causar uma complexidade na implementação e construção dos serviços, como também na própria implementação dos serviços devido a sua quantidade [3].

A figura 2.2 é um exemplo de uma aplicação construida com a arquitetura de microsserviços, apresenta-se dividido em vários microsserviços e em várias bases de dados.

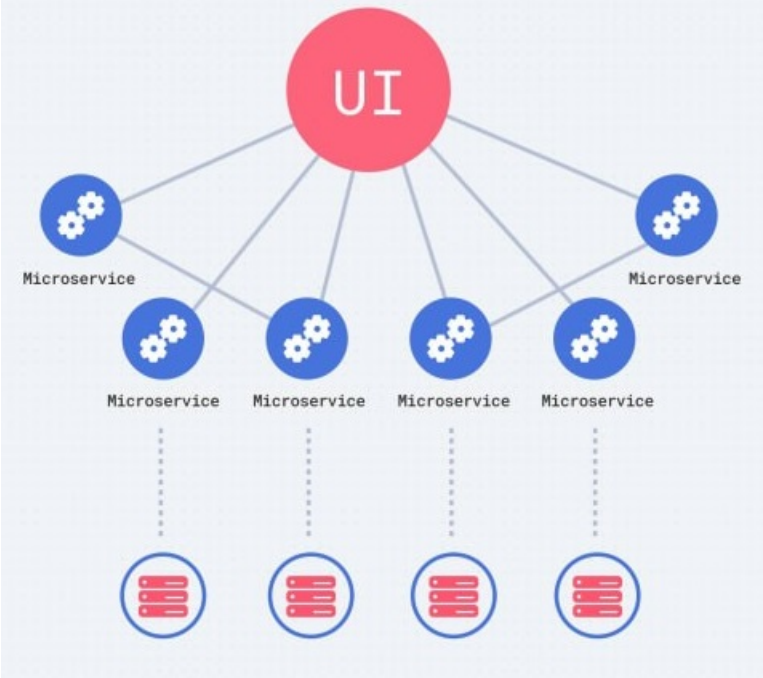

Figura 2.2: Exemplo da arquitetura de microsserviços [4].

2.1.4 Arquitetura orientada a eventos

Numa arquitetura orientada a eventos, os diferentes componentes do sistema, comunicam entre si recorrendo a eventos, eventos esses que podem ser alterações ou criação de elementos da lógica de negócio. Esta arquitetura recorre ao modelo de publicação e subscrição, onde os eventos são enviados através dos publicadores e recebidos pelos subscritores que subscreveram esse mesmo tipo de evento [11].

A figura 2.3 é um exemplo de como se processa eventos numa aplicação com uma arquitetura orientada a eventos.

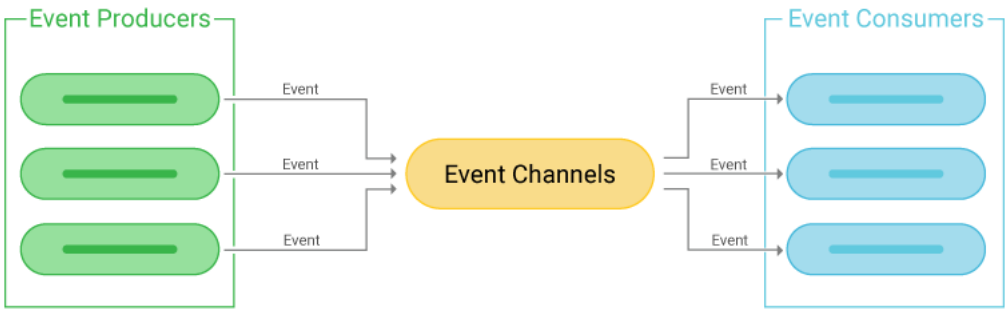

Figura 2.3: Exemplo da arquitetura orientada a eventos [12].

2.2 Comunicação entre serviços

Numa arquitetura, com várias aplicações, muita vezes é preciso fazer passar dados entre elas ou mesmo verificar a existência das mesma e por isso tem se que recorrer os vários tipos de comunicações, como, por exemplo, comunicação síncrona e comunicação assíncrona.

2.2.1 Comunicação síncrona

A comunicação síncrona é denominada comunicação bloqueante, dado que esta, está sempre a esperar uma resposta por parte do recetor do pedido, independentemente do tipo de dados, seja ele uma mensagem de erro como uma mensagem a confirmar a criação de algum tipo de dado e os respetivos atributos.

Começando com o *Remote Procedure Call* (RPC), dado que é o mais antigo deles todos, neste caso a comunicação é feita mediante procedimento remoto entre servidor e cliente. Visto que o *RPC* é uma comunicação de alto nível, esta facilita a implementação da mesma, pois a comunicação ao nível do protocolo Transmission Control Protocol (TCP) é escondida, fazendo com que a desempenho seja maior. Sendo que é feita uma chamada de procedimento remoto, é necessário voltar a desenvolver ou escrever código, mas o esforço necessário é mínimo, devido que se pode reutilizar funcionalidades [13].

Passando ao *Simple Object Access Protocol* (SOAP), este já é um protocolo especifico para comunicação de dados estruturados, com a limitação de só poder receber dados do tipo *Extensible Markup Language* (XML). Comparando com *RPC*, este apresenta uma complexidade maior na sua arquitetura [14].

No que diz respeito ao *Google Remote Procedure Call* (gRPC), este é uma evolução do *RPC*, de outro modo, uma *framework*(ferramenta que auxilia no desenvolvimento de aplicação visto já conter código já definido) do *RPC* tendo sido desenvolvido para combater alguns dos problemas do seu antecessor. A empresa *GOOGLE*, a empresa que a desenvolveu, decidiu implementar, *Protocol Buffers*(descreve a estrutura dos dados) para serializar informação e fazendo assim que microsserviços consigam comunicar entre si com auxílio desta ferramenta [15].

Por meio de *Representational State Transfer* (REST), o servidor disponibiliza ao cliente uma interface de operações bem definidas, com ajuda do protocolo HTTP. Normalmente são usadas operações *Create, Read, Update and Detroy* (CRUD), ou seja, operações de criação, leitura, atualização e destruição. Uma das vantagens é que o servidor permite que o cliente acesse as funcionalidades sem que este tenha conhecimento da implementação da mesma [14].

2.2.2 Comunicação assíncrona

A comunicação assíncrona é denominada de uma comunicação não bloqueante, sendo que a aplicação não fica a espera de uma resposta imediata e prossegue com o normal funcionamento do programa. Exemplos desse tipo de comunicação são:

- *Rabbit Message Queue* (RabbitMQ)
- *Apache Kafka*

Sendo o *RabbitMQ* um método de comunicação assíncrona, a sua utilização é normalmente associada a uma arquitetura orientada a eventos, sendo que, a esta pode também estar associada a uma arquitetura de microsserviços. Neste método a mensagem pode ser propagada ou enviada para um consumidor em específico. No caso de ser enviada para vários remetentes, a mensagem parte de um produtor, no qual, envia para uma *exchange*, que por sua vez vai contar o número de serviços que subscreveram aquele tipo de mensagem, cria *queues* respondente ao número contado com a informação enviada pelo produtor, os consumidores detetam uma nova entrada na *queue*, consumem-na e tratam-na a seguir. Por outro lado, se a mensagem for enviada em que só haverá um remetente, o processo é semelhante, a mensagem parte do consumidor é enviado para a *exchange*, a *exchange* cria só uma *queue* com a informação, o consumidor deteta a nova entrada na *queue*, consume a e trata a seguir [16].

O *Apache Kafka* tem um funcionamento parecido ao *RabbitMQ* apesar que as mensagens em vez de serem armazenadas em *queues* são armazenadas em *logs*. No caso das queues, as mensagens são eliminadas após a entrega das mesma aos consumidores e receberem um *acknowledgment* por parte do mesmo, de outra forma, as *logs* são mantidas num ficheiro e com ajuda de um apontador, o consumidor sabe qual foi a última mensagem consumida, evitando assim a duplicação da mesmas mensagens [17].

Comparando as duas opções, o *Apache Kafka* devido as suas capacidades é mais adequado para casos em que há uma abundante de dados a serem transferidos enquanto, por outro lado, o *RabbitMQ* é usado em casos que se espera que as transferências de mensagens sejam feitas com uma latência baixa.

2.3 Tecnologias

Nesta secção será feita uma introdução às tecnologias envolventes no projeto desenvolvido, explicando as suas funcionalidades, importância e alternativas disponíveis, tendo sempre em consideração alternativas *open source*.

Esta secção está dividida nas seguintes subsecções:

1. Gestor de *containers*

2. Formato de troca de dados
3. Autenticação
4. Base de dados
5. Cliente *web*
6. Ferramenta de teste de desempenho

2.3.1 Gestor de *containers*

No que diz respeito aos gestores de *containers*, estes são pedaços de *software* que ajudam na construção, desenvolvimento e gestão de *containers*.

Cada *container* é um espaço isolado, no qual, está incluído o código e dependências necessárias para o seu funcionamento. Os *containers* são leves, portáveis e dado as suas caraterísticas funciona independentemente do ambiente no qual está inserido.

Exemplos de alguns dos gestores de *containers* mais utilizados são:

- *Podman* [18]
- *Docker* [19]

O *Podman* e o *Docker* são duas ferramentas usadas para gestão de *containers* em *DevOps*. Embora tenham funções semelhantes, existem diferenças importantes entre eles [20].

Em termos de arquitetura, o *Podman* não requer um *daemon* (software que executa como um processo em plano de fundo [21]) para chamar e gerir *containers*, enquanto o Docker depende do *daemon* para lidar com imagens, *containers*, redes e armazenamento. O *Podman* usa Pods para gerir *containers* e permite a execução dos mesmos sem a necessidade de permissões de *root* [20].

O Docker requer permissões de *root* para gerir *containers*, por depender do *daemon*. No entanto, numa atualização recente, o Docker introduziu a execução sem permissões de *root*. No entanto, ainda são necessárias algumas configurações e pacotes de terceiros para executar *containers* sem permissões de *root* no Docker [20].

Relativamente à segurança, o Docker apresenta riscos de segurança, pois, se alguém ganhar acesso a um *container*, poderá comprometer os vários componentes com o mesmo acesso de *root*. O *Podman* é considerado mais seguro nesse aspeto, pois um invasor só poderá prejudicar os *containers* aos quais tem acesso, sem ganhar acesso de *root* [20].

O *Docker* não é apenas capaz de gerir *containers*, mas também de criar imagens. Já o *Podman* é projetado apenas para executar e gerir *containers* [20].

Em termos de suporte externo, o *Docker* suporta o *Docker Swarm* para orquestrar de *containers*, permitindo executar um cluster de nós *Docker* e implantar aplicações escaláveis sem dependências externas. O *Podman* não suporta o *Docker Swarm*, mas pode ser combinado com o *Nomad*, que inclui um *driver* para o *Podman*. Além disso, o *Docker* suporta o *Docker Compose* para gerir aplicativos com vários *containers* num único *host*, enquanto o *Podman* possui o *Podman Compose* como alternativa [20].

A abordagem de trabalho também difere entre o *Podman* e o *Docker*. O *Docker* é independente e pode executar todas as tarefas necessárias de forma autónoma, enquanto o *Podman* adota uma abordagem modular e recorre a várias integrações para realizar diferentes funções [20].

2.3.2 Formato de troca de dados

Relativamente a formatos de troca de dados em aplicações *web*, estes são independentes da plataforma, com uma importância elevada, dado que, conseguem ser lidos e usados por outras aplicações, ou mesmo tempo que, permitem que essa comunicação entre aplicações ou sistemas sejam feitas de forma segura e confiável. Além disso, o uso destes tipos de formatos de trocas de dados proporcionam uma certa flexibilidade.

Alguns exemplos de formatos de troca de dados são: XML, *JavaScript Object Notation* (JSON), *YAML ain't markup language* (YAML) e *Comma-Separated Values* (CSV).

O formato de trocas de dados XML foi uma das primeiras formatos a surgirem, precisamente em 1998. Apesar de novas linguagens de configuração estarem a ultrapassar o XML ainda existem muitas situações em que a natureza estruturada do XML e a sua flexibilidade funcionam melhor para configurações complexas [22].

Na Listagem 2.1 é apresenta um exemplo de forma XML.

```
<people>
  <person>
    <id>1</id>
    <nome>Exemplo1</nome>
    <idade>25</idade>
  </person>
  <person>
    <id>2</id>
    <nome>Exemplo2</nome>
    <idade>30</idade>
  </person>
  <person>
    <id>3</id>
    <nome>Exemplo3</nome>
    <idade>40</idade>
  </person>
</people>
```

Listagem 2.1: Exemplo do formato XML.

Em relação a desvantagens e vantagens, as vantagens do XML, são:

- Existem esquemas para a validação e criação de tipos personalizados
- Permite facilmente a realização de diferentes formatos a partir de uma sintaxe comum [22]

Por sua vez as desvantagens do XML são:

- Não é considerado facilmente legível por humanos devido à natureza descritiva dos elementos.
- Detalhada (aumenta a capacidade de armazenamento e as necessidades de largura de banda) e contém frequentemente sintaxe redundante [22]

No caso do JSON, este existe desde do início dos anos 2000 e reconhecido como uma especificação formal em 2013. É derivado da linguagem de programação JavaScript, mas independente dessa linguagem. Apresenta-se como uma alternativa mais simples em relação a muitos outros formatos e como uma das linguagens mais utilizados no momento [22].

Na Listagem 2.2 é apresenta um exemplo de forma JSON.

```
[
  {
    "id": 1,
    "nome": "Exemplo1",
    "idade": 25
  },
  {
    "id": 2,
    "nome": "Exemplo2",
    "idade": 30
  },
  {
    "id": 3,
    "nome": "Exemplo3",
    "idade": 40
  }
]
```

Listagem 2.2: Exemplo do formato JSON.

Em relação a desvantagens e vantagens, as vantagens do JSON, são:

- Facilmente legível por humanos
- Sintaxe simples
- Rápido para ser analisado dado a sua marcação limitada [22]

Por sua vez as desvantagens do JSON são:

- Suporte limitado em relação a tipo de dados
- Não suporta configurações complexas
- Não suporta comentários de suporte aos atributos [22]

Por outro lado, o YAML foi criado em 2001. Embora o YAML tenha um aspeto diferente do JSON, o YAML é um superconjunto do JSON. Como um superconjunto de JSON, um ficheiro YAML válido pode conter JSON. Além disso, o JSON também pode transformar-se em YAML. O próprio YAML também pode conter JSON nos seus ficheiros de configuração [22].

Na Listagem 2.3 é apresenta um exemplo de forma YAML.

```yaml
- id: 1
  nome: Exemplo1
  idade: 25
- id: 2
  nome: Exemplo2
  idade: 30
- id: 3
  nome: Exemplo3
  idade: 40
```

Listagem 2.3: Exemplo do formato YAML.

Em relação a desvantagens e vantagens, as vantagens do YAML, são:

- Sintaxe Legível por humanos
- Sintaxe compacta
- Suporta tipos de objetos independentes de idioma [22]

Por sua vez as desvantagens do YAML são:

- O formato de indentação é propenso a erros de sintaxe e validação
- A portabilidade com certos tipos pode não existir devido à falta de recursos em todos os idiomas
- A depuração é difícil
- Pontos de interrupção e funcionalidade semelhante não existem [22]

Por último, o CSV. Os arquivos CSV são arquivos de texto que utilizam vírgulas para separar os valores, e são amplamente utilizados em softwares de escritório como o Microsoft Excel [23]. Eles normalmente contêm uma linha de cabeçalho que fornece nomes de coluna para os dados, mas de outra forma, são considerados semiestruturado [24].

Na Listagem 2.4 é apresenta um exemplo de forma CSV.

```csv
id,nome,idade
1,Exemplo1,25
2,Exemplo2,30
3,Exemplo3,40
```

Listagem 2.4: Exemplo do formato CSV.

Em relação a desvantagens e vantagens, as vantagens do CSV, são:

- O formato CSV é simples e de fácil compreensão

- O CSV é amplamente suportado por diferentes softwares e sistemas operativos
- Os arquivos CSV tendem a ter um tamanho menor em comparação com outros formatos Os arquivos CSV são formatados como texto legível [25]

Por sua vez as desvantagens do CSV são:

- Nenhuma restrição no modelo de dados, pode resultar em dados corrompidos
- Não se adapta bem ao trabalhar com *Big Data*. Geralmente, eles não podem ser divididos em partições para processamento paralelo e não podem ser compactados tão bem quanto formatos binários
- Talvez seja necessário aplicar um esquema nos dados semiestruturados para facilitar a consulta e análise [24]

2.3.3 Autenticação

A decisão de fazer a implementação de um sistema de autenticação é crucial para a segurança do sistema inteiro, seja em aplicações públicas como em aplicações privadas. Estes sistemas de autenticação tem o propósito de assegurar a manipulação de dados não desejados e a visualização dos mesmos se assim for o objetivo da mesma aplicação.

Para isso existem alguns formatos de autenticação que ajudam nessa tarefa de realizar a autenticação, como, por exemplo:

- *JSON Web Token* (JWT)
- Chaves *Application Programming Interface* (API)

Começando com o JWT, este é um padrão que define a transferência de dados entre duas partes recorrendo a uma token passado mediante um *HTTP Header*. Este mesmo token é gerado por meio de uma chave privada e acessados com uma chave publica sendo composto por três componentes: *Header*, *Payload* e *Signature*. O *Header* contém informações sobre o algoritmo usado para criar o *hash* e o tipo de *token*, o *Payload* contém *claims* como, por exemplo, objetos *JSON*, data de expiração, quem emitiu o *token* e outras informações adicionais, e a *Signature* é a junção dos *hashes* gerados [26].

E finalmente, as chaves API estas são códigos únicos fornecidos ao utilizador que identificam o mesmo. As chaves API são um bom meio de controlo e supervisão do sistema, visto que, pode prevenir o abuso da API por meio de restrições e utilizadores maliciosos. Comparando com o exemplo apresentado anteriormente, este tipo de autenticação é menos segura [27].

2.3.4 Base de dados

No que diz respeito, a base de dados é uma ferramenta necessária para o armazenamento e consulta desses mesmo dados. Atualmente são utilizadas dois tipos de bases de dados, bases de dados relacionais e as bases de dados não relacionais.

Em relação a bases de dados relacionais, como o nome indica, os dados são armazenados recorrendo a relações entre tabelas, enquanto bases de dados não relacionais armazenam os dados de forma não estruturada.

Fazendo uma análise mais profunda, uma base de dados relacional é uma boa escolha para aplicações que envolvem a manipulação de várias transações. Apesar de não ter sido desenvolvida com a arquitetura *Object Oriented Programming* (OOP) em mente, é preciso fazer uma configuração das chaves primárias e estrangeiras para contornar o problema.

Em contrapartida, uma base de dados não relacional é usada em situações que as aplicações geram um grande tráfego de dados, devido à sua alta escalabilidade e a ter um desempenho superior em comparação ao as bases de dados relacionais.

Alguns exemplos de base de dados são:

- *My Structured Query Language* (MySQL) (Base de dados relacional) [28]
- *Maria Database* (MariaDB) (Base de dados não relacional) [29]

2.3.5 Cliente *web*

Relativamente a clientes *web* estes são ferramentas importantes de interação com servidores *web*. Por pedidos http ou https (métodos do tipo *GET*, *POST*, *PUT*, *DELETE*, etc) é possível obter a resposta ao pedido feito, guardá-la e até comparar a resposta com testes desenvolvidos para assegurar o funcionamento normal da aplicação e que as regras de negócio foram implementadas. Além disso, também é possível averiguar se houve regressão na qualidade do código por meio de testes de regressão, dado que a adição de uma nova funcionalidade pode causar problemas numa funcionalidade já existente.

Alguns exemplos de clientes *web* são:

- *Postman* [30]
- *Insomnia (Open source)* [31]

Comparando as duas aplicações, *Postman* aparenta ter um conjunto de funcionalidades mais maduras relativamente à aplicação *Insomnia*.

A seguir será enumerado as funcionalidades que fazem o *Postman* se destingir das outras ferramentas de teste:

- Documentação de API: Consegue gerar facilmente documentação da *API* [32]

- Execuções de coleções: Recorrendo a coleções é possível executar um grupo de pedidos como uma série, sendo esta funcionalidade bastante necessária para os testes automáticos [32]
- Monitorização: Periodicamente é possível executar uma coleção e verificar as suas respostas e desempenho [32]
- Testes em JavaScript: Permite desenvolver testes recorrendo a *JavaScript* [32]
- Servidores fictícios: Fornece servidores fictícios que permitem a simulação de cada um dos *endpoints*, sem que seja preciso desenvolver o próprio serviço [32]

Por outro lado, será enumerado as funcionalidades únicas ao *Insomnia*:

- Suporte a *plugins*: Permite o uso e a criação de novos *plugins* [32]
- Certificados de cliente e validação SSL: suporta a atribuição de certificados de cliente a espaços de trabalho e fornece opções de validação SSL [32]
- Geração de pedaços de código: Consegue gerar pedaços de código em 12 linguagens de programação diferentes [32]
- Visualização avançada de respostas: Permite aos utilizadores visualizar respostas para além dos formatos *JSON* e *XML*. Suporta vários tipos de conteúdo, como páginas *HTML*, imagens, *SVGs*, ficheiros de áudio e até documentos *PDF* [32]

2.3.6 Ferramenta de teste de desempenho

Apesar de ser importante testar a aplicação relativamente às regras de negócio, da mesma forma é importante testar a desempenho da mesma, utilizando aplicações de teste de desempenho que simulam a quantidade de utilizadores definida pelos requisitos não funcionais a usar uma certa funcionalidade ou várias funcionalidades de uma certa aplicação, ou aplicações em simultâneo. A mesma consegue apresentar relatórios, incluindo na mesma, gráficos dos pedidos feitos ao longo do tempo em que o, tempos máximos, mínimos e médios.

Exemplo de aplicações de teste de desempenho:

- JMeter [33]
- *Simple Object Access Protocol User Interface* (SOAPUI) [34]

O *SoapUI* e o *JMeter* têm objetivos e pontos fortes diferentes. No caso, do *SoapUI*, este é utilizado principalmente para testes funcionais de serviços Web e APIs, enquanto o *JMeter* é especializado em testes de desempenho, nomeadamente testes de carga. Por outro lado, o *SoapUI* oferece uma interface amigável com fácil

importação e o *JMeter* tem uma interface de utilizador baseada em formulários com características especificamente concebidas para APIs e aplicações Web. Em relação a relatórios, o *SoapUI* gera relatórios automaticamente e a sua versão paga oferece informações mais detalhadas e o *JMeter* não possui funcionalidades de relatório incorporadas, exigindo que os utilizadores realizem os relatórios manualmente [35].

Tanto o *SoapUI* como o *JMeter* têm limitações. No *SoapUI*, os testes de carga exigem um esforço significativo de programação para criar testes e simulações de cenários. O *SoapUI* também consome uma grande quantidade de memória ao gerar uma carga pesada, levando a problemas de desempenho. O *JMeter* não possui recursos de relatório integrados, essenciais para identificar gargalos de desempenho e otimizar o desempenho. A sua interface de utilizador também é considerada menos fácil de utilizar e pode ser difícil de trabalhar [35].

2.4 Padrões

2.4.1 *Database Per Service*

Imaginemos que está a ser desenvolvida uma loja de comércio online em que é recorrido à arquitetura de microsserviços. Cada serviço precisa de persistir as informações. Como, por exemplo, o serviço que disponibiliza funcionalidades para a gestão de encomendas grava informações acerca das encomendas enquanto outro serviço como, por exemplo, o serviço que disponibiliza funcionalidades para a gestão de produtos grava informações acerca dos produtos [36].

O problema que surge neste caso é o seguinte: Qual será a arquitetura da base de dados numa aplicação de microsserviços? [36].

Tendo em conta que:

- Os serviços devem ser desacoplados entre si, ou seja, ser desenvolvidos, implementados e dimensionados de forma independente [36]
- Pode ser necessário efetuar consulta de dados a outros serviços [36]
- Pode ser necessário unir dados pertencentes a outros serviços [36]
- As bases de dados podem ser replicadas [36]
- Cada serviço pode ter requisitos diferentes dos outros serviços [36]

A solução para o problema descrito em cima passa por cada serviço ter a responsabilidade de persistência dos próprios dados. Estes mesmos, passam a estarem privados e só podem ser acedidos através do serviço que os mantém, ou seja, os dados não podem ser acessados diretamente pelos os outros serviços [36].

Existem três opções para manter esses dados privados como, por exemplo:

- Tabelas privadas por serviço: Cada serviço possui um conjunto de tabelas que devem ser acessadas apenas por esse serviço [36]
- Esquema por serviço: Cada serviço possui um esquema de banco de dados privado para esse serviço [36]
- Servidor de base de dados por serviço: Cada serviço possui um servidor de banco de dados [36]

Para uma menor sobrecarga as melhores opções são as tabelas privadas por serviço e esquema por serviço, por outro lado, se houver a necessidade em que alguns serviços precisem de alto desempenho a melhor é o servidor de base de dados por serviço [36].

O uso de uma base de dados por serviço tem as seguintes vantagens:

- Os serviços tornam-se desacolados [36]
- Alterações na base de dados de um serviço não afetam os outros serviços [36]
- Cada serviço pode usar a base de dados mais adequada às necessidades do serviço [36]

Por sua vez tem as seguintes desvantagens:

- Dificuldade na junção de dados de várias bases de dados [36]
- Complexidade na gestão de várias bases de dados [36]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *API Composition*: Permite a junção de dados
- *CQRS*: Divide responsabilidades em serviços de consulta e serviços de escrita
- *Saga*: Implementa as transações entre serviços
- Arquitetura de microsserviços: Cria a necessidade para o uso deste padrão

2.4.2 *Saga*

Imaginemos que foi aplicado o padrão descrito anteriormente, ou seja, cada serviço tem a sua própria base de dados. Sendo que algumas transações são espalhadas pelos vários serviços é preciso um mecanismo para fazer a implementação dessas mesmas. Como, por exemplo, usando o paradigma anterior, o cliente tem alguma restrição na conta, quando este fizer algum pedido a aplicação deve garantir que a encomenda tenha em conta a restrição do cliente [37].

O problema que surge neste caso é o seguinte: Como será implementada as transações? [37]

A solução para o problema descrito em cima passa por implementar cada transação que abrange vários serviços como um *SAGA*, sequencia de transações locais. Cada transação local atualiza o banco de dados do serviço onde se encontra e publica o evento para acionar a próxima transação. No caso de alguma transação local violar as regras de negócio será executado transações que vão reverter as alterações feitas realizadas pelas transações anteriores [37].

O uso deste padrão tem as seguintes vantagens:

- Possível manter a consistência de dados entre serviços [37]

Por sua vez tem as seguintes desvantagens:

- Sistema mais complexo [37]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *Database Per Service*
- *Event sourcing*
- *Domain event*

2.4.3 *CQRS*

Imaginemos que foi implementado a arquitetura de microsserviços e *Database Per Service*, como resultado pode resultar em problemas de divergência, escabilidade e desempenho [38].

O problema que surge neste caso é o seguinte: Como será feita a implementação dos serviços? [38]

A solução para o problema descrito em cima passa por dividir um serviço, com todas as funcionalidades (*POST/GET/PUT/DELETE*) em dois serviços em que um dele terá a responsabilidade de fazer leituras e o outro por fazer escritas [38].

O uso deste padrão tem as seguintes vantagens:

- Separação de responsabilidades [38]
- Promove a possível de escabilidade dos serviços [38]

Por sua vez tem as seguintes desvantagens:

- Aumenta a complexidade do projeto [38]
- Duplicação de código [38]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *Database Per Service*
- *API Composition*

2.4.4 *API Composition*

Imaginemos que foi implementado a arquitetura de microsserviços e *Database Per Service*. Como resultado, a implementação de pesquisas para juntar dados de vários serviços torna-se mais difícil [39].

O problema que surge neste caso é o seguinte: Como implementar essas pesquisas numa arquitetura de microsserviços? [39]

A solução para o problema descrito em cima passa por implementar um *Composer API*, que invocará os serviços necessários e realizará a junção desses mesmos dados [39].

O uso deste padrão tem as seguintes vantagens:

- Torna-se mais simples consultar dados [39]

Por sua vez tem as seguintes desvantagens:

- Algumas consultas resultariam em junções ineficientes de grandes conjuntos de dados [39]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *Database Per Service*
- *CQRS*

2.4.5 *Messaging*

Imaginemos que foi implementado a arquitetura de microsserviços e que os serviços precisam de um protocolo de comunicação entre processos [40].

O problema que surge neste caso é o seguinte: Como os serviços comunicam-se entre si? [40]

A solução para o problema descrito em cima passa por usar mensagens assíncronas para a comunicação entre serviços. Existem vários estilos de comunicação assíncrona, tais como:

- *Request/response*: Um serviço envia uma mensagem a outro serviço e espera receber uma resposta imediatamente [40]
- *Notifications*: Um remetente envia uma mensagem a um destinatário, mas não espera uma resposta. Nem um é enviado [40]
- *Request/asynchronous response*: Um serviço envia uma mensagem de solicitação a outro serviço e espera receber uma resposta eventualmente [40]
- *Publish/subscribe*: Um serviço publica uma mensagem para zero ou mais destinatários (subscritores) [40]

- *Publish/asynchronous response*: Um serviço publica um pedido a um ou mais destinatários, alguns dos quais enviam uma resposta [40]

O uso deste padrão tem as seguintes vantagens:

- Desacopla o remetente da mensagem [40]
- Suporta variedade de padrões de comunicação [40]
- Aumenta a disponibilidade, dado que, o *message broker* faz de *buffer*[40]

Por sua vez tem as seguintes desvantagens:

- Aumenta a complexidade do sistema [40]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *SAGA*
- *CQRS*

2.4.6 *API Gateway*

Imaginemos que foi implementado a arquitetura de microsserviços no mesmo contexto de uma loja online e está a ser implementado uma certa pagina, por exemplo, uma página de um certo produto [41]. Para cada página dessas é preciso obter, por exemplo, as seguintes informações:

- Informações do produto, como nome, descrição, preço e fabricante [41]
- Número de compras já realizadas [41]
- Quantidade disponível em armazém [41]
- Comentários efetuados pelos clientes que compraram o produto [41]
- Data disponível para entrega do produto [41]
- Entre outras informações [41]

Para obter as várias informações e como a loja utiliza o padrão em cima referido, as informações estão espalhadas pelos diversos serviços [41]. Por isso seria necessário realizar pedidos aos seguintes serviços:

- Serviço das informações do produto
- Serviço de compras
- Serviço de inventário

- Serviço de comentários
- Serviço de encomendas

O problema que surge neste caso é o seguinte: Como os clientes acedem a cada um dos serviços numa aplicação baseada em microsserviços? [41]

Tendo em conta que:

- As informações diferem de cliente para cliente, a versão *desktop* é mais elaborada que a versão *mobile* [41]
- O desempenho da rede difere de cliente para cliente [41]
- As instâncias e o seu número mudam dinamicamente [41]
- A divisão da aplicação pelos serviços deve ser oculto para os clientes [41]
- Os serviços podem recorrer a um conjunto de diferentes protocolos [41]

A solução para o problema descrito em cima passa por implementar uma *API Gateway*, ou seja, um único ponto de acesso para todos os clientes [41]. Os pedidos recebidos serão encaminhados para o correspondente serviço. Conforme apresentado na figura 2.4.

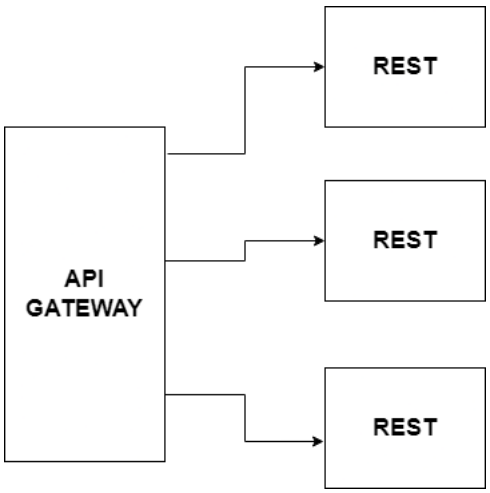

Figura 2.4: Exemplo de uma API Gateway.

Por outro lado, uma variação à estrutura apresentada na figura anterior é a seguinte, cada tipo de cliente (Web, mobile e aplicação externa de terceiros) pode ter a sua própria *API Gateway* [41]. Conforme apresentado na figura 2.5.

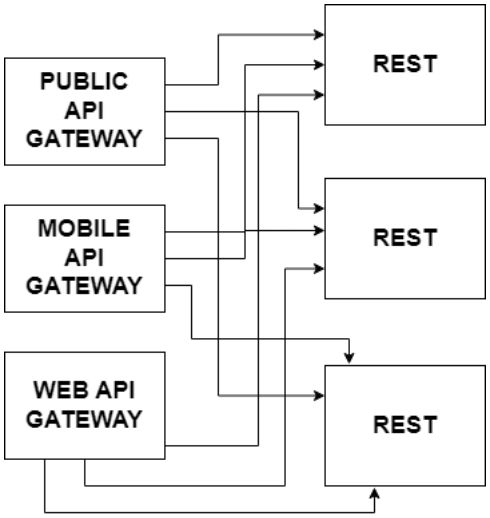

Figura 2.5: Exemplo de uma API Gateway.

O uso deste padrão tem as seguintes vantagens:

- Isola os clientes para a divisão dos microsserviços [41]
- Adequa a necessidade de cada cliente [41]
- Simplifica a chamada dos serviços [41]
- Pode reduzir o número de pedidos [41]
- O cliente não precisa de saber o endereço das instâncias dos serviços [41]

Por sua vez tem as seguintes desvantagens:

- Aumenta a complexidade do sistema [41]
- O tempo de resposta aumenta devido aos saltos adicionais [41]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *Access token*
- *API Composition*

2.4.7 *Access token*

Imaginemos que foi implementado a arquitetura de microsserviços e o padrão *API Gateway* [42].

O problema que surge neste caso é o seguinte: Como é comunicado a identidade do solicitador do pedido? [42]

A solução para o problema descrito em cima passa pela *API Gateway* autenticar o pedido e passar o token de acesso (por exemplo, *JWT*) que identifica a identidade do solicitador aos serviços [42].

O uso deste padrão tem as seguintes vantagens:

- Promove a segurança no sistema dado que assim a identidade do solicitador é conhecida [42]
- Os serviços aos saberem a identidade do solicitador podem dependendo das suas permissões, autorizar ou não autorizar a execução de uma operação/ação [42]

E finalmente, relativamente a padrões relacionados temos os seguintes:

- *API Gateway*

Capítulo 3

Ballerina

Neste capitulo será explorado a linguagem de programação *Ballerina*. Será feita uma introdução a mesma, enumerando as características distintivas, bem como é estruturado um projeto, como é construir, testado e implementado um serviço *web*.

3.1 Apresentação

Ballerina, é uma linguagem *open source*, projetada pela empresa *Web Services Oxygenated 2* (WSO2) e que teve o seu lançamento no ano de 2019, visa facilitar o desenvolvimento de aplicações modernas baseadas em nuvem, ou seja, serviços altamente escaláveis e orientados a conexão. Apresenta várias semelhanças, com as linguagens C, C++, Java e JavaScript.

Ballerina não é só uma linguagem de programação, mas sim um ecossistema, esta disponibiliza também um repositório no qual, este contém vários pacotes oficiais como pacotes feitos pela comunidade, que podem ser utilizados pelos desenvolvedores para facilitar a implementação de um certo problema, como, por exemplo, *drivers* de bases de dados, autenticação, criptografia, entre outros. Também é uma maneira dos desenvolvedores poderem, gerir versões, verificar alterações nos pacotes e resolver conflitos de dependências entre pacotes. A esse repositório é-lhe dado o nome de *Ballerina Central* [43].

3.2 Características

Sendo que, *Ballerina* recorre a um modelo de recorrência baseado em mensagens, cada serviço é executado simultaneamente, dado que, cada um deles é executado num processo.

Outra das suas caraterísticas é que esta é estaticamente tipada, o que significa que o tipo de uma variável é determinado em tempo de compilação e permanece o mesmo durante a execução do programa. Isso permite a capacidade de identificar problemas durante a compilação, o que pode reduzir o tempo e evitar que ocorram problemas durante a execução do programa. Por exemplo, se uma variável tiver sido definida como um inteiro e tentarmos atribuir-lhe uma *string*, o compilador indicará um erro durante a compilação e não durante a execução [44].

Em *Ballerina*, os dados são tratados como os outros tipos de dados funcionais como *strings* e inteiros, dado que, são-lhes concedidos a mesma importância, permitindo que os desenvolvedores os manipulem e criem sem grande esforço. Uma das formas de isso ser obtido é mediante tipos de dados compostos como registos e objetos, aliados aos recursos de manipulação dos mesmos, como mapeamentos e filtros, que ajudam a processar um conjunto de informações grandes de forma eficiente e eficaz.

Comparando com outras linguagens, tem uma particularidade, no qual, é possível por meio de fluxogramas visualizar e a compreender o fluxo de dados num programa. Por outro lado, também pode servir para se entender como é feita a comunicação entre serviços.

E por último, tem suporte nativo com os formatos *JSON* e *XML*. [45]

3.3 Estrutura de um projeto

Um projeto em *Ballerina* é organizado numa única unidade compartilhável denominada pacote. Por sua vez, um pacote pode ser constituído por vários módulos. Cada módulo é um conjunto de ficheiros do código-fonte, testes e recursos. É comum um projeto pequeno só ter um, mas à medida que o projeto vai ficando mais complexo nasce a necessidade de se criar mais, para melhor organizar o código e separar responsabilidades. Vale a pensa mencionar que cada ficheiro desta linguagem é acompanhado por uma extensão *.bal* [46]. Na figura 3.1 é demonstrado um exemplo simples daquilo que foi mencionado em cima.

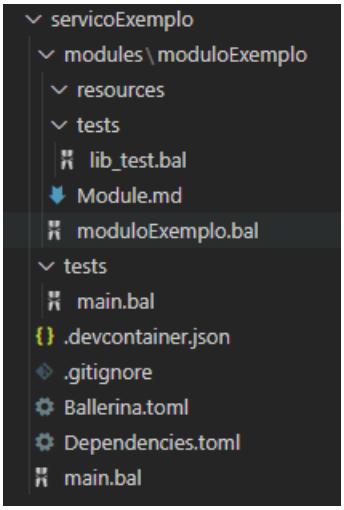

Figura 3.1: Exemplo da estrutura de um projeto.

3.4 Suporte num *Integrated Development Environment* (IDE)

O ambiente de desenvolvimento aconselhado para o desenvolvimento de código *Ballerina* é o *Visual Studio Code* (VSC), dado, a sua integração nativa com a linguagem. Adicionalmente, esta disponibiliza uma extensão que auxilia o programador no desenvolvimento de código, uma vez que, disponibiliza funcionalidades tais como, correção de erros, *noteboooks*, preenchimento automático de código e representação gráfica do código [47].

3.5 Implementar serviços na *cloud*

Feito o código necessário, este precisa de passar por um processo de implementação para passar a produção e começar a ser usado pelo utilizador final do sistema. Esse programa pode ser implementado na *cloud* por meio de contentores *docker* ou *kubernetes*, para gerir e escalar os serviços em nuvens públicas ou privadas. Este processo descrito anteriormente, pode ser automatizado e simplificado com os recurso disponibilizados da linguagem, facilitando desta forma, o processo de implementação num ambiente de produção e fazendo com que, os desenvolvedores se concentrem mais a desenvolver os serviços do que com a implementação desses mesmos [48].

3.6 Exemplos

3.6.1 Criação de um serviço

Na Listagem 3.1, o código descrito importa o módulo *HTTP*, que contém as funções para encarregar-se com pedidos *HTTP*. Posteriormente, é criado um serviço que ouve pedidos na porta 9090 e na qual contém uma função do tipo *GET* com o nome *greeting* e que retorna uma *string "Hello, World"*.

```
1 import ballerina/http;
2
3 service on new http:Listener(9090) {
4     resource function get greeting() returns string {
5         return "Hello, World!";
6     }
7 }
```

Listagem 3.1: Exemplo de um serviço. [49]

3.6.2 Conexão à base de dados

Na Listagem 3.2, o código descrito importa o módulo *MySQL* que contém funções que permitem o acesso a uma base de dados. Posteriormente é inicializado um cliente MySQL com as configurações necessárias, o anfitrião, o nome de utilizador, a palavra-passe e a porta. É importante mencionar que estes dados deveriam estar num ficheiro à parte e no qual seriam chamados. Para realizar isso seria preciso criar um ficheiro com o nome *Config.toml* [50].

Já na função *init()*, nesta é instanciado um novo cliente MySQL, cujo objetivo é criar uma base de dados através da função *execute()* com a respetiva *query*. Ao criar-se a base de dados, já é possível criar-se uma tabela, visto isso, é utilizada outra instância já declarada anteriormente para interagir com a base de dados e executar comandos SQL na mesma. A tabela criada tem duas colunas, *ingredient_id* (declarado como inteiro, é a chave primária e incrementa automaticamente) e *designation* (declarado como string).

```
1  import ballerinax/mysql;
2  import ballerinax/mysql.driver as _;
3  import ballerina/sql;
4
5  string USER = "myUser";
6  string PASSWORD = "myPassword";
7  string HOST = "localhost";
8  int PORT = 3306;
9
10 final mysql:Client dbClient;
11
12 function init() returns error? {
13     mysql:Client dbClientCreate = check new(host=HOST, user
           =USER, password=PASSWORD, port=PORT);
14     sql:ExecutionResult _ = check dbClientCreate->execute(`
           CREATE DATABASE IF NOT EXISTS Ingredient`);
15     check dbClientCreate.close();
16     dbClient = check new(host=HOST, user=USER, password=
           PASSWORD, port=PORT, database="Ingredient");
17     sql:ExecutionResult _ = check dbClient->execute(`CREATE
            TABLE IF NOT EXISTS Ingredient.Ingredients (
18                     ingredient_id INT AUTO_INCREMENT,
19                     designation VARCHAR(255),
20                     PRIMARY KEY (ingredient_id)
21                                         )`);
22 }
```

Listagem 3.2: Exemplo de uma coneção a base de dados.

3.6.3 Uso do *RabbitMQ*

Na Listagem 3.3, o código descrito importa o módulo *RabbitMQ* que contém funções que permitem realizar a comunicação entre serviços.

Na função *main()* é instanciado um novo cliente *RabbitMQ*, no qual, são definidas o *host* e porta de acesso ao servidor. E finalmente através da função *queueDeclare* é criada uma *queue* com o nome *"TestQueue"*.

```
1  import ballerinax/rabbitmq;
2
3  public function main() returns error? {
4      rabbitmq:Client client = check new (rabbitmq:
           DEFAULT_HOST, rabbitmq:DEFAULT_PORT);
5      check client->queueDeclare("TestQueue");
6  }
```

Listagem 3.3: Exemplo de criação de uma queue.

Na Listagem 3.4, como na listagem anterior, importa o módulo *RabbitMQ* e o módulo *HTTP*. O presente código é um exemplo de como combinar um serviço *web* com *RabbitMQ*. Começa-se por declarar um registo *"Test"*, que descreve um objeto, será este o objeto enviado na mensagem para o servidor. Depois disso é feita a declaração do serviço juntamente da porta em que ouvirá os pedidos. Dentro deste são instanciados um cliente *RabbitMQ* e descritas duas funções. A função *"init"* é usada para inicializar uma conexão com o servidor, e a função *"tests"* é responsável por receber o pedido do cliente *web* e publicá-lo na *queue "TestQueue"* criada anteriormente na listagem 3.3.

```
1  import ballerina/http;
2  import ballerinax/rabbitmq;
3
4  type Test readonly & record {
5      int id;
6      string name;
7  };
8
9  service / on new http:Listener(9092) {
10     private final rabbitmq:Client client;
11     function init() returns error? {
12         self.client = check new (rabbitmq:DEFAULT_HOST,
               rabbitmq:DEFAULT_PORT);
13     }
14     resource function post tests(Test test) returns http:
           Accepted|error {
15         check self.client->publishMessage({
16             content: test,
17             routingKey: "TestQueue"
18         });
19
20         return http:ACCEPTED;
21     }
22 }
```

Listagem 3.4: Exemplo de publicação de mensagem para o servidor.

Na Listagem 3.5, como na listagem anterior, importa o módulo *RabbitMQ* e o módulo *Log*. Começa-se por declarar um registo *"Test"*, que descreve um objeto, será este o objeto recebido na mensagem enviada pelo servidor.

Na função *main()* é declarado um novo cliente *RabbitMQ* com o *host* e porta de acesso ao servidor. A seguir, a mensagem enviada é consumida através da função *consumePayload()* da *queue "TestQueue"*. Depois do objeto *"Test"* ser validado, uma mensagem aparecerá na consola devido à função *"printInfo"* disponibilizada pelo módulo *Log*.

```
1  import ballerina/log;
2  import ballerinax/rabbitmq;
3
4  public type Test record {
5      int id;
6      string name;
7  };
8
9  public function main() returns error? {
10     rabbitmq:Client client = check new (rabbitmq:
           DEFAULT_HOST, rabbitmq:DEFAULT_PORT);
11     Test 'test = check client->consumePayload("TestQueue");
12     if 'test.isValid {
13         log:printInfo(string `Received valid test for ${'
               test.name}`);
14     }
15 }
```

Listagem 3.5: Exemplo da consumição de uma mensagem do servidor.

3.6.4 Concorrência

Na Listagem 3.6, o código descrito importa o módulo *io*, que contém módulos de interação com ficheiros e módulos de *output* e *input*. É definido um registo com o nome *"Person"* com dois campos, *"name"* e *"employed"* sendo os seus tipos, *string* e *boolean*, respetivamente. Também define uma função chamada *"process"* que recebe dois parâmetros, uma lista de objetos *"Person"* com o nome *"members"* e uma lista de inteiros com o nome *"quantities"*.

Na função descrita referida anteriormente, esta utiliza dois *workers* para processar os dados recebidos como parâmetros na função. No caso do *worker* com denominação "w1"é feito uma contagem dos membros empregado através do parâmetro *"employed"* de cada objeto *"Person"* na lista *"members"*, essa contagem é envida a seguir para o *worker* com denominação "w2"através do uso da ação (->) e conclui enviando um mensagem do tipo *string* para o *endpoint* da função com o valor contabilizado. Por outro lado, no w2, é inicialmente feito a soma de todos os valores recebidos no parâmetro *"quantities"* e fica a espera do resultado retornado pelo "w1", recorrendo a ação (<-), dado esse recebimento, é a seguir calculado a média para os membros empregados e conclui enviando um mensagem do tipo *string* para o *endpoint* da função com esse valor calculado.

Após a definição do "w1"e "w2", a função fica à espera dos valores de cada *worker* e imprime-as usando a função *io:println*.

```
import ballerina/io;

type Person record {|
    string name;
    boolean employed;
|};

function process(Person[] members, int[] quantities) {
    worker w1 {
        Person[] employedMembers = from Person p in members
            where p.employed
            select p;
        int count = employedMembers.length();
        count -> w2;
        string `Employed Members: ${count}` -> function;
    }
    worker w2 {
        int total = int:sum(...quantities);
        int employedCount = <- w1;
        int avg = employedCount == 0 ? 0 : total /
            employedCount;
        string `Average: ${avg}` -> function;
    }
    string x = <- w1;
    io:println(x);
    string y = <- w2;
    io:println(y);
}
```

Listagem 3.6: Exemplo de concorrência. [51]

3.6.5 Testes de código

Na Listagem 3.7, o código descrito importa o módulo *test*, que contém módulos que possibilitam a realização de testes ao código feito. Uma função chamada *"intAdd"* é definida com dois parâmetros, a e b, ambos inteiros, e retorna a soma dos valores inteiros. E outra função chamada *"intAddTest"* é definida com a anotação *"@test:Config"*, que especifica a configuração para a função de teste, para testar a função anteriormente construida. Através da função *"test:assertEquals"* é possível comparar o retorna da função com o resultado esperado.

```
import ballerina/test;

public function intAdd(int a, int b) returns (int) {
    return a + b;
}

@test:Config {}
function intAddTest() {
    test:assertEquals(intAdd(1, 3), 4);
}
```

Listagem 3.7: Exemplo de um teste desenvolvido. [52]

3.7 Alternativas à linguagem

As alternativas à linguagem em estudo para a construção de microsserviços são diversas. Algumas necessitam de recorrer a módulos externos para o desenvolvimento dos mesmos, enquanto outras através dos seus módulos nativos conseguem desenvolver estes sem necessitarem de importar módulos externos. Alguns exemplos são:

- *Java* recorrendo a *Spring Boot* [53]
- *Python* recorrendo a *Flask* [54] ou *Django* [55]
- *Node.js* recorrendo a *Express.js* [56] ou *Fastify.js* [57]
- *Rust* recorrendo a *Actix-web* [58]
- *C#* tem módulos nativos para o desenvolvimento de microsserviços [59]
- *Go* recorrendo *Fiber* [60]

Capítulo 4

Análise

Neste capitulo será descrito o cenário requisitado, os requisitos funcionais, as restrições, atributos de qualidade e casos de uso. Também serão apresentados tabelas e diagramas que representam a arquitetura do protótipo.

4.1 Descrição do cenário

Um grupo de antigos alunos, alguns com experiência em Engenharia Informática, decidiram criar uma empresa dedicada à comercialização de sanduíches saudáveis numa loja física. Para isso é preciso criar uma solução informática que suporte as necessidades da mesma. É pretendido um sistema que tenha as capacidades necessárias para a gestão das várias dimensões da empresa. No desenvolvimento deste protótipo deve ser considerada uma arquitetura de microsserviços, bases de dados única para cada microsserviço e que esses mesmos devem ser chamados por meio de uma *API Gateway*. A aplicação deve ser constituída por:

- Gestão de sanduíches
- Gestão de ingredientes
- Gestão de críticas
- Gestão de lojas
- Gestão de encomendas

- Gestão de clientes

No sistema, cada sanduíche é identificada por uma designação, preço de venda e uma lista de ingredientes. Além disso, pode ter várias descrições, mas para idiomas autorizados, detetados pela própria aplicação. Qualquer sanduíche deve ser criada com o mínimo de 3 ingredientes, sendo obrigatório, conter um tipo da categoria de pão. Os ingredientes na lista devem ser verificados quanto a sua existência.

Um ingrediente possui uma designação e uma categoria, este é usado na formação da sanduíche.

Ao cliente é permitido criar uma crítica, onde é descrito a sua experiência após efetuar um pedido de sanduíche recorrendo a um comentário e uma escala de 0 a 5 estrelas. Deve ser feito uma filtragem inicial do conteúdo do comentário. Outros clientes podem indicar se gostaram ou não dessa crítica, mas também podem efetuar uma denúncia no caso de acharem que aquela crítica contém conteúdo ofensivas, entre outros. Posteriormente será revista por um administrador. Deve ser acompanhada da hora de criação.

Uma loja inclui uma designação da mesma, endereço e a pessoa responsável. Uma loja tem um horário de funcionamento que pode variar de dia para dia. A pessoa responsável possui um nome e um endereço eletrónico. O endereço da loja é constituído pela morada, número da porta, código postal, localidade e país.

Cada encomenda é composta pelo identificador do cliente que a efetuou, estado da encomenda, por uma lista de sanduíches e respetivas quantidades, bem como dia de entrega e a loja especifica de entrega, além disso, o protótipo deve informar o preço total da encomenda, deve-se ter em consideração que uma sanduíche não pode ser vendida abaixo de zero, apesar das promoções aplicadas.

Cada cliente inclui os seus detalhes pessoais, como nome, número de identificação fiscal, endereço, endereço eletrónico, dados de autenticação e idiomas que saiba falar. O endereço do cliente é constituído pela morada, número da porta, código postal, localidade e país.

As funcionalidades do administrador do sistema incluem criar, alterar e eliminar sanduíches, ingredientes e lojas, adicionar descrições, adicionar ingredientes às sanduíches e consultar informações dos clientes. A pessoa responsável pela loja tem como funcionalidades consultar informações dos clientes, verificar encomendas e alterar o estado das mesmas. Já as funcionalidades do cliente incluem visualizar informações sobre os seus dados de cliente, sanduíches, lojas, ingredientes e as suas encomendas.

O sistema deve responder em menos de 3 segundos a 10 utilizadores em simultâneo, conter um sistema de autenticação e autorização, ser modificável no que diz respeito a introdução de novas funcionalidades ou alteração de algumas já existentes, devem ser adotados testes nas várias dimensões, isto inclui as regras de negócio capturadas e o correto funcionamento da aplicação, ser implementado um sistema de

mensagens via um *message broker* e por último só é permitido o uso de tecnologias e ferramentas open source.

Dado que a empresa abrirá em breve, cerca de 3 meses, o *software* deve estar disponível até a data de abertura.

4.2 Modelo de Domínio

Um modelo de domínio é usado para representar conceitos, objetos e relações com outros objetos que sejam importantes num determinado domínio de negócios. O modelo correspondente ao presente sistema pode ser visto na figura 4.1.

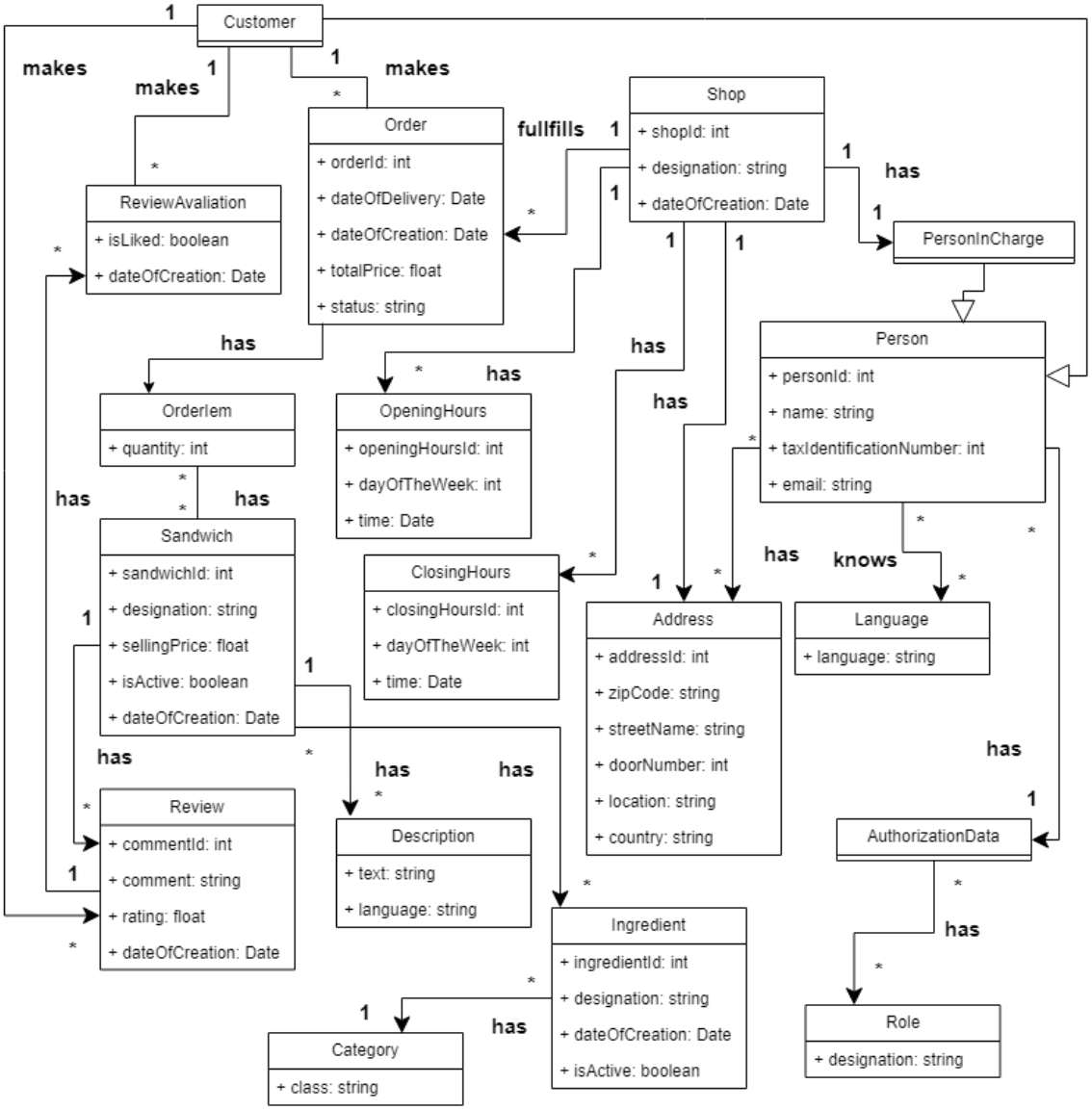

Figura 4.1: Modelo de Domínio

4.3 Requisitos Funcionais

Na tabela 4.1 são apresentados os requisitos funcionais da aplicação, apresentam as caraterísticas e funcionalidades do sistema pretendido.

Tabela 4.1: Requisitos Funcionais

Requisitos Funcionais		
Referência	**Requisito Funcional**	**Descrição**
RF1	Gestão de sanduíches	O sistema disponibiliza funcionalidades para a gestão de sanduíches
RF2	Gestão de ingredientes	O sistema disponibiliza funcionalidades para a gestão de ingredientes
RF3	Gestão de críticas	O sistema disponibiliza funcionalidades para a gestão de críticas
RF4	Gestão de lojas	O sistema disponibiliza funcionalidades para a gestão de lojas
RF5	Gestão de encomendas	O sistema disponibiliza funcionalidades para a gestão de encomendas
RF6	Gestão de clientes	O sistema disponibiliza funcionalidades para a gestão de clientes

4.4 Restrições

Na tabela 4.2 são apresentadas as restrições que afetam o desenvolvimento do projeto de *software*.

Tabela 4.2: Restrições

Restrições		
Referência	**Restrição**	**Descrição**
R1	Linguagem	O protótipo deve ser escrito recorrendo a Ballerina
R2	Arquitetura de microsserviços	A solução deve utilizar a arquitetura de microsserviços
R3	Aplicações e ferramentas open source	Só são permitidas aplicações e ferramentas open source no desenvolvimento do sistema
R4	Tempo	A aplicação deve estar concluída em três meses

4.5 Atributos de Qualidade

Na tabela 4.3 são apresentadas os atributos de qualidade que o sistema deve apresentar recorrendo à norma ISO 25010, que define a qualidade do *software* como um conjunto de características que o mesmo deve possuir para atender às necessidades pretendidas. Esta norma é representada por 8 categorias:

- Adequação funcional
- Eficiência de desempenho
- Compatibilidade
- Usabilidade
- Confiabilidade
- Segurança
- Manutenibilidade
- Portabilidade

[61]

Tabela 4.3: Atributos de Qualidade

Atributos de Qualidade		
Referência	**Atributo**	**Descrição**
Q1	Segurança	Mecanismos de autenticação e autorização devem ser implementados
Q2	Desempenho	A aplicação deve responder as pedidos realizados por 10 utilizadores em simultâneo em menos de 3 segundos
Q3	Manutenibilidade	Pretende-se que a aplicação possa evoluir, pelo que, deverão ser aplicados padrões de *design* apropriados

Adicionalmente, relativamente ao atributo de qualidade com referência Q1, deve ser mencionado que deverão ser seguidos as dez principais práticas para o desenvolvimento de código seguro. As dez principais práticas são constituídas por:

- Validação de todas as entradas de fontes de dados não seguras
- Ter atenção aos avisos do compilador e recorrer a ferramentas de análise estática e dinâmica para identificar e eliminar falhas de segurança
- Desenhar um sistema o mais simples e pequeno possível
- O acesso deve ser baseado em permissões em vez de exclusões
- Cada processo deve ter o menor conjunto de privilégios necessários para concluir o trabalho
- Os dados enviados para um sistema complexos devem passar por um processo de limpeza
- Deve ser feito uma gestão de risco com múltiplas estratégias defensivas para prevenir falhas de segurança e limitar as consequências de um possível *exploit*.
- Deve ser incorporado testes de qualidade do código-fonte, testes de penetração e testes de *fuzz*
- Desenvolver um padrão de codificação seguro para a linguagem e plataforma de desenvolvimento utilizada

[62]

4.6 Casos de Uso

Os casos de uso são uma ferramenta valiosa na engenharia de *software* para documentar e compreender os requisitos funcionais de um sistema. Por outro lado, os diagramas de caso de uso são as representações gráficas dos casos de uso, estes detalham as interações do sistema com todos os atores do mesmo. Podem ser definidas por meio de *Unified Modeling Language* (UML), uma linguagem de modelagem que permite representar um sistema padronizada que pode ser utilizada na visualização, construção e na documentação de artefactos.

A seguir, serão apresentados os casos de uso por área, recorrendo a uma tabela que contém a referência do caso de uso e o diagrama que associa os mesmos aos respetivos atores, o agrupamento destes foi feito tendo em conta os vários elementos apresentados no cenário apresentado.

4.6.1 Sanduíches

Na tabela 4.4 são apresentados os casos de usos capturados para a gestão de sanduíches. Está acompanhada pelo diagrama 4.2, diagrama de casos de usos para a administração de sanduíches.

Tabela 4.4: Sanduíches

Casos de Uso	
Referencia	**Descrição**
UC05	Criar uma sanduíche
UC06	Obter uma sanduíche por ID
UC07	Obter todas as sanduíches
UC09	Adicionar descrição(ões) a uma sanduíche
UC10	Obter todas as sanduíches que não possuam um ingrediente específico
UC30	Alterar estado de um sanduíche
UC34	Alterar os dados de uma sanduíche

Figura 4.2: Sanduíches

4.6.2 Ingredientes

Na tabela 4.5 são apresentados os casos de usos capturados para a gestão de ingredientes. Está acompanhada pelo diagrama 4.3, diagrama de casos de usos para a administração de ingredientes.

Tabela 4.5: Ingredientes

Casos de Uso	
Referencia	**Descrição**
UC01	Criar um ingrediente
UC02	Obter um ingrediente por ID
UC03	Obter um ingrediente por designação
UC04	Obter todos os ingredientes
UC29	Alterar estado de um ingrediente
UC35	Obter todos os ingredientes de uma certa categoria

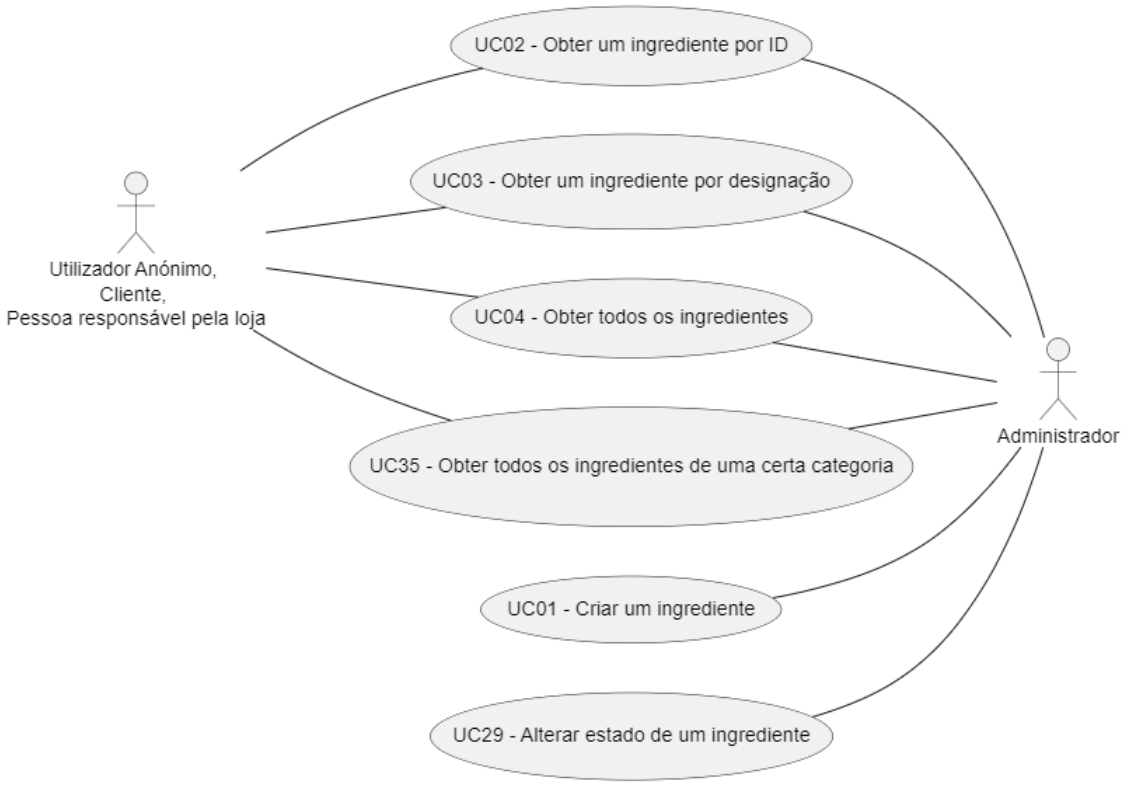

Figura 4.3: Ingredientes

4.6.3 Críticas

Na tabela 4.6 são apresentados os casos de usos capturados para a gestão de críticas. Está acompanhada pelo diagrama 4.4, diagrama de casos de usos para a administração de críticas.

Tabela 4.6: Críticas

Casos de Uso	
Referencia	**Descrição**
UC19	Adicionar crítica a uma sanduíche
UC20	Votar numa crítica de uma sanduíche
UC21	Listar todas as críticas de uma sanduíche
UC32	Listar todas as críticas denunciadas
UC33	Eliminar uma crítica denunciada por ID
UC36	Denunciar uma crítica
UC43	Editar uma crítica feita
UC44	Listar todas as críticas feitas pelo cliente
UC45	Eliminar critica feita pelo cliente

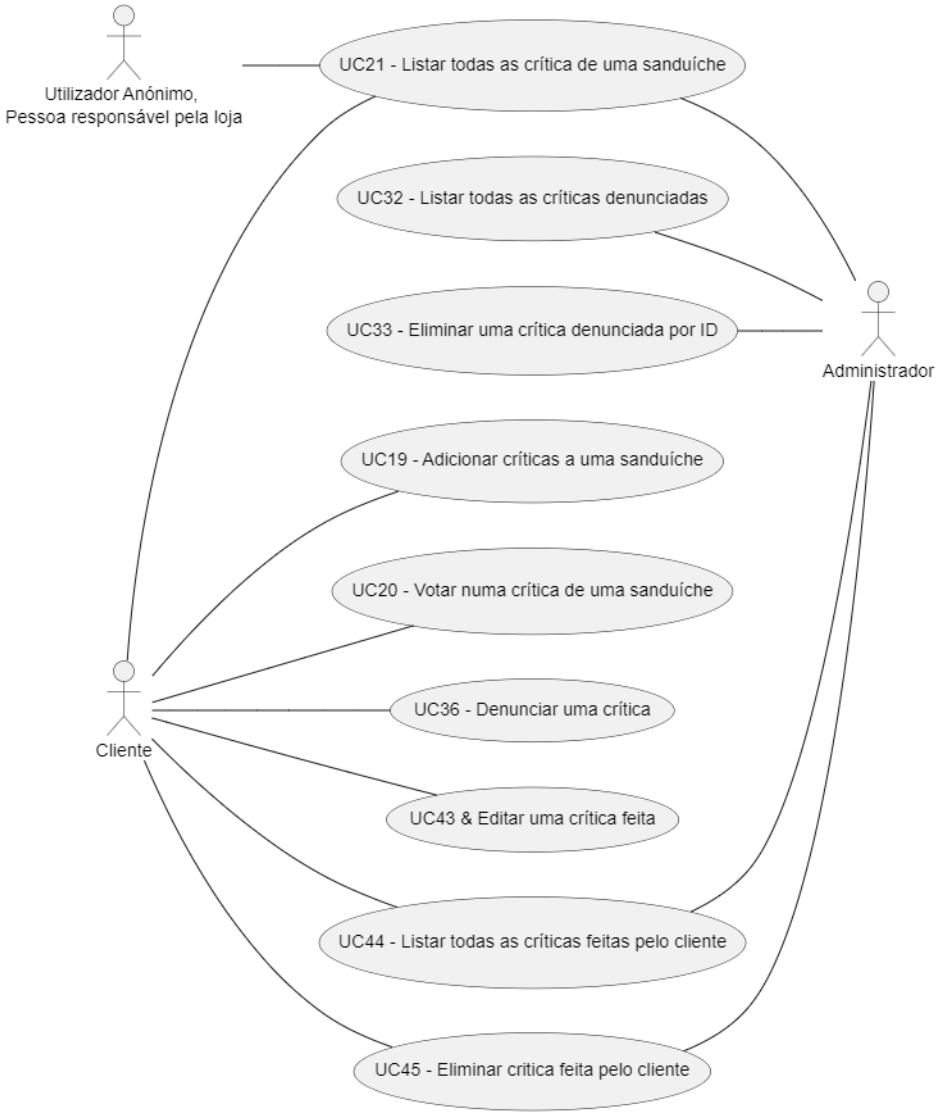

Figura 4.4: Críticas

4.6.4 Lojas

Na tabela 4.7 são apresentados os casos de usos capturados para a gestão de lojas. Está acompanhada pelo diagrama 4.5, diagrama de casos de usos para a administração de lojas.

Tabela 4.7: Lojas

Casos de Uso	
Referencia	**Descrição**
UC22	Criar loja
UC23	Listar todas as lojas
UC24	Listar loja por ID
UC25	Listar loja por designação
UC26	Listar loja por endereço
UC31	Eliminar lojas
UC42	Editar os dados de uma loja

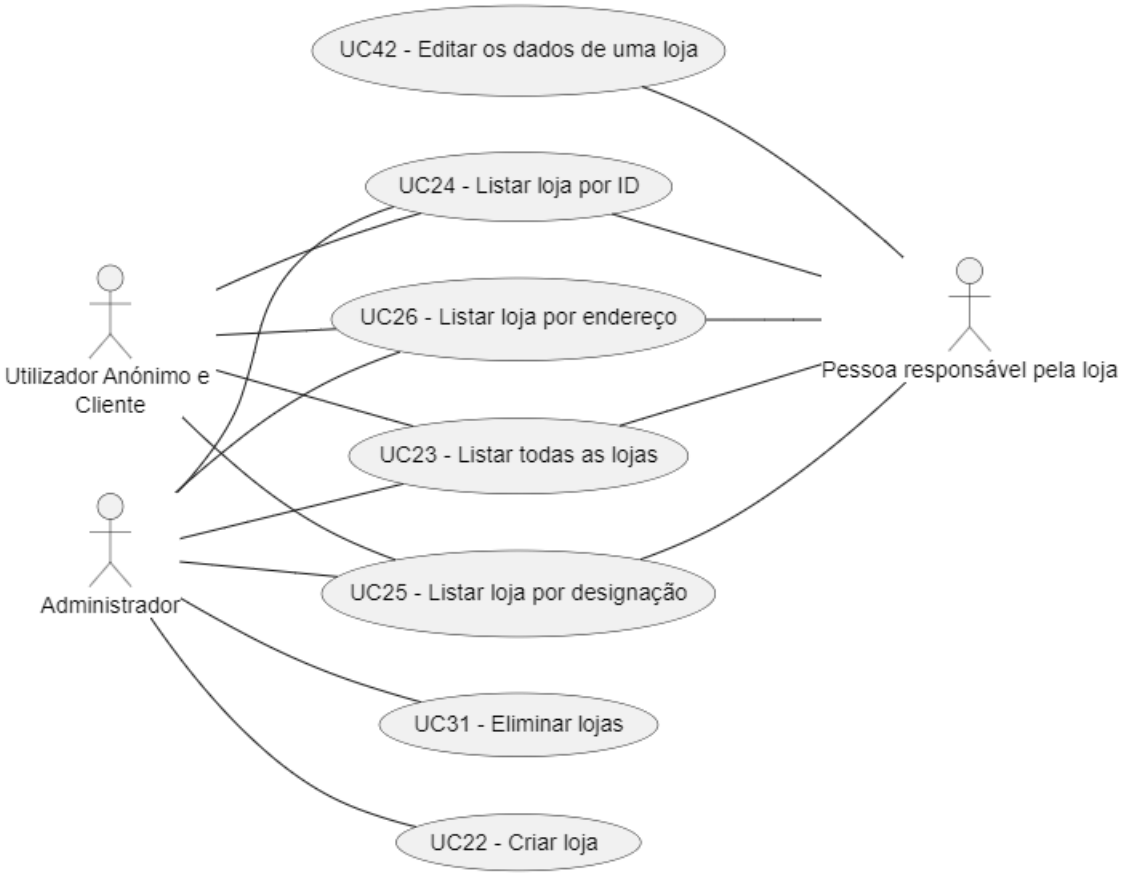

Figura 4.5: Lojas

4.6.5 Encomendas

Na tabela 4.8 são apresentados os casos de usos capturados para a gestão de encomendas. Está acompanhada pelo diagrama 4.6, diagrama de casos de usos para a administração de encomendas.

Tabela 4.8: Encomendas

Casos de Uso	
Referencia	**Descrição**
UC27	Criar uma encomenda
UC28	Listar encomendas de uma loja
UC37	Alterar a estado da encomenda
UC39	Listar as encomendas do cliente

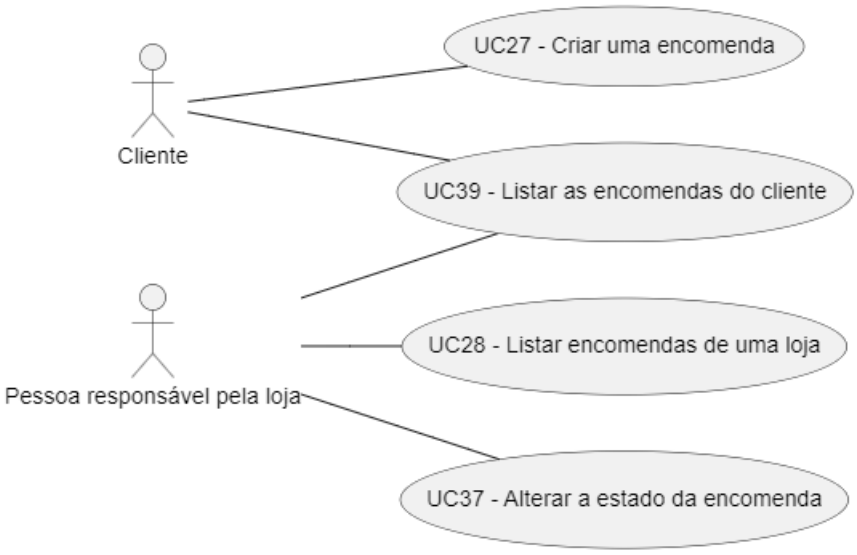

Figura 4.6: Encomendas

4.6.6 Clientes

Na tabela 4.9 são apresentados os casos de usos capturados para a gestão de clientes. Está acompanhada pelo diagrama 4.7, diagrama de casos de usos para a administração de clientes.

Tabela 4.9: Sanduíches

Casos de Uso	
Referencia	**Descrição**
UC11	Adicionar cliente
UC12	Obter cliente por ID
UC13	Obter cliente por e-mail
UC14	Obter cliente por número de identificação fiscal
UC15	Obter dados de autenticação do cliente
UC16	Obter todos os clientes
UC17	Adicionar permissões a um cliente
UC18	Fazer *login*
UC40	Alterar os dados do cliente
UC41	Visualizar os dados do cliente

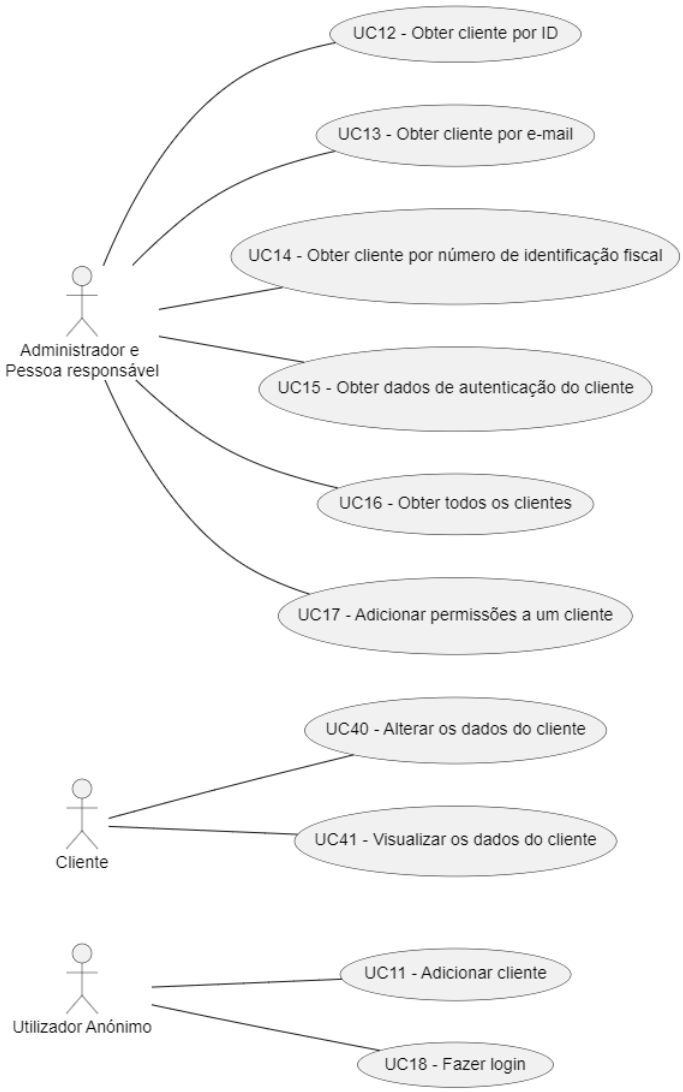

Figura 4.7: Clientes

Capítulo 5

Design

Neste capitulo será apresentado e descrito o *design* a ser implementado, mencionando também os padrões e princípios que serão usados e recorrendo as várias vistas para representar o sistema graficamente.

5.1 Padrões e princípios a serem utilizados

Para garantir a implementação do sistema como sucesso é fundamental existir a definição de princípios e padrões em antemão à construção do *software* como, por exemplo:

- Uso de um *message broker* : Deve ser utilizado um *message broker* para comunicação entre microsserviços
- Base de dados por microsserviço: Cada microsserviço deve ter a sua própria base de dados
- Escalabilidade horizontal: Projetar o sistema para ser escalável
- *Design* orientado a eventos: Deve ser implementado um *design* orientado a eventos
- Testes unitários, integração, desempenho e carga: O sistema deve ser testado recorrendo a esses testes

- Versões: O uso de *software* de versões é importante, dado as mudanças que pode haver no sistema e caso haja algum impacto negativo por parte de uma nova atualização pode-se sempre voltar para a versão anterior que funcionava sem problemas
- Garantia de consistência de dados: Deve ser feita a garantia da consistência de dados
- Modularidade e reuso: O sistema deve ser projetado de uma forma modular permitindo o reuso de componentes
- Padronização de estruturas de dados: Deve ser feita uma definição dos padrões das estruturas de dados compartilhados entre serviços
- Documentação clara e atualizada: Para serviço deve ser mantida documentação atualizada e clara
- Separação de Responsabilidades: Cada serviço deve ser responsável pelos seus próprios dados
- Adoção de objeto de transferência de dados: Devem ser adotados objetos de transferência de dados para o recebimento e envio de dados entre serviços
- Uso de *Domain-Driven Design* (DDD): Deve ser utilizado o conceito de *DDD* na construção do sistema
- Uso de *API Gateway*: A *API Gateway* deve chamar os vários serviços, aumentando desta forma, a segurança do próprio sistema com a implementação de mais uma camada, com esta também é possível aglomerar as informações de todos os serviços num só

5.2 Conceção arquitetural

O modelo *"4+1"* é um padrão de *software* que fornece várias perspetivas relativamente ao sistema proposto, sendo que, cada uma foca-se em diferentes aspetos da arquitetura a ser implementada. Este é divido por quatro principais vistas (vista lógica, vista processo, vista de implementação, vista física) e uma adicional (vista de cenários).

- Vista lógica: Descreve os componentes do *software*, ou seja, a estrutura interna do sistema
- Vista processo: Decompõe o sistema em vários processos e serviços, além disso, descreve como estes se comunicam entre si

- Vista de implementação: Descreve como o sistema está organizado nas várias camadas e hierarquias
- Vista física: Descreve a distribuição física do sistema e as suas conexões físicas
- Vista de cenários: Descreve os vários elementos que pretendem atender as regras de negócio e as várias funcionalidades que o sistema disponibiliza

Na figura 5.1 é apresentada as 5 vistas que o modelo propõem, as referidas anteriormente.

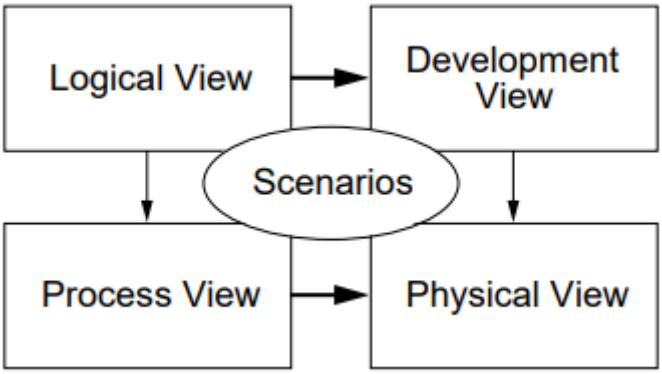

Figura 5.1: Modelo *"4+1"* [63]

5.2.1 Vista lógica

Nas figuras 5.2 e 5.3 pode-se observar a decomposição do sistema nos vários serviços. O mesmo é divido em seis serviços responsáveis pela logística e um serviço responsável pela deteção de linguagem.

A primeira figura representa um sistema mais primordial, onde a comunicação entre serviços é efetuada por pedidos *HTTP*, o padrão de base de dados por serviço e a *API Gateway* são implementados.

Diagrama da vista lógica

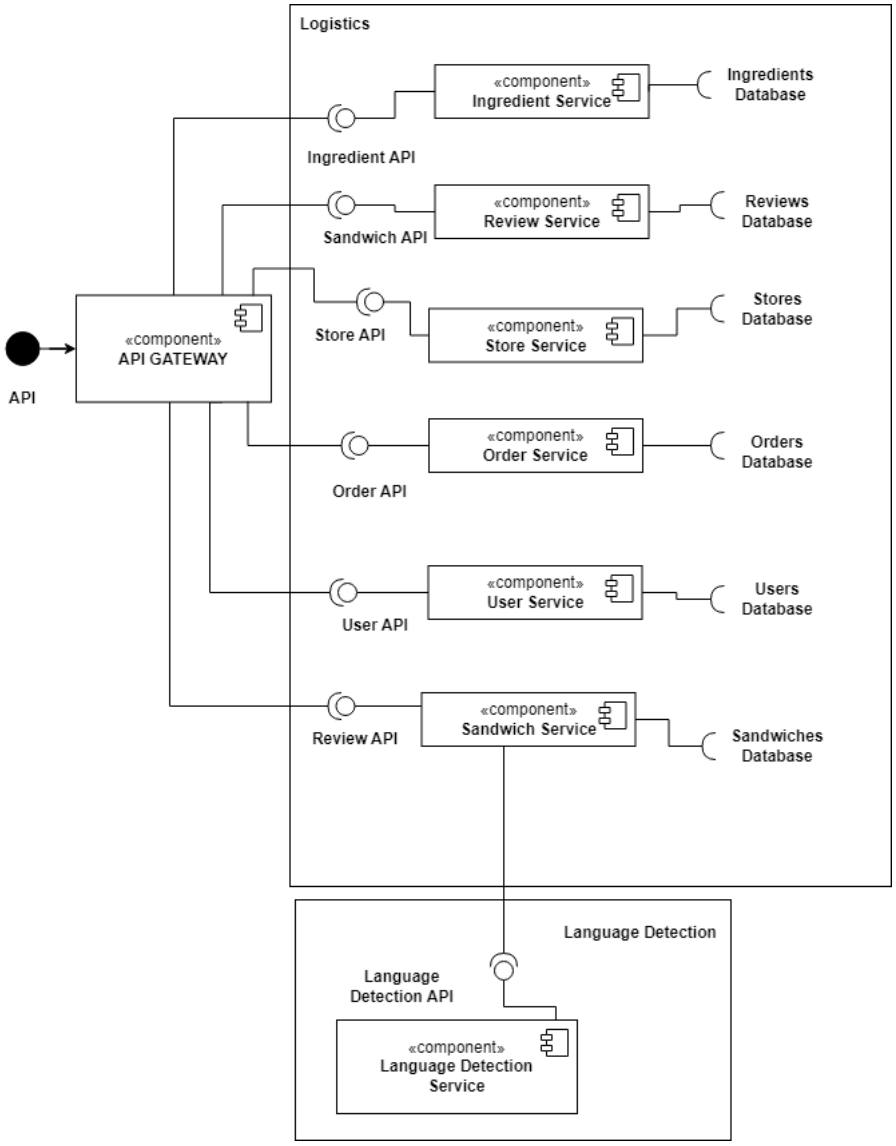

Figura 5.2: Diagrama da vista lógica

Em comparação à figura 5.2, a figura 5.3, apresenta uma adição na implementação, nesta foi adicionada o *Message Broker*, responsável pelo envio e recebimento de mensagens entre serviços, a troca de destes é realizado através do uso do padrão de *Publish/Subscriber*. O uso deste torna-se importante devido à ineficiência dos pedidos *HTTP* no que diz respeito a desempenho e, por outro lado, torna o processo de escalabilidade horizontal mais simples e eficiente.

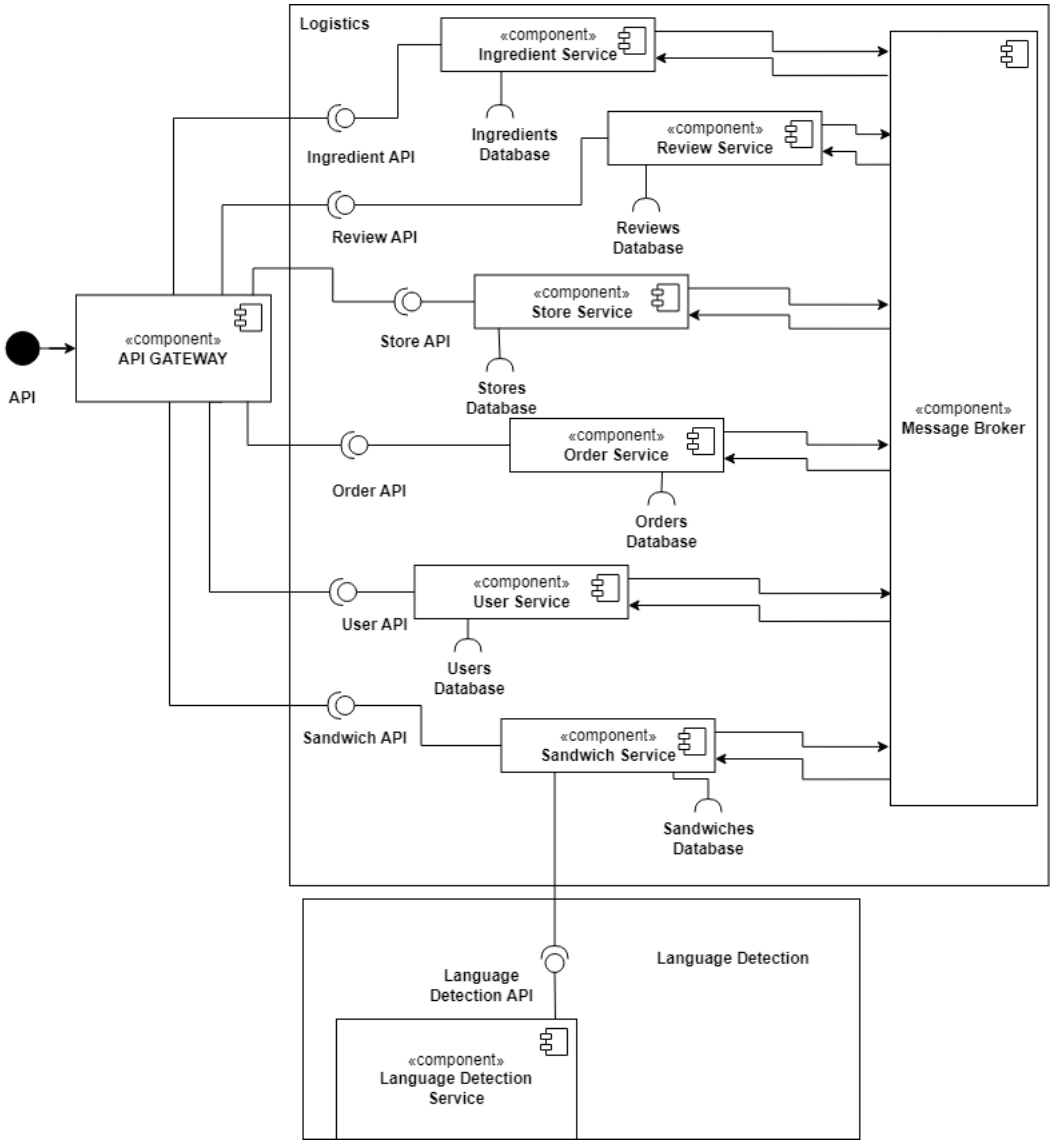

Figura 5.3: Diagrama da vista lógica

Diagrama de Estados

Um diagrama de estados representa os vários estados ou comportamento, que um certo objeto pode ter ao longo do tempo no sistema. O diagrama a seguir, figura 5.4, apresenta os possíveis estados de uma encomenda. Começa com um estado inicial, onde a encomenda é criada e passa para o estado "Criada". O cliente deverá decidir se contínua com a encomenda ou a cancela até a data da mesma. No caso de a cancelar passa para o estado "Cancelada", por outro lado, caso o cliente pretenda continuar com a encomenda, então a encomenda é preparada para o processamento, passando para o estado "Em processo". Após preparada, a encomenda é dada como processada e por isso passa para o estado "Processada". A partir deste estado, a encomenda está pronta para ser entregue e passa para o estado "Entregue".

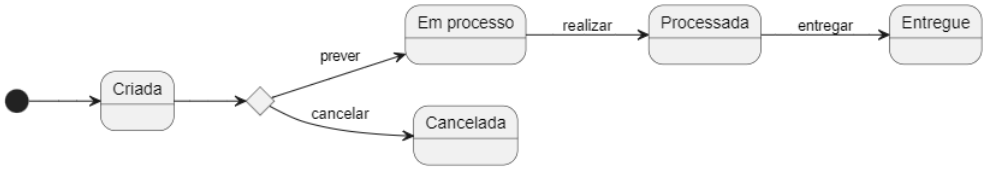

Figura 5.4: Diagrama de estados para uma encomenda

Outro diagrama necessário, figura 5.5, é o diagrama de estados para uma crítica, neste é apresentado os possíveis estados para a mesma. O diagrama começa com a criação da crítica, estado “Criada”. A partir deste estado, o cliente pode realizar a denúncia a uma crítica, levando a um novo estado chamado “Denunciada”. Neste estado, o administrador verificará o conteúdo da mesma, dessa situação pode resultar dois resultados, se a crítica contiver palavras ofensivas é eliminada, o que leva ao estado “Eliminada”, por outro lado, se não contiver nada ofensivo voltará ao estado “Criada”.

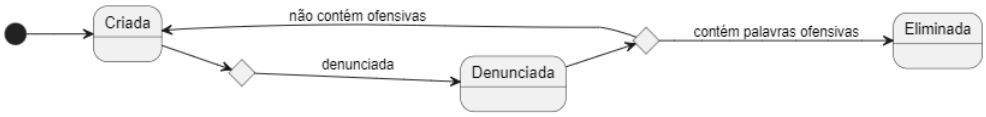

Figura 5.5: Diagrama de estados para uma crítica

O diagrama começa com a criação da crítica, estado “Criada”. A partir deste estado, o cliente pode realizar a denúncia a uma crítica, levando a um novo estado chamado “Denunciada”. Neste estado, o administrador verificará o conteúdo da mesma, dessa situação pode resultar dois resultados, se a crítica contiver palavras ofensivas é eliminada, o que leva ao estado “Eliminada”, por outro lado, se não contiver nada ofensivo voltará ao estado “Criada”.

Por último, figura 5.6, neste diagrama é apresentado os estados que uma sanduíche e um ingrediente podem tomar. O diagrama começa com um estado inicial, estado “Ativo”. A partir deste estado, é possível desativar o ingrediente ou a sanduíche, levando-o ao estado “Inativo”.

A partir do estado "Inativo", é possível reativar o ingrediente ou a sanduíche, levando-o de volta ao estado “Ativo".

Figura 5.6: Diagrama de estados para uma sanduíche e um ingrediente

5.2.2 Vista de implementação

As figuras 5.7 e 5.8 representam as vistas de implementação nível 1 e nível 2, respetivamente. Com estas figuras é possível perceber-se a relação entre componentes e a estrutura do sistema, no que diz respeito, no contexto da implementação.

Relativamente à figura 5.7 é possível observar-se as dependências entre componentes ao nível mais alto, ao nível dos serviços.

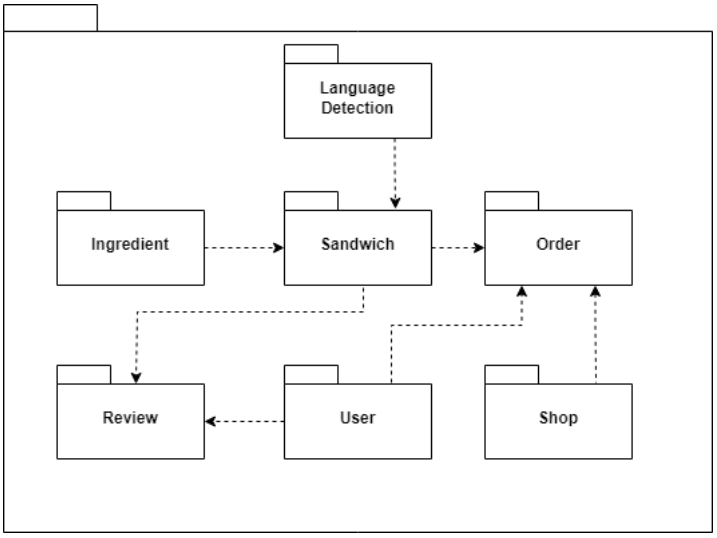

Figura 5.7: Diagrama da vista de implementação nível 1

Agora na figura 5.8, a um nível mais baixo do que na figura anterior, é possível testemunhar a estruturação de um serviço e as suas dependências entre os elementos do próprio serviço. Todos os serviços seguem esta tipologia. Cada serviço é constituído por quatro divisões:

- *Controllller*: Ponto de entrada para o serviço e responsável pelo encaminhamento dos pedidos para o *Service*
- *Service*: Responsável por toda a lógica do serviço
- *Repository*: Responsável pela conexão à base de dados e respetivo manuseamento
- *Model*: Responsável por definir os objetos

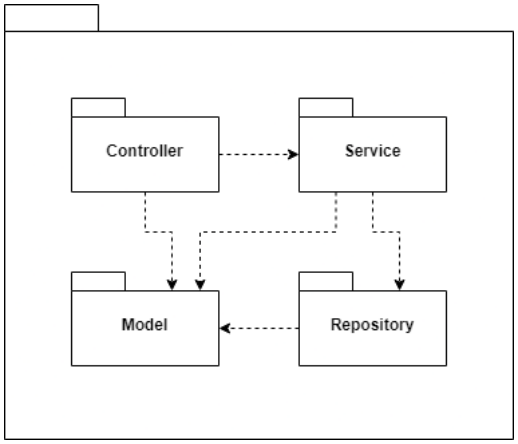

Figura 5.8: Diagrama da vista de implementação nível 2

5.2.3 Vista de processo

As seguintes figuras, representam as vistas de processo nível 1 e nível 2, respetivamente. Com estas figuras é possível observar os passos necessários para a conclusão do processo.

A figura 5.9, representa o processo de criação de um ingrediente ao mais alto nível. O diagrama começa por representar o pedido feito, que será efetuado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço "Ingredient". Este por sua vez, retornará o ingrediente criado e o código que representa a criação sucedida desse mesmo ingrediente, no caso de haver conflito na autenticação é retornado só o código *HTTP* 403.

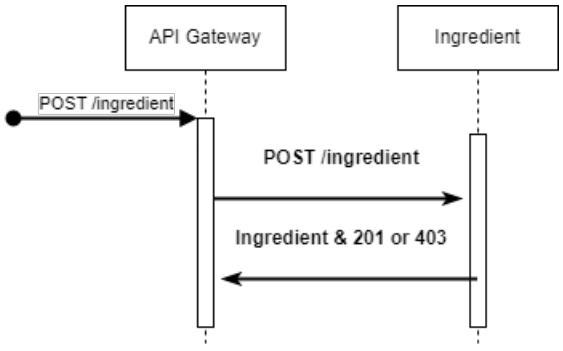

Figura 5.9: Diagrama da vista de processo do método *POST* nível 1

A figura 5.10, representa o processo de criação de um ingrediente a um nível mais baixo do que a anterior. O diagrama começa por representar o pedido feito, que será realizado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço *Ingredient*. Sendo o *IngredientController*, o ponto de partida para o serviço mencionado anteriormente, este é responsável pelo encaminhamento do pedido recebido. A seguir, este chamará

a função *createIngredient*, que enviará o objeto introduzido no pedido para o *IngredientService* onde será processado. É neste momento, que será feita a verificação do utilizador, no caso de não lhe ser permitido o acesso, será enviado o código *HTTP* 403, por outro lado, caso se verifique que tem as permissões necessárias, o mesmo objeto mencionado anteriormente, será enviado para o *IngredientRepository*, onde será feito o registo na base de dados criada para o efeito. Já o objeto criado na base de dados é retornado um objeto contendo todos os parâmetros do ingrediente, que terá de passar por um processo de transformação, ou seja, é aplicado o padrão de *software DTO*. Com este *DTO* criado será feita uma notificação ao *Message Broker* da sua criação e por último será retornado com o código *HTTP* 201.

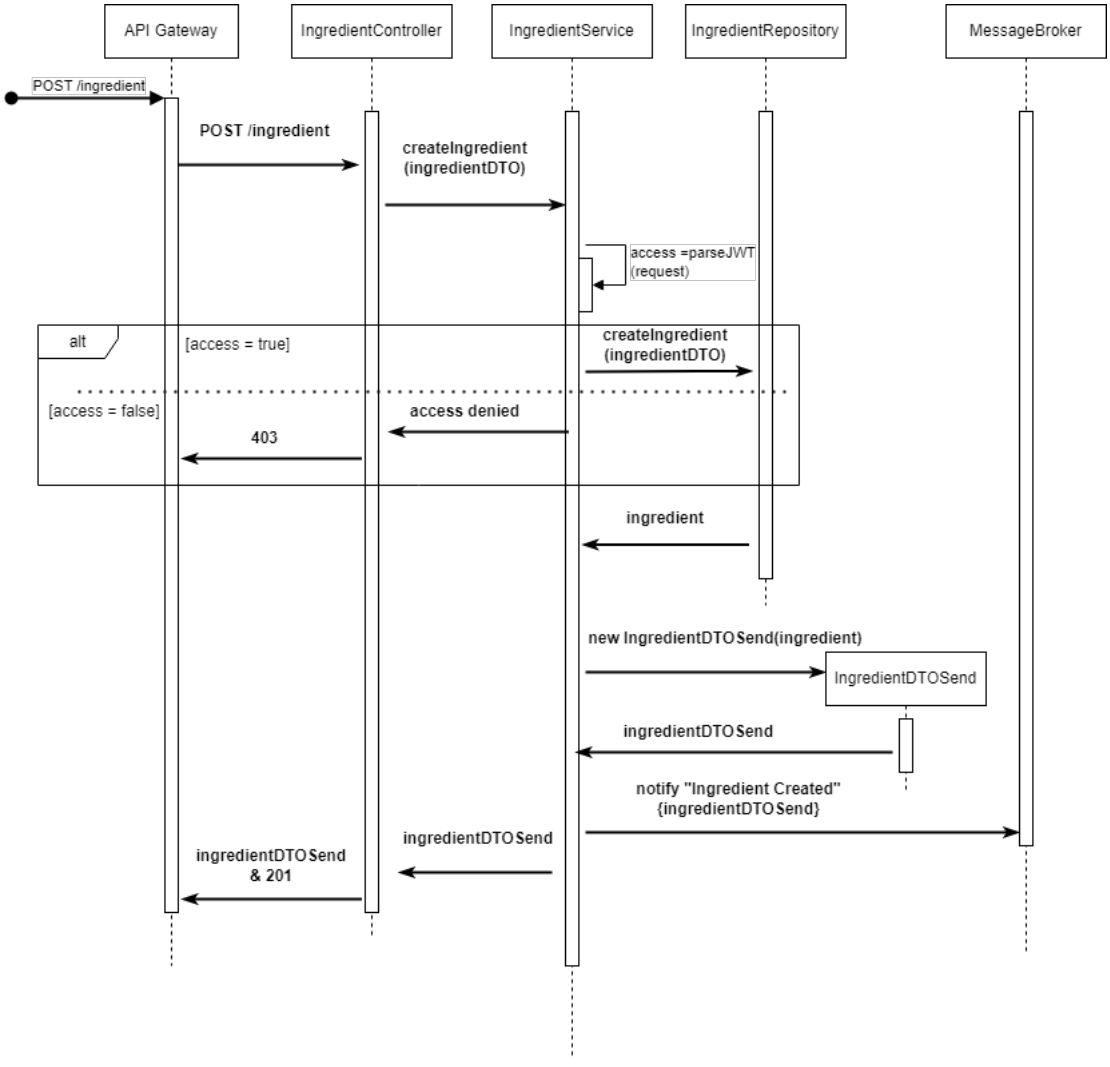

Figura 5.10: Diagrama da vista de processo do método *POST* nível 2

A figura 5.11, representa o processo para obter a lista de todos os ingredientes ao mais alto nível. O diagrama começa por representar o pedido feito, que será

efetuado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço "Ingredient". Este por sua vez, retornará a lista dos ingredientes criados e o código *HTTP* 200.

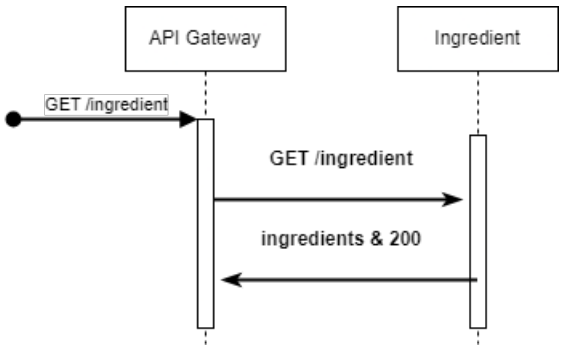

Figura 5.11: Diagrama da vista de processo do método *GET* nível 1

A figura 5.10, representa o processo para obter a lista de todos os ingredientes a um nível mais baixo do que a anterior. O diagrama começa por representar o pedido feito, que será realizado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço *Ingredient*. Sendo o *IngredientController*, o ponto de partida para o serviço mencionado anteriormente, este é responsável pelo encaminhamento do pedido recebido. A seguir, este chamará a função *getIngredients*, que pedirá ao *IngredientService* a lista de ingredientes. A seguir, o pedido é reencaminhado para o *IngredientRepository*, onde será feito o pedido à base de dados para esta retornar os ingredientes criados. Já com a lista de ingredientes retornados pela base de dados será necessário passar cada objeto dessa lista para um *DTO* e adicionar esse mesmo a outra lista que será posteriormente retornada, juntamente do código *HTTP* 200.

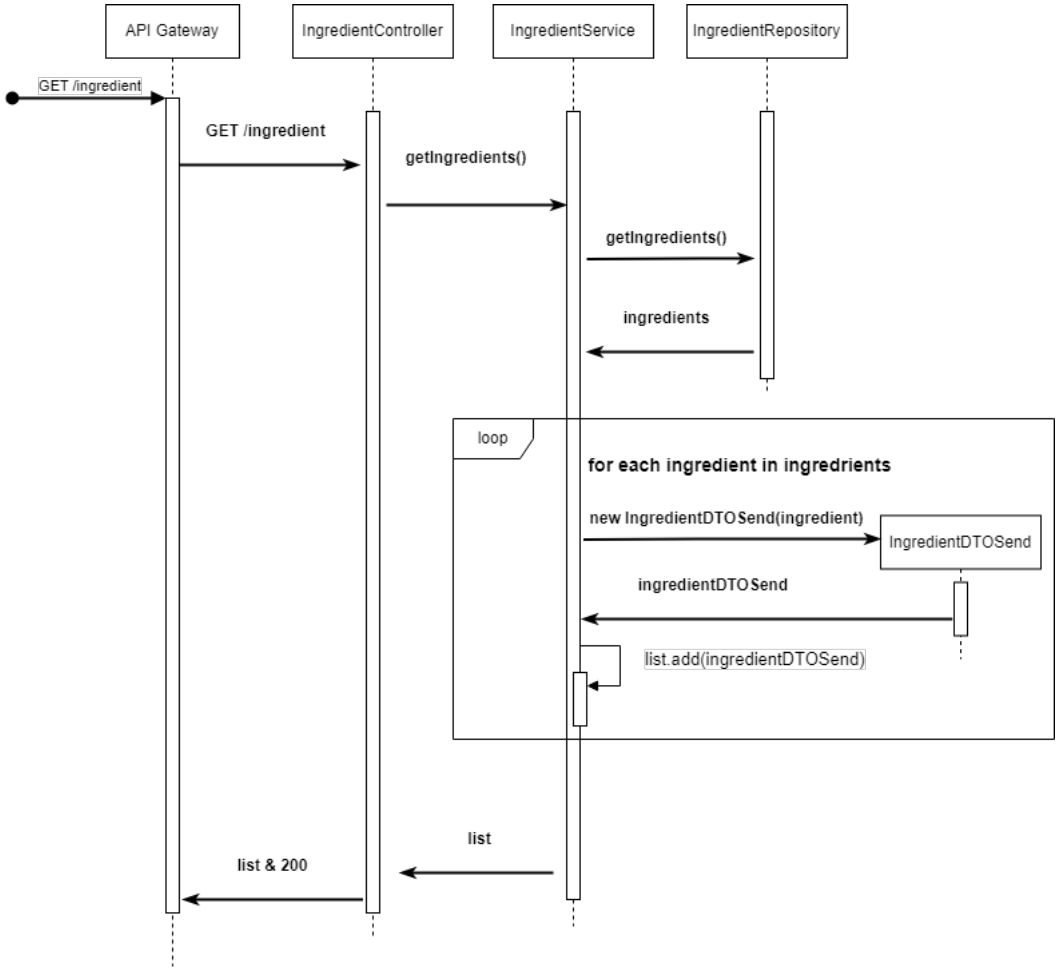

Figura 5.12: Diagrama da vista de processo do método *GET* nível 2

A figura 5.13, representa o processo de eliminação de um ingrediente ao mais alto nível. O diagrama começa por representar o pedido feito, que será efetuado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço "Ingredient". Este por sua vez, retornará no caso de haver conflito na autenticação o código *HTTP* 403 ou o código *HTTP* 204, se por eliminado com sucesso.

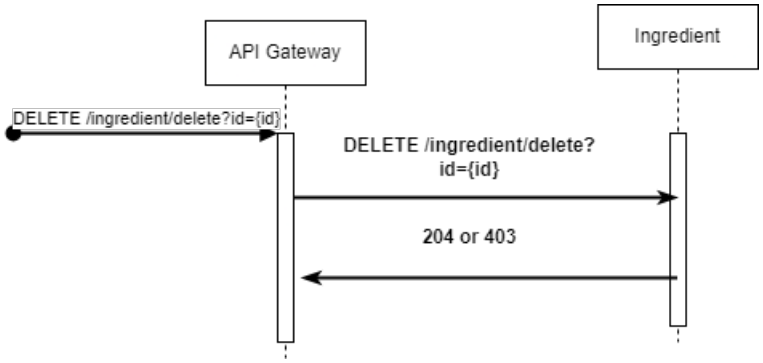

Figura 5.13: Diagrama da vista de processo do método *DELETE* nível 1

A figura 5.14, representa o processo de eliminação de um ingrediente a um nível mais baixo do que a anterior. O diagrama começa por representar o pedido feito, que será realizado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço *Ingredient*. Sendo o *IngredientController*, o ponto de partida para o serviço mencionado anteriormente, este é responsável pelo encaminhamento do pedido recebido. A seguir, este chamará a função *deleteIngredient*, que enviará o identificador do ingrediente introduzido no pedido para o *IngredientService* onde será processado. É neste momento, que será feita a verificação do utilizador, no caso de não lhe ser permitido o acesso, será enviado o código *HTTP* 403, por outro lado, caso se verifique que tem as permissões necessárias, o mesmo identificador mencionado anteriormente, será enviado para o *IngredientRepository*, onde será feito a eliminação do respetivo ingrediente na base de dados. Já o objeto eliminado na base de dados será feita uma notificação ao *Message Broker* da sua eliminação e por último será retornado com o código *HTTP* 204.

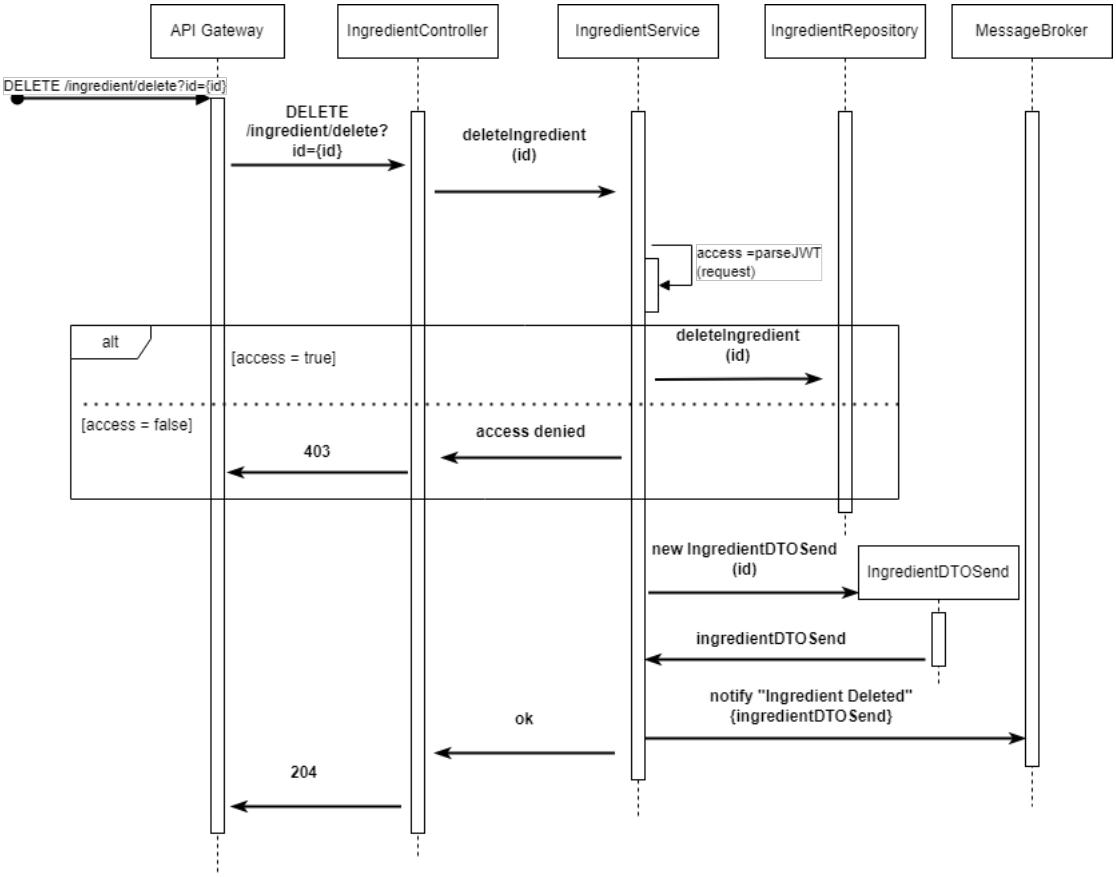

Figura 5.14: Diagrama da vista de processo do método *DELETE* nível 2

A figura 5.15, representa o processo de alteração de um ingrediente anteriormente já criado ao mais alto nível. O diagrama começa por representar o pedido feito, que será efetuado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço "Ingredient". Este por sua vez, retornará o ingrediente alterado e o código que representa a alteração sucedida desse mesmo ingrediente, no caso de haver conflito na autenticação é retornado só o código *HTTP* 403.

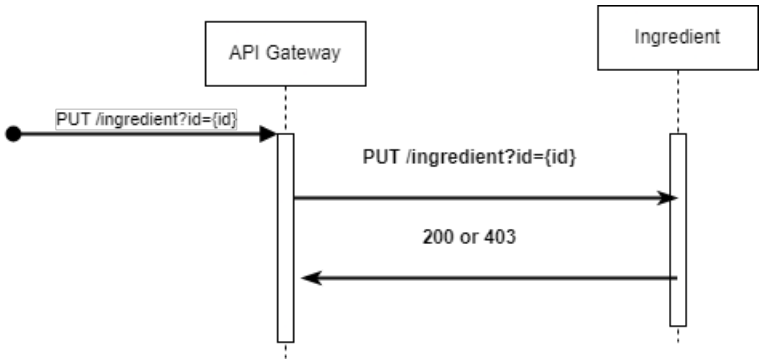

Figura 5.15: Diagrama da vista de processo do método *PUT* nível 1

A figura 5.16, representa o processo de alteração de um ingrediente a um nível mais baixo do que a anterior. O diagrama começa por representar o pedido feito, que será realizado à *API Gateway*. Posteriormente, este pedido será encaminhado para o serviço correspondente, o que neste caso, será o serviço *Ingredient*. Sendo o *IngredientController*, o ponto de partida para o serviço mencionado anteriormente, este é responsável pelo encaminhamento do pedido recebido. A seguir, este chamará a função *updateIngredient*, que enviará o objeto alterado e o identificador introduzido no pedido para o *IngredientService* onde será processado. É neste momento, que será feita a verificação do utilizador, no caso de não lhe ser permitido o acesso, será enviado o código *HTTP* 403, por outro lado, caso se verifique que tem as permissões necessárias, o mesmo objeto mencionado anteriormente, será enviado para o *IngredientRepository*, onde será feito a alteração na base de dados criada para o efeito. Já o objeto alterado na base de dados é retornado um objeto contendo todos os parâmetros do ingrediente, que terá de passar por um processo de transformação, ou seja, é aplicado o padrão de *software DTO*. Com este *DTO* criado será feita uma notificação ao *Message Broker* da sua alteração e por último será retornado com o código *HTTP* 201.

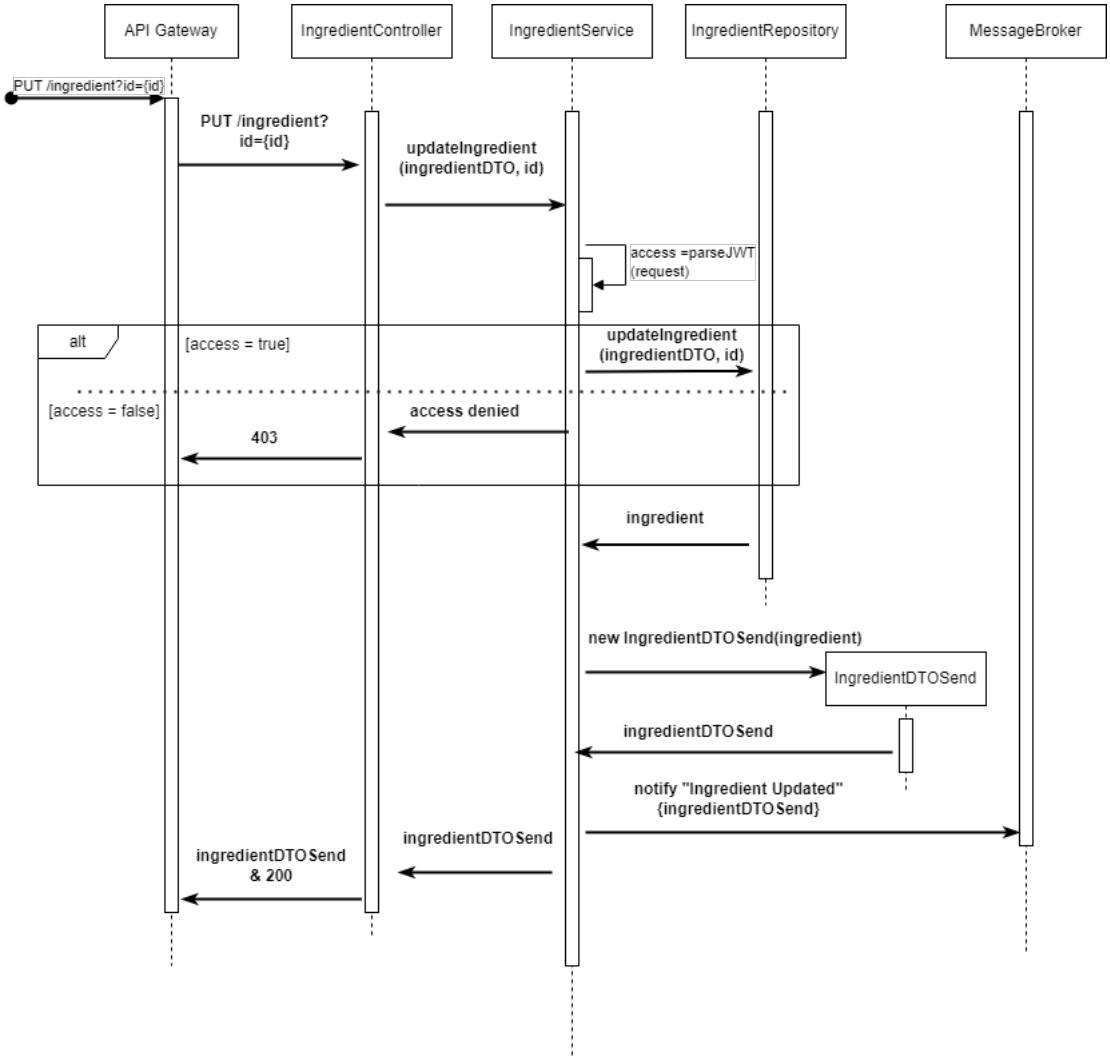

Figura 5.16: Diagrama da vista de processo do método *PUT* nível 2

5.2.4 Vista física

A figura 5.17 representa a vista física do sistema a ser implementado, ou seja, como está distribuída a infraestrutura do mesmo e como é feita a ligação entre os elementos descritos. A legenda para as linhas destacadas na imagem é:

- Linha azul: *MQTT*
- Linha vermelho: *HTTP*
- Linha amarela: *MySQL Protocol*

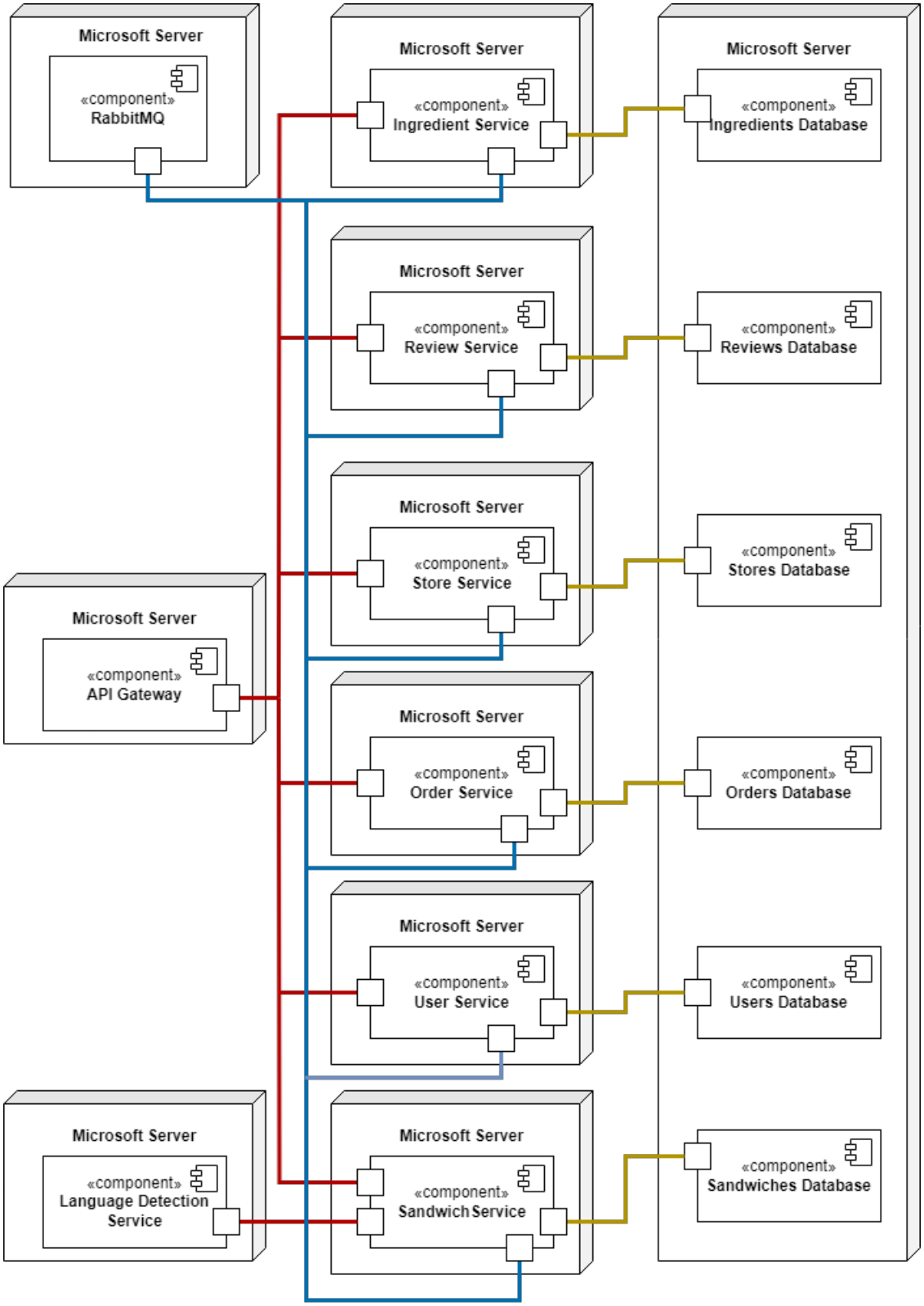

Figura 5.17: Diagrama da vista física

5.3 Descrição dos microsserviços

Na seguinte Tabela 5.1 é feita uma enumeração dos microsserviços a serem desenvolvidos juntamente da sua descrição.

Tabela 5.1: Descrição dos microsserviços

Descrição dos microsserviços	
Microsserviço	**Descrição**
SandwichService	Disponibiliza funcionalidades para a gestão de sanduíches
IngredientService	Disponibiliza funcionalidades para a gestão de ingredientes
ReviewService	Disponibiliza funcionalidades para a gestão de críticas
StoreService	Disponibiliza funcionalidades para a gestão de lojas
OrderService	Disponibiliza funcionalidades para a gestão de encomendas
CustomerService	Disponibiliza funcionalidades para a gestão de clientes
LanguageDectetionService	Disponibiliza funcionalidades para detetar a linguagem de um certo texto

5.4 Diagrama comunicação entre serviços

Num sistema distribuído, a comunicação entre serviços por muitas vezes é necessária, seja ela efetuada por HTTP ou algum outro método, como, por exemplo, Kafka ou até mesmo RabbitMQ. Dado que, a comunicação síncrona (HTTP) tem problemas como dificuldade de escalabilidade ou a acoplagem de serviços, decidiu-se utilizar a comunicação assíncrona, em concreto o *software* RabbitMQ, dado as suas caraterísticas anteriormente já referidas. O seguinte Figura 5.18 mostra as comunicações que necessitam ser feitas entre serviços.

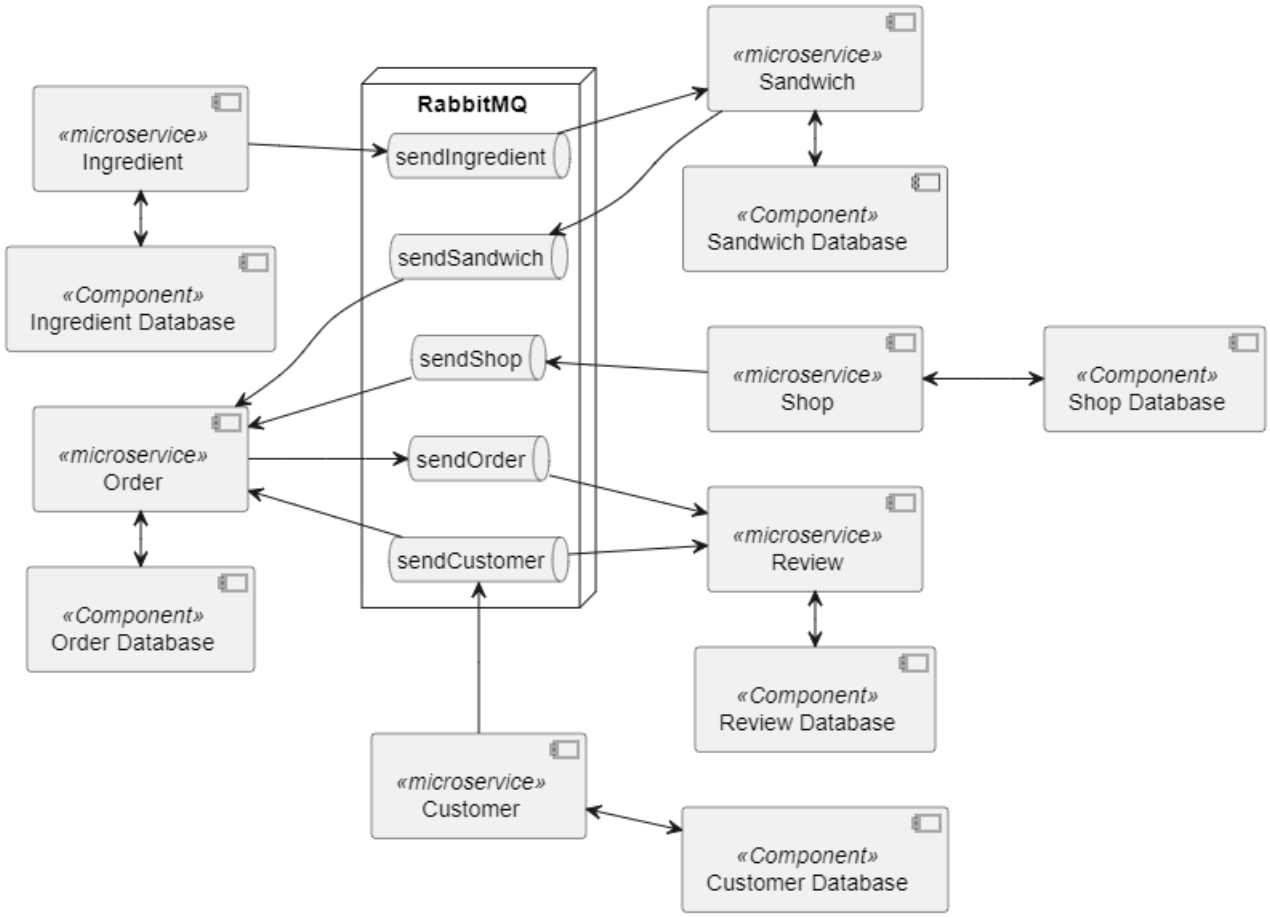

Figura 5.18: Diagrama de comunicação entre serviços

Capítulo 6

Implementação

Neste capitulo será descrita como a solução foi implementada, referindo os vários microsserviços e as aplicações de deteção de linguagem. Finalmente será feita uma lista das tecnologias e ferramentas usadas para realização deste protótipo.

6.1 Distribuição da aplicação

A Tabela 6.1 descreve a atribuição dos portos de cada um dos integrantes do sistema.

Tabela 6.1: Distribuição da aplicação

Descrição	**Porto**
SandwichService	8091
IngredientService	8090
ReviewService	8095
StoreService	8093
OrderService	8094
CustomerService	8092
LanguageDectetionService	5000
APIGateway	8000
RabbitMQ	5672
MySQLContainer	3306

6.2 *MySQL Container*

A Figura 6.1 enumera as bases de dados criadas para a aplicação recorrendo ao uso do *software Podman* que faz a gestão do *container* de um servidor de base de dados *MySQL*. As bases de dados criadas foram as seguintes:

- *Ingredients*: Armazena informações acerca dos ingredientes
- *Orders*: Armazena informações acerca das encomendas
- *Reviews*: Armazena informações acerca das críticas
- *Sandwiches*: Armazena informações acerca dos sanduíches
- *Stores*: Armazena informações acerca das lojas
- *Users*: Armazena informações acerca dos utilizadores

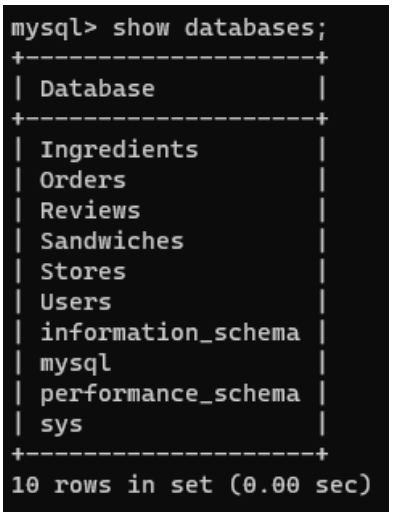

Figura 6.1: Bases de dados da aplicação

6.3 Variáveis do sistema

As variáveis do sistema são argumentos (escritas em maiúsculas) definidos tipicamente num ficheiro à parte que podem ser acedidos pela aplicação. O uso destas variáveis é importante, dado que, podem esconder informações confidenciais, tais como, informações de acesso à base de dado e/ou informações repetitivas que podem ser reduzidas num argumento, como, por exemplo, *URLs*. A Listagem 6.1 apresenta os argumentos de acesso à base de dados guardados num ficheiro chamado *Configuration.toml* com uma anotação no topo a referir o módulo que pode aceder aquela informação, que neste caso é o *repository*.

```
[ingredient.repository]
USER = "myUser"
PASSWORD = "myPassword"
HOST = "localhost"
PORT = 3306
```

Listagem 6.1: Definição de variáveis do sistema.

Já no módulo *repository*, surge o problema de como podemos aceder aos argumentos no ficheiro de configuração. Para isso é preciso recorrer à *keyword configurable* que indica à aplicação que aquela variável tem que ter um valor igual ao valor do argumento com o mesmo nome no ficheiro de configuração, como pode ser visualizado na Listagem 6.2. Por isso é importante verificar-se a igualdade dos nomes para não haver variáveis que não tenham valor atribuído.

```
configurable string USER = ?;
configurable string PASSWORD = ?;
configurable string HOST = ?;
configurable int PORT = ?;
```

Listagem 6.2: Invocação das variáveis do sistema.

6.4 Implementação dos microsserviços

6.4.1 *Endpoints* do microsserviço Sanduíches

Na Tabela 6.2 são apresentados todos os *endpoints* relativos ao microsserviço Sanduíche.

Tabela 6.2: *Endpoints* do microsserviço Sanduíche

Endpoints	
Tipo de pedido	**Endpoint**
POST	/sandwiches
GET	/sandwiches
GET	/sandwiches/searchById
POST	/sandwiches/descriptions
GET	/sandwiches/searchWithoutId
PUT	/sandwiches/state
PUT	/sandwiches/informations

6.4.2 *Endpoints* do microsserviço Ingredientes

Na Tabela 6.3 são apresentados todos os *endpoints* relativos ao microsserviço Ingredientes.

Tabela 6.3: *Endpoints* do microsserviço Ingredientes

Endpoints	
Tipo de pedido	**Endpoint**
POST	/ingredients
GET	/ingredients/searchById
GET	/ingredients/searchByDesignation
GET	/ingredients
PUT	/ingredients/state
GET	/ingredients/category

6.4.3 *Endpoints* do microsserviço Críticas

Na Tabela 6.4 são apresentados todos os *endpoints* relativos ao microsserviço Críticas.

Tabela 6.4: *Endpoints* do microsserviço Críticas

Endpoints	
Tipo de pedido	**Endpoint**
POST	/reviews
POST	/reviews/vote
GET	/reviews
GET	/reviews/reported
DELETE	/reviews/delete
PUT	/reviews/report
PUT	/reviews/searchById
GET	/reviews/searchByClient
DELETE	/reviews/clientDeleteReview

6.4.4 *Endpoints* do microsserviço Lojas

Na Tabela 6.5 são apresentados todos os *endpoints* relativos ao microsserviço Lojas.

Tabela 6.5: *Endpoints* do microsserviço Lojas

Endpoints	
Tipo de pedido	**Endpoint**
POST	/stores
GET	/stores
GET	/stores/searchById
GET	/stores/searchByDesignation
GET	/stores/searchByAddress
DELETE	/stores/delete
PUT	/stores/closingHours
PUT	/stores/openingHours

6.4.5 *Endpoints* do microsserviço Encomendas

Na Tabela 6.6 são apresentados todos os *endpoints* relativos ao microsserviço Encomendas.

Tabela 6.6: *Endpoints* do microsserviço Encomendas

Endpoints	
Tipo de pedido	**Endpoint**
POST	/orders
GET	/orders/searchByStoreId
PUT	/orders/state
PUT	/orders/seachByClientId

6.4.6 *Endpoints* do microsserviço Clientes

Na Tabela 6.7 são apresentados todos os *endpoints* relativos ao microsserviço Clientes.

Tabela 6.7: *Endpoints* do microsserviço Clientes

Endpoints	
Tipo de pedido	**Endpoint**
POST	/clients
GET	/clients/searchById
GET	/clients/searchByEmail
GET	/clients/searchByFiscalNumber
GET	/clients/authData/searchById
GET	/clients
POST	/clients/authData
POST	/clients/login
GET	/clients/data

6.4.7 *Endpoint* do microsserviço de deteção de linguagem

Na Tabela 6.8 é apresentado o *endpoint* relativos ao microsserviço de deteção de linguagem.

Tabela 6.8: *Endpoints* do microsserviço de deteção de linguagem

Deteção de linguagem	
Tipo de pedido	**Endpoint**
POST	/language

6.5 API GATEWAY

A criação de uma *API GATEWAY* ajuda na implementação de segurança num certo sistema. Normalmente é utilizado em sistemas com uma arquitetura de microsserviços, no qual os vários serviços são independentes. Com este padrão de *software*, os clientes podem interagir com os serviços por meio de uma interface única e unificada, dado que, os pedidos são encaminhados posteriormente para os respetivos serviços. Para além de algumas caraterísticas apresentadas anteriormente, este apresenta outras como, por exemplo:

- Balanceamento de carga: Os pedidos podem ser distribuídos pelas várias instâncias
- Segurança: Criptógrafa o tráfego de dados e podem ser imposta políticas de segurança

- Limitação de taxa: Pode limitar o número de pedidos que um certo cliente pode fazer por um determinado tempo, garantindo desta forma que não haja sobrecargas ou ataques maliciosos
- Armazenamento em *cache*: Pode armazenar em *cache* respostas de serviços, levando a diminuição da latência e um aumento no desempenho

A Listagem 6.3 apresenta uma porção de código da *API Gateway* desenvolvida. Para cada serviço que o sistema disponibiliza é necessária uma implementação semelhante ao código abaixo. Recorre-se a propriedade *req.originalUrl* para obter-se o *URL* do pedido original, por sua vez, é indispensável fazer a separação dos tipos de pedidos, dado que, no caso dos pedidos tipos *GET* e *DELETE*, este não precisam de um *body*.

```
app.all('/stores/*', (req, res) => {
    const serviceAUrl = `http://localhost:8093${req.originalUrl}`;

    if (req.method === 'GET' || req.method === 'DELETE') {
      axios({
        method: req.method,
        url: serviceAUrl,
        headers: req.headers
      })
        .then(response => {
          res.json(response.data);
        })
        .catch(error => {
          res.status(500).json({ error: 'An error occurred' });
        });
    } else {
      axios({
          method: req.method,
          url: serviceAUrl,
          headers: req.headers
        })
          .then(response => {
            res.json(response.data);
          })
          .catch(error => {
            res.status(500).json({ error: 'An error occurred' });
          });
    }
});
```

Listagem 6.3: API Gateway desenvolvida.

6.6 Serviços de deteção de linguagem

No que diz respeito a serviços de deteção de linguagem foram desenvolvidas duas aplicações recorrendo a diferentes tecnologias, como podem ser observados nas Listagens 6.4 e 6.5. Na primeira aplicação, Listagem 6.4, foi utilizado o *Python* com uma framework web denominada *Flask*. Já na segunda aplicação, Listagem 6.5, foi utilizado o *NodeJS* com uma framework web denominada *Express.js*.

No primeiro exemplo é primariamente importado as classes *Flask*, *jsonify* e *request* do módulo *flask*, tal como, a função *detect* do módulo *langdetect*. Na linha a seguir é instanciado a classe *Flask*.

A seguir é definida uma rota */language* que invoca a função *getLanguages()*. Quando a função for chamada a variável *content* recebe os dados JSON e uma lista vazio é criada, de seguida, um *loop* é utilizado, onde para cada um dos textos recebidos, é feita uma chamada à função *detect*, que retorna a linguagem detetada e que é posteriormente adicionada a lista criada anteriormente para o efeito. Logo após a realização do *loop* é criado um dicionário que contém as várias linguagens detetadas por ordem. Por fim, é retornado o dicionário convertido para uma resposta *JSON*.

```
1  from flask import Flask,jsonify,request
2  from langdetect import detect
3
4  app = Flask(__name__)
5
6
7  @app.route('/language', methods=['GET', 'POST'])
8  def getLanguage():
9      content = request.json
10     languages = []
11
12     for text in content:
13         language = detect(text["text"])
14         languages.append(language)
15
16     data = {
17         "languages": languages
18     }
19     return jsonify(data)
20
21
22 if __name__ == '__main__':
23     app.run()
```

Listagem 6.4: Serviço de deteção de linguagem.

Já no segundo exemplo é primariamente importado os módulos *express* e *languagedetect*, onde são posteriormente instanciados e no caso *languagedetect* é definido o tipo de linguagem a ser apresentado. O funcionamento da obtenção das linguagens tem um esquema identifico ao referido no exemplo anterior. Sendo no final determinado o porto de funcionamento do servidor *WEB*.

```
1  const express = require('express');
2  const app = express();
3  const LanguageDetect = require('languagedetect');
4  const lngDetector = new LanguageDetect();
5  lngDetector.setLanguageType("iso2");
6
7  app.use(express.json());
8
9  app.post('/language', (req, res) => {
10     const content = req.body;
11     let languages = [];
12
13     content.forEach(text => {
14         const language = lngDetector.detect(text.text,1);
15         languages.push(language[0][0]);
16     });
17
18     data = {
19         "languages":languages
20     }
21
22     res.json(data);
23 });
24
25 app.listen(5000, () => {
26     console.log('Server running on port 5000');
27 });
```

Listagem 6.5: Serviço de deteção de linguagem.

Foram realizados testes com o *software Postman* para verificar o funcionamento da aplicação de deteção de linguagem. Foi feito um pedido do tipo *POST* ao *URL http://127.0.0.1:5000/language* com uma lista de texto, um em português e outro em inglês. Ao qual foi recebido a lista das linguagens correta e na ordem correta, como se pode observar na Figura 6.2. Além disso, foi desenvolvido um teste de código para verificar o estado de saúde da aplicação *WEB* ao longo do tempo que por sua vez que também foi bem sucedido, como se pode observar na Figura 6.3.

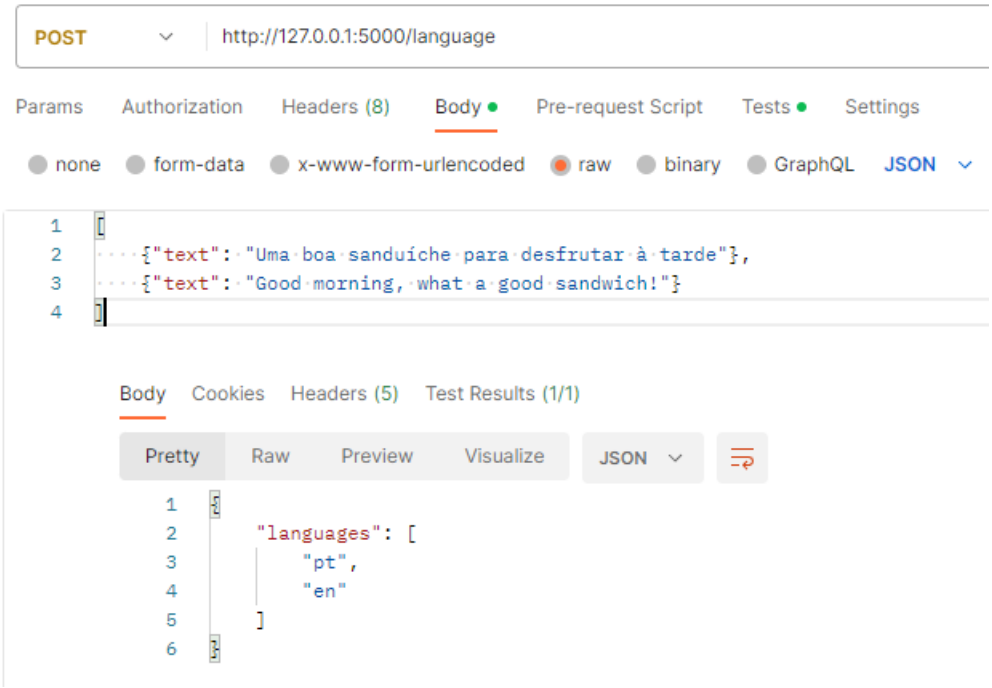

Figura 6.2: Pedido exemplo com retorno.

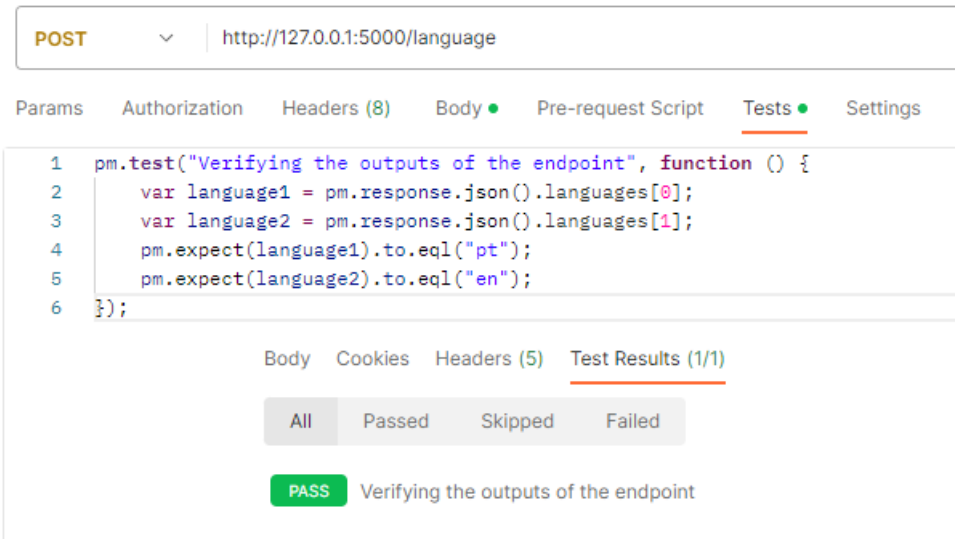

Figura 6.3: Teste realizado ao pedido.

6.7 Testes desenvolvidos

6.7.1 Teste unitário

apresenta um exemplo de um teste unitário desenvolvido no que diz respeito a criação de uma sanduíche. Começa-se por definir que esta porção de código é um teste

com a anotação *@test:Config* , as chavetas encontram-se vazias, dado que, não há configurações adicionais. Dado isso, é feita a invocação da função *testSandwich()* que contém o código a ser executado. Essa função inicializa um o objeto *SandwichDTO* com um exemplo de uma sanduíche que pode ser criada. Por fim, são feitas todas as verificações necessárias com a função *assertEquals* do módulo *test* para verificar se o objeto foi criado com as informações inseridas.

```
1  @test:Config {}
2  function testSandwich() {
3      SandwichDTO sandwich = {designation: "Veggie Delight",
           selling_price: 5.99,
4          ingredients_id: [1, 2, 3],
5          descriptions: [{text: "A delicious vegetarian sandwich"},
6                         {text: "Een heerlijk vegetarisch broodje"}]};
7      test:assertEquals(sandwich.designation, "Veggie Delight");
8      test:assertEquals(sandwich.selling_price, 5.99);
9      test:assertEquals(sandwich.ingredients_id, [1, 2, 3]);
10     test:assertEquals(sandwich.ingredients_id[0],1);
11     test:assertEquals(sandwich.ingredients_id[1],2);
12     test:assertEquals(sandwich.ingredients_id[2],3);
13     test:assertEquals(sandwich.descriptions[0].text, "A delicious
           vegetarian sandwich");
14     test:assertEquals(sandwich.descriptions[1].text, "Een heerlijk
            vegetarisch broodje");
15     test:assertEquals(sandwich.descriptions.length(), 2);
16     test:assertEquals(sandwich.ingredients_id.length(), 3);
17 }
```

Listagem 6.6: Exemplo de teste desenvolvido.

6.7.2 Testes de integração

A Figura 6.4 apresenta um excerto dos testes de integração desenvolvidos para testar as funcionalidades de cada um dos serviços que fazem parte da logística do sistema. Estes forma realizados com o auxílio do *software Postman*.

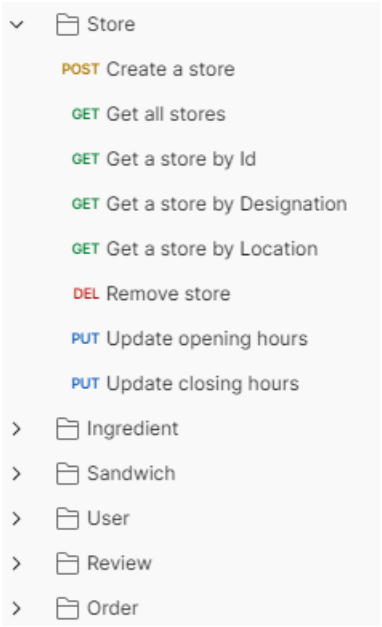

Figura 6.4: Testes desenvolvidos em Postman.

Já a Figura 6.5, nesta é demonstrado um exemplo de um *script* criado, visando gerar dados que vão ser usados, neste caso, no pedido de criação de uma loja.

Figura 6.5: Testes desenvolvidos em Postman.

Por outro lado, a Figura 6.6 apresenta um exemplo de testes efetuados no teste de integração. Nestes são feitos testes para verificar o código *HTTP* e a presença de certos campos na resposta recebida.

POST http://localhost:8093/stores

Params Authorization Headers (8) Body Pre-request Script Tests Settings

```
1  pm.test("Status code is 201", function () {
2      pm.response.to.have.status(201);
3  });
4
5  pm.test("Body contains values",() => {
6    pm.expect(pm.response.text()).to.include("store_id");
7    pm.expect(pm.response.text()).to.include("designation");
8    pm.expect(pm.response.text()).to.include("address");
9    pm.expect(pm.response.text()).to.include("openingHours");
10   pm.expect(pm.response.text()).to.include("closingHours");
11 });
12
13
14 pm.globals.set("idStore", pm.response.json().store_id);
15 console.log(pm.globals.get("idStore"));
16
```

Figura 6.6: Testes desenvolvidos em Postman.

6.7.3 Teste de desempenho e carga

A Figura 6.7 apresenta testes de desempenho e carga desenvolvidos para comparar o atual desempenho do sistema com o requisito apresentado na secção Análise. Estes foram realizados com o auxílio do *software JMeter*.

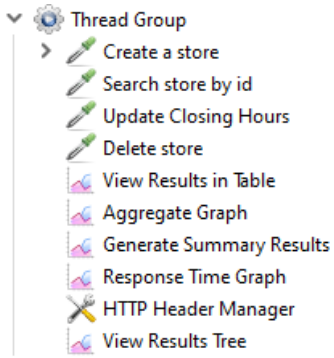

Figura 6.7: Testes de desempenho e carga.

Cada utilizador virtual executa o *Thread Group*, este é composto por vários pedidos *HTTP*, neste caso foram escolhidos um de cada tipo de pedido (*POST*, *GET*, *PUT* e *DELETE*). Este é constituído por parâmetros modificáveis, o número de utilizadores virtuais, o período de arranque e o número de vezes que o utilizador executará o mesmo. Vale a pena referir que o período de arranque representa o tempo necessário para que todos os utilizadores virtuais sejam adicionados à execução do teste.

Na Tabela 6.9 são apresentados vários cenários. Para cada cenário foram anotados os seguintes parâmetros:

- Número de ciclos
- Utilizadores virtuais
- Taxa de transferência
- Tempo decorrido
- Período de arranque utilizado

Tabela 6.9: Testes de desempenho e carga

Testes de desempenho e carga				
Número de ciclos	**Utilizadores virtuais**	**Taxa de transferência (pedidos por segundo)**	**Tempo decorrido (mm:ss.ms)**	**Período de arranque (s)**
1	10	232.6	00:00.102	0
1	100	341.0	00:01.230	1
1	1000	199.3	00:10.935	20
10	10	385.0	00:01.312	0
10	100	194.9	00:20.592	20
10	1000	332.3	02:00.744	60

Para linha da tabela são apresentadas nas figuras abaixo (Figura 6.8, Figura 6.9, Figura 6.10, Figura 6.11, Figura 6.12 e Figura 6.13) informações adicionais. Cada figura expõe a média, a mediana, o mínimo e o máximo para o número das amostras (total de pedidos realizados para cada pedido), com a adição do erro e a taxa de transferência.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	10	111	112	96	124	0,00%	80,6/sec
Search store by id	10	15	15	12	21	0,00%	256,4/sec
Update Closing Hours	10	16	15	13	21	0,00%	277,8/sec
Delete store	10	21	21	19	24	0,00%	263,2/sec
TOTAL	40	41	20	12	124	0,00%	232,6/sec

Figura 6.8: Testes de desempenho e carga linha 1.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	100	198	203	60	291	0,00%	88,5/sec
Search store by id	100	23	22	3	58	0,00%	93,3/sec
Update Closing Hours	100	35	33	6	86	0,00%	92,9/sec
Delete store	100	46	43	19	90	0,00%	91,3/sec
TOTAL	400	75	40	3	291	0,00%	341,0/sec

Figura 6.9: Testes de desempenho e carga linha 2.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	1000	62	62	51	82	0,00%	49,9/sec
Search store by id	1000	2	3	2	5	0,00%	50,0/sec
Update Closing Hours	1000	6	6	4	19	0,00%	50,0/sec
Delete store	1000	15	16	10	27	0,00%	50,0/sec
TOTAL	4000	21	12	2	82	0,00%	199,3/sec

Figura 6.10: Testes de desempenho e carga linha 3.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	100	73	73	62	91	0,00%	98,2/sec
Search store by id	100	2	3	2	4	0,00%	105,6/sec
Update Closing Hours	100	7	7	5	11	0,00%	105,2/sec
Delete store	100	18	18	12	28	0,00%	104,6/sec
TOTAL	400	25	11	2	91	0,00%	385,0/sec

Figura 6.11: Testes de desempenho e carga linha 4.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	1000	60	59	38	280	0,00%	48,8/sec
Search store by id	1000	2	2	2	5	0,00%	48,9/sec
Update Closing Hours	1000	6	6	4	14	0,00%	48,9/sec
Delete store	1000	15	15	9	29	0,00%	48,9/sec
TOTAL	4000	21	10	2	280	0,00%	194,9/sec

Figura 6.12: Testes de desempenho e carga linha 5.

Label	# Samples	Average	Median	Min	Maximum	Error %	Throughput
Create a store	10000	1060	215	36	60017	1,45%	83,1/sec
Search store by id	10000	185	32	1	60015	1,71%	83,1/sec
Update Closing Hours	10000	165	43	0	60015	1,66%	83,1/sec
Delete store	10000	149	44	0	60016	1,63%	83,1/sec
TOTAL	40000	389	45	0	60017	1,61%	332,3/sec

Figura 6.13: Testes de desempenho e carga linha 6.

6.8 Documentação da API

Após realizada a implementação dos serviços procedeu-se a documentação dos mesmos. A Figura 6.14 é um exemplo do como foi estruturada a informação para cada caso de uso. A documentação feita visa auxiliar os desenvolvedores/programadores a intenderem o funcionamento do sistema. Estes podem fornecer informações tais como:

- Correspondente caso de uso
- O *url do endpoint*
- Exemplo de pedido e resposta
- Parâmetros de consulta
- Possíveis respostas

Figura 6.14: Documentação da API.

6.9 Tecnologias e ferramentas usadas

Para a implementação desta solução foram usadas as seguintes tecnologias e ferramentas:

- *Ballerina*: Linguagem de programação
- *MySQL*: Sistema de gestão de bases de dados

- *Podman*: Plataforma de contentores
- *Postman*: Ferramenta de teste para *APIs*
- *RabbitMQ*: Programa de mensagens
- *Visual Studio Code*: Editor de código

Capítulo 7

Conclusões

7.1 Trabalho Futuro

No futuro seria interessante existir o desenvolvimento do *front end* da aplicação, ser feito a implementação do software para a gestão de pagamentos e *stock*, completando assim o sistema já desenvolvido e torna-o num produto pronto para produção, ou seja, estar a ser usado no dia a dia de alguma empresa.

7.2 Dificuldades

O desenvolvimento e implementação dos conteúdos apresentados nas secções anteriores trouxe algumas dificuldades no que diz respeito a informação, devido a Ballerina ser uma linguagem recente e ainda ter uma comunidade pequena, a documentação disponibiliza pelos criadores é insuficiente para alguns dos casos e muitas vezes as respostas encontradas para perguntas realizadas em fóruns comunitárias é desatualizada devido à constante atualização da linguagem.

Referências

[1] L. Tung, "Programming languages: These top four rule and developers are happy - for now." Disponível em https://www.zdnet.com/article/programming-languages-these-top-four-rule-and-developers-are-happy-for-now/, Nov. 2022. (Último acesso em 02/04/2023). [Citado na página 2]

[2] Wikipédia, "Aplicação monolítica." Disponível em https://pt.wikipedia.org/wiki/AplicaÃğÃčo_monolÃŋtica, 2022. (Último acesso em 15/03/2023). [Citado na página 5]

[3] P. Santos, "Quais as diferenças entre arquitetura monolítica e microsserviços, suas vantagens e desvantagens.." Disponível em https://arphoenix.com.br/quais-as-diferencas-entre-arquitetura-monolitica-e-microsservicos-suas-vantagens-e 2021. (Último acesso em 15/03/2023). [Citado nas páginas 6 e 7]

[4] P. Freitas, "Arquitetura monolítica é melhor que microsserviços ? quando usar uma ou outra ?." Disponível em https://www.linkedin.com/pulse/arquitetura-monolÃŋtica-Ãľ-melhor-que-microsserviÃğos-quando-freitas/, Nov. 2022. (Último acesso em 02/04/2023). [Citado nas páginas ix, 6 e 8]

[5] AWS, "O que é arquitetura orientada a serviços?." Disponível em https://aws.amazon.com/pt/what-is/service-oriented-architecture. (Último acesso em 15/03/2023). [Citado na página 6]

[6] H. Rzayev, "Why microservices are the future of architecture." Disponível em https://dreamix.eu/blog/microservices/why-microservices-are-the-future-of-architecture, Oct. 2022. (Último acesso em 20/03/2023). [Citado na página 7]

[7] B. tecnologia, "Elasticidade na cloud computing: entenda o que é e como funciona." Disponível em https://blog.vibetecnologia.com/elasticidade-na-cloud-computing/, June 2021. (Último acesso em 15/03/2023). [Citado na página 7]

[8] A. Lima, "Escalabilidade e elasticidade em computaÇÃo em nuvem." Disponível em https://acervolima.com/escalabilidade-e-elasticidade-em-computacao-em-nuvem/. (Último acesso em 15/03/2023). [Citado na página 7]

[9] C. Richardson, "Pattern: Command query responsibility segregation (cqrs)." Disponível em https://microservices.io/patterns/data/cqrs.html. (Último acesso em 15/03/2023). [Citado na página 7]

[10] C. Richardson, "Pattern: Database per service." Disponível em https://microservices.io/patterns/data/database-per-service.html. (Último acesso em 15/03/2023). [Citado na página 7]

[11] R. Hat, "O que é arquitetura orientada a eventos?." Disponível em https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture, 2019. (Último acesso em 15/03/2023). [Citado na página 8]

[12] ScyllaDB, "Event driven architecture." Disponível em https://www.scylladb.com/glossary/event-driven-architecture/. (Último acesso em 02/04/2023). [Citado nas páginas ix e 8]

[13] Wikipédia, "Chamada de procedimento remoto." Disponível em https://pt.wikipedia.org/wiki/Chamada_de_procedimento_remoto, 2022. (Último acesso em 02/04/2023). [Citado na página 9]

[14] RedHat, "Rest x soap." Disponível em https://www.redhat.com/pt-br/topics/integration/whats-the-difference-between-soap-rest, 2019. (Último acesso em 02/04/2023). [Citado na página 9]

[15] A. S. Mariana Berga, "grpc vs rest: comparing apis architectural styles." Disponível em https://www.imaginarycloud.com/blog/grpc-vs-rest/, 2019. (Último acesso em 02/04/2023). [Citado na página 9]

[16] RabbitMQ, "Publish/subscribe." Disponível em https://www.rabbitmq.com/tutorials/tutorial-three-python.html. (Último acesso em 02/04/2023). [Citado na página 10]

[17] Simplilearn, "Kafka vs rabbitmq: Biggest differences and which should you learn?." Disponível em https://www.simplilearn.com/kafka-vs-rabbitmq-article, Mar. 2023. (Último acesso em 02/04/2023). [Citado na página 10]

[18] Podman, "Podman." Disponível em https://podman.io/. (Último acesso em 02/04/2023). [Citado na página 11]

[19] Docker, "Docker." Disponível em https://www.docker.com/. (Último acesso em 02/04/2023). [Citado na página 11]

[20] R. Kumari, "Podman vs docker comparison: Which container tool is better?." Disponível em `https://testsigma.com/blog/podman-vs-docker/`, 2023. (Último acesso em 05/06/2023). [Citado nas páginas 11 e 12]

[21] Wikipédia, "Daemon (computing)." Disponível em `https://en.wikipedia.org/wiki/Daemon_(computing)`. (Último acesso em 05/06/2023). [Citado na página 11]

[22] Octopus, "O estado dos formatos de arquivo de configuração: Xml vs. yaml vs. json vs. hcls." Disponível em `https://octopus.com/blog/state-of-config-file-formats`. (Último acesso em 05/06/2023). [Citado nas páginas 12, 13, 14 e 15]

[23] Wikipédia, "Comma-separated values." Disponível em `https://pt.wikipedia.org/wiki/Comma-separated_values`. (Último acesso em 05/06/2023). [Citado na página 15]

[24] Microsoft, "Trabalhando com arquivos csv e json para soluções de dados." Disponível em `https://learn.microsoft.com/pt-br/azure/architecture/data-guide/scenarios/csv-and-json`. (Último acesso em 05/06/2023). [Citado nas páginas 15 e 16]

[25] ByteScout, "Csv format: History, advantages and why it is still popular." Disponível em `https://bytescout.com/blog/csv-format-history-advantages.html#3`. (Último acesso em 05/06/2023). [Citado na página 16]

[26] G. V. B. Roberto, "Jwt - explicação simples e rápida." Disponível em `https://www.linkedin.com/pulse/jwt-explicaÃğÃčo-simples-e-rÃąpida-guilherme-vilas-boas-roberto-1f/?originalSubdomain=pt`, Dec. 2019. (Último acesso em 20/03/2023). [Citado na página 16]

[27] FORTINET, "What is an api key?." Disponível em `https://www.fortinet.com/resources/cyberglossary/api-key`, Mar. 2023. (Último acesso em 22/03/2023). [Citado na página 16]

[28] MySQL, "Mysql." Disponível em `https://www.mysql.com/`. (Último acesso em 02/04/2023). [Citado na página 17]

[29] MariaDB, "Mariadb." Disponível em `https://mariadb.org/`. (Último acesso em 02/04/2023). [Citado na página 17]

[30] Postman, "Postman." Disponível em `https://www.postman.com/`. (Último acesso em 02/04/2023). [Citado na página 17]

[31] Insomnia, "Insomnia." Disponível em `https://insomnia.rest/`. (Último acesso em 02/04/2023). [Citado na página 17]

[32] P. Niedringhaus, "Postman vs. insomnia: Comparing the api testing tools." Disponível em `https://itnext.io/postman-vs-insomnia-comparing-the-api-testing-tools-4f12099275c1`. (Último acesso em 05/06/2023). [Citado nas páginas 17 e 18]

[33] JMeter, "Jmeter." Disponível em `https://jmeter.apache.org/`. (Último acesso em 02/04/2023). [Citado na página 18]

[34] SOAPUI, "Soapui." Disponível em `https://www.soapui.org/`. (Último acesso em 02/04/2023). [Citado na página 18]

[35] LoadView, "What is the difference between soapui and jmeter?." Disponível em `https://www.loadview-testing.com/blog/what-is-the-difference-between-soapui-and-jmeter/`. (Último acesso em 05/06/2023). [Citado na página 19]

[36] C. Richardson, "Pattern: Database per service." Disponível em `https://microservices.io/patterns/data/database-per-service.html`. (Último acesso em 05/06/2023). [Citado nas páginas 19 e 20]

[37] C. Richardson, "Pattern: Saga." Disponível em `https://microservices.io/patterns/data/saga.html`. (Último acesso em 05/06/2023). [Citado nas páginas 20 e 21]

[38] C. Richardson, "Pattern: Command query responsibility segregation (cqrs)." Disponível em `https://microservices.io/patterns/data/cqrs.html`. (Último acesso em 05/06/2023). [Citado na página 21]

[39] C. Richardson, "Pattern: Api composition." Disponível em `https://microservices.io/patterns/data/api-composition.html`. (Último acesso em 05/06/2023). [Citado na página 22]

[40] C. Richardson, "Pattern: Messaging." Disponível em `https://microservices.io/patterns/communication-style/messaging.html`. (Último acesso em 05/06/2023). [Citado nas páginas 22 e 23]

[41] C. Richardson, "Pattern: Api gateway / backends for frontends." Disponível em `https://microservices.io/patterns/apigateway.html`. (Último acesso em 05/06/2023). [Citado nas páginas 23, 24 e 25]

[42] C. Richardson, "Pattern: Access token." Disponível em `https://microservices.io/patterns/security/access-token.html`. (Último acesso em 05/06/2023). [Citado nas páginas 25 e 26]

[43] Ballerina, "Ballerina central." Disponível em `https://central.ballerina.io/`. (Último acesso em 29/04/2023). [Citado na página 27]

[44] Ballerina, "Flexibly typed." Disponível em `https://ballerina.io/why-ballerina/flexibly-typed/`. (Último acesso em 29/04/2023). [Citado na página 28]

[45] TheNewStack, "Why should you program with ballerina?." Disponível em `https://thenewstack.io/why-should-you-program-with-ballerina/`, Apr. 2022. (Último acesso em 02/04/2023). [Citado na página 28]

[46] Ballerina, "Organize ballerina code." Disponível em `https://ballerina.io/learn/organize-ballerina-code/`. (Último acesso em 29/04/2023). [Citado na página 28]

[47] WSO2, "Ballerina visual studio code extension documentation." Disponível em `https://wso2.com/ballerina/vscode/docs/`. (Último acesso em 29/04/2023). [Citado na página 29]

[48] Ballerina, "Code to cloud." Disponível em `https://ballerina.io/learn/run-in-the-cloud/code-to-cloud/code-to-cloud-deployment/`. (Último acesso em 29/04/2023). [Citado na página 29]

[49] Ballerina, "Hello world service." Disponível em `https://ballerina.io/learn/by-example/hello-world-service/`. (Último acesso em 02/04/2023). [Citado nas páginas xiii e 30]

[50] Ballerina, "Provide values to configurable variables." Disponível em `https://ballerina.io/learn/configure-ballerina-programs/provide-values-to-configurable-variables/`. (Último acesso em 29/04/2023). [Citado na página 30]

[51] Ballerina, "Concurrent." Disponível em `https://ballerina.io/why-ballerina/concurrent/`. (Último acesso em 02/04/2023). [Citado nas páginas xiii e 34]

[52] Ballerina, "Test a simple function." Disponível em `https://ballerina.io/learn/test-ballerina-code/test-a-simple-function/`. (Último acesso em 02/04/2023). [Citado nas páginas xiii e 35]

[53] V. Tanzu, "https://spring.io/projects/spring-boot." Disponível em `https://spring.io/projects/spring-boot`. (Último acesso em 05/06/2023). [Citado na página 35]

[54] Pallets, "Flask." Disponível em `https://flask.palletsprojects.com/en/2.3.x/`. (Último acesso em 05/06/2023). [Citado na página 35]

[55] D. S. Foundation, "Meet django." Disponível em `https://www.djangoproject.com/`. (Último acesso em 05/06/2023). [Citado na página 35]

[56] O. Foundation, "Express." Disponível em `https://expressjs.com/`. (Último acesso em 05/06/2023). [Citado na página 35]

[57] O. Foundation, "Fast and low overhead web framework, for node.js." Disponível em `https://fastify.dev/`. (Último acesso em 05/06/2023). [Citado na página 35]

[58] Actix, "Actix." Disponível em `https://actix.rs/`. (Último acesso em 05/06/2023). [Citado na página 35]

[59] Microsoft, "Creating a simple data-driven crud microservice." Disponível em `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/data-driven-crud-microservice`. (Último acesso em 05/06/2023). [Citado na página 35]

[60] G. Fiber, "Fiber." Disponível em `https://docs.gofiber.io/`. (Último acesso em 05/06/2023). [Citado na página 35]

[61] I. 25000, "Iso / iec 25010." Disponível em `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`. (Último acesso em 11/05/2023). [Citado na página 41]

[62] top10, "Top 10 secure coding practices." Disponível em `https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices`, 2018. (Último acesso em 11/05/2023). [Citado na página 42]

[63] P. B. Kruchten, "The 4+ 1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995. [Citado nas páginas ix e 55]