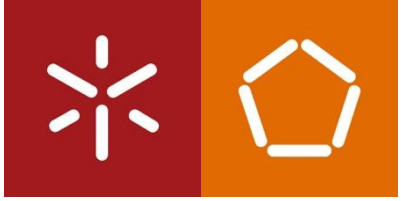
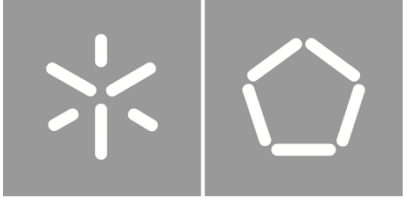


Universidade do Minho
Escola de Engenharia
Departamento de Informática

José André Martins Pereira

Estudo sobre a importância dos Princípios e Padrões das Arquiteturas orientadas a Microsserviços

abril de 2023



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José André Martins Pereira

Estudo sobre a importância dos Princípios e Padrões das Arquiteturas orientadas a Microsserviços

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sobre a orientação do
Professor Doutor António Manuel Nestor Ribeiro

abril de 2023

AGRADECIMENTOS

Agradeço à minha família e amigos por todo o apoio e dedicação, em particular aos meus pais, avô, irmã, afilhado, cunhado e amigos mais próximos. O vosso apoio a nível pessoal foi e é fundamental para alcançar os meus objetivos.

Agradeço a todos os meus professores por todo o conhecimento que partilharam comigo, por tudo o que me ensinaram, mas em especial ao meu orientador, Professor António Manuel Nestor Ribeiro, que aceitou orientar este tema sugerido por mim. Agradeço ao meu orientador por todos os conselhos e conversas, onde discutimos ideias e opiniões de forma a melhorar o resultado do trabalho.

Agradeço à entidade CGI Portugal, na qual trabalhei durante parte do processo de desenvolvimento deste trabalho, pelo apoio e disponibilidade prestada para a realização do mesmo, com especial agradecimento a Jaime Silva e Manuel Araújo.

Agradeço à entidade Devoteam, na qual trabalho atualmente, pelo apoio e disponibilidade que me prestam diariamente para a realização deste trabalho, com especial agradecimento a Rui Bento, Nuno Silva e Ricardo Ferreira.

Agradeço a todos os colegas de curso, pela vossa companhia durante o percurso académico, em especial ao Ricardo Petronilho e João Marques, não só pela ajuda a nível académico, como também pelo convívio a nível pessoal.

Por fim, deixo um agradecimento à Universidade do Minho, a toda a comunidade académica, sem o apoio de todos os envolvidos, este desfecho positivo não seria possível.

DECLARAÇÃO DE INTEGRIDADE

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

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

_____, _____

(Localização) (Data)

_____

(José André Martins Pereira)

Estudo sobre a importância dos Princípios e Padrões das Arquiteturas Orientadas a Microsserviços

Resumo

Com o atual crescimento do desenvolvimento de software e aumento da procura para solução de diversos problemas, começa-se a verificar a incapacidade de certas abordagens arquiteturais para lidarem com alguns dos desafios atuais. Efetivamente, alguns destes desafios estão relacionados com o aumento da adesão das pessoas à tecnologia, personalização das aplicações, implementação rápida de novas funcionalidades, crescimento e complexidade das aplicações, otimização da produtividade das equipas de desenvolvimento, adequação das melhores tecnologias, entre outros. Deste modo, as arquiteturas orientadas a microsserviços resolvem alguns destes problemas atuais. Este projeto de dissertação tem como objetivo estudar as arquiteturas orientadas a microsserviços, os seus princípios, padrões e testar a aplicabilidade dos mesmos a um caso de estudo do mundo real, um sistema E-commerce, com a finalidade de resolver alguns dos problemas e desafios desta abordagem arquitetural. As categorias de padrões de arquiteturas orientadas a microsserviços estudadas são: Decomposição, Manutenção de Dados, Mensagens Transacionais, APIs Externas, Descoberta de Serviços e Segurança. Dos padrões estudados e implementados, os que tiveram resultados interessantes foram o Saga e CQRS, devido a resolverem problemas relacionados com a manutenção dos dados, que se torna complexa com a característica distribuída deste tipo de arquiteturas. A avaliação da aplicabilidade destes padrões ao caso de estudo, faz-se em comparação do desenho e implementação, com o estudo do estado de arte, através dos pontos positivos e negativos, bem como são efetuados testes à aplicação desenvolvida para conclusão de resultados. Por fim, foram efetuados testes de carga, para verificar a capacidade de escalabilidade, mas principalmente o impacto que as decisões arquiteturais têm na performance e disponibilidade das funcionalidades.

Palavras-chave: Arquiteturas de Software, CQRS, Microsserviços, Padrões de Desenho, Saga

Study on the importance of the Principles and Patterns of Microservices Architectures

Abstract

With the current growth of software development and increased demand for solutions of multiple problems, we start to verify the inability of certain architectural approaches to deal with some of the current challenges. Effectively, some of these challenges are related with people's adherence to technology, application customization, quick implementation of new features, application growth and complexity, productivity's optimization of the development teams, adaption of the best technologies, etc. The microservices architectures solve some of these current problems. This dissertation project has the objective to study the microservices architectures, their principles, patterns and test their applicability in one real world case study, an E-commerce system, to solve some problems and challenges of that architectural approach. The categories of microservices architecture patterns studied are: Decomposition, Data Management, Transactional Messaging, External APIs, Service Discovery and Security. Of the patterns studied and implemented, the ones that had interesting results were Saga and CQRS, due to solving problems related with data management, which becomes complex with distribution characteristic of these architectures. The applicability of these patterns to the case study is evaluated by comparing the design and implementation with the state of the art study, with positive and negative points, as well with application testing to get the conclusion results. Finally, were carried out charge tests, to verify the scalability capacity, but mainly the impact of the architecture decisions has in features performance and availability.

Keywords: Software Architecture, CQRS, Microservices, Design Patterns, Saga

Índice

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Acrónimos

2PC	**2-Phase Commit**
ACID	**Atomicity, Consistency, Isolation, Durability**
API	**Application Programming Interface**
CPU	**Central Processing Unit**
CQRS	**Command Query Responsibility Segregation**
CRUD	**Create, Read, Update and Delete**
DDD	**Domain-driven design**
GPU	**Graphics Processing Unit**
gRPC	**Remote Procedure Calls**
HTTP	**Hypertext Transfer Protocol**
IDE	**Integrated Development Environment**
IP	**Internet Protocol**
REST	**Representational state transfer**
SOA	**Service-oriented architecture**
URI	**Uniform Resource Identifier**

1. Introdução

Com o atual crescimento do desenvolvimento de software e aumento da procura para solução de diversos problemas, começa-se a verificar a incapacidade de certas abordagens arquiteturais para lidarem com alguns dos desafios atuais.

Efetivamente, alguns dos desafios atuais estão relacionados com o aumento da adesão das pessoas à tecnologia, personalização das aplicações, implementação rápida de novas funcionalidades, crescimento e complexidade das aplicações, otimização da produtividade das equipas de desenvolvimento, adequação das melhores tecnologias, entre outros. Deste modo, as abordagens arquiteturais, como por exemplo o monolítico, tem diversas dificuldades para lidar com alguns destes desafios [1].

Uma arquitetura orientada a microsserviços vem resolver alguns destes problemas, na medida em que, existe uma decomposição das responsabilidades, sendo isto um ponto importante para a dissipação da maior parte dos problemas.

Uma boa decomposição do sistema consiste em ter microsserviços desacoplados com responsabilidades bem definidas. Deste modo, permite uma maior produtividade das equipas responsáveis, bem como, a implementação de novas funcionalidades torna-se apenas dependente dos microsserviços envolvidos.

Do mesmo modo, este tipo de arquiteturas permite aplicar de forma mais trivial mecanismos de escalabilidade com o objetivo de aumentar a disponibilidade, atributos de qualidade essenciais na maior parte dos sistemas atuais.

Outra vantagem desta abordagem, consiste em permitir a adequação ou migração facilitada das melhores tecnologias de forma independente nos microsserviços.

Por outro lado, existem algumas desvantagens e desafios associados à implementação de uma arquitetura orientada a microsserviços.

Começando pela complexidade, uma vez que, existem mais componentes a desenhar, desenvolver e operar. Do mesmo modo, a decomposição de um sistema em microsserviços não é uma tarefa trivial e em caso de uma má divisão deixa-se de ter as vantagens referidas anteriormente, bem como outras que serão abordadas nos próximos capítulos.

Outra complexidade associada à implementação destas arquiteturas, consiste em resolver problemas associados com a consistência dos dados em transações e queries ao longo de múltiplos microsserviços. Neste tipo de operações, normalmente necessita-se de envio de mensagens como meio de comunicação entre microsserviços sendo um desafio garantir que seja feita de forma atómica com as operações.

Alguns dos problemas das arquiteturas orientadas a microsserviços resolvem-se com a aplicação dos padrões, em que o seu conhecimento torna-se imprescindível quando se pretende desenhar uma arquitetura deste tipo.

Assim, os objetivos desta dissertação centram-se em demonstrar o que são as arquiteturas orientadas a microsserviços, os seus princípios, padrões e capacidades para resolverem alguns dos problemas atuais no desenvolvimento de software.

1.1. Contextualização

Torna-se importante contextualizar qual o propósito desta dissertação. Deste modo, o propósito consiste em adquirir conhecimento sobre o estado de arte das arquiteturas orientadas a microsserviços.

Efetivamente, pretende-se entender em que consiste uma arquitetura orientada a microsserviços, quais os princípios que devem ser levados em conta no momento do desenho de uma arquitetura deste tipo. No entanto, dado que este conhecimento já se pode considerar como adquirido, será feita apenas uma breve abordagem nessa temática. Deste modo, a temática mais fulcral nesta dissertação consiste no estudo dos padrões das arquiteturas orientadas a microsserviços.

Deste modo, são estudados uma seleção de padrões das arquiteturas orientadas a microsserviços com a finalidade de perceber como e quais problemas resolvem para diferentes casos de uso e as suas vantagens e desvantagens.

De seguida, o objetivo consiste em selecionar um caso de estudo, isto é, um problema do mundo real para se efetuar o processo de desenvolvimento, designado *unified process* sobre o mesmo, com a finalidade de utilizar como abordagem arquitetural uma arquitetura orientada a microsserviços.

O processo inicia-se com a análise e levantamento dos requisitos, seguido do desenho da arquitetura do sistema onde serão aplicados os conhecimentos adquiridos, principalmente os padrões das arquiteturas orientadas a microsserviços.

De seguida, proceder-se-á à implementação da arquitetura desenhada, por forma a verificar-se a aplicabilidade da mesma num contexto prático.

São efetuados alguns testes, desde a implementação de uma aplicação cliente para testar a aplicabilidade do sistema backend implementado.

Por fim, são efetuados testes de carga para verificar algumas das vantagens das decisões arquiteturais efetuadas, desde a utilização de uma arquitetura orientada a microsserviços, mas principalmente a implementação de alguns padrões.

1.2. Objetivos

Nesta secção são apresentados os objetivos desta dissertação:

1. Estudo do estado da arte da temática sobre arquiteturas orientadas a microsserviços, os seus princípios e padrões mais relevantes.
2. Análise do domínio do caso de estudo.
3. Desenho da arquitetura do caso de estudo, isto é, uma arquitetura orientada a microsserviços.
    a. Verificar a resposta desta arquitetura a alguns dos problemas das arquiteturas alternativas.
    b. Implementar os princípios e padrões de arquiteturas orientadas a microsserviços mais relevantes para solucionar os problemas/desvantagens deste tipo de arquiteturas.
    c. Garantir mecanismos de escalabilidade para permitir aumentar a disponibilidade, entre outros atributos de qualidade.
4. Implementação do backend da arquitetura desenhada, onde será testada na prática a aplicação dos conhecimentos adquiridos no estudo do estado da arte, bem como a respetiva arquitetura.
5. Implementação da aplicação cliente para testar a aplicabilidade do backend implementado no caso de estudo.
6. Preparar o sistema da arquitetura para instalação (p.e. deployment utilizando Docker e GCP (Google Cloud Platform)).
7. Efetuar testes de carga às funcionalidades mais críticas (mais requeridas e as que devem ter tempos de resposta mais reduzidos) do caso de estudo.

8. Conclusões sobre a aplicabilidade dos conhecimentos adquiridos no estudo do estado de arte sobre arquiteturas orientadas a microsserviços.

1.3. Estrutura do documento

A dissertação está organizada em 8 capítulos.

O capítulo 2, começa com o estudo do estado de arte sobre as arquiteturas orientadas a microsserviços, iniciando-se por um teor histórico do surgimento destas com base nos autores predecessores da abordagem, Martin Fowler e James Lewis. De seguida, são apresentadas a definição destas arquiteturas, comparações às arquiteturas alternativas, bem como, vantagens e desvantagens.

Por conseguinte, abordam-se os padrões mais relevantes das arquiteturas orientadas a microsserviços, com início na perceção em que consiste um padrão, seguido da exposição das diferentes categorias de padrões, como se organizam, relacionam, entre outros pontos importantes. Em cada categoria são apresentados os problemas que as mesmas resolvem, bem como aborda-se cada padrão especificamente, desde a sua explicação, casos de uso, vantagens, desvantagens e em determinados casos as tecnologias mais recomendadas.

O capítulo 3, apresenta o caso de estudo selecionado, os objetivos que se pretendem com o mesmo, a análise de domínio, levantamento de requisitos, entre outros pontos importantes para a contextualização do caso de estudo.

O capítulo 4, apresenta a arquitetura, componentes de software e padrões genéricos a todos os microsserviços, por forma a não se repetir os mesmos para cada microsserviço.

O capítulo 5, apresenta cada microsserviço, organizados por subdomínios, onde são apresentadas as responsabilidades e arquiteturas da lógica de negócio de cada microsserviço. Ainda, nestes subdomínios de microsserviços são apresentados os padrões implementados entre os mesmos.

O capítulo 6, demonstra a estrutura, artefactos necessários e processo de instalação, do sistema implementado do caso de estudo.

O capítulo 7, apresenta os resultados e conclusões dos testes de carga efetuados às funcionalidades consideradas críticas do caso de estudo.

O capítulo 8, apresenta as conclusões finais desta dissertação relativamente aos objetivos apresentados anteriormente.

2. Estado da Arte

Neste capítulo, apresenta-se o estado da arte das temáticas sobre as arquiteturas orientadas a microsserviços e os padrões mais relevantes das mesmas.

Efetivamente, apresentam-se as arquiteturas orientadas a microsserviços, desde o surgimento destas, o que são, princípios e as suas vantagens e desvantagens, bem como a comparação com as arquiteturas alternativas.

De seguida, são descritas as categorias dos padrões mais relevantes sobre arquiteturas orientadas a microsserviços, no entanto, um subconjunto menor destes com alguns dos padrões cruciais para qualquer sistema.

Por fim, em cada categoria são apresentados os padrões mais relevantes com explicação do problema e como o resolvem, casos de uso, vantagens e desvantagens associadas aos mesmos, bem como as tecnologias mais recomendadas.

2.1. Arquiteturas orientadas a Microsserviços

O termo arquitetura orientada a microsserviços, surgiu nos últimos anos para descrever uma forma de desenhar sistemas de software como uma coleção de serviços independentes, desacoplados e facilmente instaláveis [2]. De acordo com James Lewis e Martin Fowler, predecessores da abordagem aos microsserviços, este tema foi discutido pela primeira vez em 2011 num workshop de arquiteturas de software [3]. No entanto, previamente a este acontecimento, alguns profissionais da indústria exploraram algumas ideias próximas deste tipo de arquiteturas, como por exemplo: Werner Volgels da Amazon[1] "*encapsulating the data with the business logic that operates on the data, with the only access through a published service interface*" [4] enquanto que Adrian Cockcroft da Netflix[2] "*loosely coupled service-oriented architecture with bounded contexts*" [5]. Outros termos utilizados na indústria nesta fase relacionados com arquiteturas *SOA* (*Service-oriented architecture*), tais como: "fine-grained SOA" e "SOA done right", revelam conceitos similares a arquiteturas orientadas a microsserviços, sendo provas do facto de que estes estão firmemente enraizados em SOA [2, 6].

[1] A amazon, é uma companhia multinacional de tecnologia, que se foca no e-commerce, cloud computing, digital streaming e inteligência artificial.

[2] A Netflix é uma plataforma de produção de conteúdos (filmes e séries), segundo uma arquitetura orientada a microsserviços, sendo desta forma, uma organização de renome nesta área.

De acordo com o apresentado nas referências [2, 3, 1, 7, 8, 9, 10], os microsserviços são uma abordagem para arquiteturas de software, que criam o conceito de modularização enfatizando os limites tecnológicos. Cada módulo, isto é, cada microsserviço, é desenvolvido e operado como um sistema pequeno e independente, permitindo o acesso à sua lógica interna e dados por uma API bem definida. A aplicação desta abordagem aumenta a agilidade, pois como já foi referido cada microsserviço torna-se numa unidade independente de desenvolvimento, *deployment,* operações, controlo de versões e escalabilidade.

O surgimento das arquiteturas orientadas a microsserviços ocorrem como uma alternativa ao estilo monolítico. O estilo arquitetural monolítico consiste em desenvolver um sistema em camadas bem definidas que apenas dependem da camada subsequente, no qual a interface do utilizador, a lógica de negócio e o acesso aos dados são combinados num único artefacto a partir de uma única plataforma [11]. No entanto, este estilo arquitetural contém algumas desvantagens em determinados domínios, principalmente com o crescimento e aumento de complexidade dos sistemas. Desta forma, o estilo arquitetural orientado a microsserviços resolve algumas dessas desvantagens e problemas.

Como qualquer abordagem, as arquiteturas orientadas a microsserviços tem as suas vantagens e desvantagens, desta forma, a sua aplicação deve ser ponderada, uma vez que, existem casos onde uma arquitetura monolítica torna-se a melhor escolha [12]. Ainda, existe o caso onde uma arquitetura orientada a microsserviços é adequada ao problema, no entanto, mal implementada, que por consequência origina um sistema ineficiente.

Em seguida, apresenta-se uma lista com algumas das vantagens mais importantes da implementação de uma arquitetura orientada a microsserviços [1]:

- Permitem o *continuous delivery, deployment* e desenvolvimento de sistemas grandes e complexos.
- Os serviços são pequenos e fáceis de manter.
- Os serviços são instalados independentemente.
- Os serviços são escaláveis independentemente.
- Arquiteturas orientadas a microsserviços permitem que as equipas sejam autónomas.
- Permite de forma fácil a experiência e adoção de novas tecnologias de forma independente.

- Os serviços têm um melhor isolamento em relação a faltas.

Uma das vantagens mais importantes de uma arquitetura orientada a microsserviços, consiste em a mesma permitir de forma mais trivial o *continuous delivery* e *deployment*, rápido, frequente e confiável de software de forma automática. O continuous delivery *e* deployment consiste em continuamente e de forma automática, colocar com frequência novas funcionalidades, alterações de configuração, correção de erros ("bugs"), efetuar experiências com novas tecnologias, entre outras atividades, em produção de forma confiável permitindo testar mais facilmente estas alterações, uma vez que, cada microsserviço executa o delivery e deployment independentemente.

Dada a dimensão dos microsserviços, os testes automáticos são triviais de escrever e eficientes de executar, resultando em menos erros para o sistema. Ainda permite aos programadores de um determinado microsserviço colocarem em produção alterações e decisões locais sem necessitar de coordenação com programadores de outros microsserviços tornando ágil o processo. No entanto, em outra perspetiva, necessita-se de manter uma boa comunicação e coordenação entre as equipas. Na verdade, as equipas devem ter presente que as suas alterações locais têm de ser coerentes e justificadas no contexto do sistema e objetivos de negócio, para que não o prejudiquem.

Resumidamente, esta vantagem permite às equipas de desenvolvimento serem mais autónomas no desenvolvimento, deployment e escalabilidade dos microsserviços em sistemas grandes e complexos.

Em relação ao tamanho dos microsserviços, existem múltiplas vantagens que contribuem para a produtividade dos programadores. Primeiramente, um serviço micro, corresponde a código reduzido, que por consequência, torna-se mais compreensível por parte dos programadores. Com um código base reduzido, verifica-se que as ferramentas de desenvolvimento, como por exemplo *IDEs* (*Integrated Development Environment*) não ficam pesadas e morosas. Por fim, o tamanho dos microsserviços permite que, a inicialização dos mesmos seja mais rápida, do que num sistema monolítico.

A escalabilidade torna-se um ponto forte das arquiteturas orientadas a microsserviços, dividindo-se em três vertentes, como se pode observar na Figura 1:

- **Eixo Y:** dividir funcionalidades, isto é, passar de uma arquitetura monolítica para orientada a microsserviços.
- **Eixo X:** duplicação horizontal, ou seja, aumento do número de instâncias.

- **Eixo Z:** partição dos dados por diferentes serviços

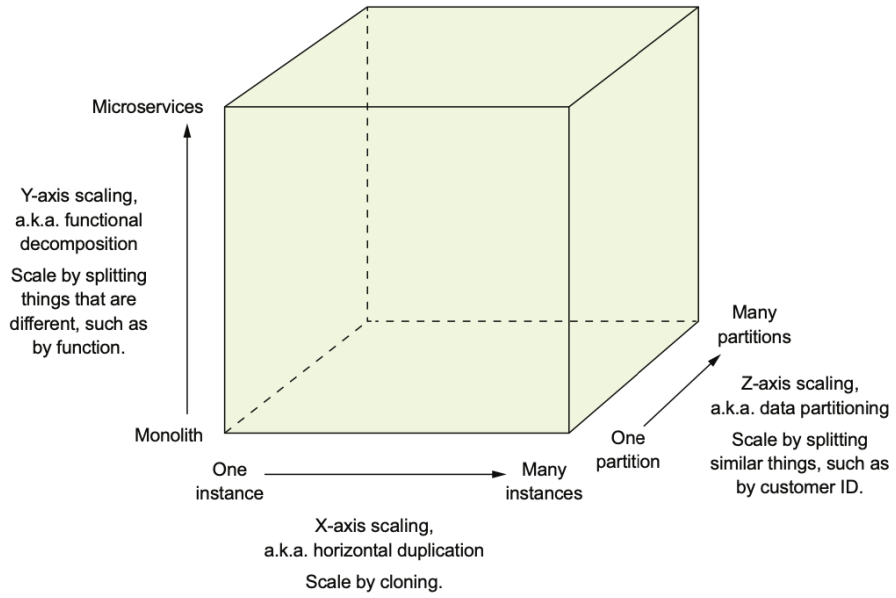


Figura 1: O cubo da escalabilidade, que define as três formas de escalar um sistema [1].

Cada microsserviço numa arquitetura orientada a microsserviços, pode ser escalável de forma independente dos outros. A escalabilidade nas suas vertentes distintas, permite adequar melhor os recursos, dado os requisitos (p.e. uma funcionalidade ou serviço que necessita mais de CPU do que GPU), adequar diferentes capacidades de atendimento de pedidos (p.e. existem funcionalidades mais requeridas que outras, logo a escalabilidade deve ser diferente), resistência a faltas e falhas de serviços, entre outras vantagens, que por outro lado, numa arquitetura monolítica torna-se mais difícil de conseguir.

A vantagem de isolamento, presente nos microsserviços, deve-se ao facto de cada unidade ser independente e isolada. Desta forma, o isolamento tem benefícios, como por exemplo a ocorrência de uma falha na memória em um microsserviço, apenas irá afetar esse microsserviço. Outros microsserviços vão continuar a responder a pedidos normalmente, ao contrário de um sistema monolítico, em que a falha de um componente irá afetar todo o sistema podendo levar à sua falha total.

Por último, mas não menos importante, outra vantagem dos microsserviços consiste na sua trivialidade na integração e migração de novas tecnologias individualmente em cada microsserviço. Desta forma, consegue-se adequar as melhores tecnologias face aos requisitos de cada microsserviço, quer a nível de lógica de negócio ou dados.

Em seguida, apresenta-se uma lista com algumas das desvantagens e problemas a ter em conta quando se decide implementar uma arquitetura deste tipo [1]:

- Decompor o conjunto correto de microsserviços é desafiante
- Sistemas distribuídos são complexos, tornando o desenvolvimento, teste e deployment difíceis
- Adicionar funcionalidades que envolvem múltiplos microsserviços necessitam de uma coordenação cuidadosa
- Decidir quando se deve aplicar uma arquitetura orientada a microsserviços é difícil
- Necessidade de mais recursos, do que uma arquitetura monolítica

Um dos desafios do desenho de arquiteturas orientadas a microsserviços consiste em decompor o sistema em microsserviços, pois não existe um algoritmo concreto bem definido para o fazer. Se um sistema for incorretamente decomposto, na verdade, o mesmo será um monolítico distribuído, isto é, um sistema com muitas dependências entre serviços, que dessa forma, deveriam ser colocados juntos. O grande problema de um monolítico distribuído consiste em o mesmo conter o conjunto das desvantagens de uma arquitetura orientada a microsserviços e monolítica.

Outro problema numa arquitetura orientada a microsserviços consiste na complexidade de ser um sistema distribuído. Os microsserviços necessitam de mecanismos para comunicarem entre si, sendo estes mais complexos de aplicar e implementar, do que numa arquitetura monolítica onde apenas necessita-se de chamar métodos. Adicionalmente, os microsserviços necessitam de ser desenhados para lidarem com falhas parciais, indisponibilidades e altas latências.

Outro desafio são as funcionalidades que constituem transações ou *queries* que abrangem múltiplos microsserviços, pois requerem o conhecimento de técnicas e padrões de desenho, como por exemplo a Saga, CQRS, API composition, entre outros, abordados nas seções seguintes, para garantir que as mesmas são executadas de forma a seguir um conjunto de propriedades, como a atomicidade, consistência, isolamento e durabilidade.

O desenvolvimento dos microsserviços em determinadas situações torna-se mais complexo, pois existe o problema de as ferramentas de desenvolvimento serem mais focadas para construção de sistemas monolíticos. Por exemplo, existe dificuldade na criação de testes que envolvem múltiplos microsserviços.

De acordo com apresentado na referência [1], motivado pelo facto de uma arquitetura orientada a microsserviços ser um sistema distribuído complexo, com numerosas instâncias, a gestão das mesmas introduz uma complexidade adicional, que devido à dimensão destas arquiteturas tende a ser o mais automatizada possível. Desta forma, destacam-se algumas ferramentas que auxiliam nesta automatização, tais como: Spinnaker[3], Docker Swarm[4], Kubernetes[5], Red Hat OpenShift[6], entre outras.

Tal como já foi referido em relação às vantagens, arquiteturas orientadas a microsserviços, permitem uma maior autonomia por parte das equipas de desenvolvimento. No entanto, noutra perspetiva existem alterações locais a um microsserviço que podem afetar outros que sejam dependentes deste. Nestes casos, isto é, quando as necessidades abrangem vários microsserviços, necessita-se de uma coordenação cuidadosa entre as equipas de desenvolvimento. Por outro lado, num sistema monolítico torna-se mais trivial proceder a uma alteração atómica que modifique vários componentes.

Por fim, outro desafio associado a arquiteturas orientadas a microsserviços consiste em decidir adequação de adotar esta abordagem. Numa fase inicial do sistema dificilmente encontram-se problemas ao aplicar uma arquitetura deste tipo. No entanto, a implementação destas, principalmente numa fase inicial, implica um desenvolvimento mais demorado em relação às arquiteturas alternativas. Desta forma, muitas *startups*, onde objetivo central consiste em quão rápido envolvem-se no modelo de negócio, isto é, desenvolver o MVP (Mínimo Valor de Produto), tendem a começar por sistemas monolíticos e mais tarde, caso se justifique, tentar migrar as mesmas para uma arquitetura orientada a microsserviços.

Assim, torna-se imprescindível ter o conhecimento das desvantagens das arquiteturas orientadas a microsserviços, pois comprometem o sucesso do sistema.

2.2. Padrões em Arquiteturas orientadas a Microsserviços

Previamente a listar e descrever os padrões de desenho para arquiteturas orientadas a microsserviços mais relevantes, torna-se importante inicialmente perceber o que são realmente e para que servem os padrões.

[3] https://spinnaker.io/
[4] https://docs.docker.com/engine/swarm/
[5] https://kubernetes.io/
[6] https://www.openshift.com/

Segundo Chris Richardson, autor do livro Microservice Patterns [1, 13], uma referência importante desta dissertação, um padrão consiste numa solução reutilizável para um problema conhecido que ocorre num contexto particular. Na verdade, os padrões consistem numa ideia que surgiu no mundo real das arquiteturas (p.e. desenho de edifícios) e após aplicada em software a nível de arquiteturas e desenho provou ser deveras útil.

Desde o surgimento das arquiteturas orientadas a microsserviços, foram propostos padrões para resolverem problemas recorrentes em determinados contextos, sendo os mesmos divididos em categorias. Segundo Chris Richardson, devem ser categorizados da seguinte forma:

1. Padrões de arquiteturas de aplicação
2. Decomposição
3. *Refactoring* para microsserviços
4. Manutenção de dados
5. Mensagens transacionais
6. Testes
7. Padrões para deployment
8. *Cross cutting concerns*
9. Estilo de comunicação
10. API externa
11. Descoberta de serviços
12. Confiabilidade
13. Segurança
14. Observabilidade
15. Padrões da interface do utilizador

A lista sugerida pelo autor contém diversas categorias importantes no desenho e desenvolvimento de uma arquitetura orientada a microsserviços. No entanto, para esta dissertação selecionou-se um subconjunto de categorias para abordar (1, 2, 4, 5, 10, 11, 13).

Adicionalmente às categorias, existem ainda relações entre os padrões como: predecessor, sucessor, alternativa, generalização e especialização, sendo algumas destas representadas na Figura 2.

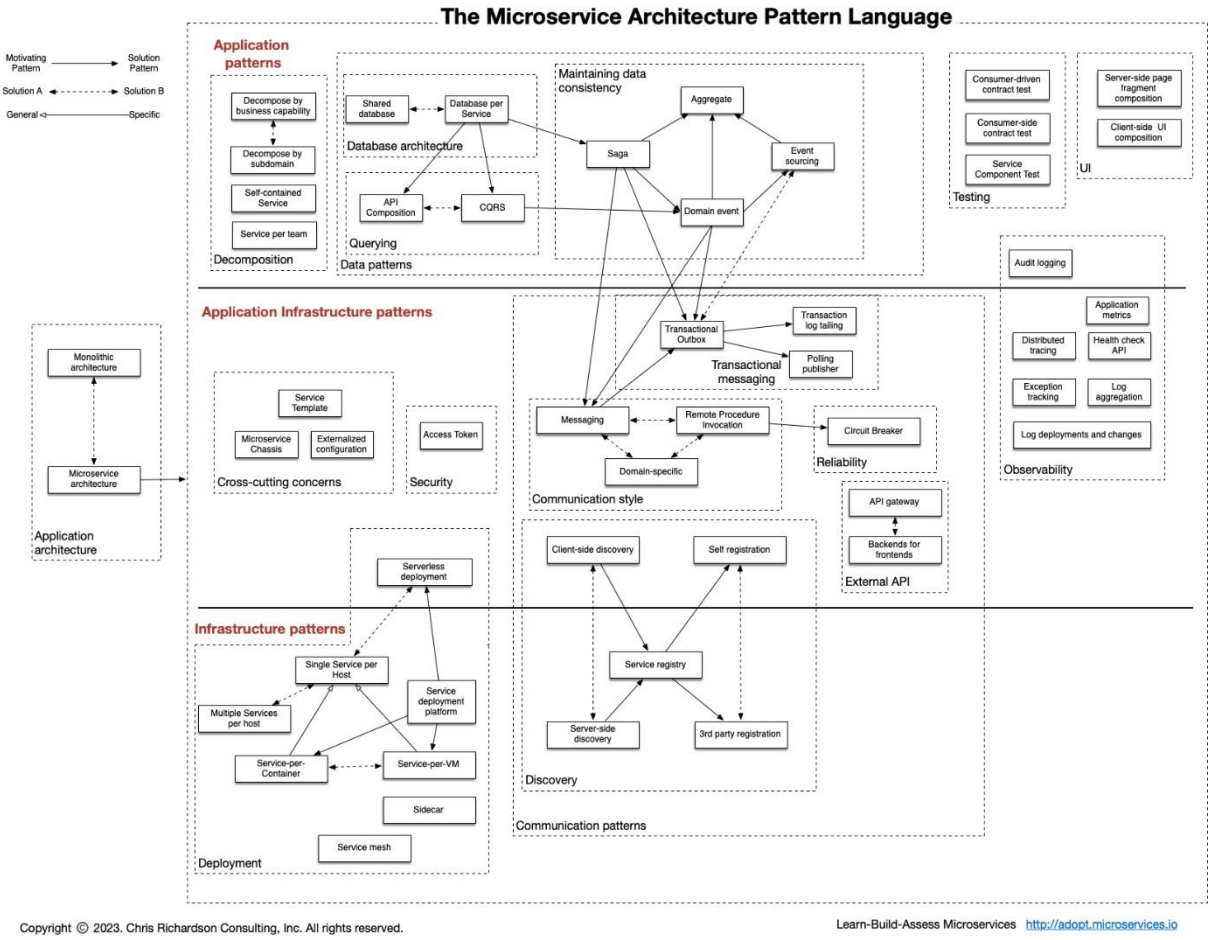


Figura 2: Categorias e relacionamentos dos padrões de microsserviços propostos por Chris Richardson [13].

Desta forma, a aplicação dos padrões tem o objetivo de resolver algumas das desvantagens descritas anteriormente, bem como alguns obstáculos que se encontram no desenvolvimento e implementação de uma arquitetura orientada a microsserviços, sendo alguns destes os apresentados de seguida:

- Latência da rede.
- Redução da disponibilidade em comunicações síncronas.
- Manter os dados consistentes ao longo de múltiplos microsserviços.
- Obter os dados consistentes.

De seguida, apresentam-se as categorias selecionadas e os respetivos padrões mais relevantes abordados nesta dissertação.

2.2.1. Decomposição

De acordo com o apresentado nas referências [1, 9, 13, 14], uma arquitetura orientada a microsserviços estrutura um sistema num conjunto de microsserviços. A determinação deste conjunto torna-se um desafio não existindo propriamente um algoritmo bem definido para o fazer. No entanto, sendo uma tarefa recorrente em todas as arquiteturas deste tipo, elaboraram-se alguns padrões que ajudam nessa divisão.

A existências destes padrões torna-se importante, pois uma boa decomposição dos microsserviços, permite que, alterações no futuro ou novos requisitos apenas afetem o menor conjunto possível de microsserviços. Do mesmo modo, todas as vantagens anteriormente apresentadas das arquiteturas orientadas a microsserviços, apenas se conseguem com uma boa decomposição dos mesmos.

Padrão Decompose by Business Capability

Tal como o nome do padrão sugere, a ideia deste consiste em dividir os microsserviços por capacidades de negócio [1, 9, 13, 14]. Uma capacidade de negócio, consiste numa atividade que gera valor para o mesmo correspondendo normalmente aos objetos de negócio. Desta forma, a fase de observação, análise e modelação do domínio, tornam-se muito importantes pois auxiliam no conhecimento das capacidades de negócio.

Supondo um sistema de uma loja online que contém a lista de produtos, detalhes dos mesmos, respetivas encomendas e informações de clientes. Desta forma, listam-se os típicos objetos de negócio (capacidades do negócio), que corresponderão a microsserviços, onde se verifica uma clara divisão de responsabilidades:

- Manutenção do catálogo dos produtos
- Manutenção do inventário
- Manutenção das encomendas
- Manutenção das entregas
- Manutenção dos clientes

Um ponto bastante importante a ter em conta em relação a esta divisão, consiste em que, este conjunto de serviços e objetos de negócio devem ser bastante estáveis, no entanto, cada um deles internamente pode mudar múltiplas vezes. Para descrever este ponto, o autor Chris Richardson [1] apresenta um exemplo sobre o depósito de cheques, onde há

relativamente pouco tempo o processo era feito presencialmente nos balcões dos bancos. Depois passou a ser um processo possível nas caixas de multibanco e atualmente é possível através de um *smartphone*. Estes exemplos mostram que em pouco tempo, a forma como determinadas ações e objetos de negócio são executados pode mudar drasticamente.

As vantagens de utilização deste padrão para decomposição dos microsserviços, são de uma arquitetura estável, visto que, as capacidades de negócio também o são. Permite às equipas de desenvolvimento serem multifuncionais, autónomas e organizadas em fornecer valor de negócio, ao contrário de características tecnológicas. Por fim, os microsserviços resultantes desta decomposição são mais coesos, isto é, contidos, bem definidos e desacoplados de outros microsserviços.

Por outro lado, existem alguns problemas associados a este padrão que se centram na necessidade do conhecimento do domínio. No entanto, como já foi referido anteriormente, através da observação, análise e modelação, atividades essenciais no desenvolvimento de software, será possível conhecer melhor as capacidades de negócio.

Padrão Decompose by Subdomain

Numa abordagem de modelação tradicional normalmente cria-se um único modelo que descreve todo o domínio, sendo desta forma desnecessariamente complexo. A dificuldade associada à utilização desta abordagem, deve-se ao facto de tentar que diferentes partes do negócio concordem com um único modelo, onde em certos casos, pode ser demasiado complexo para os requisitos dessas partes.

Desta forma, o *Domain-driven design* (DDD) [1] resolve este problema, pois define múltiplos modelos de domínio (subdomínios), cada um com diferentes âmbitos e objetivos, permitindo às partes do negócio adequarem o modelo de domínio aos seus requisitos. Daqui advém dois conceitos muito importantes do DDD, no contexto de arquiteturas orientadas a microsserviços: os subdomínios (equivalente a microsserviços) e os limites de contextos (limites dos microsserviços).

A aplicação do padrão Decompose by Subdomain [1, 9, 13, 14], passa por efetuar a divisão desses múltiplos domínios, que na realidade correspondem à decomposição do sistema em microsserviços e dos seus limites bem definidos, sendo uma alternativa ao padrão apresentado anteriormente.

Por forma a perceber-se a aplicabilidade deste padrão, recorre-se ao mesmo exemplo do sistema de uma loja online. Desta forma, ter-se-á de dividir o domínio da loja online por subdomínios, onde cada um destes corresponderá a um microsserviço, tal como se pode observar na Figura 3.

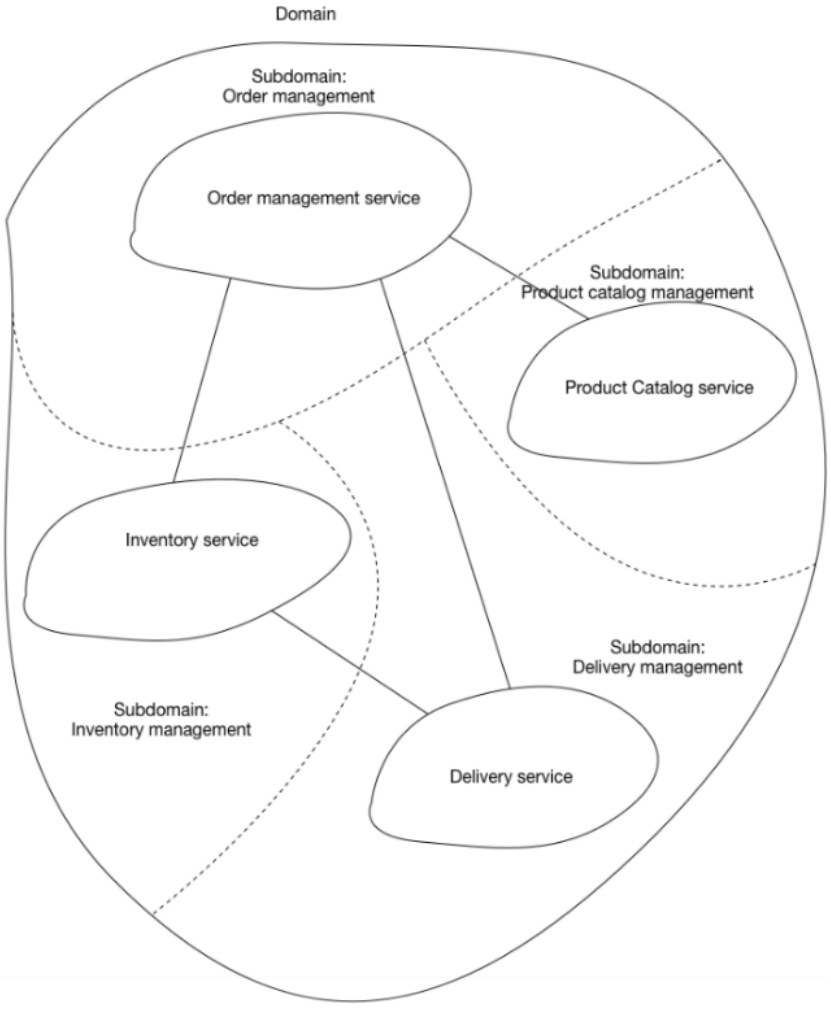


Figura 3: Decomposição por subdomínios do sistema de loja online [13].

As vantagens da aplicação deste padrão são idênticas às do padrão anterior, exceto na estabilidade da arquitetura, que neste caso, dependerá da estabilidade dos subdomínios.

Do mesmo modo, as desvantagens e problemas da aplicação deste padrão consiste na necessidade de um bom conhecimento do domínio para efetuar a divisão do mesmo em subdomínios.

Padrão Self-contained Service

O padrão *Self-contained Service* [1, 13] resolve o problema dos microsserviços que necessitam da colaboração de outros para responder a pedidos síncronos de um cliente, tornando este microsserviço independente.

Supondo um negócio de distribuição de comida, que utiliza um sistema composto pelos seguintes microsserviços:

- Microsserviço de Encomendas - responsável pelo processo de criar uma encomenda.
- Microsserviço do Restaurante - responsável pela lista de menus e preços.
- Microsserviço do Cliente - responsável pelo estado do cliente.
- Microsserviço da Cozinha - responsável por criar tickets para o chefe de cozinha.
- Microsserviço de Compras - responsável por autorizar o cartão de crédito do cliente.

Um cliente faz um pedido síncrono *HTTP* (*Hypertext Transfer Protocol*) de uma encomenda (POST /encomendas) ao microsserviço de encomendas, esperando uma resposta imediata (aproximadamente 600 milissegundos). Como referido anteriormente, a funcionalidade de criar uma encomenda, apesar de ser da responsabilidade do microsserviço de encomendas, a mesma necessita da colaboração de outros microsserviços.

Numa primeira abordagem, sem aplicar o padrão, a colaboração com outros microsserviços faz-se através de comunicações síncronas, tal como se pode observar na Figura 4. Na verdade, utilizar comunicações síncronas entre microsserviços reduz a disponibilidade dos mesmos, uma vez que, podem estar indisponíveis, levando a que a encomenda não seja criada retornando-se um erro para o cliente.

Ao retornar um erro, significa que a funcionalidade total de criar uma encomenda não está disponível para qualquer cliente. Desta forma, a melhor solução consiste em tentar resolver o problema ao invés de o devolver ao cliente, que neste caso, consiste em aguardar que os microsserviços voltem a estar disponíveis. Ainda em relação a esta abordagem, existe o problema de tempos de resposta morosos com espera ativa entre os microsserviços, bem como com o cliente, devido às comunicações síncronas.

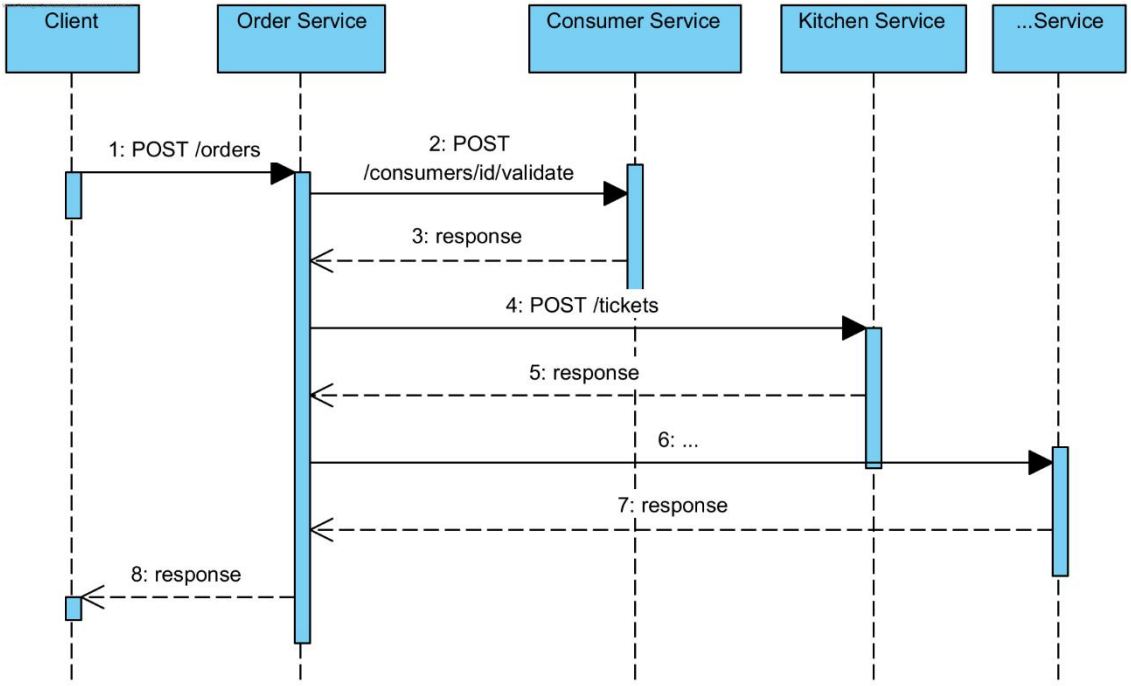


Figura 4: Criação de uma encomenda com colaboração síncrona [13].

A segunda abordagem, consiste em eliminar todas as comunicações síncronas entre o microsserviço de encomendas (microsserviço independente) e os seus colaboradores, como se pode ver na Figura 5, utilizando para isso outros padrões, tais como: Saga e CQRS, que serão abordados com detalhe nas próximas secções.

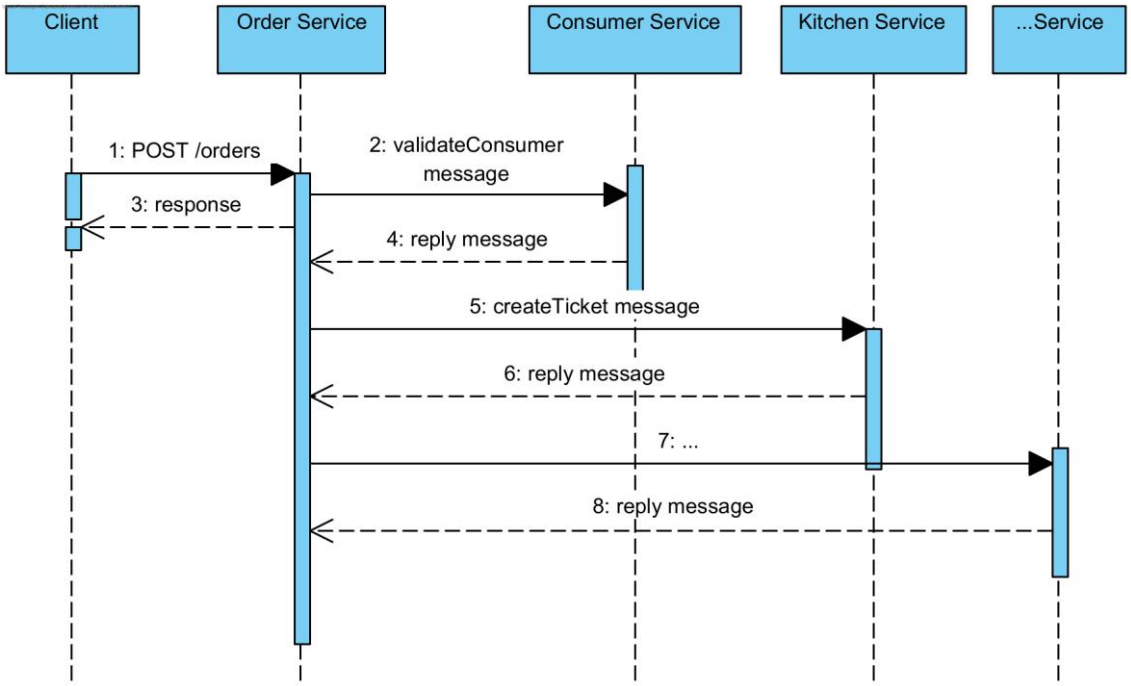


Figura 5: Criação de uma encomenda com colaboração assíncrona [13].

Deste modo, após invocar assincronamente o primeiro colaborador (Consumer Service), o microsserviço das encomendas pode responder de imediato ao cliente que o processo foi iniciado com sucesso, mas que ainda não está completo retornando um código HTTP 202[7] e os dados necessários. O estado do processo pode ser consultado pelo cliente ou ser o sistema a notificar o estado do mesmo.

As vantagens de aplicação do padrão Self-contained Service, consistem no aumento da disponibilidade dos microsserviços, uma vez que, com comunicações assíncronas o microsserviço de encomendas não aguarda a resposta dos colaboradores tornando-o mais independente. Do mesmo modo, a implementação de comunicações assíncronas permite ao cliente realizar outras tarefas, ao invés de aguardar pela resposta do microsserviço sendo principalmente benéfico para funcionalidades morosas.

Por outro lado, a aplicação deste padrão tem algumas desvantagens, como o aumento da complexidade e custo ao aplicar os padrões CQRS e Saga. Ao utilizar o padrão Saga, a forma de comunicação torna-se mais complexa de implementar ao invés de uma comunicação HTTP. Por fim, outra desvantagem consiste na necessidade de se desenvolver mais funcionalidades para aplicação do padrão, ao invés de focar apenas nos requisitos do microsserviço.

2.2.2. Manutenção de Dados

De acordo com apresentado nas referências [1, 13], a categoria de manutenção de dados contém um conjunto de padrões com o objetivo da resolução de problemas relacionados com a organização dos dados numa arquitetura orientada a microsserviços, bem como a execução de queries e transações que abrangem múltiplos microsserviços.

Uma arquitetura orientada a microsserviços, implica inevitavelmente transações e queries para operações que envolvem múltiplos microsserviços. Desta forma, uma transação consiste num conjunto de procedimentos a serem executados de forma sequencial, onde devem ser respeitadas as seguintes propriedades [15]:

1. **atomicidade:** todos os passos de uma transação são concluídos com sucesso para que esta seja efetivada

[7] https://tools.ietf.org/html/rfc2616#page-59

2. **consistência:** os dados estão num estado consistente quando a transação começa e acaba (p.e. transferência de fundos entre contas, a soma total tem de ser igual no início e fim da transação)
3. **isolamento:** estado intermédio das transações é invisível para outras, pelo que, transações concorrentes ou sequências obtém o mesmo resultado
4. **durabilidade:** os resultados de uma transação são permanentes, apenas podem ser alterados por transações subsequentes

Ao contrário de uma transação numa base de dados, transações para operações que envolvem múltiplos microsserviços não garantem as propriedades acima descritas, pelo que, são necessárias abordagens por forma a conseguir as mesmas.

Do mesmo modo, as queries utilizadas na obtenção de dados que podem ser de múltiplos microsserviços necessitam de garantir a propriedade de consistência.

Padrão Database per Service

De acordo com as referências [1, 13], o problema que o padrão Database per Service resolve consiste em como definir a arquitetura da base de dados, isto é, como são persistidos os dados de um sistema que tem uma arquitetura orientada a microsserviços. Na verdade, uma abordagem inicial seria colocar uma única base de dados partilhada por todos os microsserviços, no entanto, esta contraria o princípio da independência e desacoplamento dos mesmos.

A segunda abordagem, referente à aplicação do padrão Database per Service, consiste em definir uma base de dados privada para cada microsserviço. Na verdade, a base de dados passa a ser um componente do microsserviço, uma vez que, nenhum outro microsserviço pode aceder diretamente à mesma, apenas pela API que este expõe.

Importa realçar que, a aplicação deste padrão não necessita diretamente de um servidor de base de dados por cada microsserviço, uma vez que, existem diferentes formas de garantir os dados persistentes privados, tais como:

- Tabelas privadas por microsserviço (em base de dados relacionais)
- *Schema* por microsserviço
- Servidor de base de dados por microsserviço;

A implementação de tabelas privadas ou *schema* por microsserviço partilham algumas das vantagens, tais como, melhor aproveitamento dos recursos, bem como diminuição do *overhead*.

Por outro lado, as desvantagens consistem em não permitirem uma melhor adequação dos recursos e tecnologias de base de dados face aos requisitos dos microsserviços. Nestas duas abordagens, acresce ainda a complexidade e coordenação necessária para garantir que não são quebrados os limites dos microsserviços. Isto é, não poderá haver interação pela base de dados entre os dados privados de diferentes microsserviços.

A implementação de um servidor de base de dados por microsserviço, torna-se a forma mais viável pois permite apropriar a melhor tecnologia face aos requisitos dos microsserviços. Outra vantagem consiste em não haver partilha de capacidade de resposta com as bases de dados de outros microsserviços. Do mesmo modo, verifica-se a possibilidade de aplicar determinados atributos de qualidade de forma individual aos respetivos servidores de bases de dados de cada microsserviço, tais como: escalabilidade, resiliência, disponibilidade, entre outras. Por fim, garante o desacoplamento e independência, uma vez que, alterações na base de dados de um microsserviço não terá impacto em outros.

Por outro lado, as desvantagens associadas são a necessidade de mais recursos, bem como o aumento da complexidade, uma vez que se torna necessário manter, controlar e colocar em produção mais componentes para a arquitetura.

Padrão Saga

De acordo com o apresentado nas referências [16, 1, 13, 17, 18, 19, 20, 21, 22], implementar transações que abrangem múltiplos microsserviços torna-se bastante complexo, uma vez que, cada microsserviço contém uma base de dados privada.

Uma possível solução seriam transações distribuídas, no entanto, não devem ser utilizadas no contexto de uma arquitetura orientada a microsserviços, em consequência de utilizarem um algoritmo bastante penoso, demorado e complexo, denominado *2PC* (*Two-Phase commit*), tornando-se pior com o aumento de microsserviços. Do mesmo modo, as transações distribuídas não são triviais para garantir a consistência dos dados, utilizam comunicações síncronas que reduzem a disponibilidade [23, 24]. Ainda, algumas tecnologias

mais recentes não suportam as transações distribuídas, como por exemplo *RabbitMQ*[8] e *Apache Kafka*[9], causando uma limitação na escolha das mesmas.

Desta forma, para solucionar o problema surgiu o padrão Saga para manter os dados consistentes numa arquitetura orientada a microsserviços sem utilizar as transações distribuídas. Desta forma, define-se uma saga para cada operação que necessite de atualizar múltiplos microsserviços, também denominados por participantes.

A saga consiste numa sequência de transações locais realizadas pelos participantes envolvidos na mesma. Cada microsserviço executa uma transação local na própria base de dados e publica uma mensagem ou evento para ativar a próxima transação local em outro participante, como se pode observar na Figura 6.

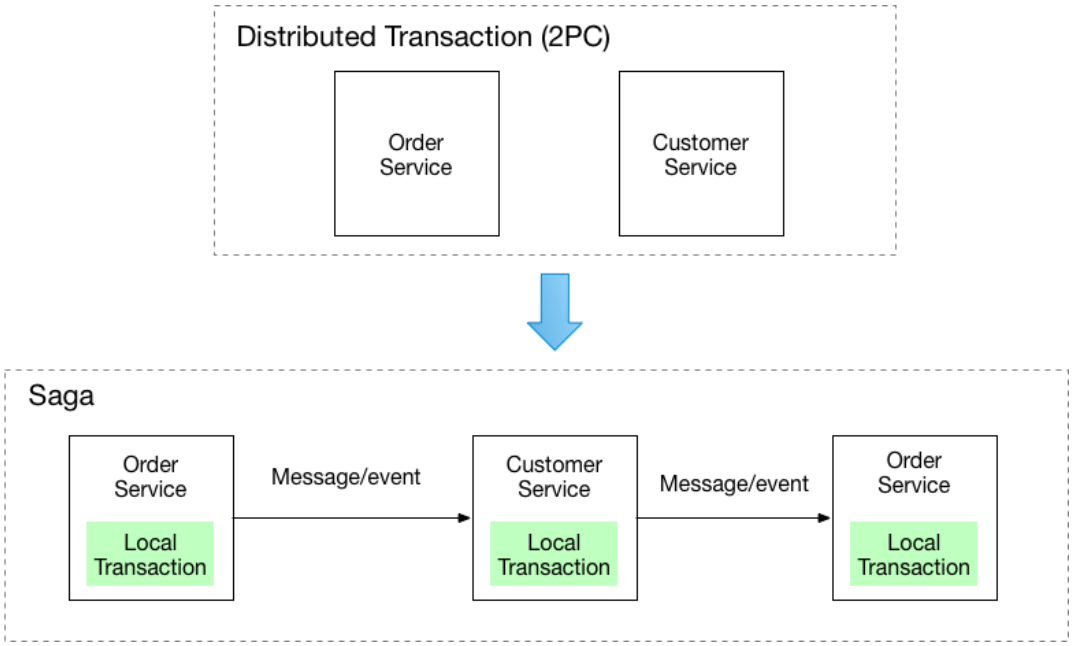


Figura 6: Saga num exemplo de uma loja online [13].

A lógica necessária para implementação do padrão da Saga consiste na seleção e coordenação dos passos dos participantes. Deste modo, a saga ativa-se devido a um comando do sistema, implicando que o coordenador da saga defina e invoque o primeiro participante a executar a transação local. Após a conclusão desta transação local, o coordenador da saga seleciona e invoca o próximo participante.

[8] https://www.rabbitmq.com/
[9] https://kafka.apache.org/

Se um participante obtiver uma falha a nível de negócio, aquando da execução da transação local, o coordenador da saga executa um conjunto de transações de compensação para reverter as alterações feitas pelos microsserviços precedentes. As mesmas correspondem a executar uma ação ou funcionalidade contrária à que foi executada, como se pode observar na Tabela 1. Desta forma significa que, se os participantes subsequentes são concorrentes a falhas de negócio, deve-se garantir que, as transações locais precedentes tenham uma transação de compensação correspondente.

Passo	**Participante**	**Transação local**	**Transação de compensação**
1	Serviço de Encomenda	criarEncomendaPendente()	rejeitarEncomenda()
2	Serviço Cliente	reservarCredito()	-
3	Serviço de Encomenda	aprovarEncomenda()	-

Tabela 1- Exemplo de processo saga, com transações locais e transações de compensação [13].

Existem duas abordagens distintas de coordenar o processo da saga, a coreografia e orquestração. As grandes diferenças entre as abordagens consistem no tipo de coordenação e comunicação assíncrona.

A coordenação destas abordagens implica uma comunicação assíncrona entre microsserviços, uma vez que, contém a vantagem de garantir que todos os procedimentos da saga são executados, até mesmo quando um ou mais microsserviços estão temporariamente indisponíveis, tornando-os mais independentes e desacoplados.

A abordagem **coreografia**, define-se pela coordenação descentralizada, bem como a comunicação assíncrona ser orientada a eventos.

Um evento representa uma alteração do estado de um determinado objeto, sendo focado no emissor do mesmo, isto é, a alteração do estado refere-se a um objeto do emissor, tal como se pode observar na Figura 7 (os triângulos representam eventos).

A abordagem define-se pelo facto de não haver uma coordenação centralizada, na verdade, cada participante da saga após concluir a transação local determina qual o próximo participante, isto é, envia um evento para o mesmo.

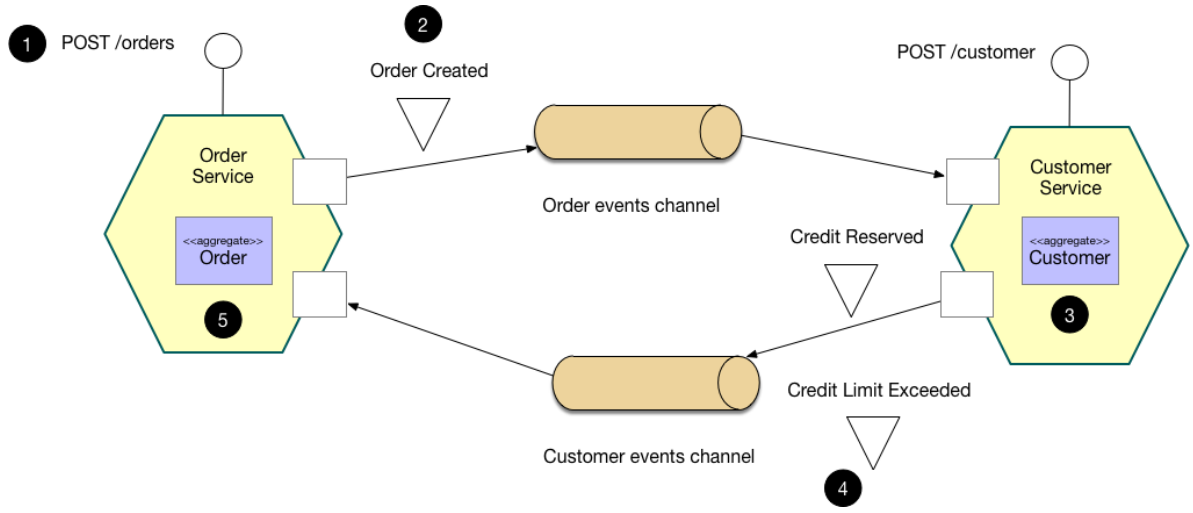


Figura 7: Saga seguindo a coordenação em coreografia [13].

Supondo um sistema onde exista uma gestão de encomendas, a implementação de uma saga seguindo uma coordenação em coreografia para a funcionalidade de "criação de uma encomenda" processa-se da forma representada na Figura 7. Deste modo, o processo inicia com um pedido HTTP POST /encomendas (passo 1) à API do microsserviço de encomendas, por este motivo, a primeira transação local ocorre neste microsserviço. Após a criação da encomenda num estado pendente, o microsserviço publica um evento (passo 2) para a fila de encomendas, denominado por "encomenda criada", que representa o estado atual da encomenda.

O microsserviço do cliente, como subscreve o evento denominado por "encomenda criada", após este ser emitido consome o mesmo e verifica o limite do crédito do cliente associado à encomenda, implicando a reserva ou não do crédito (passo 3). Neste caso, o microsserviço do cliente pode publicar dois eventos distintos (passo 4) para a fila de eventos do cliente, o "crédito foi reservado" ou "limite do crédito foi excedido".

O microsserviço de encomendas, como subscreveu os eventos anteriormente referidos, irá consumi-los quando estes forem emitidos e com base no estado destes verifica se a encomenda pode ser aprovada ou rejeitada.

Uma das vantagens da aplicação desta abordagem consiste na simplicidade, uma vez que, os microsserviços apenas publicam eventos quando criam, alteram ou apagam objetos de negócio. Outra vantagem concerne num maior desacoplamento e independência dos microsserviços, uma vez que, estes subscrevem-se a eventos e não a microsserviços. Dada a

sua simplicidade, esta abordagem caracteriza-se por ser mais eficiente em termos de tempo de execução em comparação com a abordagem orquestradora. Deste modo, deve-se optar pela mesma quando existe esse mesmo requisito.

Por outro lado, existem desvantagens como uma maior dificuldade de entender esta abordagem, uma vez que, a lógica da saga distribui-se pelos vários participantes, não permitindo um conhecimento geral do processamento desta, bem como possibilita a existência de ciclos de dependências.

Em resumo, a abordagem coreografia deve ser utilizada apenas para sagas simples, isto é, com um número reduzido de microsserviços envolvidos, como consequência das suas desvantagens. Esta abordagem deve ser equacionada também quando a performance da saga é um requisito. Sagas mais complexas, torna-se melhor a adoção da abordagem orquestradora.

A abordagem **orquestração**, define-se pela coordenação centralizada, bem como a comunicação assíncrona ser orientada a mensagens.

Uma mensagem pode ser do tipo comando ou resposta. Uma mensagem do tipo comando, corresponde a uma funcionalidade a executar, agregada de um conjunto de dados enviados para um microsserviço destino. Uma mensagem do tipo resposta contém os dados resultantes da execução de uma operação. A abordagem das mensagens ao contrário da orientada a eventos foca-se no destinatário da mesma, isto é, o comando a executar e os dados são aplicados no destinatário, tal como se pode observar na Figura 8.

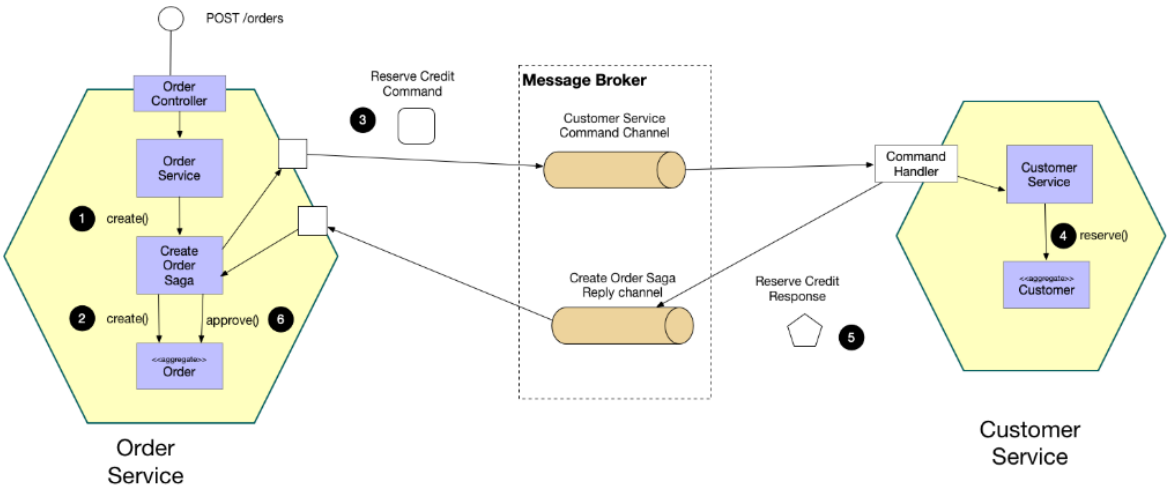


Figura 8: Saga seguindo a coordenação em orquestração (os quadrados representam mensagens de comando, e pentágonos mensagens de resposta) [13].

A abordagem define-se pelo facto de haver uma coordenação centralizada, isto é, existe um microsserviço responsável por coordenar os participantes da saga denominado por orquestrador.

O microsserviço orquestrador pode ser um microsserviço exclusivamente responsável pela orquestração ou o microsserviço com funcionalidade base a ser executada na saga, isto é, ao qual se faz o primeiro pedido. Após concluir a transação local no microsserviço orquestrador, este determina qual o próximo participante, isto é, envia uma mensagem do tipo comando com os dados necessários.

Supondo o mesmo sistema, onde existe uma gestão de encomendas e uma funcionalidade para "criação de uma encomenda", o processo inicia-se com um POST /encomendas à API do microsserviço de encomendas. Uma vez que o microsserviço de encomendas orquestra a saga de "criação da encomenda", o mesmo inicializa o processo de coordenação (passo 1). De seguida, a primeira transação local ocorre neste microsserviço (passo 2). Após a criação da encomenda num estado pendente (passo 2), o microsserviço envia uma mensagem do tipo comando (passo 3) para o canal de entrada do microsserviço do cliente.

O manipulador de comandos do microsserviço cliente, após receber a mensagem na fila de entrada verifica que funcionalidade corresponde à mensagem comando. Após efetuar a verificação, executa a funcionalidade correspondente que consiste na "verificação e reserva do crédito" (passo 4). De seguida, envia uma mensagem de resposta para a fila de respostas da saga de criação de encomenda (passo 5).

O microsserviço de encomendas, ao receber a mensagem de resposta, verifica os dados da mesma aprovando ou rejeitando a encomenda (passo 6). Por fim, altera-se o estado pendente da encomenda para aprovada ou rejeitada.

É importante realçar que, neste exemplo a saga contém apenas dois participantes, sendo que um deles também tem o papel de orquestrador, no entanto, caso fosse um conjunto maior, o orquestrador, isto é, o microsserviço de encomendas, após receber a resposta do microsserviço do cliente determinaria qual o próximo passo da saga, isto é, o próximo microsserviço participante.

A vantagem de aplicação desta abordagem consiste na simplicidade de dependências, isto é, o orquestrador invoca os participantes, mas estes não o invocam. Desta forma, apenas

existe dependência unidirecional entre o orquestrador e os participantes, que ao contrário da abordagem anterior não ocorrem ciclos de dependências.

Outra vantagem consiste no desacoplamento, uma vez que, cada microsserviço implementa uma API que o orquestrador invoca, e desta forma, os mesmos não necessitam de conhecer as mensagens publicadas pelos participantes.

A separação de conceitos e simplificação da lógica de negócio torna-se uma vantagem em relação à abordagem alternativa, uma vez que, os objetos desta apesar de participarem na mesma não conhecem ou interferem no processo saga. Deste modo, o processo de desenvolvimento da lógica de negócio torna-se mais simples.

Por outro lado, a desvantagem desta abordagem consiste na centralização da orquestração, uma vez que, torna-se um ponto único de falha. Caso o orquestrador fique indisponível todo o processo da saga será interrompido.

Um problema que existe quando se implementa uma saga, consiste em garantir a atomicidade da transação local e publicação da mensagem ou evento, respetivamente nas abordagens orquestração e coreografia. No entanto, existem padrões que resolvem este problema e serão abordados nas secções seguintes.

As propriedades atomicidade, consistência e durabilidade são garantidas com o padrão da saga, no entanto, o mesmo não acontece com o isolamento. Na verdade, esta falha de isolamento deve-se ao facto de cada alteração feita por cada transação local da saga ficar imediatamente visível para outras. No entanto, existe um conjunto de contra medidas para contrariar esta falha, das quais o *semantic lock* [1, 18]. A contramedida *semantic lock* a nível aplicacional consiste em colocar um atributo de controlo em cada registo, que permita verificar se o mesmo está bloqueado por alguma saga.

Em resumo, esta abordagem deve ser equacionada em situações em que a diferença de performance em relação à abordagem alternativa não é um problema. Por outro lado, esta abordagem é adequada para cenários complexos.

Padrão API Composition

De acordo com o apresentado nas referências [1, 7, 13, 18, 25, 9], o padrão API Composition tem como objetivo implementar queries que agreguem dados provenientes de múltiplos microsserviços. Na verdade, o padrão implementa uma operação *query* que invoca um conjunto de queries em microsserviços, combinando os resultados das mesmas.

Existem dois tipos de participantes nesta abordagem, o microsserviço compositor da API e o fornecedor. O microsserviço compositor da API tem a responsabilidade da exposição e implementação da operação query, que obtém os dados agregados através de invocações de várias queries nos microsserviços fornecedores. O microsserviço fornecedor expõe uma API, com a finalidade de fornecer os dados necessários ao compositor, para que este os agregue aos dados de outros microsserviços fornecedores.

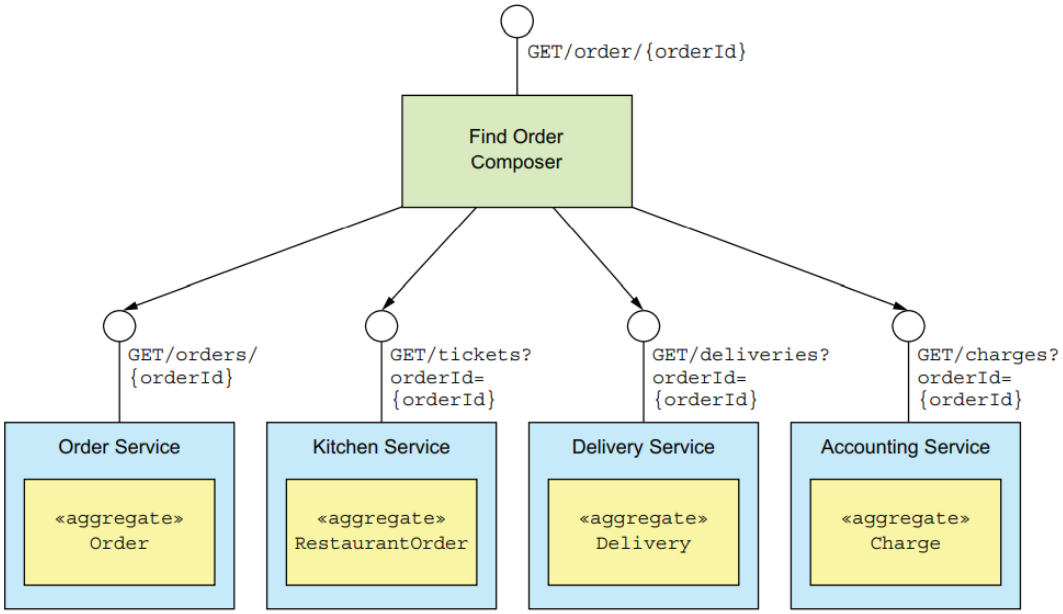


Figura 9: Aplicação do padrão API Composition a um sistema de entregas de refeições, com gestão de encomendas, para obter a informação de uma encomenda, que abrange múltiplos serviços [1].

Supondo o exemplo descrito na Figura 9, que consiste na funcionalidade de "obter os dados de uma encomenda" num sistema de entregas de refeições.

O microsserviço compositor da API (figura esverdeada), denominado por "obter encomenda", expõe uma operação query (GET /encomendas/{IdEncomenda}) numa REST API responsável por obter todos os dados de uma encomenda. No entanto, obter os dados totais de uma encomenda abrange múltiplos microsserviços. Desta forma, o compositor tem a responsabilidade de invocar esses microsserviços com a finalidade de obter as agregações simples dos dados de cada fornecedor, através da chave estrangeira (id da encomenda). Após obter todas as agregações simples de cada microsserviço fornecedor, o microsserviço compositor através do id da encomenda procede à combinação dessas agregações numa única agregação, retornando o resultado ao cliente.

A vantagem associada a este padrão consiste na simplicidade de implementação por parte da equipa de desenvolvimento, bem como da execução por parte do cliente, uma vez que, obtém todos os dados de um registo, neste caso da encomenda, proveniente de diferentes microsserviços com apenas um pedido.

Por outro lado, uma desvantagem associada ao padrão consiste no aumento de peso computacional no lado do *backend*, uma vez que, são necessários mais pedidos e queries à base de dados para obter dados proveniente de diversos microsserviços. Desta forma, implicará mais recursos de computação e rede ao contrário de uma arquitetura monolítica onde apenas se necessita de um pedido.

Outra desvantagem, consiste na redução de disponibilidade, uma vez que, a mesma diminuí com o aumento de microsserviços envolvidos numa funcionalidade.

Supondo que a disponibilidade média de um microsserviço consiste em 99.5% durante um ano, analisando o exemplo acima descrito na Figura 9, a disponibilidade total da funcionalidade que envolve um microsserviço compositor e quatro microsserviços fornecedores de API pode ser calculada da seguinte forma:

$$\boldsymbol{disponibilidade_{microsserviço}} = \mathbf{99.5}\% \ (\boldsymbol{ano})$$

Cálculo 1:

$N\º\ microsserviços = 5$

$disponibilidade_{obter\ encomenda} = 0.995^{\ 5} \times 100 = 97.5\ \%\ (ano)$

$indisponibilidade_{obter\ encomenda} = (1 - 0.975) \times (N\º\ horas\ ano) \approx\ 36\ min/dia$

Cálculo 2:

$N\º\ microsserviços = 1$

$disponibilidade_{obter\ encomenda} = 0.995^{\ 1} \times 100 = 99.5\ \%\ (ano)$

$indisponibilidade_{obter\ encomenda} = (1 - 0.995) \times (N\º\ horas\ ano) \approx\ 7\ min/dia$

Analisando os resultados dos cálculos acima constata-se que, existe uma redução da disponibilidade da funcionalidade "obter encomenda" devido ao aumento da dependência de microsserviços envolvidos para a execução da mesma.

Uma forma de ultrapassar a desvantagem da disponibilidade, consiste em ter dados em *cache*, para que, em caso de falha possam ser utilizados. No entanto, estes podem estar desatualizados. Outra forma consiste, no caso de falha de um dos microsserviços fornecedores, o microsserviço compositor da API pode omitir os dados do mesmo, uma vez que, em certas situações os restantes podem ser suficientes para o cliente.

Por fim, apesar da simplicidade da aplicação do padrão, existem alguns problemas associados com a sua implementação, que consiste em decidir que componente da arquitetura corresponderá ao compositor da API, bem como a implementação da lógica das agregações de forma eficiente.

Padrão *Command and Query Responsability Segregation* (CQRS)

De acordo com o apresentado nas referências [1, 13, 18, 26, 27, 28, 29, 9], os microsserviços normalmente são compostos por uma base de dados e um modelo para as operações de criação, leitura, atualização e remoção, normalmente designadas por *CRUD* (*Create, Read, Update and Delete*), ocorrendo em determinados contextos um modelo extra para operações de leitura específicas, como se pode observar à esquerda da Figura 10.

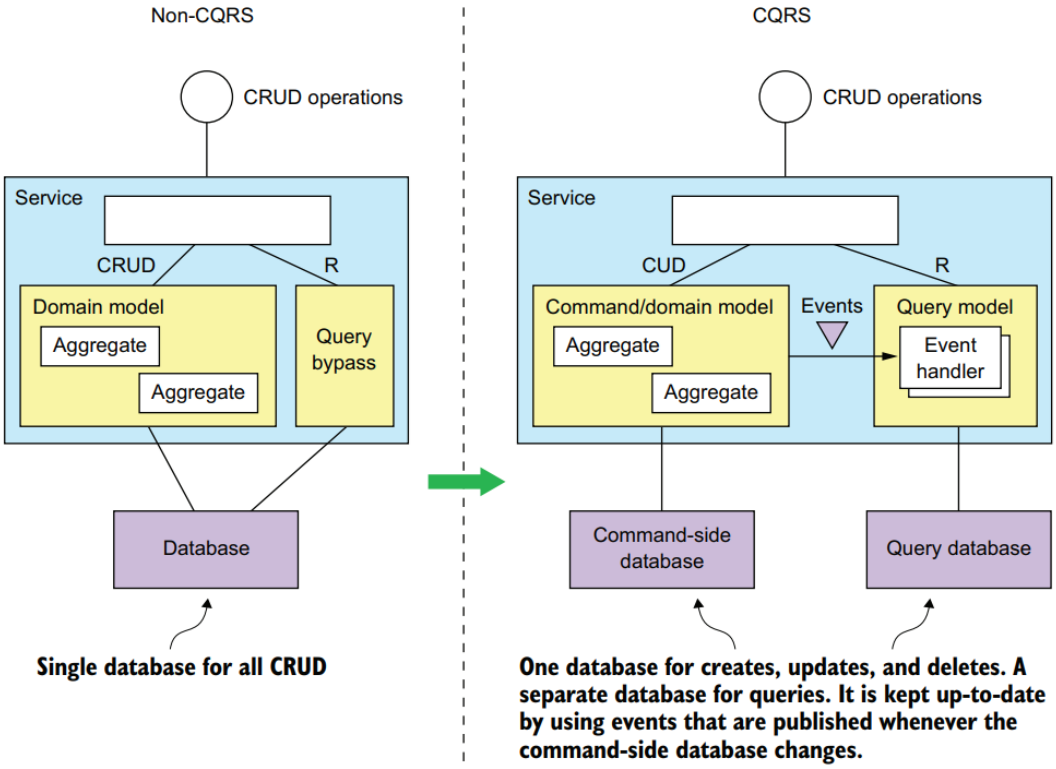


Figura 10: Padrão Non-CQRS versus CQRS [1].

Existem três problemas comuns que ocorrem quando se implementam operações de leitura numa arquitetura orientada a microsserviços:

1. Aplicação desadequada do padrão API composition, abordado anteriormente, uma vez que, o microsserviço compositor efetua a combinação (*joins*) de numerosos dados em memória oriundos de múltiplos microsserviços de forma morosa e ineficiente.
2. Um microsserviço que persiste os dados numa estrutura ou tecnologia não adequada e ineficiente para execução de uma determinada operação de comando ou leitura.
3. Falta de separação de conceitos, uma vez que, apesar de um microsserviço conter os dados requeridos para uma operação, não implica que a implementação desta seja da responsabilidade do mesmo.

O padrão *CQRS* (*Command and Query Responsability Segregation*) resolve estes problemas, na medida em que, num domínio de dados separa as operações de criação, atualização e remoção (operações de comando) das de leitura (*query*), em modelos e base de dados diferentes, como se pode observar à direita da Figura 10. Segundo Betrand Meyer, autor do livro *Object-Oriented Software Construction* [28], tem a seguinte afirmação, "*asking a question should not change the answer*", para demonstrar que não existe problema em separar as operações de comando das operações de leitura, uma vez que, as de leitura não alteram o estado dos dados. Do mesmo modo, a aplicação desta abordagem pode originar a separação das operações a nível de microsserviços, isto é, implementação de um microsserviço para leituras (*query service*), que será abordado mais adiante.

A aplicação do padrão CQRS implica a sincronização da base de dados correspondente às leituras com a de operações de comando. Desta forma, a cada operação de criar, atualizar e remover, o modelo de comando publica um evento, previamente subscrito pelo modelo das leituras, para que sincronize a base de dados do mesmo a cada alteração.

Torna-se importante realçar que, apesar de haver uma separação das operações criar, atualizar e remover, das de leitura, o modelo de comando poderá conter algumas operações de leitura simples baseadas na chave primária, que não impliquem a junção (joins) de numerosas quantidades de dados.

Em relação ao primeiro problema apresentado anteriormente, o mesmo refere-se ao contexto de uma desadequada implementação do padrão API composition. Na verdade,

ocorre quando o microsserviço compositor necessita de efetuar junção e filtragem de numerosos dados em memória, devido ao facto de que, os microsserviços invocados não o fazem nas suas bases de dados, tornando a operação principal de leitura bastante morosa e ineficiente.

Exemplo disso são operações de leitura dado um conjunto de atributos como critério, que não estão presentes em todos os microsserviços invocados no API composition.

Supondo a funcionalidade de "obter a lista de encomendas de um cliente" dado um conjunto de palavras-chave, através da implementação de um API Composition que envolve múltiplos microsserviços, tal como se pode observar na Figura 11.

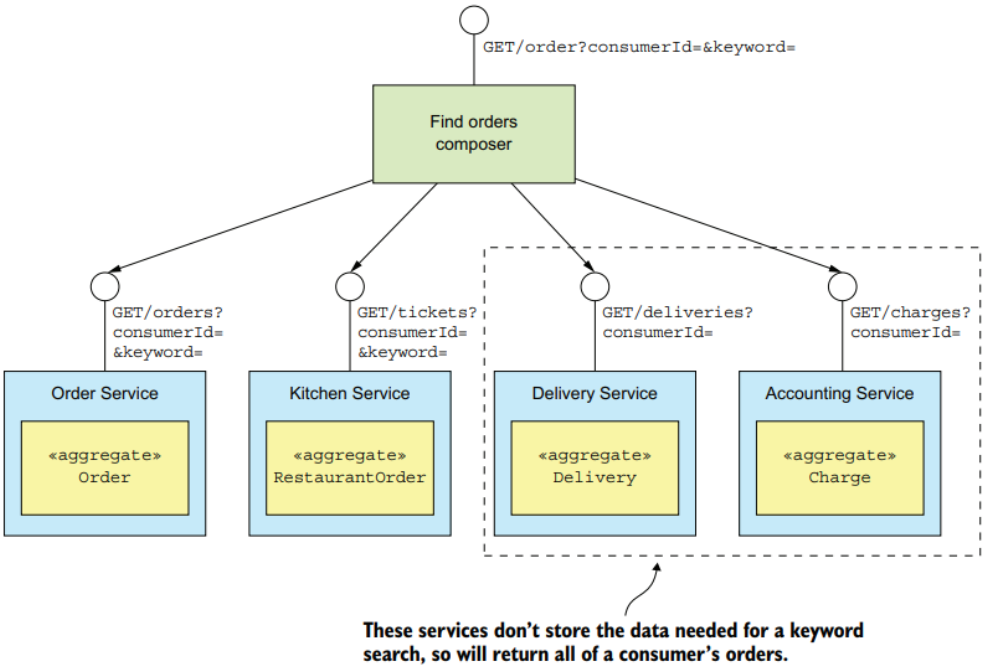


Figura 11: Aplicação incorreta do padrão do API Composition [1].

Como os microsserviços de "entregas" e "pagamentos" não conseguem efetuar a operação de leitura através das palavras-chave nas suas bases de dados, implica que, serão retornados para o microsserviço compositor todos os registos associados ao cliente.

Devido a este facto, no microsserviço compositor (figura esverdeada), acresce à funcionalidade normal a complexidade de efetuar em memória a filtragem dos numerosos dados provenientes dos microsserviços de "entrega" e "pagamentos", com os dados já filtrados pelos microsserviços de "encomendas" e "cozinha", seguida da funcionalidade

normal de junção dos resultados. Desta forma, este acréscimo de complexidade neste tipo de contextos, torna a aplicação do padrão API Composition ineficiente.

A resolução do problema apresentado anteriormente, passa por aplicar o padrão CQRS como alternativa ao API composition. Na verdade, a aplicação deste padrão neste contexto, consiste na implementação de um microsserviço de leitura (*query service*).

Um microsserviço de leitura, tal como se pode observar na Figura 12, tem a responsabilidade de implementar as operações de leitura sobre uma visão de dados que são provenientes de um ou mais microsserviços. Deste modo, este microsserviço persiste os dados necessários para as operações de leitura numa estrutura e tecnologia adequada, sincronizados com os microsserviços proprietários dos mesmos, na medida em que, subscreve eventos emitidos a cada alteração.

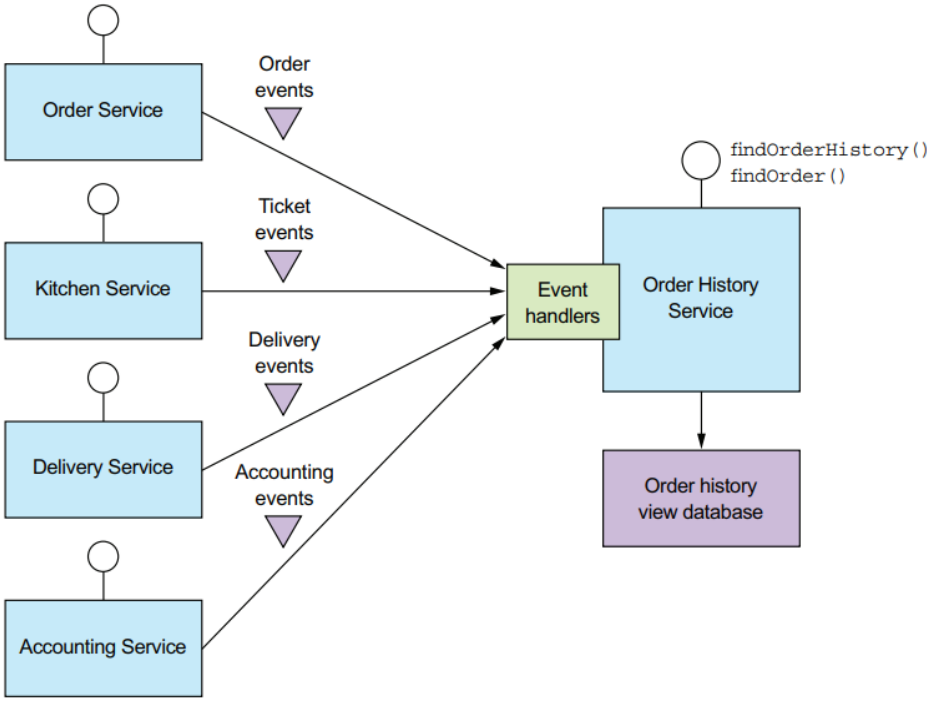


Figura 12: Aplicação do padrão CQRS, segundo a abordagem do microsserviço de leituras (query service) [1].

A aplicação do padrão CQRS para resolução do segundo problema torna-se muito semelhante à solução anterior, uma vez que, do mesmo modo permite adequar as melhores estruturas de dados e tecnologias de base de dados, com a finalidade de garantir que as operações de leitura sejam mais eficientes.

Na verdade, a adequação da estrutura e tecnologia, pode referir-se por exemplo, a operações de leitura que envolvem tipos e índices de localizações geográficas, existindo determinadas tecnologias de base de dados que não as suportam. Desta forma, não se justifica modificar a base de dados de um microsserviço devido a uma operação deste tipo, mas aplicar o padrão CQRS com a abordagem referida no problema anterior, isto é, a criação de um microsserviço de leituras (*query service*). Deste modo, o microsserviço de leituras passa a implementar uma estrutura e tecnologia de base de dados que suporte as localizações geográficas e permita executar a operação de leitura de forma mais eficiente.

Por fim, o último problema relaciona-se com a separação de conceitos, uma vez que, a decisão de que microsserviço deverá implementar uma operação de leitura, não se rege apenas pela verificação da propriedade dos dados envolvidos na mesma. Na verdade, para determinadas operações de leitura, apesar de um microsserviço conter maior parte dos dados requeridos na mesma, não implica que, seja o mais adequado para a responsabilidade dessa operação, evitando-se assim uma sobrelotação de responsabilidades.

Supondo como exemplo um sistema de entrega de refeições, onde se necessita de implementar a operação de leitura que obtém os restaurantes mais próximos de um determinado utilizador.

O sistema contém um microsserviço denominado por restaurante, responsável pela manutenção dos dados necessários para execução desta operação, isto é, a localização dos restaurantes. No entanto, a responsabilidade principal deste microsserviço, como referido, consiste em permitir aos proprietários dos restaurantes manter os dados dos mesmos, como: nome, localização, cozinhas, menus e horários, entre outros. A execução da operação que encontre os restaurantes mais próximos não encaixa nas responsabilidades deste microsserviço, uma vez que, consiste na execução de uma operação crítica de grandes volumes de dados.

A aplicação do padrão neste contexto para resolução do problema, consiste na implementação de um microsserviço de leituras (query service), similar às soluções anteriores, uma vez que, delega a responsabilidade de execução desta operação, para um novo microsserviço ou um microsserviço existente mais adequado para a mesma.

Com os problemas apresentados e respetivas soluções, verifica-se que, os desafios de execução de operações de leitura em arquiteturas orientadas a microsserviços, nem sempre

são execuções que abrangem múltiplos microsserviços. Na verdade, como se verifica com o segundo e terceiro problema, os mesmos referem-se a um microsserviço singular.

Uma das vantagens da aplicação deste padrão, consiste na eficiência de implementação de operações de leitura que retornam dados de múltiplos microsserviços. Na verdade, este padrão evita problemas de junção de numerosos dados em memória que ocorrem com o padrão alternativo, tendo como consequência operações de leitura mais eficientes.

Outra vantagem consiste em permitir adequação das melhores estruturas de dados, tecnologias e recursos, face aos requisitos das operações. Por exemplo, a nível de recursos, normalmente existem mais pedidos de leitura do que escrita, desta forma, com a separação das responsabilidades das operações consegue-se escalar independentemente as mesmas.

A separação de conceitos torna-se uma vantagem, uma vez que, tal como referido anteriormente, existe uma divisão dos modelos das operações de comando e leitura, bem como das respetivas bases de dados. Desta forma, permite que, as responsabilidades das operações sejam segregadas, permitindo aos programadores um desenvolvimento dos modelos mais focado e eficiente.

Por outro lado, a implementação deste padrão tem como consequência algumas desvantagens. A primeira desvantagem consiste no aumento da complexidade, uma vez que, se necessita de operacionalizar e manter mais componentes, como modelos de operações, base de dados, novos microsserviços, entre outros.

Outra desvantagem consiste na comunicação entre os modelos de comando e leitura depender da rede, uma vez que, podem ocorrer falhas, latências, entre outros problemas que podem implicar uma desatualização dos dados obtidos numa leitura. No entanto, uma possível solução para diminuir o impacto deste problema consiste em retornar os dados adicionados ou alterados na operação de comando.

Segundo Martin Fowler [26], o mesmo conhece situações da aplicação do padrão CQRS, em contextos com equipas de programadores muito competentes onde houve um arrasto da produtividade e consequente aumento do risco para o projeto. Desta forma, torna-se importante realçar que, sempre que possível, isto é, quando não existem os problemas anteriormente descritos, deve-se aplicar o padrão API composition. Por outro lado, o padrão CQRS apenas deve ser aplicado quando estes problemas ocorrem, uma vez que, a

desadequação da implementação deste padrão, tem como consequência o aumento da complexidade, risco e redução da produtividade.

Por fim, a *framework open source* Axon[10] pode ser utilizada para implementação do padrão CQRS, do mesmo modo, como referido anteriormente, este padrão necessita do envio de mensagens ou eventos, pelo que, podem ser implementados os padrões de mensagens transacionais, abordados de seguida e event sourcing[11] para esse efeito, existindo respetivamente as frameworks EventuateTram[12] e Eventuate Local[13].

2.2.3. Mensagens Transacionais

De acordo com o apresentado nas referências [1, 7, 13], a categoria de mensagens transacionais contém um conjunto de padrões que visam garantir a atomicidade do processo de execução de transações locais e publicação de mensagens.

Normalmente um microsserviço necessita de publicar uma mensagem quando efetua uma transação de atualização da base de dados, como por exemplo no padrão Saga. Desta forma, torna-se crucial garantir que essa atualização e envio da mensagem para outro microsserviço seja executada como uma única transação.

A razão da necessidade desta combinação na mesma transação, deve-se ao facto de garantir que, caso ocorra uma falha na atualização da base de dados não seja publicada a mensagem e vice-versa, para não haver inconsistências no sistema.

Padrão Transactional Outbox e publicação de mensagens por Polling ou Log Tailling

A primeira abordagem, isto é, Transactional Outbox [1, 7, 13], tem como objetivo garantir a atomicidade das transações de atualização da base de dados e a publicação de mensagens, através da persistência conjunta das mesmas com os dados atualizados.

Por outro lado, necessita-se de aplicar adicionalmente à Transactional Outbox, um padrão para a publicação das mensagens, existindo duas abordagens: Polling e Log Tailling.

[10] https://axoniq.io/product-overview/axon-framework#0
[11] https://microservices.io/patterns/data/event-sourcing.html
[12] https://github.com/eventuate-tram/eventuate-tram-core
[13] https://github.com/eventuate-local/eventuate-local

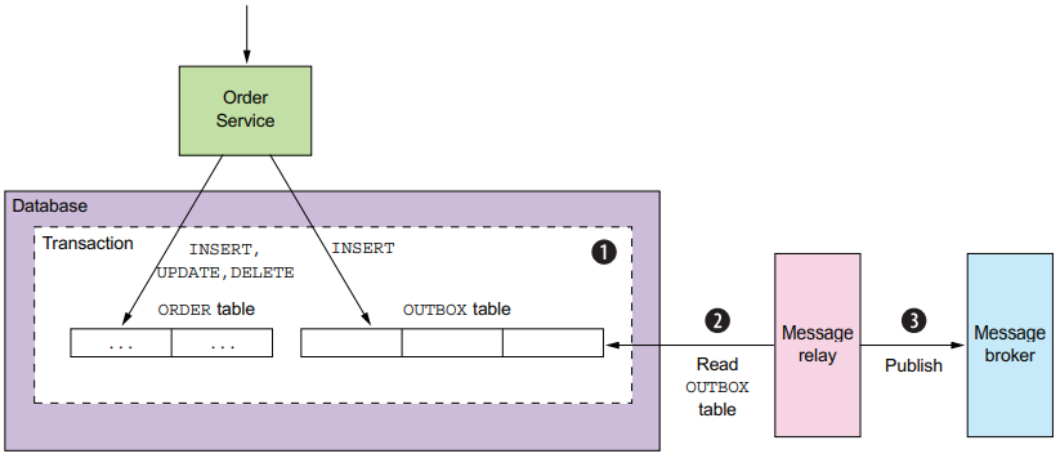


Figura 13: Padrão caixa de saída transacional [1].

Numa base de dados relacional, a abordagem Transactional Outbox consiste em criar uma tabela específica, denominada por OUTBOX, para persistir temporariamente as mensagens. Desta forma, as operações de criação, atualização e remoção de dados, devem nas suas transações, inserir a respetiva mensagem na tabela OUTBOX, tal como se pode observar na Figura 13 (passo 1). Deste modo, garante-se a atomicidade, uma vez que, os dois processos passam a ser executados numa única transação, que em base de dados relacionais contém a propriedade atómica.

Do mesmo modo, numa base de dados não relacional, pode-se aplicar o padrão Transactional Outbox, na medida em que, cada registo na base de dados contém um atributo de lista de mensagens a serem publicadas após execução da operação. Neste caso, a atomicidade garante-se, na medida em que, a operação de inserção, atualização ou remoção dos dados e a inserção das mensagens implementa-se como uma única operação.

Em relação à publicação das mensagens, a abordagem de Polling numa base de dados relacional consiste em implementar um componente, denominado por retransmissor de mensagens (*message relay*), tal como se pode observar na Figura 13.

O retransmissor de mensagens tem a responsabilidade de ler os dados da tabela OUTBOX (passo 2), seguido da conversão e publicação das mesmas no agente de mensagens (passo 3), e por fim a remoção da mensagem através de uma transação atómica.

Em relação à publicação de mensagens através da abordagem Polling numa base de dados não relacional torna-se desafiante, uma vez que, depende da capacidade de leitura desta, para efetuar a pesquisa por mensagens em todos os registos.

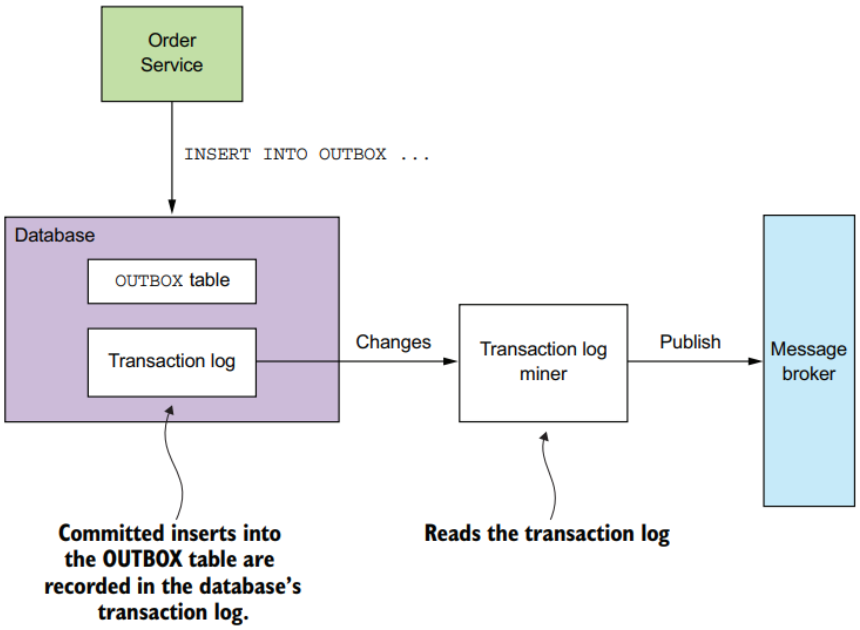


Figura 14: Publicação de mensagens através de *logs* de transações [1].

Por outro lado, outra alternativa para a publicação das mensagens em bases de dados relacionais consiste na abordagem Log Tailing, que implementa um componente denominado por *Transaction Log Miner*.

O Transaction Log Miner, tem a responsabilidade de através dos *log commits*, presentes nas bases de dados relacionais, identificar inserções na tabela OUTBOX, convertendo-as em mensagens, seguida da publicação das mesmas num agente, tal como se pode observar na Figura 14.

A aplicação da abordagem Log Tailing, para publicação de mensagens em bases de dados não relacionais, torna-se mais eficiente do que a Polling, uma vez que, não necessita de efetuar uma pesquisa por todos os registos.

Algumas tecnologias que podem ser utilizadas para aplicação dos padrões abordados nesta secção são a *Debezium*[14], *LinkedIn Databus*[15], *DynamoDB streams*[16] e *Eventuate Tram*[17]. A última tecnologia, desenvolvida pelo autor Chris Richardson [1], resolve alguns problemas

[14] O Debezium consiste num projeto open source de uma ferramenta que publica alterações de uma base de dados, para o agente de mensagens Apache Kafka (https://debezium.io/).
[15] O LinkedIn Databus consiste num projeto open source, que analisa o log das transações da Oracle e publica eventos das alterações (https://github.com/linkedin/databus).
[16] A DynamoDB streams consiste numa ferramenta que contém uma sequência ordenada de alterações feitas a uma tabela DynamoDB, nas últimas 24 horas (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html).
[17] A Eventuate Tram consiste numa ferramenta open source, que utiliza a abordagem tailling nas bases de dados MySQL, Postgres ou a abordagem polling para ler as alterações da tabela OUTBOX e publicá-las no Apache Kafka (https://github.com/eventuate-tram/eventuate-tram-core).

encontrados na Debezium para aplicação neste contexto, que fornece APIs para mensagens utilizando as abordagens Tailing e Polling.

2.2.4. APIs Externas

De acordo com o apresentado nas referências [1, 7, 13, 18, 30, 31, 25, 9], a grande motivação desta categoria deve-se ao facto de resolver um conjunto de problemas, que surgem devido às aplicações cliente efetuarem pedidos diretamente às APIs dos microsserviços.

Padrão API Gateway

O padrão API Gateway, como se pode observar na Figura 15, consiste num serviço que implementa o único ponto de entrada do exterior da *firewall* com os microsserviços do backend. As responsabilidades deste serviço são o encapsulamento da arquitetura, roteamento e balanceamento dos pedidos efetuados fora da firewall ao sistema, API composition e outras funcionalidades, que habitualmente são colocadas nas periferias do backend (*edge functions*) tais como: autenticação, autorização, limitação de pedidos, *caching*, coleção de métricas, *logging,* descoberta de serviços, circuit breakers[18], entre outras.

Efetivamente, o padrão API Gateway resolve um conjunto de problemas presentes em arquiteturas orientadas a microsserviços, relacionados maioritariamente com a execução de pedidos diretamente às APIs dos microsserviços por parte de aplicações cliente.

Um desses problemas consiste na existência de uma diversidade de utilizadores, uma vez que, os mesmos têm diferentes necessidades e recursos. Por exemplo, uma aplicação *mobile* normalmente tem uma interface de utilizador reduzida e menos recursos de rede, do que uma aplicação *desktop.* Deste modo, a quantidade e qualidade (p.e. imagens) dos dados obtidos através dos microsserviços por aplicações deste tipo deve ser mais reduzida.

Desta forma, a aplicação do API Gateway resolve este problema, na medida em que, permite implementar diferentes APIs para diferentes tipos de clientes adequadas às necessidades destes, tal como se pode observar na Figura 16.

[18] O padrão circuit breaker, tem como objetivo aumentar a confiança do sistema, contra falhas em cascata provocadas por outros serviços, para mais informações consultar https://microservices.io/patterns/reliability/circuit-breaker.html.

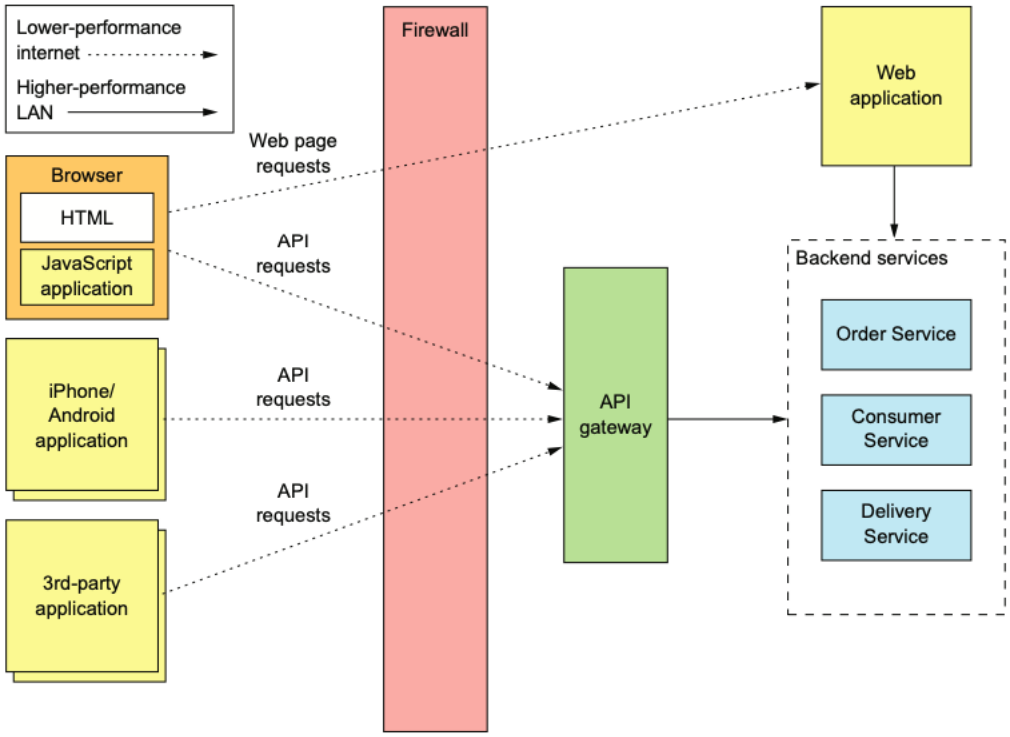


Figura 15: Padrão API Gateway, para um sistema com diferentes tipos de clientes [1].

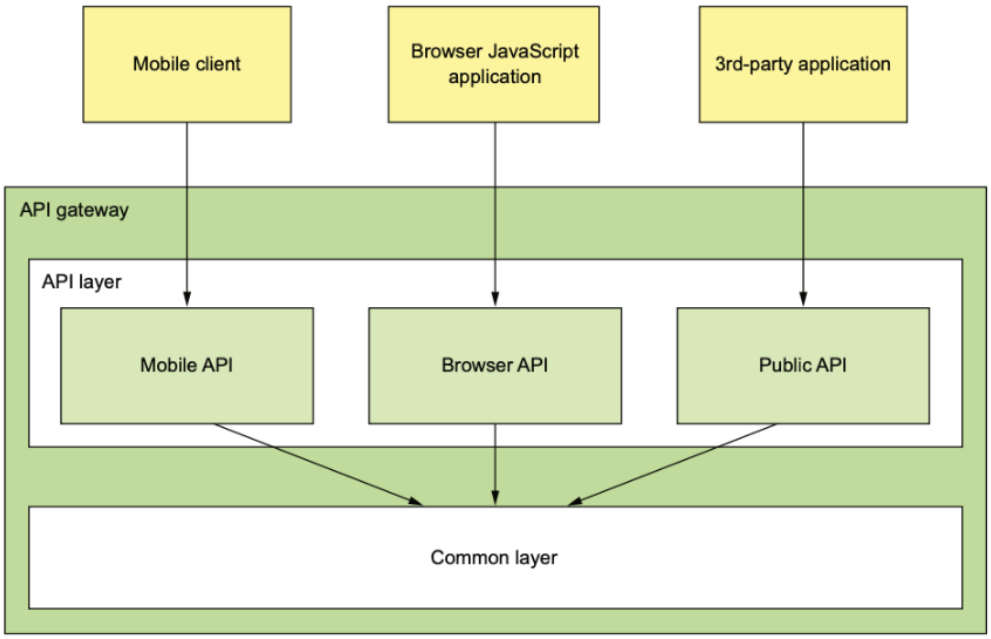


Figura 16: API Gateway, com APIs adequadas a cada tipo de cliente [1].

Outro problema, refere-se à falta de encapsulamento, isto é, o acesso direto das aplicações cliente às APIs dos microsserviços implica uma exposição da arquitetura, complexidade na manutenção dos microsserviços, entre outros problemas. Na verdade, as

equipas de desenvolvimento estão constantemente a modificar, adicionar e dividir microsserviços. Desta forma, caso não exista encapsulamento, torna-se mais complexo efetuar estas modificações sem implicar alterações nas aplicações clientes.

Outro problema, refere-se à falta de encapsulamento, isto é, o acesso direto das aplicações cliente às APIs dos microsserviços implica uma exposição da arquitetura, complexidade na manutenção dos microsserviços, entre outros problemas. Na verdade, as equipas de desenvolvimento estão constantemente a modificar, adicionar e dividir microsserviços. Desta forma, caso não exista encapsulamento, torna-se mais complexo efetuar estas modificações sem implicar alterações nas aplicações clientes.

Deste modo, a implementação do serviço API Gateway resolve este problema, uma vez que, encapsula e abstrai as aplicações cliente da arquitetura interna. Desta forma, desde que a API do Gateway seja mantida, os programadores do backend podem efetuar alterações internamente sem implicar a necessidade de modificações na aplicação cliente. Do mesmo modo, permite manter versões antigas das APIs dos microsserviços, para não prejudicar determinadas aplicações cliente que não sejam atualizadas com as novas modificações.

Por outro lado, existe um problema relacionado com os mecanismos de comunicação, uma vez que, as aplicações cliente normalmente comunicam através de HTTP ou *websockets*[19], no entanto, os microsserviços podem utilizar outros protocolos menos familiares, tais como RPC[20], AMQP, entre outros. Desta forma, o API Gateway resolve este problema, na medida em que, comportar-se como um adaptador do protocolo de comunicação, uma vez que, comunica com o exterior da firewall por HTTP ou websocket, enquanto, internamente os microsserviços utilizam os mecanismos mais adequados, como HTTP, websocket, RPC, AMQP, entre outros.

Algumas das tecnologias mais relevantes para implementação do padrão API Gateway e funcionalidades de periferia (*edge functions*) são: *NGINX*[21], *Kong*[22], *Traefik*[23], *Netflix Zuul*[24], *Spring Cloud Gateway*[25], *AWS API Gateway*[26], *AWS Load Balancer*[27].

[19] RFC 6455 https://tools.ietf.org/html/rfc6455
[20] RFC 5531 https://tools.ietf.org/html/rfc5531
[21] https://www.nginx.com/
[22] https://konghq.com/kong/
[23] https://traefik.io/solutions/api-gateway/
[24] https://github.com/Netflix/zuul
[25] https://spring.io/projects/spring-cloud-gateway
[26] https://aws.amazon.com/api-gateway/
[27] https://aws.amazon.com/elasticloadbalancing/gateway-load-balancer/

As vantagens da aplicação do padrão API Gateway, como referido anteriormente, consiste no encapsulamento da arquitetura interna do sistema, adequação das APIs face às necessidades e recursos das diferentes aplicações clientes, bem como redução da necessidade de alterações no lado do cliente, face a modificações na arquitetura interna. Do mesmo modo, simplifica o código da aplicação cliente, uma vez que, não necessita de efetuar múltiplos pedidos (API composition) quando necessita de efetuar operações de leitura de dados que envolvem múltiplos microsserviços.

Por outro lado, existem desvantagens, como aumento da complexidade, uma vez que, torna-se necessário desenvolver, instalar e manter, mais componentes. Sendo um ponto de entrada, tem o risco do efeito de gargalo (*bottleneck*) de pedidos.

A nível de desafios, existe a necessidade de os programadores atualizarem as APIs do Gateway em tempo de execução. Desta forma, torna-se necessário que estes processos sejam leves e que não tenham impacto negativo na utilização do sistema.

Padrão Backends for Frontends

O surgimento do padrão Backends for Frontends vem como melhoria ao API Gateway, na medida em que, ao invés deste conter as diferentes APIs para cada tipo de cliente num único serviço, as mesmas são divididas em serviços API Gateways separados, tal como se pode observar na Figura 17.

A separação das responsabilidades nesta abordagem torna-se uma vantagem, uma vez que, permite uma maior independência e desacoplamento dos serviços e das equipas que mantém as APIs do Gateway, seguindo a filosofia "if you build it you own it".

A divisão dos API Gateways em diferentes serviços permite uma maior fiabilidade, uma vez que, a falha num determinado Gateway não afeta os restantes. Do mesmo modo, também permite uma maior independência e desacoplamento na observação e escalabilidade, entre outros atributos de qualidade dos diferentes Gateways.

As primeiras aplicações desta abordagem foram na aplicação SoundCloud[28] [1], no entanto, devido às vantagens desta abordagem algumas organizações de renome no mundo das arquiteturas orientadas a microsserviços estão a migrar para a mesma, tais como a Netflix.

[28] A SoundCloud é uma plataforma de distribuição de áudio e música.

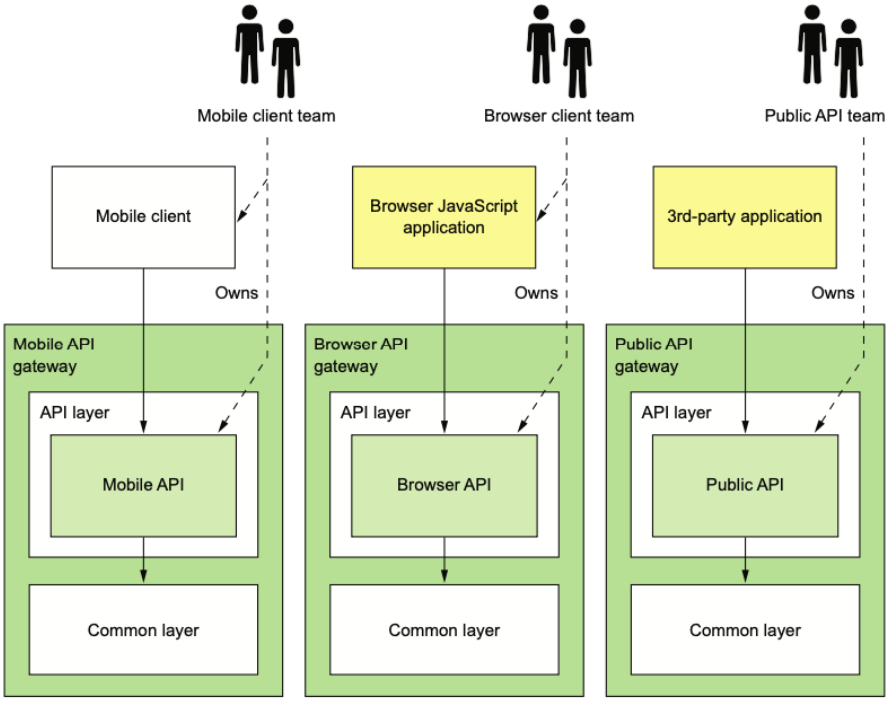


Figura 17: Aplicação do padrão *frontends for backends* a API Gateways [1].

2.2.5. Descoberta de Serviços

Numa arquitetura orientada a microsserviços, durante o desenvolvimento dos microsserviços, bem como das aplicações cliente, torna-se inevitável invocar funcionalidades de outros serviços como se verifica nos padrões abordados anteriormente.

Efetivamente, para se invocar um determinado serviço no código necessita-se do conhecimento dos endereços IP (*Internet Protocol*) das instâncias dos serviços. No entanto, tipicamente numa arquitetura orientada a microsserviços as invocações não podem ser efetuadas de forma estática, uma vez que, os IPs das instâncias dos serviços são atribuídos dinamicamente. Por outro lado, normalmente neste tipo de arquiteturas podem ocorrer falhas, atualizações, mecanismos de escalabilidade, entre outras situações, que implicam a mudança da localização das instâncias dos serviços.

Deste modo, de acordo com o apresentado nas referências [1, 7, 13, 30, 9], os padrões desta categoria resolvem os problemas acima descritos.

Padrão Service Registry

Um Service Registry consiste num serviço para registo de serviços, que contém uma base de dados de serviços, isto é, as instâncias e respetivas localizações, implicando a atualização desta a cada inicialização ou suspensão de um serviço da arquitetura.

Deste modo, aplicações cliente, serviços de roteamento ou outros, fazem um pedido pelo nome ao service registry para obter a localização das instâncias disponíveis de um determinado serviço.

Por fim, cada serviço da arquitetura deverá implementar o padrão de verificação da disponibilidade do serviço (*health check API*[29]), que consiste em implementar um método para permitir ao service registry determinar se os mesmos estão aptos para receber e manipular pedidos, acesso à base de dados, entre outras verificações.

Padrão Self-registration e 3rd Party Registration

Efetivamente, os serviços quando são inicializados ou suspensos devem ser respetivamente registados, atualizados e removidos do Service Registry.

Deste modo, existem dois padrões para efetuar o registo, atualização e remoção da localização das instâncias dos serviços, isto é, o *Self-Registration,* onde o registo é feito pelo próprio serviço ou *3rd Party Registration,* onde o registo é feito por terceiros.

Por fim, a nível de tecnologias, isto é, plataformas de deployment como Docker[30], Kubernetes[31], entre outras, utilizam a abordagem 3rd Party Registration.

Padrão Server-side Discovery

Adicionalmente à abordagem Server-side Discovery, numa arquitetura orientada a microsserviços requer-se a combinação dos padrões acima descritos, isto é, Service Registry e a abordagem de realização do registo mais adequada.

A abordagem Server-side Discovery, difere se o pedido se efetua por aplicações cliente ou serviços internos à arquitetura.

[29] https://microservices.io/patterns/observability/health-check-api.html
[30] https://www.docker.com/
[31] https://kubernetes.io/

Como mostra a Figura 18, no contexto de aplicações cliente, como referido na secção 2.2.4, sobre o padrão API Gateway, os pedidos exteriores à firewall são direcionados para o serviço com a responsabilidade de roteamento dos pedidos.

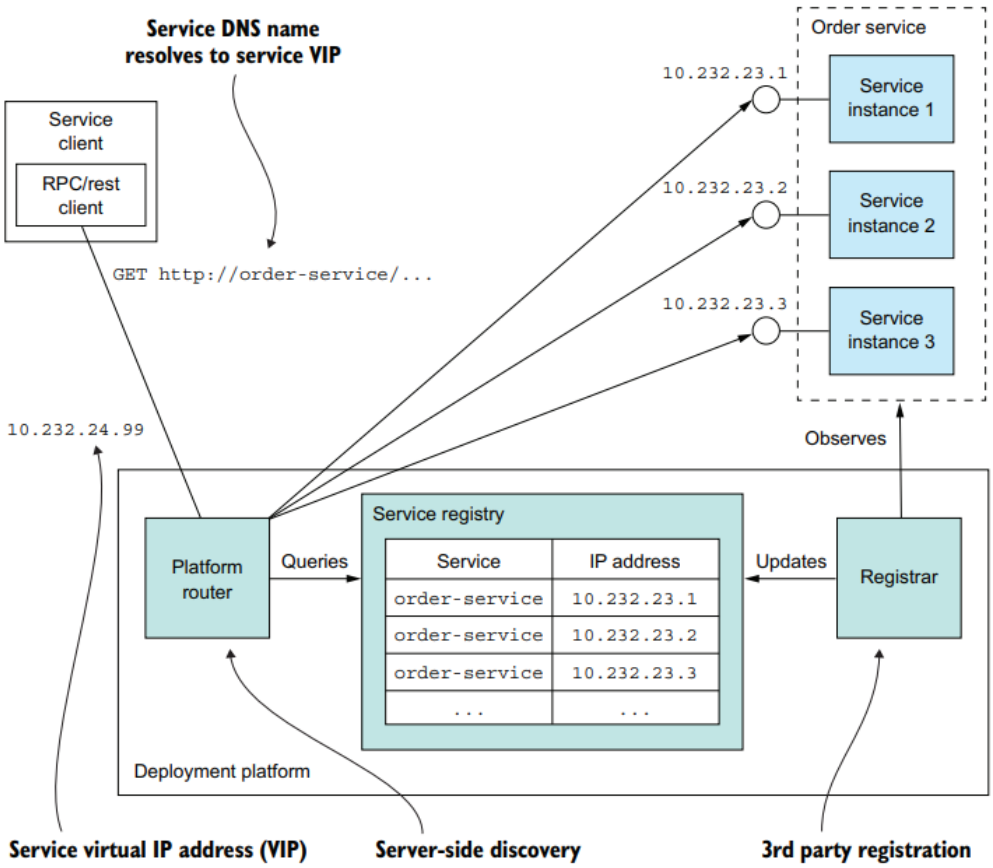


Figura 18: Implementação dos padrões de service registry, 3rd party registration, server-side discovery [1].

Deste modo, o serviço de roteamento tem a responsabilidade de efetuar a lógica de Server-side Discovery, isto é, efetuar o pedido ao Service Registry através do nome do serviço, seguido de mecanismos de balanceamento, que através da informação retornada determinam a qual instância se deve direcionar o pedido.

Por outro lado, quando existe comunicação entre serviços internos à firewall, não se torna necessário efetuar o pedido ao serviço de roteamento. Na verdade, a abordagem consiste nos próprios serviços executarem um pedido ao registo de serviços, para obter a localização das instâncias do serviço que requerem.

Uma desvantagem inerente a esta abordagem consiste no aumento da complexidade, uma vez que, necessita de se desenvolver, manter e instalar mais componentes. Do mesmo

modo, verifica-se um aumento de saltos devido à necessidade de vários pedidos, no entanto, o aumento do tempo de resposta devido a estes saltos, torna-se insignificante face às vantagens que esta abordagem proporciona.

Por fim, como referido anteriormente, as tecnologias de deployment como Docker, Kubernetes, entre outras, implementam o Service Registry, Server-side Discovery e roteamento de pedidos seguindo a abordagem 3rd Party Registration. Deste modo, a utilização deste tipo de tecnologias permite que todas as abordagens acima descritas sejam garantidas pela plataforma de deployment, retirando essa responsabilidade do programador.

2.2.6. Segurança

De acordo com o apresentado nas referências [1, 13, 18, 32], esta categoria contém um conjunto de abordagens, que com a sua implementação permitem resolver problemas como a identificação do utilizador ao longo de microsserviços, bem como garantir o atributo de qualidade de segurança numa arquitetura orientada a microsserviços.

Padrão Acess Token

Uma das motivações para o surgimento deste padrão, teve como consequência a necessidade dos sistemas com arquiteturas orientadas a microsserviços identificarem o utilizador que está a efetuar pedidos, uma vez que, estes passam de microsserviço em microsserviço iniciando-se no API Gateway. Acrescida a esta necessidade, deve-se garantir o atributo de qualidade de segurança, existindo várias abordagens para este tipo de arquiteturas, das quais a autenticação e autorização.

A autenticação consiste em verificar a identidade de um utilizador da API ou utilizador por login, que pretende ter acesso ao sistema. Normalmente, um utilizador por login, como o nome sugere, necessita de efetuar primeiramente o pedido de login enviando as credenciais, como id e password, antes de efetuar qualquer outro pedido, enquanto um utilizador da API envia as credenciais, por exemplo um *secret* em cada pedido.

Existem diferentes abordagens para implementação da autenticação numa arquitetura orientada a microsserviços apresentadas de seguida.

A primeira abordagem consiste em delegar a lógica necessária para autenticação a cada microsserviço, no entanto, existem alguns problemas associados a esta implementação.

Um dos problemas, consiste em permitir que pedidos não autenticados consigam entrar na rede interna do sistema.

Um outro problema, tem como consequência a necessidade de as equipas de desenvolvimento dos microsserviços implementarem corretamente a lógica de segurança, podendo criar um grande risco para o sistema devido às vulnerabilidades que possam surgir.

Por fim, outro problema, consiste na implicação dos microsserviços necessitarem de implementar diferentes formas de autenticação, para os distintos tipos de utilizadores.

A segunda abordagem consiste na implementação da autenticação no API Gateway, sendo considerada uma melhor abordagem, uma vez que, ultrapassa os problemas descritos nos parágrafos acima.

Na verdade, permite autenticar o pedido antes de ser direcionado para os microsserviços, impedindo que pedidos não autenticados entrem na rede interna.

A centralização desta lógica de segurança no API Gateway assegura que existe apenas um lugar para o sucesso da autenticação, bem como permite uma redução da probabilidade de vulnerabilidades de segurança, uma vez que, o código se encontra exclusivamente num componente.

Por fim, permite delegar apenas para API Gateway a responsabilidade de autenticar diferentes utilizadores, abstraindo esta complexidade dos microsserviços do backend.

Supondo o exemplo do sistema de entrega de refeições, apresentado na secção 2.2.2 sobre o padrão API Composition, a funcionalidade de "obter os dados de uma encomenda" processa-se de forma diferente para cada tipo de cliente, como se pode observar na Figura 19.

Caso seja um utilizador autenticado por login, como referido anteriormente, necessita-se primeiramente de efetuar o login. Deste modo, o processo inicializa-se com um pedido HTTP POST /login enviando as credenciais id e password (passo B1).

De seguida, o API Gateway ao receber este pedido, prossegue com a autenticação do utilizador retornando em caso de sucesso um token para o mesmo (passo B2), para que este o envie em cada pedido subsequente (passo B3), como por exemplo "obter os dados da encomenda" com um HTTP GET /orders/1.

Após o API Gateway receber o pedido com o token, verifica a identidade do utilizador e reencaminha o pedido juntamente com o token para os microsserviços responsáveis (passo B4), para que estes consigam identificar e autenticar o utilizador.

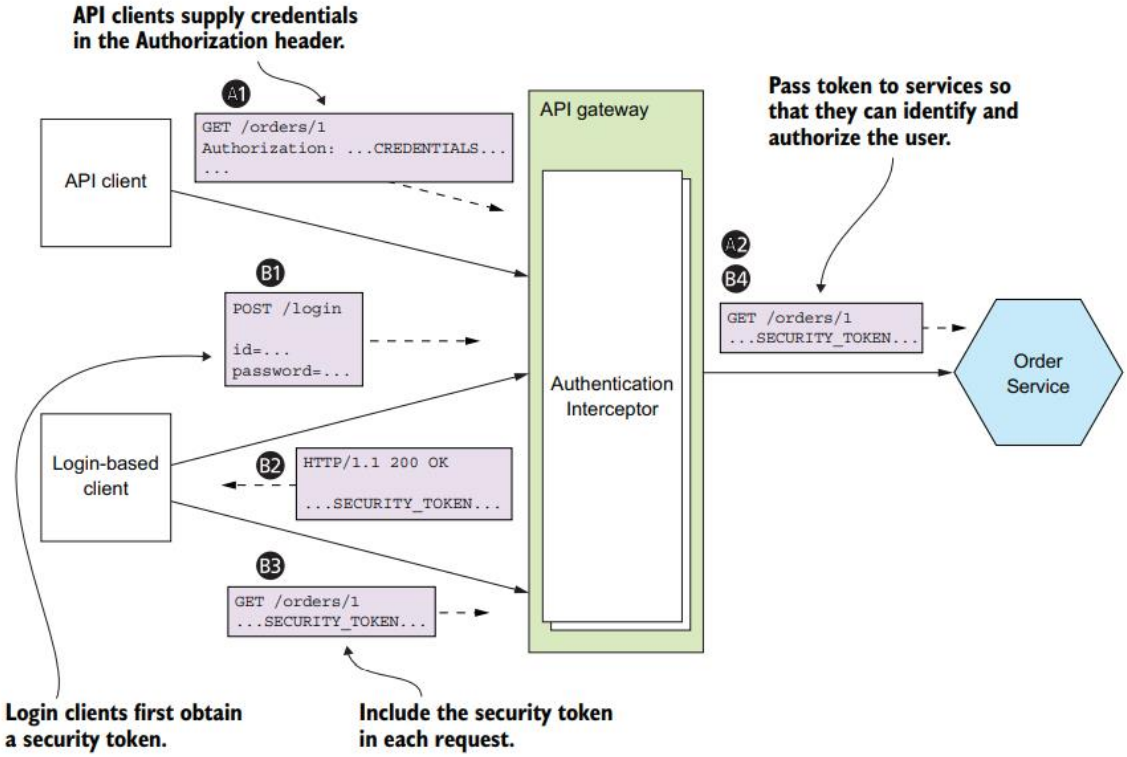


Figura 19: Autenticação no API Gateway, implementando o padrão access token [1].

Por outro lado, caso seja um utilizador de API, como referido anteriormente o mesmo não necessita de efetuar o pedido de login.

Deste modo, o processo inicializa-se diretamente com o pedido que o utilizador pretende efetuar, que neste caso consiste no pedido dos dados da encomenda, isto é, HTTP GET /orders/1 (passo A1) enviando as credenciais no cabeçalho.

De seguida, o API Gateway ao receber este pedido executa a autenticação do utilizador com base nas credenciais fornecidas. Em caso de sucesso, gera um token que é passado aos microsserviços, para que, estes consigam identificar e autenticar o utilizador (passo A2).

A autorização, consiste em verificar se um utilizador tem permissão para efetuar a operação requerida sobre determinados dados.

As abordagens utilizadas para implementação deste mecanismo são a *role-based* e *ACLs* (*Access Control Lists*).

A role-based, consiste em marcar cada utilizador com um ou mais papeis, que concedem permissões para determinadas invocações de operações.

Por outro lado, as ACLs definem utilizadores específicos ou papeis com permissões para executarem uma operação sobre um objeto ou agregações de negócio.

A motivação para o mecanismo de autorização, deve-se ao facto de que determinados pedidos ou objetos de negócio apenas podem ser acedidos por utilizadores específicos. Supondo o exemplo da "obtenção dos dados de uma encomenda", a mesma apenas pode ser consultada pelo utilizador que a criou e os agentes de serviço que auxiliam os utilizadores no processo de encomendas.

Do mesmo modo que a autenticação, existem diferentes abordagens para implementação da autorização apresentadas de seguida.

A primeira abordagem consiste em implementar a autorização no API Gateway, e por esse motivo contém as mesmas vantagens que a abordagem da autenticação quando implementada no mesmo componente.

No entanto, existe uma desvantagem desta abordagem, que consiste em apenas poder ser usado o mecanismo de role-based para os pedidos, como consequência do desafio de implementar ACLs, uma vez que, o API Gateway necessita de conter um conhecimento dos objetos de negócio dos microsserviços.

Por fim, existe um problema relacionado com a implementação desta abordagem, que consiste na necessidade de atualização do API Gateway a cada alteração das rotas ou permissões dos pedidos, implicando um risco de dependência com os microsserviços.

A segunda abordagem consiste em cada microsserviço implementar a lógica de autorização, utilizando os mecanismos de role-based e ACLs. As ACLs ao contrário da abordagem anterior tornam-se mais triviais a sua implementação, uma vez que, permite controlar o acesso aos objetos de negócio no próprio microsserviço, sendo que este tem total conhecimento dos mesmos. Do mesmo modo que a abordagem equivalente da autenticação, esta implementação contém as mesmas desvantagens.

Como apresentado anteriormente, as abordagens de autenticação e autorização utilizam tokens para identificação do utilizador que está a efetuar o pedido. Deste modo, torna-se importante conhecer para que servem, como funcionam e a diferença entre os tokens opacos e transparentes.

Um token opaco normalmente utiliza *UUIDs*[32] (*Universally unique identifier*) para identificação do utilizador. No entanto, não são muito recomendados para arquiteturas orientadas a microsserviços, uma vez que, reduzem a performance e aumentam a latência,

[32] RFC 4122 https://tools.ietf.org/html/rfc4122

em consequência de ser necessário efetuar um pedido síncrono a um serviço de segurança para validar e obter os dados do utilizador.

A validação e obtenção de dados de um token transparente não implica a realização de pedidos a serviços terceiros, sendo deste modo mais vantajoso, uma vez que, resolve os problemas do token opaco. Um padrão recomendado para implementação desta abordagem consiste no JWT[33] (*JSON Web Token*), que representa de forma segura a identidade e papeis de um utilizador entre duas entidades.

Um token JWT, tem no seu conteúdo um objeto JSON que contém dados como a identidade e o papel do utilizador, bem como informações adicionais sobre o token, como por exemplo a data de expiração do mesmo. Por forma a garantir-se a segurança da informação que o token representa, utiliza-se criptografia assinando os tokens com uma chave que apenas o criador e recetor do token conhecem.

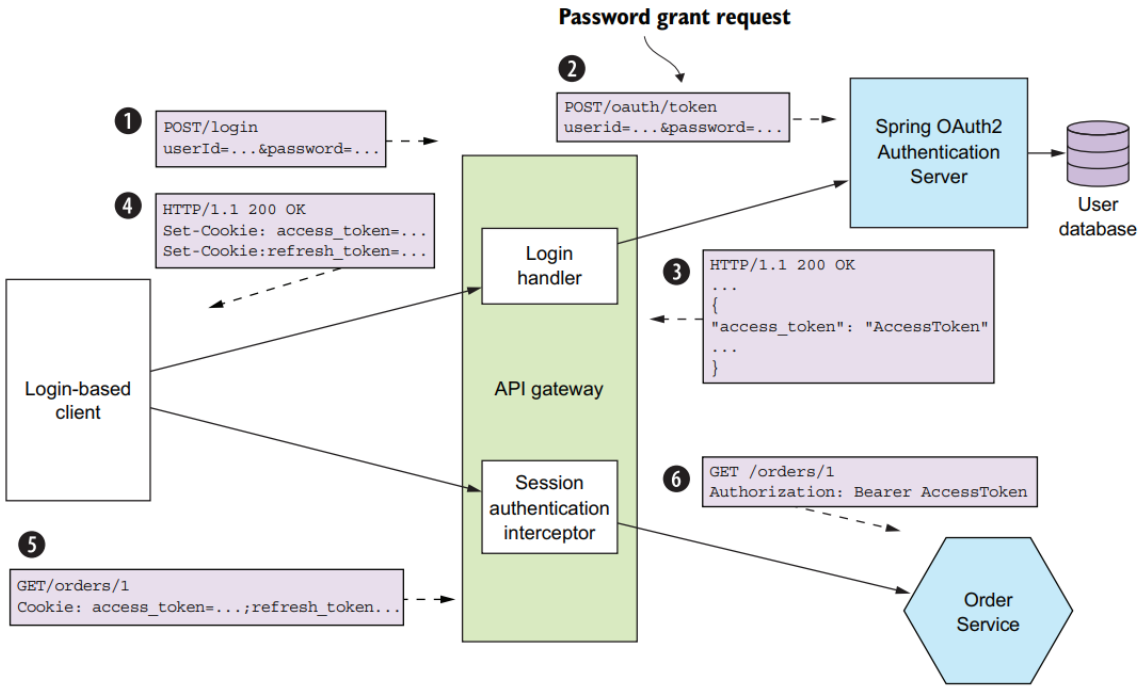


Figura 20: Implementação da autenticação e autorização utilizando Spring OAuth2 [1].

Em relação às várias abordagens apresentadas nesta secção, torna-se importante referir o protocolo de autorização *OAuth 2.0*[34]. Apesar do objetivo primordial ser de permitir

[33] RFC 7519 https://tools.ietf.org/html/rfc7519
[34] RFC 6749 https://tools.ietf.org/html/rfc6749

o acesso de aplicações terceiras a serviços por HTTP, o mesmo pode ser utilizado para a implementação da autenticação e autorização em arquiteturas orientadas a microsserviços, quer para utilizadores de APIs ou por login.

Por fim, a utilização do protocolo OAuth 2.0 torna-se vantajoso, uma vez que, implementa a infraestrutura necessária para a garantir a segurança de forma confiável retirando essa responsabilidade dos programadores, tal como se pode observar na Figura 20, onde se utiliza a framework *Spring OAuth2*[35]. Deste modo, torna-se importante reforçar, tal como se pode observar na Figura 19, que algumas tecnologias API Gateway implementam na sua base ou por *plugins* os mecanismos de autenticação e autorização.

2.3. Conclusão

Em conclusão deste capítulo, o estudo das arquiteturas orientadas a microsserviços já se dá como adquirido, pelo que, não foram elaboradas grandes pesquisas sobre o mesmo. Por outro lado, não existe uma lista oficial bem definida dos padrões destas arquiteturas, pelo que, inicialmente a pesquisa dos mesmos foi bastante desafiante.

No entanto, com base na lista proposta por Chris Richardson, no seu livro *Microservices Patterns* [1], foi possível selecionar um conjunto dos mesmos, para os quais se achou importante a sua análise. Esta seleção teve em conta aqueles que são necessários e base para maior parte dos sistemas. Abrangendo nestas arquiteturas a decomposição, manutenção dos dados, comunicação e segurança. De seguida, fez-se uma pesquisa comparativa dos mesmos analisados e descritos por outros autores para se perceber a sua relevância.

Relativamente aos padrões estudados, a categoria que foi mais desafiante foi a manutenção de dados, sendo muito importante, uma vez que, um dos pontos essenciais em qualquer sistema são os dados. Na verdade, a sua dificuldade está relacionada com a natureza distribuída de uma arquitetura orientada a microsserviços, que aumenta a complexidade para se garantir algumas propriedades dos dados.

Assim relativamente ao estudo do estado da arte, o mesmo foi bastante importante para aumentar o conhecimento das arquiteturas orientadas a microsserviços. Mas principalmente tornou-se importante, para conhecer alguns dos princípios e padrões

[35] https://spring.io/guides/tutorials/spring-boot-oauth2/

existentes, bem como a sua necessidade e importância face a resolver alguns dos problemas e desafios, que existem na implementação deste tipo de arquiteturas.

3. Análise do Caso de estudo

Neste capítulo, apresenta-se a análise do caso de estudo, seguindo uma especificação de requisitos com uma estrutura Volere. No entanto, esta estrutura contém diversas secções, pelo que, abordar-se-á apenas as mais relevantes para o caso de estudo.

Efetivamente, começar-se-á pela apresentação dos propósitos e objetivos, seguida das partes interessadas, isto é, as entidades afetadas direta ou indiretamente, termos técnicos e convenções importantes para a análise e levantamento dos requisitos e observação do domínio do caso de estudo.

Por fim, torna-se importante realçar a importância deste capítulo, uma vez que, tal como foi abordado na secção 2.2.1, sobre os padrões para decomposição do sistema em microsserviços, o conhecimento do domínio do caso de estudo torna-se implícito para efetuar uma decomposição estável.

3.1. Propósito e Objetivos do Caso de estudo

Por forma a aplicar os conhecimentos adquiridos durante o estudo do estado da arte, dos princípios e padrões das arquiteturas orientadas a microsserviços, decidiu-se escolher como caso de estudo uma aplicação para comércio eletrónico, também designado normalmente por e-commerce. A escolha deste domínio deve-se ao facto de o mesmo ter uma dimensão e complexidade relevantes, permitindo aplicar alguns dos princípios e padrões das arquiteturas orientadas a microsserviços abordados no capítulo 2.

Efetivamente, para além de ser um caso de estudo para aplicação do conhecimento adquirido anteriormente, pretende-se que, esta aplicação e-commerce a desenvolver contenha a base necessária a qualquer problema deste domínio, para tornar o mesmo mais reutilizável evitando-se redundâncias.

Desta forma, justifica-se a aplicação deste tipo de arquiteturas orientadas a microsserviços neste tipo de domínios, uma vez que, tendencionalmente existe um aumento do crescimento de utilização, isto é, maior número de transações, dados e utilizadores. Os utilizadores interagem cada vez mais em dispositivos diferentes, pelo que, este tipo de arquiteturas tende a ser mais dinâmica e independente, do que as alternativas permitindo maior facilidade de adaptar a estas diferenças.

Por fim, tal como abordado ao longo da secção 2.1, as vantagens das arquiteturas orientadas a microsserviços, como por exemplo: permitir uma maior facilidade e rapidez em colocar em produção novas funcionalidades, escalabilidade, disponibilidade, entre outras, são imprescindíveis nestes modelos de negócio.

3.2. Partes Interessadas (Stakeholders)

Nesta secção, apresenta-se na Tabela 2 as partes interessadas (*stakeholders*), que são afetadas direta ou indiretamente pelo caso de estudo. Importante realçar que, podem existir mais partes interessadas, dado que se trata de um domínio bastante complexo. No entanto, fez-se uma seleção das que são mais importantes no contexto desta dissertação.

Designação	Motivo
Cliente	Entidade organizacional, que tem um comércio físico ou eletrónico e pretende uma aplicação trivialmente mais dinâmica, independente, escalável, entre outras vantagens anteriormente abordadas.
Consumidor	Utilizador da aplicação para compra de produtos, e deste modo terá uma melhor plataforma, visto que, esta consegue atender um maior número de pessoas com qualidade.
Gestor	Utilizador da aplicação para efetuar a manutenção do conteúdo da mesma.
Administrador	Utilizador da aplicação que efetua a manutenção da mesma, quanto à correção, melhoria e adição de funcionalidades, bem como garantir que todo o sistema está com bom funcionamento.
Organizações Colaboradoras	A aplicação de e-commerce não tem como objetivo substituir outras aplicações já existentes, que sejam excelentes a fazer o seu trabalho (p.e. gestão de inventários, CRM (Customer Relationship Management), entre outras), pelo que, existe um interesse de colaboração para integrar as mesmas neste sistema e-commerce. Por outro lado, também existem organizações que fornecem ferramentas e serviços essenciais para o desenvolvimento, deployment e manutenção do sistema.
Organizações Concorrentes	O cliente pode já ter uma aplicação e-commerce, deste modo, poderá ser necessário em alguns casos efetuar uma migração dessa aplicação.

Tabela 2 - Partes Interessadas no projeto.

3.3. Convenções e Terminologia

Nesta secção, apresenta-se na Tabela 3 o vocabulário mais importante deste caso de estudo, desde nomenclatura, termos, abreviações, siglas e acrónimos, para facilitar uma melhor compreensão do mesmo.

Vocabulário	Definição
Cliente	Entidade organizacional, que requisita o serviço do sistema e-commerce, para implementação do seu negócio (p.e. PCDiga, Worten, Wook, Fnac, entre outas.)
Consumidor	Utilizador da aplicação que procede desde a pesquisa/procura de produtos, adição dos mesmos ao carrinho de compras, até à finalização da compra destes.
Gestor	Utilizador da aplicação que procede à manutenção do conteúdo da mesma.
Administrador	Utilizador da aplicação que procede à manutenção do sistema.
Produto	Item para venda com características específicas (p.e. especificações, cores, tamanhos, materiais, entre outras).
Categoria	Refere-se a uma coleção de produtos, com características semelhantes. Ainda contém um conjunto de subcategorias de características mais específicas dos produtos.
EAN	European Article Number - identificador padrão, em formato de código de barras ou sistema de números, para identificar o tipo, a configuração de empacotamento e fabricante do produto.
PN	Part Number - identificador de uma parte de um design ou material completo, sendo o seu principal objetivo simplificar a sua referência.
SKU	Stock Keeping Unit - utilizado para identificar e monitorizar o inventário ou stock de um produto.
VAT	Value-added-tax - taxas sobre um Produto.
Carrinho de Compras	Guarda os produtos que o Consumidor deseja comprar.
Encomenda	Registo que representa um conjunto de produtos que o consumidor comprou.
Inventário	Específica a quantidade de stock de cada produto pelo seu SKU.
Stock	Quantidade em uma determinada unidade de um Produto com o SKU específico.
Loja	Espaço físico ou virtual, onde são expostos Produtos para venda.

Tabela 3- Vocabulário do projeto

3.4. Modelo de Domínio

Nesta secção, proceder-se-á à explicação do modelo de domínio que se pode observar na Figura 21, referente ao caso de estudo apresentado na secção 3.1, o qual foi obtido através da observação de aplicações de e-commerce mais comuns em Portugal, como por exemplo: PCDiga, Fnac, Worten, Wook, PcComponentes, Amazon, entre outras, bem como a consulta de alguns recibos e faturas provenientes destas organizações.

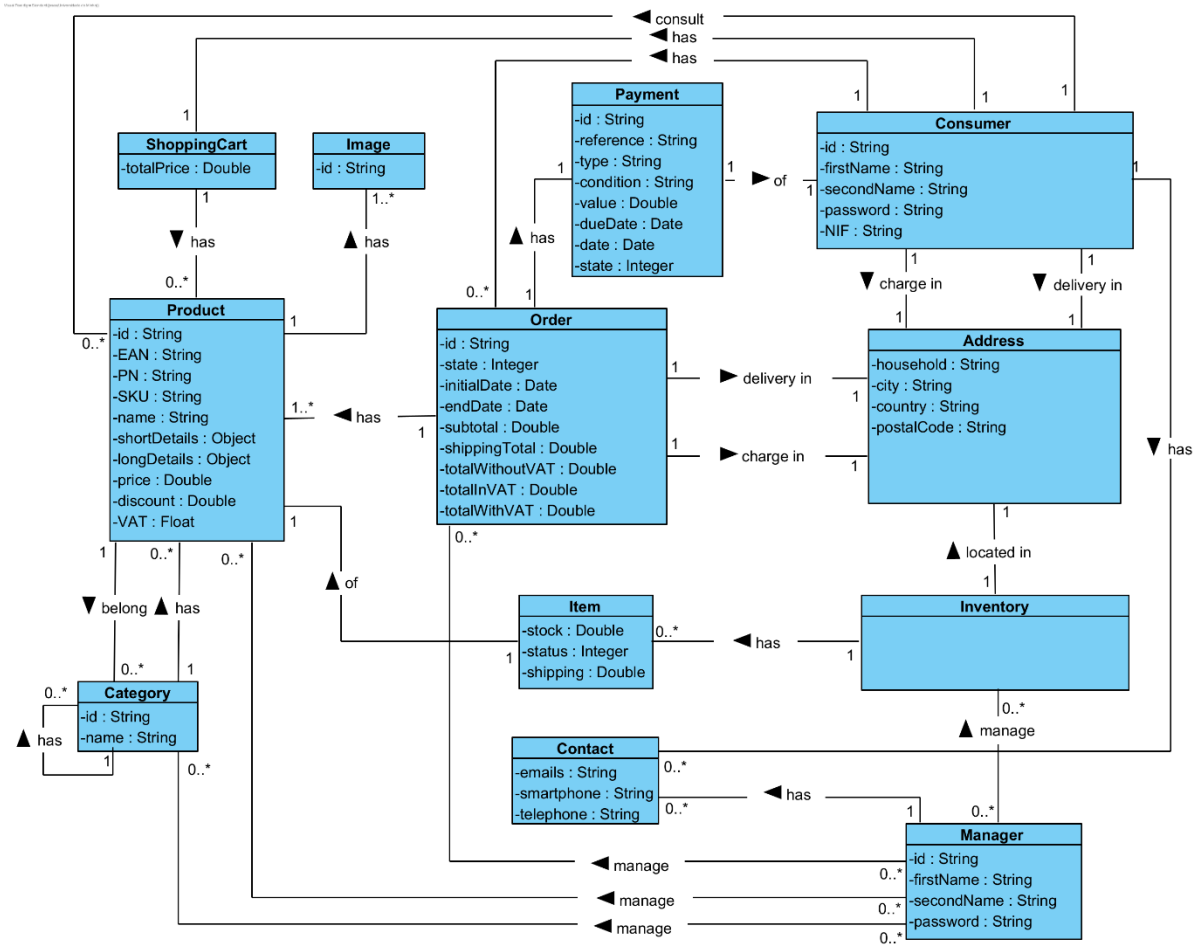


Figura 21: Modelo domínio do caso de estudo e-commerce.

Importante relembrar que, a análise e observação efetuada não foi detalhada ao pormenor, uma vez que, trata-se de um caso de estudo. Isto é, o mais importante na implementação deste sistema consiste na análise e aplicação do conhecimento obtido no estudo do estado da arte e não o modelo de negócio em si. No entanto, foram consideradas algumas noções essenciais para a conceção deste sistema de e-commerce abordadas de seguida.

Os sistemas de e-commerce, normalmente fornecem produtos ou serviços de um ou múltiplos vendedores. Neste caso, por forma a simplificar, visto que, para aplicar o conhecimento adquirido não se justifica aumentar a complexidade nesse sentido, decidiu-se focar apenas o estudo do domínio nos produtos e apenas um único vendedor. No entanto, sendo uma arquitetura orientada a microsserviços, tornar-se-á trivial em acrescentar a noção de serviço e múltiplos vendedores entre outras ideias no futuro.

A análise do domínio deve começar pela entidade central do mesmo, ou seja, o produto. Considera-se uma entidade central a nível de modelo de negócio, bem como arquitetural, uma vez que, contém relações diretas e indiretas com quase todas as outras entidades. Desta forma, prevê-se já na fase de análise, algo que se irá revelar mais adiante, que existem diversas decisões arquiteturais a serem tomadas para garantir de forma mais eficiente estas relações entre as entidades e o produto.

A entidade produto, constitui-se por um atributo id que permite identificar o produto no sistema. Do mesmo modo, existem três atributos abordados anteriormente que permitem a identificação do produto em outros contextos de negócio que são: o EAN (European Article Number), PN (Part Number) e o SKU (Stock Keeping Unit).

Por outro lado, a entidade produto contém atributos que permitem detalhar o mesmo, isto é, definir as suas características, tais como: name, shortDetails, longDetails e listas de imagens. O atributo denominado por name refere-se ao nome do produto e permite ao consumidor a identificação do mesmo. O atributo shortDetails refere-se à descrição sucinta das características mais importantes do produto permitindo ao consumidor obter essa informação de forma simples e rápida. O atributo longDetails refere-se à descrição detalhada das características do produto, onde existe normalmente textos completos a explicar as mesmas. A relação da entidade produto com a Imagem representa a lista de imagens do produto desde as miniaturas, bem como imagens detalhadas.

Ainda em relação à entidade produto, existe um conjunto de atributos que representam valores, tais como: price - preço do produto, discount - desconto/promoção sobre o produto, VAT - imposto sobre o produto.

A entidade categoria tem como objetivo categorizar/organizar os produtos em conjuntos com características semelhantes. A categoria tem um id como identificador e um name que normalmente designa a relação que existe entre os produtos de uma categoria (p.e. Telemóveis, Computadores, Android, iOS, etc.).

Além destes atributos, esta entidade contém uma relação recursiva para representar as subcategorias de uma categoria tornando esta categoria hierarquicamente superior a estas. Deste modo, exceto a categoria raiz (root), todas as outras categorias contêm sempre uma relação com a categoria superior. Por outro lado, todas as categorias têm um conjunto de subcategorias, das quais são superiores, formando assim caminhos até às categorias mais específicas de uma determinada característica.

Após apresentar as entidades produto e categoria individualmente, importa realçar a relação que existe entre estas duas entidades. A partir da navigabilidade do produto, o mesmo pode pertencer a nenhuma ou muitas categorias de produtos relacionados em termos de características. A partir da navigabilidade da categoria, a mesma contém uma lista de produtos, sendo que, esta relação permite chegar aos produtos a partir de uma categoria. Na verdade, isto significa que, esta relação entre estas entidades constitui numa funcionalidade de pesquisa/procura de produtos por categorias.

A entidade inventário, representa os locais físicos onde existe stock de itens de produtos. A distinção entre produto e item, serve para, no contexto do inventário distinguir o que representa os dados do produto e a noção de quantidade e preço de distribuição do produto.

A entidade consumidora, consiste num tipo de utilizador da aplicação constituído por um identificador id, bem como alguns dados pessoais como o nome, número de contribuinte e contactos, que normalmente são utilizados para verificações de login ou comunicações de alterações de estado ou novidades (p.e. promoções). Adicionalmente, esta entidade tem dois tipos de endereços, um para o pagamento e outro para o envio da encomenda, no entanto, normalmente ambos podem ser alterados no momento da compra.

Em relação aos comportamentos, a entidade consumidora pode consultar produtos, o seu carrinho de compras e as encomendas que efetuou.

A entidade carrinho de compras, como referido anteriormente, está relacionada com o consumidor e os produtos, uma vez que, representa a lista de produtos adicionados pelo consumidor. O mesmo contém associado o somatório dos preços dos produtos que estão incluídos nesse mesmo carrinho.

Cada consumidor contém um carrinho de compras, que por sua vez, contém uma lista de produtos que o mesmo pretende encomendar.

Após encomendar um carrinho de compras, é criada uma encomenda que guarda todas as informações associadas a este processo, tais como toda a informação atual dos produtos que estavam no carrinho de compras, o estado, a data em que se executou, e todos os valores importantes referentes às totalidades de preços, distribuições, taxas, e endereços físicos de pagamento e destino.

Por fim, a entidade gestor consiste num tipo de utilizador da aplicação constituído por um identificador id, bem como alguns dados pessoais, como o nome e contactos, que do mesmo modo que no consumidor, são utilizados para verificações de login ou comunicações de alterações de estado. Ainda referente a esta entidade, o mesmo tem diversos comportamentos de manutenção dos produtos, categorias, encomendas e o stock dos produtos que estão em inventário.

3.5. Requisitos Funcionais

Nesta secção, apresenta-se a lista resultante do levantamento e análise dos requisitos funcionais deste caso de estudo organizados por requisitos de utilizador e sistema.

3.5.1. Requisitos de Utilizador

Nos requisitos de utilizador, os mesmos estão agrupados por tipo de utilizador, isto é, consumidor não autenticado, consumidor autenticado, gestor e administrador, sendo que, este último não é apresentado nesta lista, uma vez que, poderá efetuar qualquer requisito apresentado.

Consumidor não autenticado

1. Como consumidor, registo-me.
2. Como consumidor, faço login.
3. Como consumidor, procuro produtos.
4. Como consumidor, consulto os dados (detalhes) de um produto.

Consumidor autenticado

5. Como consumidor, faço logout.
6. Como consumidor, procuro produtos.
7. Como consumidor, consulto os dados (detalhes) de um produto.

8. Como consumidor, adiciono, consulto, edito e removo dados da minha conta.
9. Como consumidor, adiciono, consulto, edito e removo a morada de cobrança.
10. Como consumidor, adiciono, consulto, edito e removo a morada de envio.
11. Como consumidor, adiciono, consulto, edito e removo produtos do carrinho de compras.
12. Como consumidor, encomendo os produtos do carrinho de compras.
13. Como consumidor, consulto as minhas encomendas.
14. Como consumidor, adiciono, consulto, edito e removo produtos como favoritos.

Gestor

15. Como gestor, faço login e logout.
16. Como gestor, adiciono, consulto, edito e removo dados da minha conta.
17. Como gestor, procuro produtos.
18. Como gestor, consulto os dados (detalhes) de um produto.
19. Como gestor, adiciono, edito e removo produtos.
20. Como gestor, consulto e edito o valor de stock dos produtos.
21. Como gestor, adiciono, consulto, edito e removo categorias.
22. Como gestor, adiciono produtos a categorias.
23. Como gestor, removo produtos de categorias.
24. Como gestor, consulto, edito e removo encomendas dos consumidores.

3.5.2. Requisitos do Sistema

25. O sistema deve ser modelado e implementado segundo uma arquitetura orientada a microsserviços.
26. O sistema deve utilizar os padrões de desenho mais adequados, quer a nível de microsserviço, quer a nível da arquitetura orientada a microsserviços.

3.6. Requisitos Não Funcionais

Nesta secção, apresenta-se a lista resultante do levantamento e análise dos requisitos não funcionais deste caso de estudo organizados por classificações.

Desempenho

27. O sistema deve conter mecanismos de escalabilidade.
28. O sistema deve conter mecanismos para garantir a consistência dos dados.
29. O sistema deve conter mecanismos de resiliência e tolerância a faltas.
30. O sistema deve conter mecanismos para garantir um aumento da disponibilidade das funcionalidades críticas, face à sua disponibilidade base.

Operacional

31. O sistema deve implementar um estilo arquitetural de software REST.
32. O sistema deve permitir a interoperabilidade com aplicações/sistemas terceiros (p.e. sistemas/aplicações de pagamentos, gestão de inventários, CRM, etc.).

Manutenção

33. O sistema deve permitir a manutenção, melhoria e adição de novas funcionalidades na aplicação de forma eficiente.
34. O sistema deve ser modelado e documentado para facilitar a sua compreensão quer por componente, quer a nível arquitetural.

Segurança

35. O sistema deve garantir que não são introduzidos dados incorretos.
36. O sistema deve garantir o acesso a determinadas funcionalidades e dados apenas a utilizadores autenticados e autorizados.

3.7. Casos de Uso

Nesta secção, apresenta-se a modelação dos requisitos funcionais, através de um diagrama de casos de uso do sistema e-commerce, tal como se pode observar na Figura 22.

Começando pelos atores deste sistema, identificam-se três tipos principais: o consumidor (Consumer), gestor (Manager) e administrador (Admin).

O consumidor pode conter duas generalizações, que são, de não autenticado e autenticado, que servem para distinguir que casos de uso implicam autenticação do consumidor. Achou-se importante dar relevância a esta diferença de autenticado e não

autenticado, uma vez que, existem muitos utilizadores que utilizam este tipo de aplicações sem estarem autenticados ou até mesmo terem uma conta de utilizador.

Por outro lado, todos os casos de uso do gestor, bem como do administrador requerem a autenticação, exceto o caso de uso login.

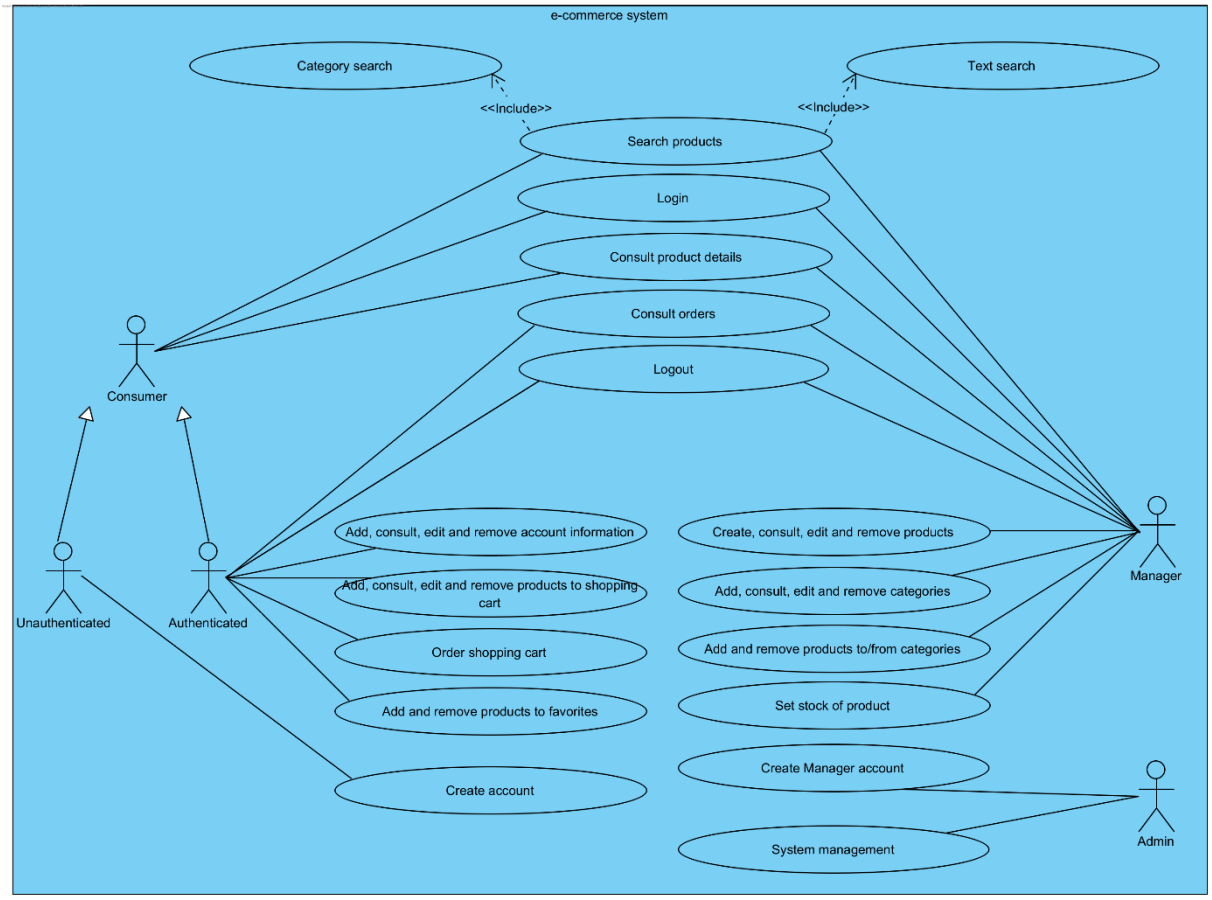


Figura 22: Diagrama de casos de uso do sistema e-commerce.

Em relação ao ator consumidor o mesmo contém um conjunto de casos de uso que podem ser efetuados quer autenticado ou não. Alguns casos de uso deste conjunto são a procura de produtos, que inclui dois tipos de procuras, por texto e categorias. Ainda nesse conjunto, existe o caso de uso de acesso aos detalhes de um produto específico, bem como o login que permite ao consumidor autenticar-se.

Em relação ao ator generalizado do consumidor, designado por não autenticado, o mesmo herda todos os casos de uso anteriormente abordados do consumidor. No entanto, o consumidor não autenticado pode efetuar um caso de uso exclusivo do mesmo que consiste na criação de uma conta de utilizador.

Em relação ao ator generalizado do consumidor, designado por autenticado, do mesmo modo, herda todos os casos de uso anteriormente abordados do consumidor. No entanto, contém outros casos de uso exclusivos, que consistem na gestão da informação da conta, definir produtos favoritos, adicionar e remover produtos do carrinho de compras, proceder à compra do mesmo e consultar as encomendas resultantes.

Em relação ao ator gestor, o mesmo partilha um conjunto de casos de uso com o consumidor base e autenticado, seguidos de casos de uso específicos a este ator que são a gestão de produtos, categorias, as relações entre estes e atribuição de valor de stock a um determinado produto.

Por fim, em relação ao ator administrador, o mesmo contém dois casos de uso que consiste na criação da conta de utilizador de gestores e a gestão do sistema. A gestão do sistema, consiste num caso de uso genérico para representar todas as funcionalidades utilizadas para esse efeito desde configuração, desenvolvimento, teste e deployment.

Assim, com o estudo dos casos de uso através da modelação com o diagrama de casos de uso consegue-se segregar as funcionalidades por utilizador, bem como perceber as que podem ser comuns e neste caso, entender previamente a existência de utilizadores que podem usar a aplicação sem estar autenticados. Ainda se torna importante realçar que, dos casos de uso apresentados existem muitos que na realidade podem ser segregados, uma vez que, foram colocados no mesmo caso de uso as diversas operações CRUD sobre uma entidade, como por exemplo: os produtos, as categorias, entre outras.

3.8. Conclusão

Em conclusão, a seleção deste caso de estudo tornou-se desafiante, na medida em que, tem de ter a complexidade suficiente para permitir aplicar a maior parte dos princípios e padrões estudados. Deste modo, foram estudados alguns casos de estudo, no entanto, aquele que pareceu mais completo foi um sistema e-commerce, uma vez que, tem complexidade relevante. Ainda, verifica-se que este caso de estudo é muitas vezes utilizado como exemplo para explicar e apresentar os princípios e padrões dos microsserviços.

A conceção do caso de estudo, como referido neste capítulo, seguiu o método *Unified Process*. Deste modo, realizou-se a observação do domínio e-commerce para se conhecer o mesmo, obtendo-se como resultado o modelo de domínio, seguido da análise e levantamento dos requisitos. Em relação à análise do domínio, a mesma foi elaborada com recurso à

consulta de aplicações web de e-commerce comuns em Portugal, tais como: PCDiga, PcComponentes, Worten, Fnac, Wook, Amazon, entre outros. Ainda foram analisados alguns recibos de faturação provenientes de algumas das lojas referidas anteriormente. No entanto, o processo de análise passou muito pela visão do consumidor, uma vez que, não existe fácil acesso às funcionalidades dos gestores. Deste modo, a visão do domínio do gestor foi prevista com base nas necessidades do consumidor.

Em suma, o processo de análise torna-se implícito em qualquer sistema, no entanto, verificou-se que, numa arquitetura orientada a microsserviços torna-se crucial, uma vez que, tal como foi abordado nos padrões da categoria de decomposição, secção 2.2.1, o conhecimento do domínio é o que permite uma decomposição estável dos microsserviços.

4. Arquitetura Genérica dos Microsserviços do Sistema E-commerce

Neste capítulo, abordar-se-á todo o processo e decisões efetuadas para o desenho de uma arquitetura orientada a microsserviços para o sistema e-commerce, como caso de estudo da aplicação dos conhecimentos estudados no capítulo 2.

É importante referir que, a ordem pela qual são apresentados nos próximos capítulos e secções, os estudos, decisões, modelos entre outros métodos para definição da arquitetura, não corresponde na sua totalidade à ordem pela qual foram efetuados. Na verdade, por forma a perceber-se melhor os conceitos e não se repetir informações decidiu-se começar pela apresentação das abordagens, componentes e padrões genéricos a qualquer microsserviço. De seguida, no capítulo 5, serão descritas e apresentadas as arquiteturas de cada microsserviço, bem como, os padrões, princípios e tecnologias aplicados.

4.1. Padrões arquiteturais implementados no Caso de Estudo

Nesta secção, são apresentados e justificados os padrões arquiteturais implementados no caso de estudo, dos quais a arquitetura hexagonal, muito referenciada para arquiteturas orientadas a microsserviços e o princípio de inversion of control utilizando o padrão de desenho dependecy injection.

4.1.1. Princípio de Inversion of Control com padrão Dependecy Injection

O princípio de Inversion of Control, tal como o nome em inglês sugere, consiste em inverter o controlo do processo de desenvolvimento do programador, isto é, este deixa de ser responsável por reimplementar a base arquitetural dos microsserviços. Na verdade, a inversão de controlo do programador, tem como consequência, uma maior reutilização de componentes de software, bem como uma manutenção de código mais trivial. Desta forma, a arquitetura base de vários microsserviços pode ser reutilizada evitando-se redundâncias e garantindo uma base confiável de software.

O dependecy injection surge como um padrão que contribui para o inversion of control do programador, dado que, permite a injeção de dependências o que garante uma abstração das mesmas. Na verdade, delega-se a instanciação de objetos para outras instâncias, separando esta responsabilidade do programador permitindo que este se foque apenas na utilização de objetos e serviços.

Efetivamente, justifica-se a aplicação do inversion of control utilizando dependecy injection neste caso de estudo. Na verdade, motivado por ser uma arquitetura orientada a microsserviços onde a redução de dependências, aumento de produtividade e facilidade de manutenção são características importantes. Do mesmo modo, como se abordará mais adiante, na secção 4.1.2, os microsserviços são característicos por facilitarem a migração de tecnologias (p.e. base de dados, message brokers, entre outras), pelo que, o padrão dependecy injection permite uma maior abstração dessas alterações.

4.1.2. Arquitetura Hexagonal em Microsserviços

Uma arquitetura hexagonal, consiste numa alternativa à arquitetura por camadas sendo constituída, tal como se pode observar na Figura 23, pela lógica de negócio (região amarelo) centrada e adaptadores (regiões verdes) nas periferias.

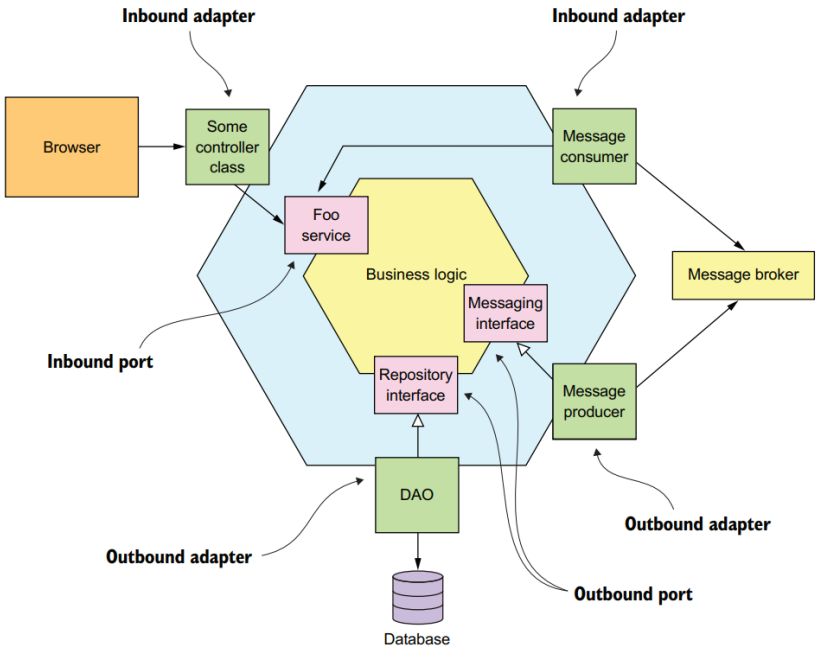


Figura 23: Ilustração conceptual de uma arquitetura hexagonal de um serviço *[1]*.

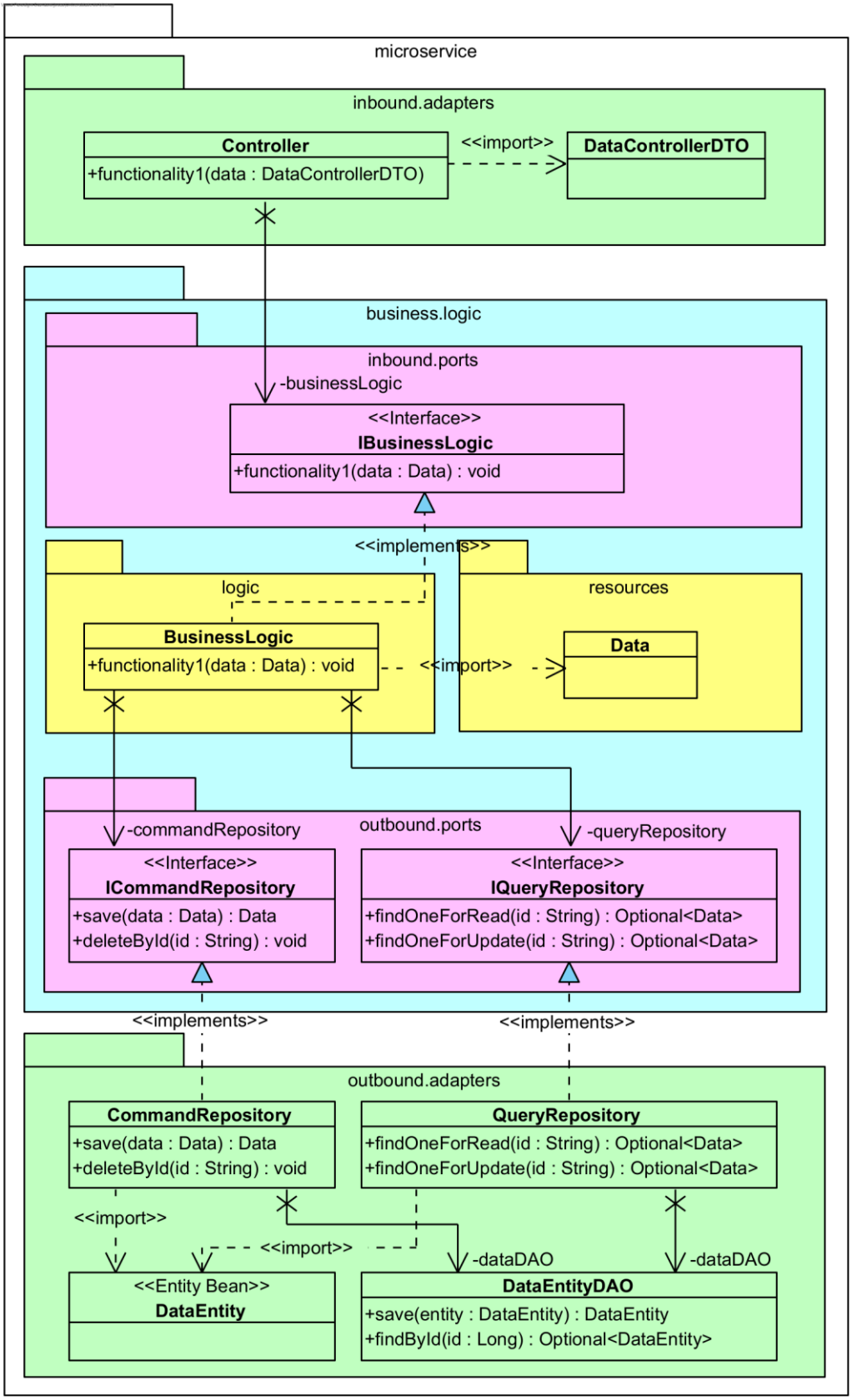


Figura 24: Diagrama de classes, como exemplo genérico de uma arquitetura hexagonal.

A Figura 24, vai ser utilizada como exemplo na explicação da arquitetura hexagonal, devido a ser um diagrama de classes, ou seja, mais completo. No entanto, o mesmo foi colorido para permitir identificar os packages do diagrama de classes na ilustração conceptual da arquitetura hexagonal da Figura 23.

Os adaptadores são responsáveis pela comunicação com o exterior do microsserviço e adaptação dessa comunicação com a lógica de negócio. Na verdade, esta adaptação passa por receber os dados num método ou protocolo (p.e. HTTP, AMQ, RPC, entre outros) mais adequado ao cliente e mapear os mesmos para os objetos que a lógica de negócio utiliza. O objeto que permite esta adaptação de dados chama-se DTO (Data Transfer Object) e está definido nos adaptadores, onde são implementados mecanismos para mapear os mesmos para os objetos que a lógica de negócio utiliza.

Ainda em relação aos adaptadores, os mesmos organizam-se em adaptadores de entrada (inbound adapters) e saída (outbound adapters), onde a diferença persiste em quem executa o adaptador. Um adaptador de entrada significa que está a ser executado por uma entidade exterior ao microsserviço, como por exemplo: controllers, recetores de message brokers, entre outros, onde o objetivo consiste em executar uma funcionalidade da lógica de negócio. Por outro lado, um adaptador de saída significa que está a ser executado pela lógica de negócio, onde o objetivo consiste em esta executar um serviço externo, tais como: base de dados, produtores de message brokers, entre outros.

Ainda em relação aos adaptadores, torna-se importante referir que, está inerente a implementação do padrão de desenho denominado por Adapter [33], que tem como objetivo adaptar objetos por forma a permitir a colaboração entre duas partes.

Efetivamente, para garantir que a lógica de negócio não depende dos adaptadores de entrada e saída, entre estes (entre a lógica de negócio e os adaptadores), existem portas (regiões rosa) respetivamente de entrada e saída. Como se pode observar na Figura 24, as portas de entrada (inbound ports), são implementadas na preferia da lógica de negócio e definem as funcionalidades expostas por esta, a serem executadas pelos adaptadores de entrada. Por outro lado, as portas de saída (outbound ports), são implementadas também na periferia da lógica de negócio, mas neste caso definem as funcionalidades a serem executadas pela lógica de negócio em outros serviços externos.

Ainda, torna-se importante reforçar que, a abstração criada pelas interfaces das portas de entrada e saída, entre a lógica de negócio e os adaptadores, apenas se consegue com a

utilização do padrão dependecy injection, abordado anteriormente na secção 4.1.1. Na verdade, na lógica de negócio estão definidas as interfaces das portas, ou seja, a dependência com estas, no entanto, o conhecimento das classes concretas e o processo de instanciação dessas torna-se totalmente abstrato. Deste modo, consegue-se o objetivo primordial da arquitetura hexagonal que consiste em tornar a lógica de negócio independente dos adaptadores de entrada e saída.

Em conclusão, uma arquitetura hexagonal, permite que a lógica de negócio seja independente dos adaptadores, possibilitando que estes possam ser alterados, sem implicar alterações na lógica de negócio, pelo que, estas vantagens justificam a utilização desta abordagem na arquitetura orientada a microsserviços para o caso de estudo e-commerce.

4.2. Componentes e Tecnologias em Comum nos Microsserviços

Nesta secção, são apresentados os componentes arquiteturais e tecnologias comuns em todos os microsserviços, modelados e implementados para o caso de estudo e-commerce. Importante reforçar que, os padrões arquiteturais abordados anteriormente na secção 4.1, também são considerados componentes comuns a qualquer microsserviço.

4.2.1. Comunicação Assíncrona por Mensagens entre Microsserviços

Nas arquiteturas orientadas a microsserviços, a comunicação e interação entre estes torna-se algo inevitável, pelo que, são necessários mecanismos para lidar com diversos problemas que ocorrem nestas situações, abordados ao longo do capítulo 2, dos quais, a disponibilidade e latências dos microsserviços entre outros.

Existem dois tipos de comunicações possíveis entre microsserviços, síncronas, que se caracterizam por esperas ativas entre a entidades que se comunicam e assíncronas, que se caracterizam por não ter uma espera ativa entre as entidades.

Efetivamente, para contornar os problemas das comunicações entre microsserviços normalmente opta-se por comunicações assíncronas, enquanto as comunicações síncronas são usadas maioritariamente entre o utilizador e o microsserviço.

Estrutura da Mensagem Utilizada nas Comunicações Assíncronas

A comunicação entre microsserviços ocorre frequentemente e existirá neste caso de estudo a aplicação de diversas decisões e padrões das arquiteturas orientadas a

microsserviços, que implicam interação entre os mesmos. Deste modo, decidiu-se definir uma estrutura para as mensagens trocadas entre microsserviços, para tornar este processo mais estrutural, consistente e organizado, tal como se pode observar na Listagem 1.

```
{
        "owner": "saga",
        "exchange": "saga.publisher.exchange",
        "routing": "create.product.saga.0.cp.create.product",
        "type": "COMMAND",
        "status": "",
        "code": "",
        "message": "",
        "method": "create.product.saga.0.cp.create.product ",
        "metadata": {
                "token": "...",
                "sagaId": "..."
        },
        "data": {
                "id": "0",
                "name": "Smartphone XPTO"
                "shortDetails": "CPU XPTO, 8 GB RAM, ...",
                ...
        }
}
```

Listagem 1: Estrutura de uma mensagem, enviada entre microsserviços, de forma assíncrona.

A mensagem compõe-se por um campo denominado owner que identifica o microsserviço origem onde a mensagem foi produzida. Para além do uso de monitorização, este campo permite aos microsserviços consumidores saberem para que microsserviço têm de responder. Apesar de na própria lógica poder-se definir para que microsserviço o mesmo tem de responder, tornar este processo dinâmico, isto é, usar o campo owner para saber o destino da resposta contribui para garantir a escalabilidade por particionamento de dados. Na verdade, neste tipo de escalabilidade, existem múltiplas instâncias do mesmo microsserviço,

sendo que, cada instância terá a sua própria designação de owner o que permite enviar a mensagem de resposta para a instância correta.

Os campos exchange e routing permitem definir e direcionar o destino da mensagem, concedendo maior capacidade para filtrar as mesmas dentro de um conjunto de consumidores de um determinado exchange.

O campo type diferencia os dois tipos de mensagens existentes: comando (*COMMAND*) e resposta (*REPLY*). Uma mensagem do tipo comando, significa que irá executar um comando/funcionalidade no microsserviço destino, pelo que, por vezes poderá gerar uma mensagem do tipo resposta. Uma mensagem do tipo resposta, como referido anteriormente, ocorre sempre após uma mensagem comando e normalmente serve para devolver o resultado da operação executada pela mesma.

Os campos status, code e message, ocorrem normalmente nas mensagens do tipo resposta e tem como finalidade avaliar o resultado da mensagem comando que antecede sempre uma mensagem resposta.

O campo method permite especificar o método a executar no microsserviço que recebe a mensagem, quer seja de comando ou resposta.

O campo metadata, ao contrário dos anteriores, tem um formato json, o que significa que, aceita qualquer estrutura de dados. Deste modo, a utilização deste campo tem como finalidade, tal como o nome sugere, de dados auxiliares aos dados da mensagem, como por exemplo: tokens, identificadores, versionamento de mensagens, entre outros.

Por fim, o campo data, também tem o formato json, pelo que, aceita qualquer tipo de estrutura de dados e tem como finalidade transportar os dados essenciais para a execução de um comando no consumidor ou no caso de uma mensagem do tipo resposta o output desse comando executado.

Em suma, a definição de uma estrutura para as mensagens torna-se importante, pois permite uma maior consistência e organização da comunicação. No entanto, um ponto crítico desta estrutura consiste em algum overhead, uma vez que, em termos de execução da funcionalidade, o que interessa a ambos os microsserviços, produtor e recetor, são os metadados e dados, os restantes apenas são utilizados para gestão das mensagens. Todavia, neste tipo de arquiteturas, torna-se importante balancear entre o overhead e a consistência, estrutura e organização da comunicação entre microsserviços.

Arquitetura da tecnologia RabbitMQ em Microsserviços

A definição da arquitetura de comunicação assíncrona por mensagens entre microsserviços começa pela seleção da tecnologia broker mais adequada. A tecnologia selecionada que satisfaz os requisitos trata-se do RabbitMQ[36] devido ao seu equilíbrio entre independência, estruturação e performance, três características importantes de acordo com os objetivos deste caso de estudo.

Desta forma, torna-se necessário perceber como funciona a tecnologia RabbitMQ a nível arquitetural e de que forma a mesma deve ser implementada nos microsserviços, para que não seja quebrado nenhum princípio dos mesmos.

Tal como se pode observar na Figura 25, a tecnologia RabbitMQ constitui-se por três grandes componentes: exchanges, routing keys e queues. Ao contrário de outras tecnologias de envio de mensagens, nesta tecnologia o produtor não envia as mensagens diretamente para as queues, mas sim para o exchange definindo uma determinada routing key, que por sua vez, interliga o exchange às queues. Desta forma, torna-se totalmente abstrato para o produtor que consumidores existem e respetivas queues. Por outro lado, as queues, subscrevem-se a exchanges utilizando uma routing key.

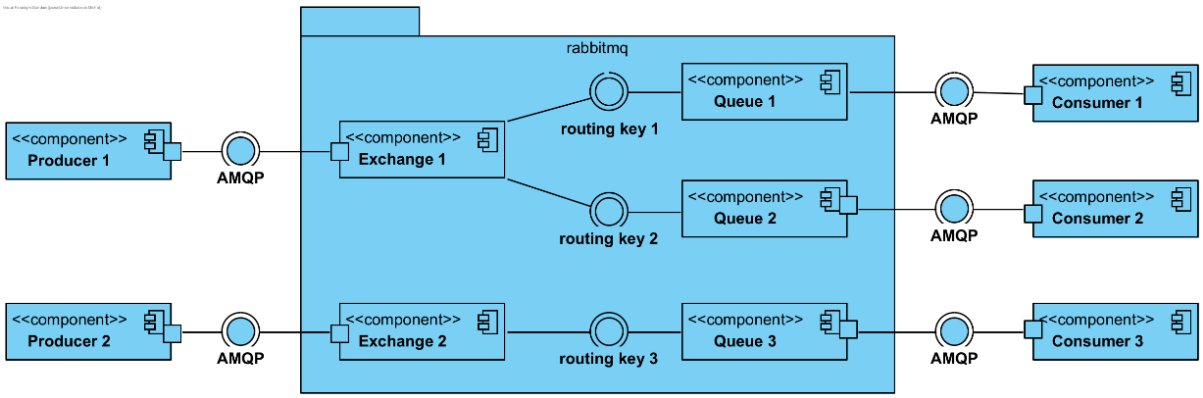


Figura 25: Arquitetura da tecnologia RabbitMQ.

Com a análise da arquitetura da tecnologia RabbitMQ, conclui-se que o exchange está fortemente ligado ao produtor, isto é, este apenas conhece o mesmo e as possíveis routing keys, enquanto o consumidor, por outro lado, está fortemente ligado às queues.

Tal como se pode observar na Figura 26, em consequência da arquitetura do RabbitMQ e dada a importância dos limites bem definidos de um microsserviço, decidiu-se especificar

[36] https://www.rabbitmq.com/tutorials/amqp-concepts.html

que, qualquer microsserviço deste caso de estudo e-commerce que comunique de forma assíncrona através de mensagens, seja proprietário e responsável de uma queue que utilizará para subscrever exchanges de outros microsserviços (p.e. Producer 1 e Producer 2) dos quais tenha interesse em consumir as mensagens. Por outro lado, cada microsserviço também será proprietário de um exchange que utilizará para produzir mensagens que outros microsserviços (p.e. Consumer 1 e Consumer 2) pretendam consumir e por esse motivo subscrevem as suas queues a este exchange.

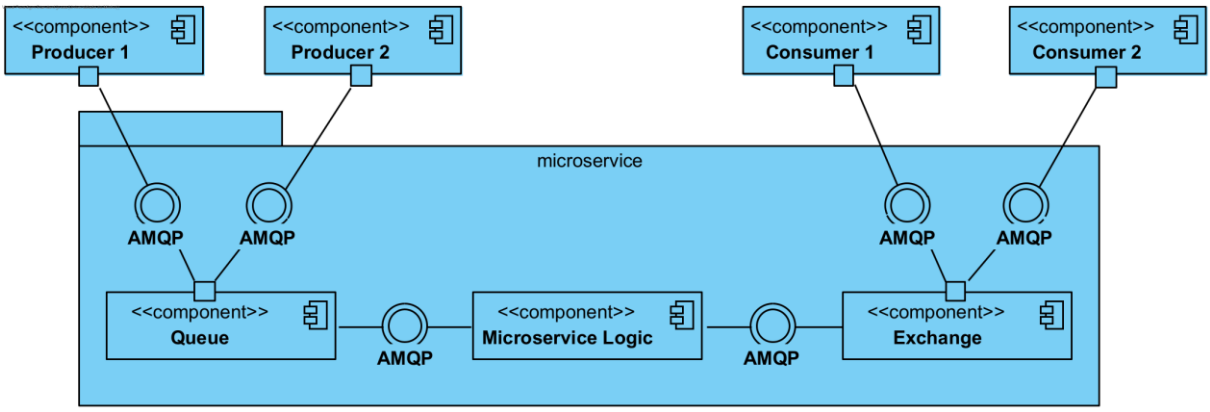


Figura 26: Limites bem definidos dos microsserviços, quanto à comunicação assíncrona por mensagens, utilizando a tecnologia RabbitMQ.

Arquitetura tipo para Manipulação das Mensagens nos Microsserviços

O processo de manipulação de uma mensagem, quer seja a produzir ou receber, torna-se num exercício repetitivo ao longo dos microsserviços. Por este motivo, decidiu-se definir uma arquitetura a nível de software que pode ser reutilizada em todos os microsserviços por forma a evitar-se código redundante.

Previamente a especificar a arquitetura de software desenhada, torna-se importante perceber o fluxo das mensagens num microsserviço.

Quando uma mensagem é recebida num microsserviço a mesma passa por um processo repetitivo que consiste na receção da mensagem, validação da sua estrutura de acordo com a apresentada anteriormente, seguido do mapeamento da mesma para um objeto que a lógica de negócio utiliza. Por outro lado, o envio da mensagem torna-se num processo mais simplificado, pois apenas cria e envia a mensagem preenchendo a sua estrutura com os dados necessários.

A Figura 27, consiste no diagrama de classes que representa a arquitetura tipo a nível de software presente em todos os microsserviços para a manipulação de mensagens.

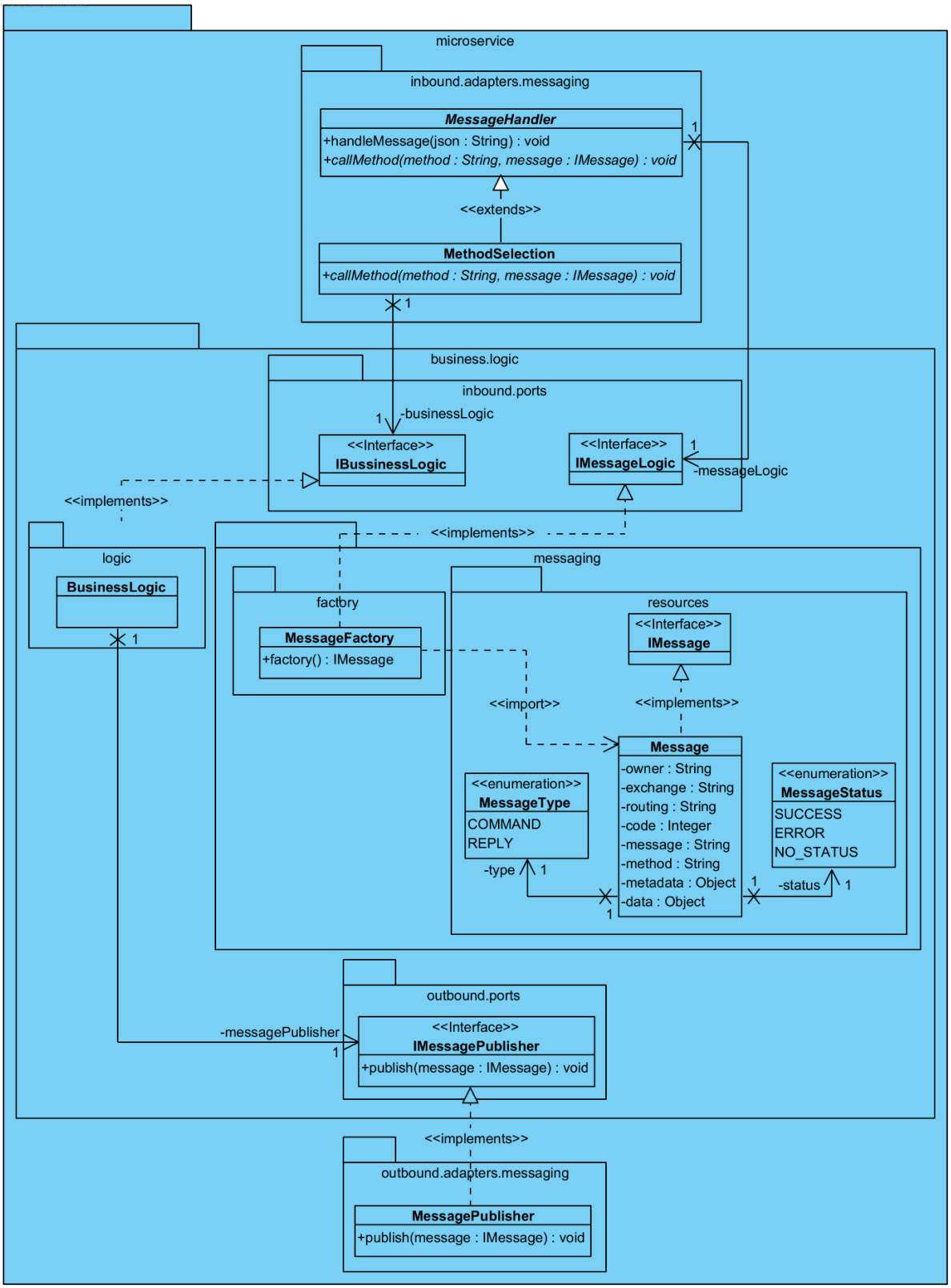


Figura 27: Diagrama de classes da arquitetura de manipulação de mensagens ("handling", "publishing"), que será implementada em todos os microsserviços do problema e-commerce.

Começando pela estrutura de dados utilizada para representar a mensagem, a classe Message, decidiu-se que a mesma ao contrário do que seria suposto será definida na lógica de negócio, uma vez que, a mesma utiliza-se quer nos adaptadores de entrada, quer nos adaptadores de saída para mapeamento das mensagens recebidas, mas principalmente na própria lógica de negócio relacionada com a implementação de certos padrões (p.e. Saga, CQRS, entre outros). Por estes motivos, decidiu-se que a lógica de negócio deve ter a responsabilidade por esta estrutura de dados, dado que, a mesma não pode depender dos adaptadores. No entanto, percebe-se que a noção de mensagem está muito relacionada com os adaptadores, pelo que, seria esperado que a mesma fosse definida nos mesmos e que a lógica de negócio apenas deveria receber os dados que vem na mensagem.

Porém, os princípios do padrão hexagonal são mantidos, na medida em que, o adaptador de entrada recebe a mensagem num formato String e a mesma é mapeada para o objeto do tipo IMessage (responsável pelo encapsulamento da classe Message) da lógica de negócio garantindo-se o objetivo principal que consiste em tornar a lógica de negócio independente dos adaptadores.

Ainda em relação à mensagem, implementou-se o padrão de desenho denominado por Factory Method [33] para instanciação do objeto Message, que por sua vez, também encapsulado por uma interface denominada IMessage, para que não se criem dependências com a classe concreta Message. Na verdade, esta vantagem está representada no diagrama de classes com a relação de dependência "import" entre a classe concreta do MessageFactory com a classe concreta do Message, para demonstrar que apenas esta a conhece. Ainda existe uma camada de encapsulamento sobre a classe concreta da fábrica do objeto Message com uma interface definida nas portas de entrada do microsserviço, denominada por IMessageLogic, seguindo a arquitetura hexagonal abordada na secção 4.1.2.

Por fim, ainda em relação à estrutura de dados da mensagem a mesma tem uma relação de composição com duas enumerações, MessageType e MessageStatus, que respetivamente listam os tipos e estados possíveis da mensagem.

Em relação ao processo repetitivo que acontece desde a receção da mensagem até à execução de uma funcionalidade da lógica de negócio, para se conseguir reutilizar o máximo de código possível decidiu-se aplicar o padrão de desenho Template Method [33].

A implementação do padrão de desenho Template Method consiste na criação de uma framework reutilizável onde se específica o comportamento esqueleto, que neste caso corresponde ao comportamento repetitivo na manipulação da mensagem.

Desta forma, tal como se pode observar na Figura 27, no adaptador de entrada das mensagens define-se uma classe abstrata que será comum a todos os microsserviços, denominada por MessageHandler, que implementa um método chamado handleMessage. O método handleMessage define o esqueleto do processo repetitivo e chama internamente o método abstrato callMethod que não está implementado nessa classe. Na verdade, o método callMehtod será implementado numa classe concreta que estende a classe abstrata MessageHandler. Por esse motivo, terá de implementar esse método com o comportamento que é específico a cada microsserviço, que neste caso consiste na decisão de qual funcionalidade da lógica de negócio executar e por isso existe uma relação de composição com a interface da lógica de negócio IBusinessLogic.

Assim, a aplicação deste padrão explicado anteriormente, permite a reutilização do código esqueleto implementado no método handleMessage, que injeta o código que é específico a cada microsserviço nesse mesmo método através do método abstrato denominado por callMethod.

Por fim, em relação à publicação de mensagens por parte dos microsserviços, como referido anteriormente esse processo torna-se mais simplificado, pelo que, como se pode observar na Figura 27, na porta de saída, existe uma interface que expõe o método de publicação de mensagens, que por sua vez, é implementado por uma classe concreta denominada MessagePublisher.

Versionamento de Mensagens do tipo Comando

O versionamento de mensagens do tipo comando foi motivado pela aplicação do padrão CQRS neste caso de estudo de e-commerce.

Efetivamente, o objetivo do versionamento de mensagens do tipo comando consiste em identificar as mesmas com uma ordem de execução dos comandos. Na verdade, o versionamento permite garantir a ordem de execução dos comandos nos microsserviços de leitura, pela mesma ordem que foram executados nos microsserviços de comando, garantindo-se assim a consistência dos dados no CQRS. Isto torna-se necessário, uma vez que,

não existe garantia que as mensagens chegam pela mesma ordem aos microsserviços de leitura como na execução das operações de comando.

A Figura 28, representa o diagrama de classes da arquitetura do versionamento de mensagens nos microsserviços de comando e leitura.

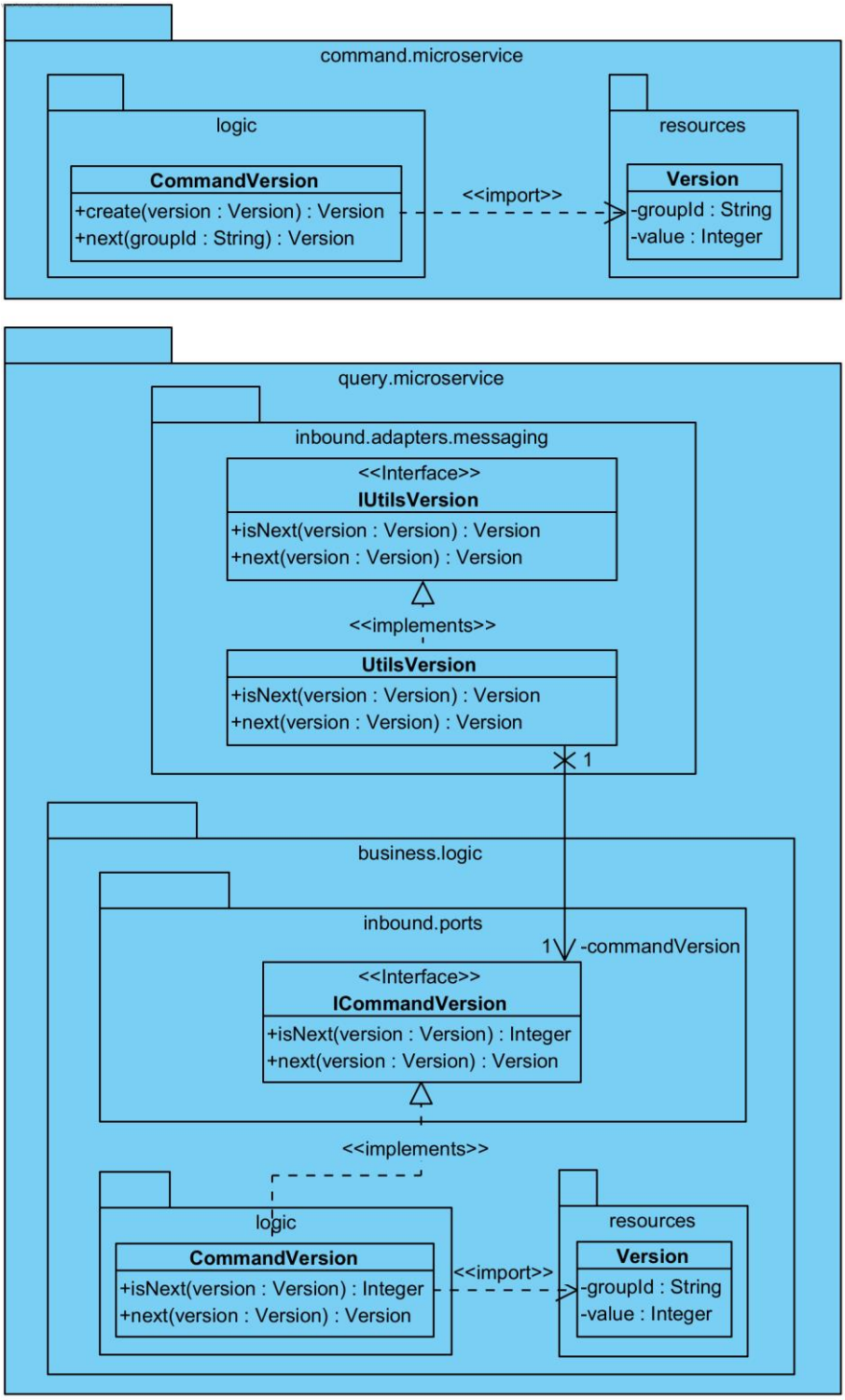


Figura 28: Diagrama de classes, que demonstra a arquitetura necessária para implementar o versionamento de mensagens num CQRS.

Deste modo, a nível de estruturas de dados em ambos os microsserviços de comando e leitura existe uma classe concreta denominada por Version, que representa o versionamento de uma ou um conjunto de entidades.

A nível de atributos, na classe Version existe um identificador denominado por groupId normalmente associado à(s) entidade(s) respetiva(s) (p.e. se a entidade for as categorias pode ser o id da categoria respetiva). O atributo value corresponde ao valor da versão, isto é, uma espécie de contador que vai sendo incrementado a cada operação comando executada sobre a(s) entidade(s) respetiva(s).

A nível de funcionalidades relativamente ao microsserviço comando ou emissor de mensagens do tipo comando, existe o create que como o nome sugere cria o primeiro registo de uma versão. Por outro lado, existe a funcionalidade next que incrementa a versão retornando a mesma para ser agregada na mensagem.

Em relação às funcionalidades dos microsserviços de leitura ou consumidores de mensagens de tipo comando, as mesmas dividem-se sobre as duas camadas, isto é, nos adaptadores de entrada e na lógica de negócio.

Começando pela lógica de negócio, uma vez que, os adaptadores de entrada utilizam a mesma, existe uma funcionalidade comum ao microsserviço de comando, o next, que tem o mesmo papel de sincronizar o valor da versão, isto é, incrementando. Por outro lado, existe uma funcionalidade denominada por isNext que verifica se a versão dada como argumento coincide com a versão corrente. Uma particularidade desta funcionalidade consiste em a mesma retornar a distância (número de versões) que a versão dada como argumento está da versão corrente.

Por outro lado, a lógica que está definida no adaptador de entrada, apesar de utilizar a lógica de negócio, relação de composição denominada por commandVersion, a mesma tem a responsabilidade de efetuar as verificações. Na verdade, executa o isNext da lógica de negócio e em caso de este retornar uma distância diferente de zero, calcula-se de forma dinâmica o tempo de espera (com base na distância) para a próxima verificação desta mesma versão. Deste modo, evita-se que mensagens de comando próximas da versão corrente fiquem muito tempo à espera, e, por outro lado, outras que estejam mais distantes da versão corrente verifiquem demasiadas vezes a sua vez.

Ainda, torna-se importante referir que, na arquitetura do versionamento está inerente a aplicação do padrão de desenho denominado por Iterator [33] que tem como objetivo iterar

pela ordem correta elementos de uma estrutura de dados. Neste caso, a estrutura de dados consiste na lista de mensagens do tipo comando que têm de ser iteradas e executadas pela ordem correta para se garantir a consistência.

Assim, com o versionamento das mensagens consegue-se garantir a consistência dos dados, após a execução das mensagens do tipo comando sendo neste caso de estudo a sua utilização mais considerável na aplicação do padrão CQRS que será abordada mais adiante. Ainda, outro ponto importante relativamente ao versionamento consiste no retorno da distância à versão corrente na verificação, permitindo que seja calculado os tempos de espera de forma dinâmica reduzindo-se o número de verificações de versão.

4.2.2. Arquitetura tipo para implementação do Padrão CQRS

Nesta secção, apresenta-se a arquitetura tipo para implementação do padrão CQRS abordado na secção 2.2.2.

Por forma a evitar-se repetitivamente apresentar a mesma arquitetura tipo nos vários microsserviços que implementam este padrão CQRS, decidiu-se apresentar a mesma nas arquiteturas comuns e as motivações da aplicação deste padrão nas descrições dos microsserviços respetivos.

A Figura 29, consiste no diagrama de classes da arquitetura tipo da aplicação do padrão CQRS, isto é, do microsserviço comando, meio de comunicação (p.e. tecnologia message broker RabbitMQ) e microsserviço de leitura.

Começando pelo microsserviço comando, o mesmo implementa a lógica de negócio respetiva às operações de comando sobre uma entidade denominada no diagrama por Entity. Seguindo a abordagem da arquitetura hexagonal, abordada anteriormente na secção 4.1.2, definem-se as interfaces para cada tipo de operação comando com um método de sincronização que será invocado a cada operação deste tipo. No pacote dos adaptadores de saída, implementam-se as classes concretas respetivas com a lógica necessária de mapeamentos de dados, versionamento e envio de mensagens.

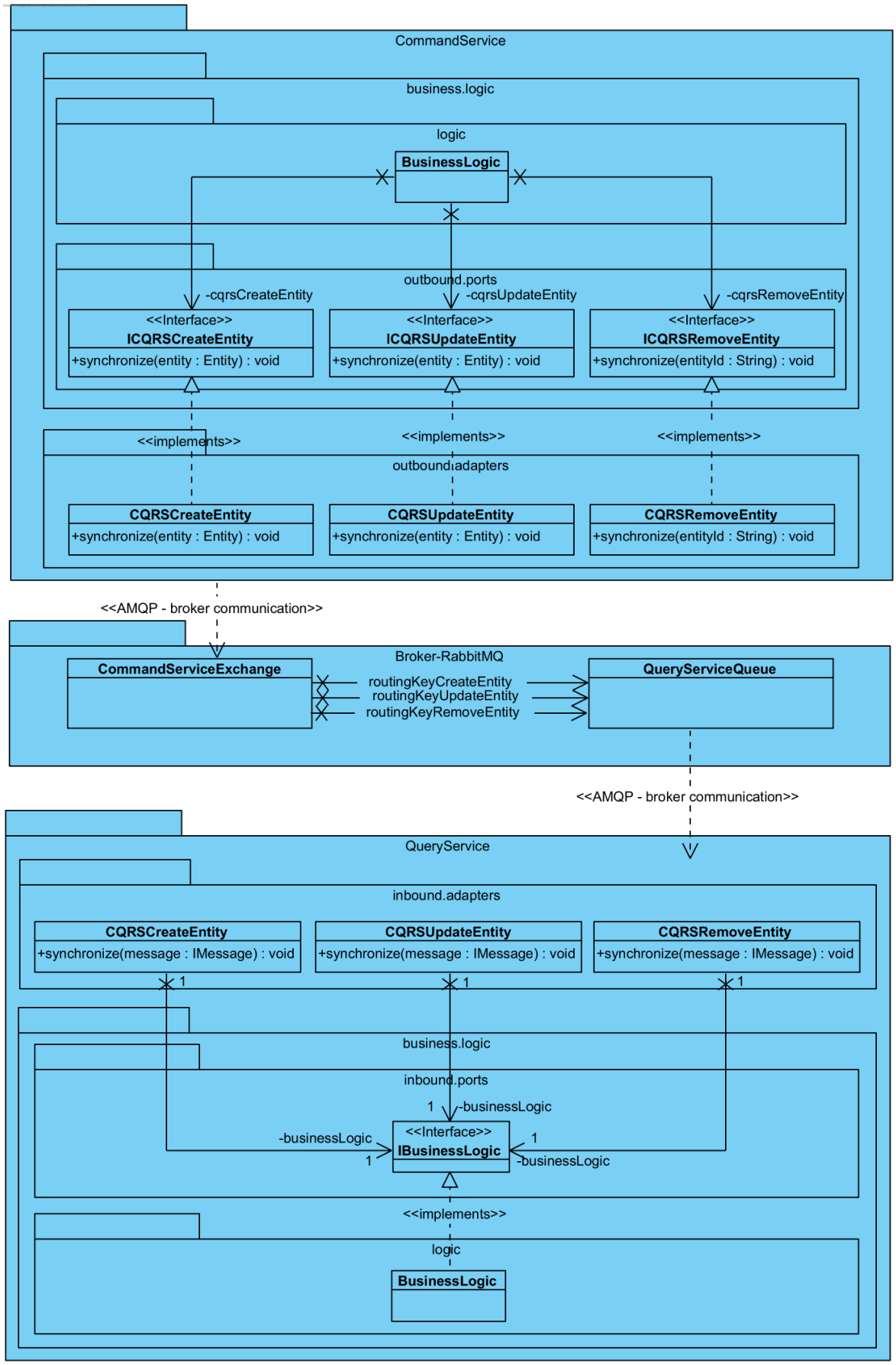


Figura 29: Diagrama de classes da arquitetura tipo, para aplicação do padrão CQRS entre um microsserviço comando e leitura.

O pacote denominado por Broker-RabbitMQ, representa a tecnologia de comunicação assíncrona que será utilizada, pelo que, como foi abordado anteriormente na secção 4.2.1, um microsserviço que publica uma mensagem faz o mesmo para um exchange (CommandServiceExchange) do qual é proprietário. Por outro lado, os microsserviços de leitura, terão de subscrever as suas queues (QueryServiceQueue) com determinadas routingKeys a este exchange para receberem as mensagens com os dados a atualizar. Importante referir que, tem que existir um acordo de interface entre estes microsserviços, para se saber que comando do CQRS corresponde a mensagem, usando para isso a routingKey (p.e. cqrs.0.create.product) e o campo method (p.e. cqrs.0.create.product) da estrutura da mensagem do tipo comando.

No microsserviço de leitura, acontece o processo inverso, ou seja, existem classes concretas para cada tipo de comando possível, no entanto, definidas nos adaptadores de entrada que implementam um método de sincronização. Do mesmo modo, este método de sincronização torna-se responsável pelo mapeamento de dados, versionamento de mensagens e invocação da lógica de negócio necessária, isto é, as operações de comando deste microsserviço de leitura com a finalidade de sincronização da base de dados deste microsserviço com o microsserviço de comando.

4.2.3. Utilização das frameworks Spring Boot e Hibernate

Os dois componentes que também são comuns a todos os microsserviços deste caso de estudo consistem nas frameworks Spring Boot[37] e Hibernate[38].

Efetivamente, a Spring Boot trata-se de uma framework baseada na linguagem Java, que normalmente é utilizada para a criação de microsserviços stand-alone. Na verdade, esta framework fornece toda uma base arquitetural suficiente para desenvolver um microsserviço de forma bastante produtiva com mínimo esforço.

Uma das vantagens que a framework Spring Boot proporciona consiste em permitir a abordagem dependecy injection, muito importante, quando se pretende implementar microsserviços seguindo uma arquitetura hexagonal.

Por outro lado, a Hibernate trata-se de uma framework ORM (Object-relation-mapping), que é utilizada na gestão das entidades da lógica de negócio na base de dados do

[37] Versão 2.5.1 (https://spring.io/projects/spring-boot)
[38] Versão 2.5.1 (https://hibernate.org/)

microsserviço. Deste modo, esta framework caracteriza-se por facilitar a integração da lógica de um microsserviço com a respetiva base de dados, fornecendo um conjunto de ferramentas para a criação do modelo lógico, bem como a gestão do mesmo, garantindo uma melhor eficiência e confiabilidade nestas atividades.

4.3. Conclusão

Em conclusão, no desenho da arquitetura a aplicação dos padrões arquiteturais Inversion of Control e Arquitetura Hexagonal foram importantes, uma vez que, contribuem para o aumento de produtividade.

Na verdade, em relação ao Inversion of Control, mais propriamente o Dependecy Injection o mesmo abstrai do programador a complexidade desnecessária de criação, invocação e integração de serviços internos ou externos.

Em relação à arquitetura hexagonal, a mesma contribui para a produtividade, pois estrutura o código, de forma que encontrar uma determinada secção de código seja simples. Do mesmo modo, faz uma abstração da complexidade dos adaptadores em relação à camada da lógica de negócio o que simplifica a implementação de funcionalidades.

Outro ponto importante relativamente ao desenho da arquitetura consiste na comunicação entre microsserviços. Por um lado, a comunicação entre microsserviços é algo indispensável tal como se verificou no estudo do estado da arte, bem como, dada a natureza distribuída torna-se um desafio. Deste modo, o estudo e desenho da mesma torna-se muito importante. A seleção da comunicação por mensagens dada a sua simplicidade, bem como a definição prévia da estrutura das mensagens foram importantes para o sucesso da comunicação entre microsserviços, que se tornou num processo mais simplificado com a implementação de uma framework própria para manipular as mensagens de igual forma em todos os microsserviços.

5. Arquitetura dos Microsserviços e Padrões do Sistema E-commerce

Neste capítulo, apresentar-se-á a decomposição do caso de estudo e-commerce em um conjunto de microsserviços. Deste modo, definir-se-á cada microsserviço quanto às suas responsabilidades bem definidas, característica importante de uma arquitetura orientada a microsserviços, bem como a apresentação da sua arquitetura.

Importante referir que, nesta decomposição foram utilizados os padrões abordados na secção 2.2.1, denominados respetivamente por Decompose by Business Capability e Decompose by Subdomain.

Ainda, nesta secção serão apresentados os padrões das arquiteturas orientadas a microsserviços aplicados neste sistema e-commerce como caso de estudo desta dissertação, pelo que, a sua abordagem ao longo da apresentação da arquitetura torna-se o método mais eficiente para apresentar os mesmos.

É importante realçar que, todos os diagramas que são apresentados ao longo desta secção não são a totalidade da arquitetura do microsserviço. Na verdade, como foi apresentado anteriormente, existem componentes e uma arquitetura hexagonal base comum a todos os microsserviços com adaptadores de entrada e saída que são omitidos caso não sejam importantes, uma vez que, se tornam repetitivos.

Ainda se torna importante referir que, a seleção de algumas tecnologias, como por exemplo base de dados nos microsserviços não significa que são as escolhas mais adequadas, uma vez que, não foi feito um estudo detalhado sobre este tema, dado que não é o foco da dissertação. Do mesmo modo, a utilização de diferentes tecnologias entre microsserviços tem como objetivo demonstrar a característica que os mesmos têm de serem independentes a esse nível.

A lista dos doze microsserviços decompostos nesta arquitetura que se apresentam ao longo deste capítulo são os seguintes:

1. Subdomínio dos Utilizadores:
    a. Consumer
    b. Manager
2. Subdomínio das Sagas:

    a. Saga
3. Subdomínio dos Produtos:
    a. Command Product
    b. Query Product
4. Subdomínio das Categorias:
    a. Command Category
    b. Query Category Visible Products
    c. Query Category All Products
    d. Query Category Tree
5. Subdomínio das Encomendas:
    a. Order
    b. Inventory
    c. Shopping Cart

5.1. Subdomínio dos Utilizadores

Nesta secção, apresenta-se os microsserviços relacionados com subdomínio dos utilizadores do sistema e-commerce, neste caso consumidores (consumer) e gestores (manager).

5.1.1. Microsserviços Consumer e Manager

Os microsserviços Consumer e Manager são muito semelhantes a nível arquitetural, pelo que, serão abordados simultaneamente.

Primeiramente, torna-se importante justificar a existência destes dois microsserviços. Dado que os utilizadores representam um subdomínio, justifica-se a criação de um microsserviço para a manutenção dos dados dos mesmos e implementação de mecanismos de autenticação. Por outro lado, decidiu-se segregar em dois microsserviços diferentes os dois tipos de utilizador, uma vez que, têm responsabilidades e quantidade de pedidos muito diferentes. Deste modo, justifica-se a sua segregação para se garantir uma maior independência em relação a desenvolvimento, deployment e escalabilidade.

A Figura 30, consiste no diagrama de classes da arquitetura do microsserviço Consumer mais propriamente da camada da lógica de negócio.

Analisando a estrutura de dados, a classe Consumer representa as informações do consumidor desde atributos como o id, que serve de identificador, a password e token, utilizados para fins de autenticação, entre outras informações relativas ao negócio de e-commerce como os contactos, moradas de envio, faturação e identificação fiscal.

A nível de funcionalidades destacam-se dois tipos de autenticação, por login utilizando o id e password do consumidor ou por token, que correspondem ao requisito número 2. Por outro lado, existem funcionalidades para efetuar operações sobre os dados dos consumidores segregadas por operações do tipo comando e leitura, que correspondem ao requisito número 8.

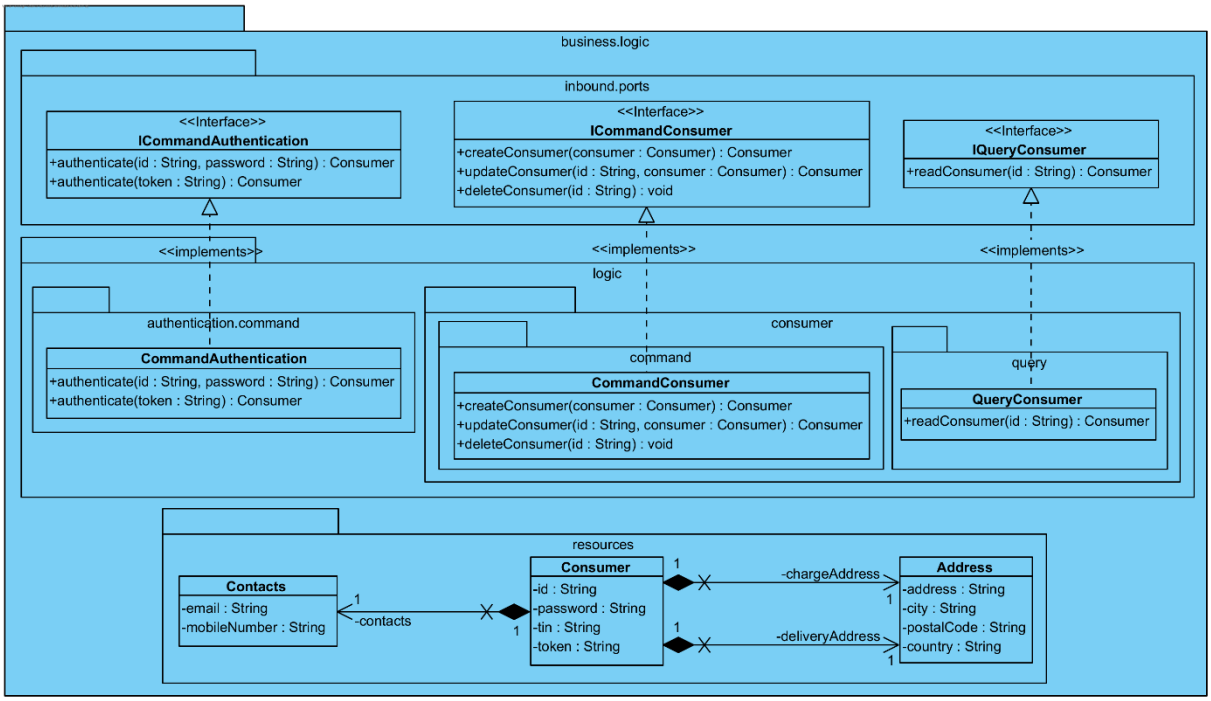


Figura 30: Diagrama de classes do microsserviço Consumer, apenas da camada da lógica de negócio.

A Figura 31, consiste no diagrama de classes da arquitetura do microsserviço Manager mais propriamente da camada da lógica de negócio.

Analisando a estrutura de dados, a classe Manager representa as informações do gestor desde atributos como o id, que serve de identificador, a password e token utilizados para fins de autenticação, entre outras informações relativas ao negócio de e-commerce.

A nível de funcionalidades destacam-se dois tipos de autenticação, por login utilizando o id e password do consumidor ou por token, que correspondem ao requisito número 15 . Por

outro lado, existem funcionalidades para efetuar operações sobre os dados dos gestores segregadas por operações do tipo comando e leitura, que correspondem ao requisito número 16.

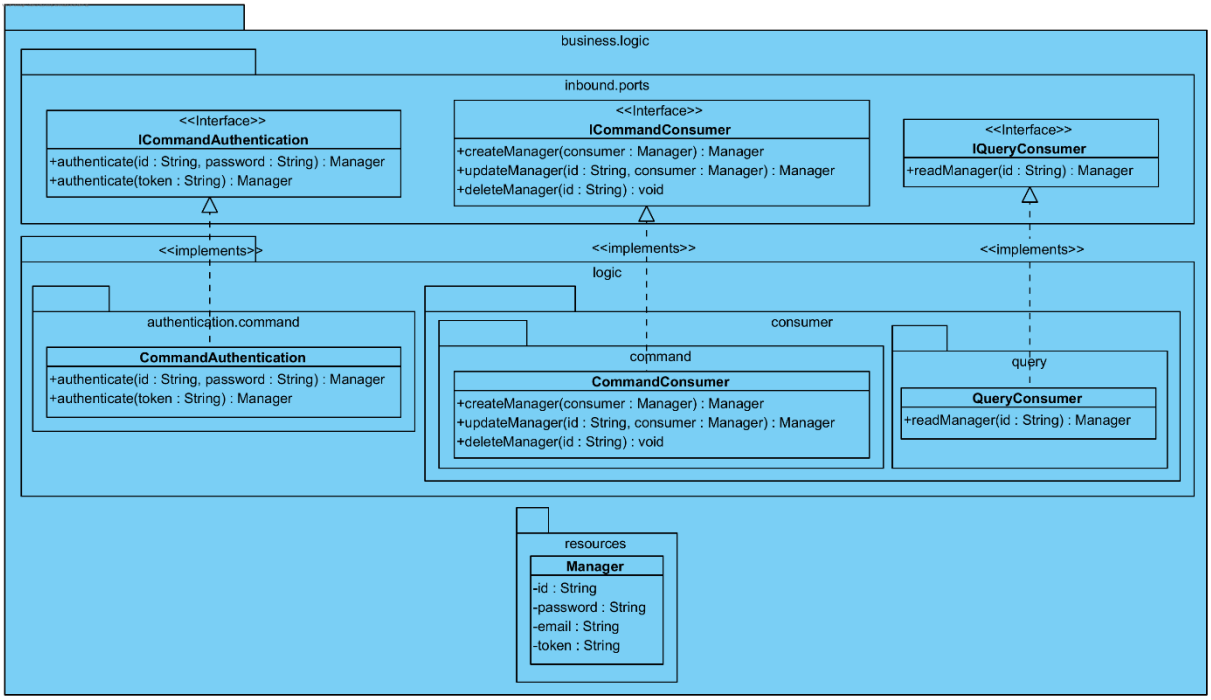


Figura 31: Diagrama de classes do microsserviço Manager, apenas da camada da lógica de negócio.

5.1.2. Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
MongoDB	Não existem grandes motivações para a seleção desta tecnologia.

Tabela 4 - Tecnologias utilizadas nos microsserviços Consumer e Manager.

5.1.3. Padrão Access Token

O padrão access token, abordado na secção 2.2.6, foi aplicado a esta arquitetura, uma vez que, necessita-se de identificar os diferentes utilizadores ao longo de microsserviços, bem como garantir a segurança.

Efetivamente, torna-se necessário garantir a autenticação por login, uma vez que, este sistema será integrado numa aplicação cliente, onde a mesma vai necessitar de efetuar o login de utilizadores. Deste modo, a funcionalidade da classe concreta CommandAuthentication, denominada por authenticate, que recebe como argumentos o id e password do utilizador será responsável por permitir a autenticação por login, onde existe uma validação das

credenciais recebidas e gera-se um token para o utilizador utilizar nos pedidos subsequentes. Nesses pedidos subsequentes a autenticação passa a ser efetuada pelo componente API Gateway com o token devolvido no login. No entanto, por vezes numa aplicação que mantém sessões iniciadas durante um período, pode ser necessário a validação de um token. Por este motivo, existe outra funcionalidade que valida se o token recebido está valido e pode ser autenticado.

Por fim, em relação a este padrão as tecnologias utilizadas a nível de microsserviço foram a framework JWT juntamente com a framework Spring Security[39] que permitem mecanismos de codificação e descodificação de tokens. Ainda em relação a tecnologias foi selecionado o API Gateway da tecnologia Kong, como será abordado mais adiante na secção 5.7.2, que contém plugins de segurança para efetuar a autenticação de utilizadores com base no token do tipo JWT, retirando esta responsabilidade do programador.

5.2. Subdomínio das Sagas

Nesta secção, apresenta-se o microsserviço orquestrador das sagas e a arquitetura respetiva dos microsserviços participantes das mesmas, utilizados para a implementação do padrão Saga abordado na secção 2.2.2.

5.2.1. Microsserviço Saga

O microsserviço Saga tem como responsabilidade orquestrar as transações que abrangem múltiplos microsserviços também denominados por participantes. Na verdade, este microsserviço assume o papel de orquestrador do processo saga, isto é, invoca os participantes sequencialmente, de forma que, essas transações ao longo de múltiplos microsserviços garantam as propriedades de atomicidade, consistência, isolamento e durabilidade nesta arquitetura.

Efetivamente, existem aqui um conjunto de decisões, desde logo, a decomposição do orquestrador num microsserviço específico ao contrário da utilização de um já existente. Na verdade, torna-se evidente as motivações desta segregação, desde separação de responsabilidades por forma a reduzir a complexidade, isto é, não sobrecarregar outros microsserviços com o papel de orquestração. Ainda, permite tornar o orquestrador num

[39] https://docs.spring.io/spring-security/reference/index.html

microsserviço independente, quer a nível de desenvolvimento e deployment, mas principalmente a nível de escalabilidade.

A nível de escalabilidade, tal como abordado na secção 2.1, o orquestrador está desenhado para permitir a escalabilidade no eixo dos YY dado que foi isolado num microsserviço, no eixo dos XX dado que este microsserviço pode ser replicado e por fim, no eixo dos ZZ dado que este microsserviço pode ser particionado em relação aos seus dados e funcionalidades. Dos tipos de escalabilidade apresentados, o primeiro e segundo são garantidos em praticamente todos os microsserviços deste caso de estudo. No entanto, em relação ao particionamento de dados e funcionalidades, apenas alguns como o microsserviço Saga foram preparados para o mesmo. Na verdade, as decisões que o tornaram possível serão abordadas mais adiante, durante a análise do diagrama de classes que representa a arquitetura do microsserviço e respetivas decisões.

A Figura 32, consiste no diagrama de classes da arquitetura do microsserviço Saga mais propriamente da camada da lógica de negócio.

A classe concreta SagaDefinition tem como objetivo guardar de forma dinâmica a configuração de um processo saga. Deste modo, dado que podem existir várias sagas e ao longo de várias instâncias deste microsserviço, para garantir a escalabilidade a mesma tem de ser identificada por um atributo id. Do mesmo modo, como o processo saga necessita de comunicação assíncrona e para garantir a escalabilidade dos eixos ZZ, necessita-se de definir em cada configuração do processo saga o exchange (componente do RabbitMQ) da instância específica onde se encontra, para o mesmo saber para onde deve enviar as mensagens. Ainda em relação à comunicação assíncrona, existem os atributos denominados por commitRoutingKey e commitMethodName, que respetivamente definem a rota e o método da mensagem do tipo comando a enviar para efetuar o commit de uma saga nos microsserviços participantes.

Ainda nesta classe, dado que se pretende tornar a configuração o mais dinâmica possível, define-se um atributo de um tipo genérico, denominado por inputSchema, que consiste na estrutura para validação do input de dados a receber para execução de uma saga.

Do mesmo modo, para tornar o processo dinâmico define-se o método HTTP que se espera receber, para se respeitar os padrões arquiteturais REST, bem como, para fins de autenticação e autorização, a lista de papeis dos utilizadores que podem executar esta funcionalidade, isto é, este processo saga.

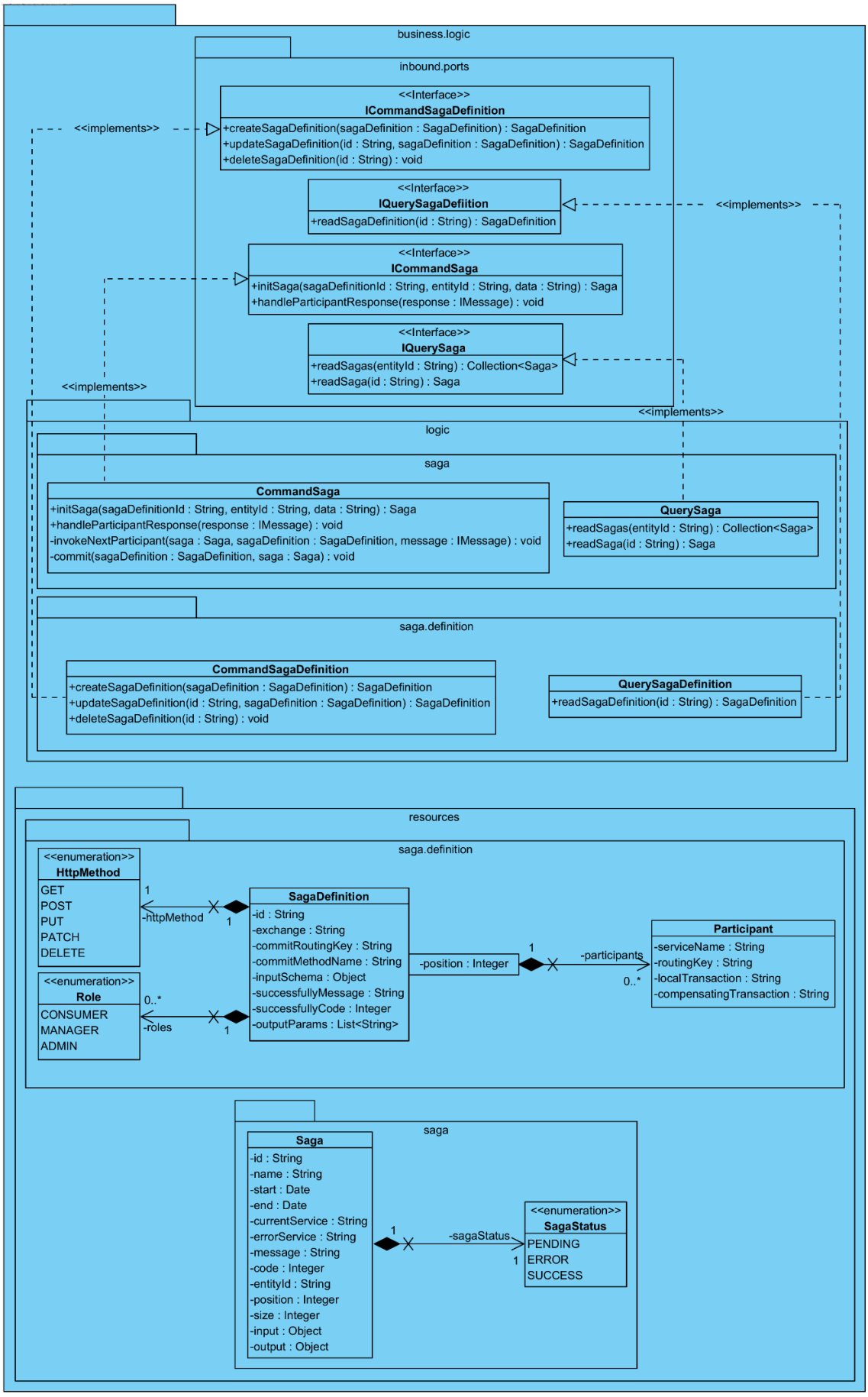


Figura 32: Diagrama de classes da camada da lógica de negócio do microsserviço Saga.

Adicionalmente, existem dois campos denominados por successfullyMessage e successfullyCode, que tem como objetivo especificar respetivamente a mensagem e código que devem ser associados ao registo da saga em caso de sucesso. Do mesmo modo, o atributo outputParams especifica os atributos que estão na mensagem, trocada entre orquestrador e os participantes, que serão associadas ao registo da saga para no final do processo chegar ao resultado. Por exemplo, se a saga for a "criação de uma encomenda" um possível atributo no outputParam será o id dessa encomenda.

Por fim, em relação à configuração das sagas existe uma classe concreta denominada Participant que representa os dados necessários para comunicar com os participantes. Os dados consistem na designação do microsserviço, rota para o envio das mensagens e por fim os nomes dos métodos correspondentes à transação local (localTransaction) e transação de compensação (compensatingTransaction). Ao contrário do commitMethodName, que está definido na classe concreta SagaDefinition, a transação local e de compensação (localTransaction e compensatingTransaction) referem-se ao microsserviço e por este motivo faz sentido estarem definidos em cada participante. Ainda em relação a estas classes, existe uma relação de chave valor entre as classes concretas SagaDefinition e Participant, que representa o conjunto de microsserviços participantes da saga, isto é, a configuração da sequência pela qual serão executados através da chave position.

Por outro lado, existe uma classe concreta Saga, que tem como finalidade guardar os dados relativos a cada registo do processo saga que ocorra, dado que se trata de um processo assíncrono, torna-se necessário guardar um conjunto de dados, para que o orquestrador coordene e continue o processo.

Efetivamente, existe um atributo id para identificar cada processo saga, quer no contexto do orquestrador, quer entre as comunicações com os participantes. Na verdade, este id permite ao orquestrador informar o processo saga aos participantes, quando estes recebem as mensagens do tipo comando para executarem as transações locais ou compensação ou commit. Por outro lado, os participantes com os ids das sagas recebidos, conseguem responder aos orquestradores com os mesmos, permitindo ao orquestrador identificar, quando recebe uma mensagem do tipo resposta a que processo saga se refere. Deste modo, o id da saga consiste num meio de interligação bastante importante entre os microsserviços orquestradores e participantes.

Ainda na classe concreta da Saga, existe um atributo denominado por name para designar o tipo de saga a que se refere esse registo, sendo o mesmo valor que está no id da SagaDefinition. Existem um conjunto de atributos que tem como objetivo a monitorização do processo saga, tais como start e end, que registam as datas de início e fim do processo saga, currentService e errorService que identificam o participante atual em que se encontra o processo saga e em caso de erro o participante onde ocorreu. O sagaStatus, message e code que em caso de sucesso ou erro permitem a avaliação do resultado do processo saga e o entityId, que identifica a entidade (utilizador ou aplicação) que solicitou a execução deste processo.

Por fim, existem três atributos que são utilizados no contexto da coordenação do processo saga, que consiste no position que permite ao orquestrador saber a posição atual no conjunto dos participantes definidos na SagaDefinition. O input, que guarda os dados recebidos para efetuar o processo saga, onde o objetivo consiste em caso de erro facilitar à entidade executadora corrigir o input e voltar a tentar de novo. O output corresponde aos campos definidos no outputParams da SagaDefinition, onde serão colocados dados para se conseguir chegar ao resultado do processo saga (p.e. identificadores).

Relativamente a funcionalidades, este microsserviço divide-se em dois subdomínios, a gestão das configurações das sagas e as sagas.

A gestão das configurações das sagas divide-se em operações de comando e leitura passando pela criação, leitura, atualização e remoção das configurações o que torna os processos sagas mais dinâmicos. Na verdade, isto significa que, podem ser adicionados, alterados e removidos os microsserviços participantes, a ordem pela qual são executados, os nomes dos métodos respetivos às transações locais, compensação e commit, entre outros dados. Em consequência da dinâmica das estruturas de dados e das funcionalidades torna-se possível distribuir diferentes tipos de sagas por diferentes instâncias do microsserviço Saga, garantindo-se como já foi abordado anteriormente a escalabilidade nos eixos dos ZZ, isto é, particionamento de dados e funcionalidades.

Por outro lado, em relação às funcionalidades sobre os processos saga dividem-se entre operações de comando e leitura. Relativamente às operações de comando apenas existem duas funcionalidades expostas para os adaptadores de entrada (definidas no inbound.ports) denominadas por initSaga e handleParticipantResponse.

Em relação à funcionalidade initSaga, como o nome sugere inicia um determinado processo saga, onde recebe o sagaDefinitionId da configuração da saga a executar, o entityId, que corresponde ao identificador da entidade que está a executar e por fim o campo data, ou seja, os dados para executar a mesma. O objetivo desta funcionalidade consiste na criação do registo de execução desta saga, que será utilizado para coordenação da mesma e invocação do primeiro microsserviço participante definido na configuração da saga (SagaDefinition).

A funcionalidade handleParticipantResponse, como o nome sugere, tem o objetivo de analisar as mensagens do tipo resposta enviadas pelos microsserviços participantes das sagas ao orquestrador e efetuar decisões sobre o resultado das mesmas.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
MongoDB	Primeiramente, esta tecnologia permite maiores quantidades de dados, o que será necessário e uma realidade neste microsserviço. Do mesmo modo, esta tecnologia permite agregar todos os dados num único registo de documento, isto é, quer os dados das configurações, quer os registos das sagas, evitando-se operações de junção de dados. Ainda, esta tecnologia torna-se mais adequada para a característica dinâmica das configurações e respetivas sagas.

Tabela 5 - Tecnologias utilizadas no microsserviço Saga.

5.2.2. Arquitetura dos Microsserviços Participantes

Nesta secção, apresenta-se a arquitetura tipo, a nível da camada da lógica de negócio de cada microsserviço participante das sagas, para se evitar repetições das explicações nos diversos microsserviços participantes existentes.

A Figura 33, consiste no diagrama de classes da arquitetura tipo dos microsserviços participantes das sagas mais propriamente da camada da lógica de negócio.

Começando pelas estruturas de dados, existe uma classe concreta Saga que da mesma forma que no orquestrador, tem o objetivo de registar um processo saga, no entanto, no contexto do participante. Deste modo, esta classe contém um atributo sagaId que corresponde ao mesmo id da classe concreta Saga do orquestrador e permite assim identificar a saga entre este e os participantes.

A relação de composição com a classe concreta EntityBackup representa um conjunto de entidades (neste diagrama corresponde à classe Entity) que sofrem alterações locais com um processo saga e por este motivo são guardados registos de backup das mesmas. Estes registos são cruciais para garantir as transações de compensação que ocorrem quando existem erros nos participantes subsequentes, onde se volta a colocar os estados anteriores das entidades antes da saga (equivalente a rollbacks de base de dados).

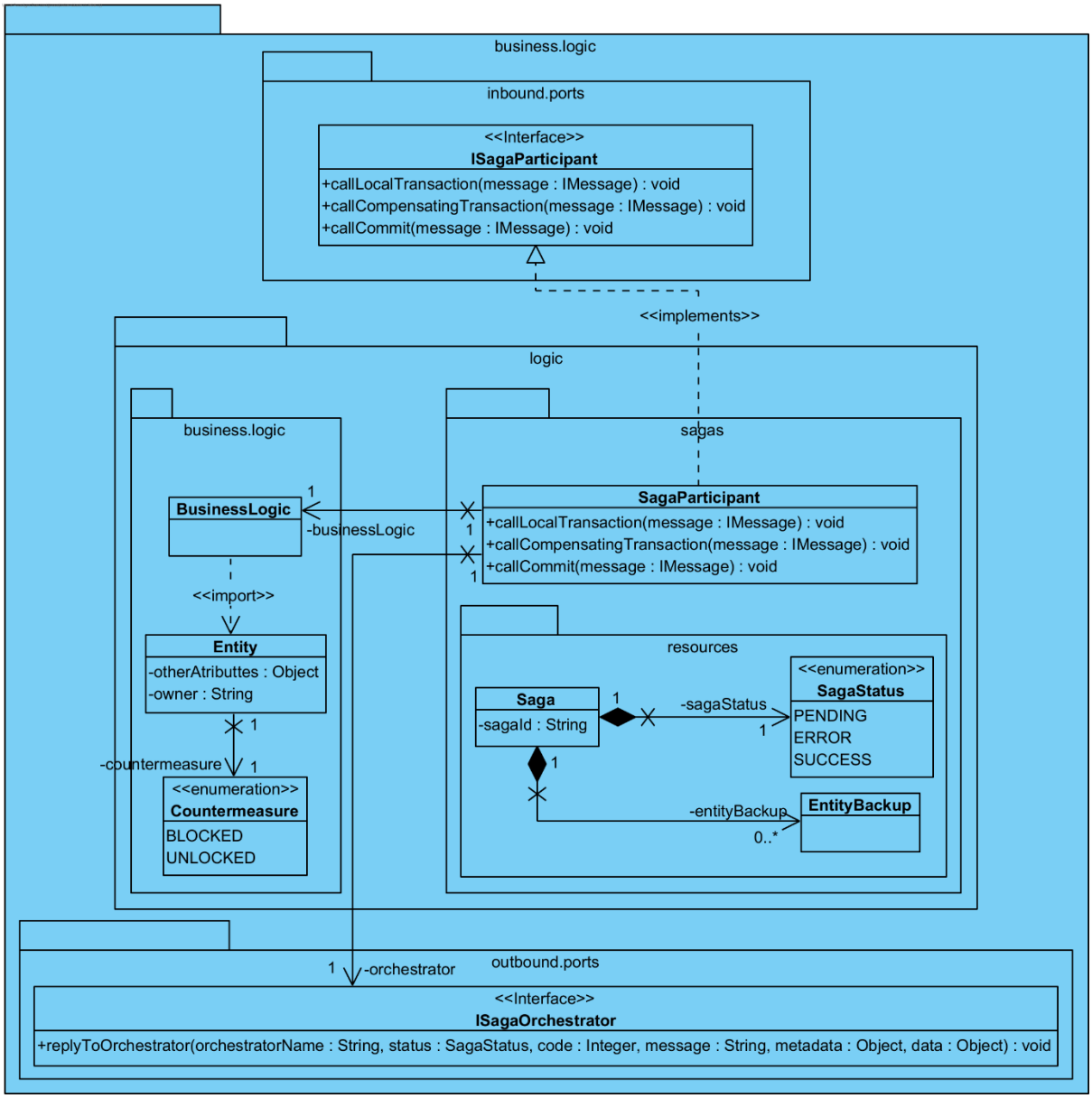


Figura 33: Diagrama de classes da arquitetura da camada de negócio dos microsserviço participantes das sagas.

Ainda em relação às estruturas de dados, torna-se importante referir que, em todos os microsserviços participantes existe uma classe do tipo enumeração, denominada por Countermeasure, que representa as regras de semântica abordadas na secção 2.2.2, sobre o padrão Saga, para garantir a propriedade de isolamento das transações ao longo de múltiplos microsserviços. Na verdade, consiste em ter um atributo denominado por countermeasure que indica se a entidade está bloqueada por algum processo saga e um atributo owner referente ao identificador da saga proprietária do countermeasure.

A nível de funcionalidades, com o objetivo de tornar esta arquitetura consistente ao longo dos microsserviços participantes, definiu-se uma interface comum a todos estes, denominada ISagaParticipant, nas portas de entrada com três métodos que representam as três ações possíveis numa saga: transação local, transação de compensação e commit. Desta forma, o objetivo destas funcionalidades consiste em tratar da lógica da saga a nível do participante e de seguida invocar os métodos da lógica de negócio necessários, existindo uma relação de composição com a lógica de negócio. A lógica da saga que estas funcionalidades tratam consiste na criação de registos saga, análise dos dados recebidos, verificações de erros e em caso disso comunicar os mesmos ao orquestrador.

A comunicação ao orquestrador representa-se pela relação de composição que existe entre a classe concreta destas funcionalidades, denominada SagaParticipant, com a interface ISagaOrchestrator que torna a forma e o meio de comunicação abstratos.

Em suma, esta arquitetura base dos microsserviços participantes das sagas permite uma maior produtividade e facilidade de manutenção, uma vez que, se torna mais fácil localizar as funcionalidades a alterar.

5.2.3. Padrão Self-contained Service

O padrão Self-contained Service foi aplicado ao microsserviço orquestrador Saga para tornar o mesmo mais independente e evitar esperas ativas do utilizador, uma vez que, os processos saga **dependem** da disponibilidade de outros microsserviços. A dependência referida anteriormente, significa que, para executar por completo um processo saga necessita-se que os microsserviços participantes estejam disponíveis. No entanto, isso nem sempre é possível, pelo que, este padrão resolve esse problema.

Efetivamente, para se implementar este padrão, primeiramente as comunicações deste microsserviço Saga com os microsserviços participantes passam a ser assíncronas,

resolvendo-se o problema da disponibilidade dos mesmos (exemplo prático no anexo III). Por outro lado, a resposta ao pedido do utilizador, que executa a saga, faz-se o mais breve possível com um código HTTP 202[40], que significa que, a iniciação do processo foi aceite, mas ainda não foi concluído. O utilizador poderá consultar o estado do processo até que o mesmo fique concluído.

5.3. Subdomínio dos Produtos

Nesta secção, apresenta-se os microsserviços que tem responsabilidades sobre a entidade produto. No entanto, e como será explicado nesta secção, os mesmos foram segregados em microsserviços de comando e leitura.

5.3.1. Microsserviço Command Product (CP)

O microsserviço Command Product tem como responsabilidade as operações de comando sobre a entidade produto.

A Figura 34, consiste no diagrama de classes da arquitetura do microsserviço Command Product mais propriamente da camada da lógica de negócio.

Em relação às estruturas de dados, define-se uma classe concreta Product com um conjunto de atributos de dados do produto. Deste modo, o atributo id identifica o produto e poderá ser auto gerado ou ser introduzido pelo gestor, sendo que, nesta última, se permite colocar um atributo do negócio, como o SKU, EAN, PN, entre outros, onde o significado destes foi apresentado na secção 3.3.

O produto contém um conjunto de atributos que definem os detalhes do mesmo, desde o name que consiste na designação do produto, o shortDetails e longDetails que especificam as características do produto de forma mais sucinta e detalhada respetivamente. Em relação aos dois últimos atributos, torna-se importante referir que, os mesmos são do tipo String, no entanto, o objetivo dos mesmos será receber estruturas json dinâmicas, permitindo assim uma maior liberdade de configuração e estruturação das especificações dos produtos. Do mesmo modo, o produto contém um atributo denominado por links que especifica os caminhos (p.e. URI) das imagens dos produtos e pelos mesmos motivos que os atributos anteriores, tem o tipo String.

[40] https://tools.ietf.org/html/rfc2616#page-59

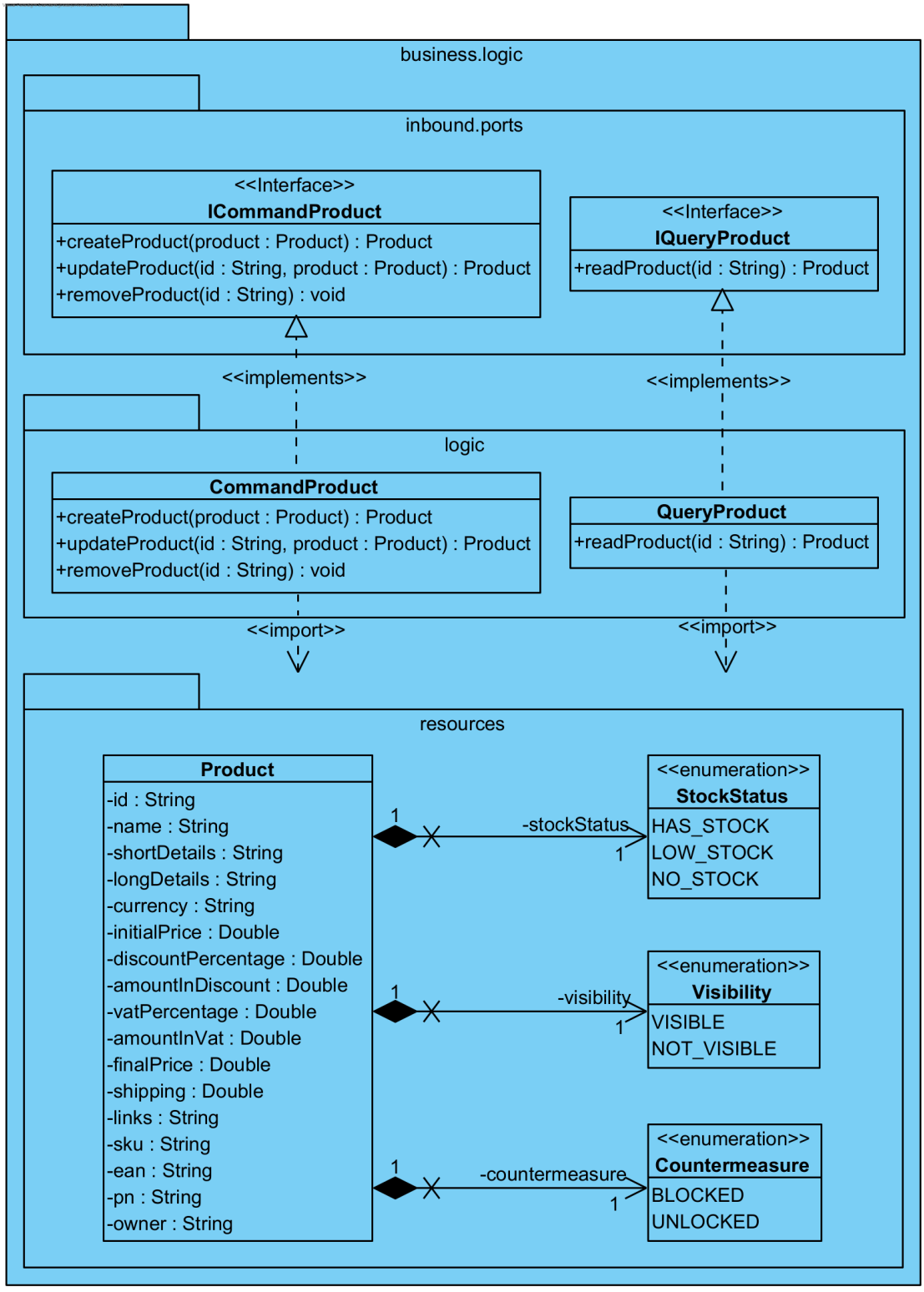


Figura 34: Diagrama de classes da camada da lógica de negócio do microsserviço Command Product (CP).

Existem um conjunto de atributos relacionados com o preço, descontos, distribuição e taxas do produto, tais como: currency, initialPrice, discountPercentage, amountInDiscount, vatPercentage, amountInVat, finalPrice e shipping. Importante referir que, alguns destes

atributos são valores derivados, isto é, o sistema calcula os mesmos através de outros, tais como: amounInDiscount, amountInVat e finalPrice.

O atributo stockStatus categoriza o estado do stock de um produto em três estados diferentes, uma vez que, do ponto de vista dos dados do produto não interessa saber o valor da quantidade em stock. Importante referir que, como será abordado na secção 5.5, este dado provém de outro microsserviço denominado por Inventory. O atributo visibility define a visibilidade de um produto em relação ao consumidor.

Por fim, existem um conjunto de atributos, identificadores no contexto do negócio, onde o seu significado foi anteriormente apresentado na secção 3.3, que são: o SKU, EAN e PN.

A nível de funcionalidades, este microsserviço contém a responsabilidade das operações de comando sobre a entidade produto. Deste modo, tal como se pode observar na Figura 34, o mesmo especifica um conjunto de operações de criação, atualização e remoção de produtos na classe concreta CommandProduct. Por outro lado, como foi abordado na secção 2.2.2, apesar de ser um microsserviço comando não significa que não contenha funcionalidades de leitura, no entanto, estas são simples e utilizam a chave principal, como está definido na classe concreta QueryProduct.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
PostgreSQL	Não existe um motivo concreto para seleção desta tecnologia

Tabela 6 - Tecnologias utilizadas no microsserviço CommandProduct (CP).

5.3.2. Microsserviço Query Product

O microsserviço Query Product tem como responsabilidade as operações de leitura sobre a entidade produto.

A Figura 35, consiste no diagrama de classes da arquitetura do microsserviço Query Product mais propriamente da camada da lógica de negócio.

Em relação às estruturas de dados as mesmas são muito semelhantes ao microsserviço Command Product, abordado anteriormente na secção 5.3, pelo que, torna-se relevante focar apenas nas grandes diferenças.

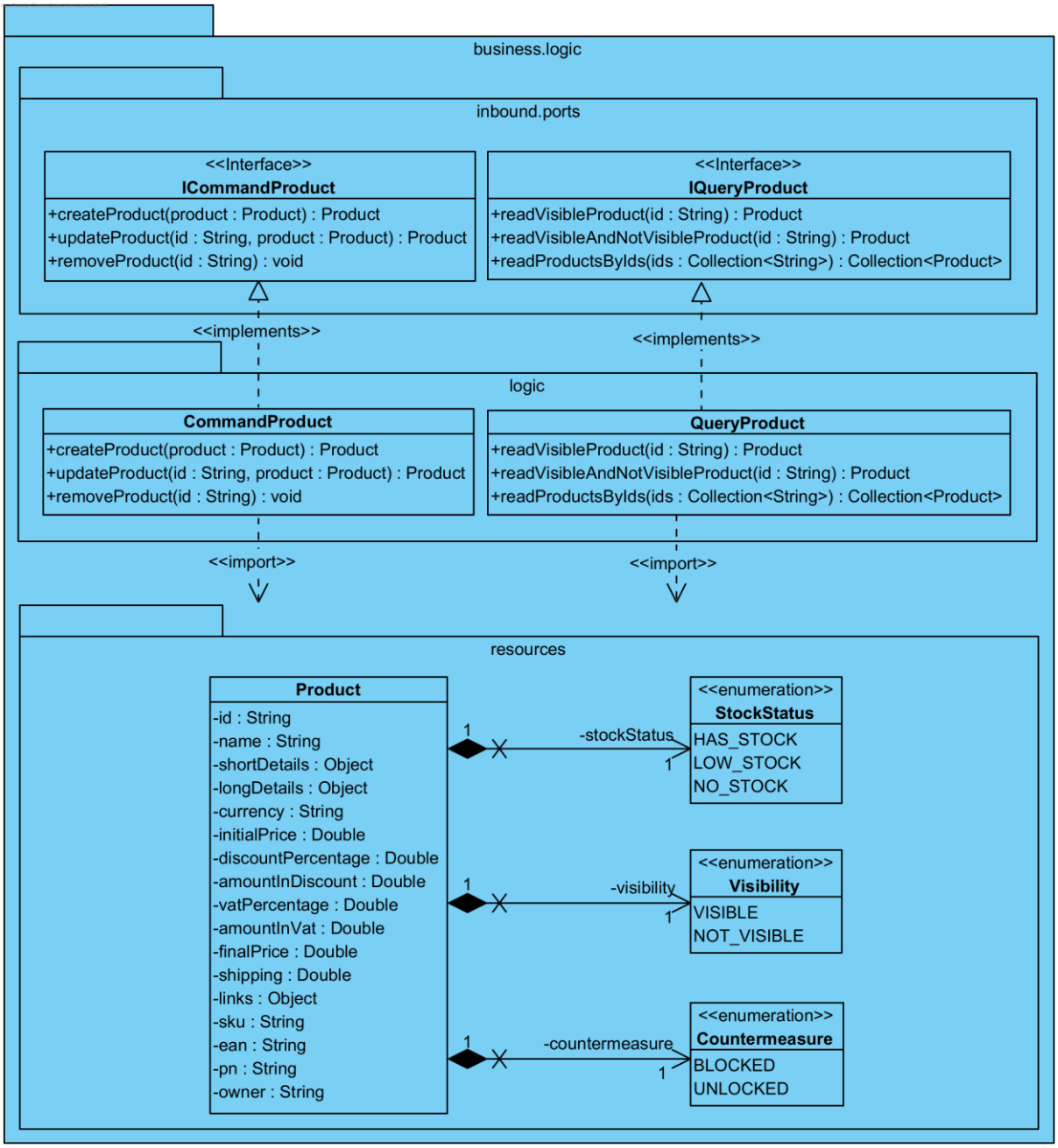


Figura 35: Diagrama de classes da camada da lógica de negócio do microsserviço Query Product (QP).

Efetivamente, as diferenças consistem nos tipos de dados de alguns atributos (p.e. shortDetails, longDetails, links) que no microsserviço Command Product eram do tipo String para guardarem dados em formato JSON. No entanto, por forma a não prejudicar as operações de leitura com mapeamentos de dados aplicou-se o padrão CQRS, com a criação deste microsserviço de leituras, que será explicado com mais detalhe no seguimento desta secção. Desta forma, esses atributos neste microsserviço são do tipo Object, pelo que, dada a

tecnologia de base de dados selecionada, são guardados diretamente sem a necessidade de mapeamentos o que também se verifica para as operações de leitura.

Em relação às funcionalidades, dado que se trata de um microsserviço de leituras, existem um conjunto de operações deste tipo.

A funcionalidade "leitura de produto visível" (readVisibleProduct) trata-se de uma funcionalidade do consumidor, requisito número 4 e 7, devolve os dados totais de um produto, mas apenas se for visível. A funcionalidade "leitura de produto visível e não visível" (readVisibleAndNotVisibleProduct) trata-se de uma funcionalidade do gestor, requisito número 18 (referente à consulta de produtos), que devolve os dados totais de um produto, quer seja visível ou não visível. A funcionalidade de leitura de produtos por ids (readProductsByIds) trata-se de uma funcionalidade do sistema que devolve os dados de um conjunto de produtos dado um conjunto de identificadores.

Apesar de se tratar de um microsserviço de leituras o mesmo necessita de gerir os dados dos produtos, pelo que, contém um conjunto de operações de comando para efetuar essa gestão, desde criar, atualizar e remover produtos.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
MongoDB	Não existe um motivo concreto para seleção desta tecnologia.

Tabela 7 - Tecnologias utilizadas no microsserviço QueryProduct (QP).

5.3.3. Padrão CQRS entre CP e QP

O padrão CQRS, abordado na secção 2.2.2 e 4.2.2, foi aplicado a esta arquitetura para segregar em dois microsserviços as operações de comando e leitura sobre a entidade produto, gerando os respetivos Command Product (CP) e Query Product(QP).

Efetivamente, as motivações para esta segregação devem-se ao facto de se considerarem responsabilidades distintas as operações de comando das de leitura sobre a entidade produto, principalmente em relação às quantidades de pedidos que se esperam em cada conjunto. Na verdade, o objetivo consiste em aumentar a disponibilidade de ambos os conjuntos de operações de forma independente sem prejudicar o conjunto oposto. Neste caso, as operações de leitura são mais críticas a nível de disponibilidade e tempos de resposta.

Outro motivo para esta segregação, como já foi referido ao longo da secção 5.3, consiste em permitir adequar as tecnologias de base de dados mais adequadas às funcionalidades deste subdomínio, que neste caso são as operações de comando e leitura sobre a entidade produto respetivamente.

Ainda, a aplicação deste padrão permite que no futuro caso haja novos ou já existentes microsserviços interessados nos dados do produto possam ser integrados nesta arquitetura CQRS entre o microsserviço CommandProduct e os respetivos microsserviços de leitura.

Por outro lado, ao aplicar este padrão existem algumas desvantagens, como aumento da complexidade e a latência, que pode levar a uma inconsistência temporária enquanto as mensagens não chegam ao destino.

5.3.4. Decisões para resolver o desafio da centralidade da entidade Produto

Tal como foi abordado na secção 3.4, da análise do domínio, o produto comporta-se como uma entidade central, uma vez que, tem relações com maior parte das entidades deste caso de estudo e-commerce. Deste modo, numa arquitetura orientada a microsserviços, naturalmente distribuída, estas relações tornam-se desafiantes.

Efetivamente, para solucionar o problema das relações entre entidades de múltiplos microsserviços, implementou-se alguns padrões como: Saga e CQRS.

O padrão Saga, como foi abordado na secção 2.2.2, permite garantir as propriedades ACID em transações ao longo de múltiplos microsserviços, que também são responsáveis por criarem as relações anteriormente descritas. Para além de criarem as relações com a entidade produto nos microsserviços que não são responsáveis por esta entidade, também são injetados os dados do mesmo que são necessários para a execução das funcionalidades daquele microsserviço. Isto promove uma grande vantagem, uma vez que, torna estas funcionalidades mais disponíveis, dado que, não necessitam de depender do microsserviço responsável pelos dados do produto.

Do mesmo modo, o padrão CQRS também contribui para garantir estas relações, uma vez que, permite propagar dados de um para N microsserviços com o objetivo de evitar pedidos recorrentes desses dados ao microsserviço responsável.

Por outro lado, isto torna-se possível com a entidade produto, uma vez que, a mesma não altera o seu estado com muita frequência, pois caso contrário, deve-se avaliar a situação,

dado que, a aplicação de alguns dos padrões tem como consequência um aumento da complexidade.

Torna-se relevante abordar este tema, uma vez que, a entidade produto surge em muitos microsserviços que tem relações com a mesma. Deste modo, poderiam ocorrer algumas dúvidas, dado que, como foi abordado anteriormente a responsabilidade da entidade produto, respetivamente as operações de comando e leitura, corresponde aos microsserviços Command Product e Query Product.

Na verdade, essa responsabilidade **não** deixa de pertencer a estes microsserviços, mas com a contribuição do padrão Saga e CQRS é possível que estes permitam criar as relações e injetar os dados do produto em outros microsserviços com o objetivo primordial de reduzir as dependências das funcionalidades.

5.4. Subdomínio das Categorias

Nesta secção, apresenta-se os microsserviços que tem responsabilidades sobre a entidade categoria. No entanto, e como será explicado nesta secção, os mesmos foram segregados em microsserviços de comando e leitura.

Uma característica deste subdomínio consiste na existência de três microsserviços de leitura devido à aplicação do padrão CQRS. Os principais objetivos são segregação de responsabilidades e independência de funcionalidades críticas, quer a nível de disponibilidade e tempos de resposta, onde se pretende implementar mecanismos escalabilidade.

5.4.1. Microsserviço Command Category (CC)

O microsserviço Command Category tem como responsabilidade as operações de comando sobre a entidade categoria. Na verdade, esta responsabilidade passa pela gestão das categorias desde a criação, atualização e remoção destas, bem como a relação entre as mesmas e com os produtos.

A Figura 36, consiste no diagrama de classes da arquitetura do microsserviço Command Category mais propriamente da camada da lógica de negócio.

Em relação às estruturas de dados, a classe concreta Category representa a entidade categoria tendo como atributos um id que se comporta como identificador e um name para designação da mesma em relação ao consumidor.

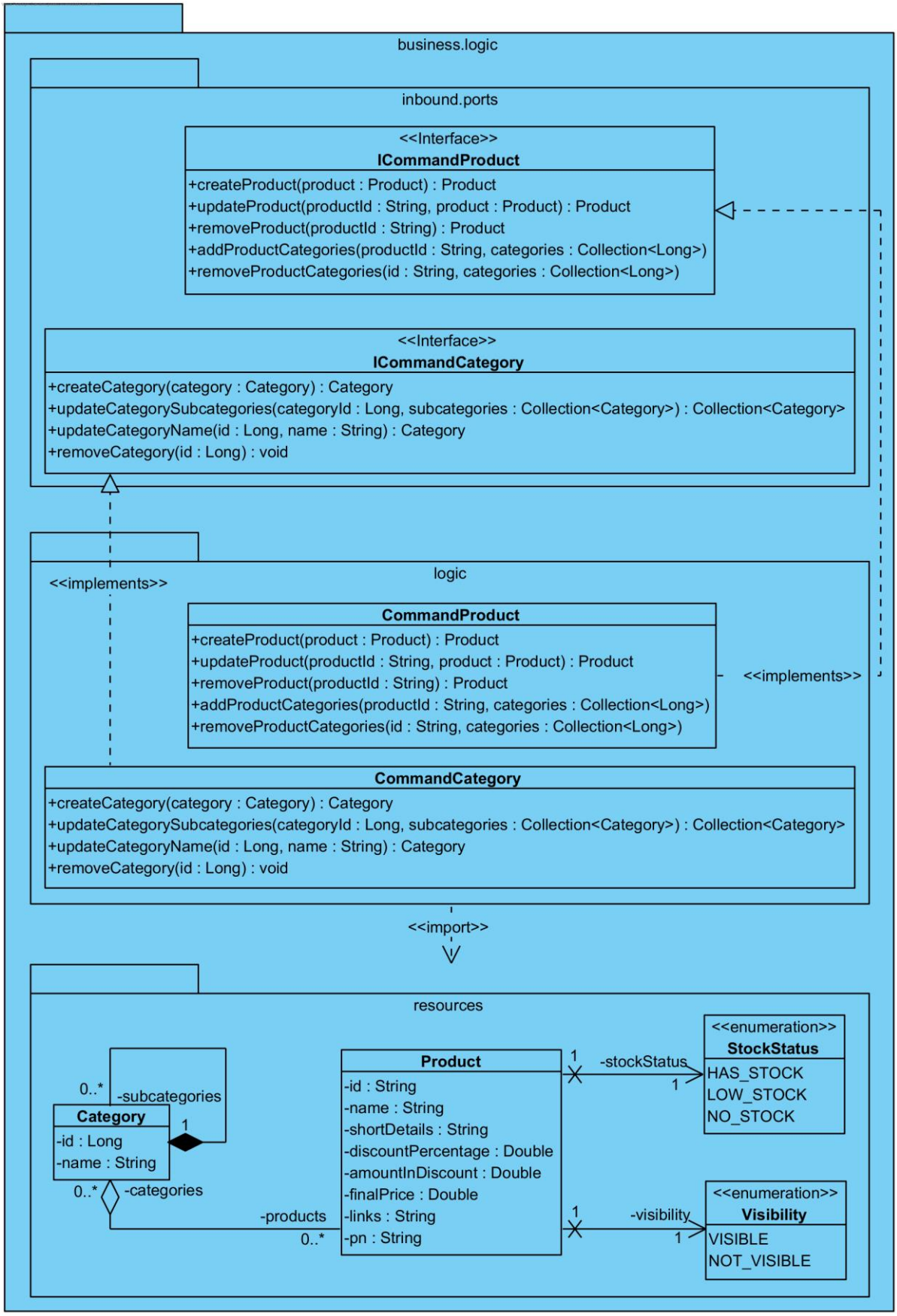


Figura 36: Diagrama de classes da camada da lógica de negócio do microsserviço Command Category (CC).

Em termos de relações, existe uma relação recursiva bidirecional com multiplicidade de um para muitos na categoria, denominada por subcategories. O objetivo desta relação consiste em organizar numa espécie de estrutura em árvore as subcategorias de uma categoria e assim sucessivamente formando caminhos, tal como acontece normalmente nas aplicações de e-commerce. No entanto, torna-se importante referir que, para obter uma categoria, independentemente de estar relacionada com outras, não é necessário percorrer caminhos para chegar à mesma, daí a importância desta estrutura e do atributo id que torna este processo mais eficiente. Todavia, a relação recursiva do tipo composição na categoria tem como objetivo representar que, uma categoria tem de ser sempre subcategoria de outra (hierarquicamente superior) exceto a categoria raiz. Por consequência, sempre que for removida uma categoria também serão removidas as suas subcategorias e assim sucessivamente até terminar os caminhos.

Por outro lado, existe uma classe concreta Product que representa os dados do produto, que se torna necessária para as funcionalidades deste microsserviço lembrando que, as categorias têm como objetivo categorizar os produtos, isto é, organizar os mesmos em conjuntos com características semelhantes. Torna-se importante relembrar que, tal como referido na secção 5.3, estes dados são injetados neste microsserviço provenientes de outros que tem a responsabilidade sobre os mesmos.

A constituição da classe concreta Product a nível de atributos é semelhante à do microsserviço CommandProduct, apresentada na secção 5.3.1, excluindo alguns campos que não são necessários para as funcionalidades do microsserviço CommandCategory, pelo que não se repetirá a descrição dos mesmos.

Existe uma relação de agregação entre a categoria e os produtos, denominada por products, bidirecional com multiplicidade de muitos para muitos, para representar a lista de produtos que estão associados a nenhuma ou várias categorias e vice-versa. Trata-se de uma agregação, uma vez que, o produto tem existência própria, dado que, pode estar relacionado com nenhuma ou várias categorias e vice-versa.

Em relação às funcionalidades, as mesmas dividem-se em operações de comando sobre as entidades categoria e produto.

Começando pelas operações de comando sobre a categoria que correspondem ao requisito número 21 (operações de adicionar, editar e remover), existe uma funcionalidade que cria uma categoria, denominada por createCategory, sendo que, a mesma tem a

particularidade de apenas permitir criar a categoria raiz. Por outro lado, para adicionar novas categorias existe a funcionalidade updateCategorySubcategories, que como o nome sugere, cria uma categoria adicionando-a como subcategoria de outra dado o categoryId dessa categoria hierarquicamente superior. Este processo pode ser efetuado em qualquer categoria começando na categoria raiz. Esta funcionalidade serve para criar a relação recursiva de composição entre as categorias e as subcategorias. Outra funcionalidade consiste na atualização do nome da categoria, denominada por updateCategoryName. Por fim, existe a funcionalidade que remove uma categoria, denominada por removeCategory, que como já foi referido anteriormente dada a existência da relação recursiva de composição, denominada por subcategories, quando se remove uma categoria as respetivas subcategorias e assim sucessivamente são também removidas.

Por outro lado, as operações de comando sobre o produto, consiste na gestão desta entidade, desde criar, atualizar e remover para permitir a injeção e sincronização destes dados neste microsserviço. Para além destas funcionalidades de gestão, destacam-se duas outras funcionalidades que tem como objetivo gerir a relação entre o produto e as categorias que são respetivamente addProductCategories e removeProductCategories. A funcionalidade que adiciona categorias ao produto recebe como argumentos o id do produto, bem como, um conjunto de ids de categorias, as quais são associadas ao produto, isto é, a relação denominada por categories. Por outro lado, a funcionalidade que remove categorias ao produto recebe os mesmos argumentos, no entanto, para dissociar categorias do produto, isto é, a relação denominada por categories.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
PostgreSQL	A existência de muitas relações entre entidades, algumas recursivas e bidirecionais, levou à escolha de uma base de dados relacional. O PostgreSQL, é uma tecnologia conhecida pelo programador.

Tabela 8 - Tecnologias utilizadas no microsserviço CommandCategory (CC).

5.4.2. Microsserviço Query Category Visible Products (QCVP) e o Query Category All Products (QCAP)

Nesta secção, são abordados dois dos três microsserviços de leitura que existem no subdomínio das categorias. Na verdade, isto deve-se ao facto de estes dois microsserviços serem muito semelhantes a nível arquitetural, no entanto, distinguem-se nas responsabilidades.

Por forma a perceber-se a diferença entre estes dois microsserviços necessita-se de relembrar a definição da visibilidade de um produto, que já foi abordada anteriormente em diversas secções. Desta forma, a visibilidade de um produto define se um determinado produto pode ser consultado pelos consumidores. Por outro lado, um gestor e administrador pode consultar sempre todos os produtos.

Efetivamente, o microsserviço Query Category Visible Products, tem a responsabilidade das operações de leitura sobre os produtos de diferentes categorias que estejam definidos como visíveis para o consumidor. Deste modo, este microsserviço na sua base de dados apenas contém produtos definidos como visíveis. Desta forma, a funcionalidade de consulta dos produtos de uma determinada categoria não tem de implementar mecanismos de autorização e filtragem de produtos que o consumidor seja autorizado a consultar, tornando a mesma mais eficiente. Assim, o microsserviço QCVP apenas implementa funcionalidades de leitura que são direcionadas para os consumidores. Na verdade, como será abordado em detalhe mais adiante a aplicação do padrão CQRS neste subdomínio, para além das motivações da redução de lógica desnecessária de autorização e filtragem, existem outras vantagens em segregar estes dois microsserviços.

A Figura 37, consiste no diagrama de classes da arquitetura do microsserviço Query Category Visible Products mais propriamente da camada da lógica de negócio.

A nível de estrutura de dados, as mesmas distinguem-se das definidas no microsserviço CommandCategory e por este motivo justifica-se a necessidade desta segregação de microsserviços de comando e leitura. Na verdade, dado que a funcionalidade principal deste microsserviço, em termos da sua responsabilidade, consiste na "consulta dos produtos visíveis de uma categoria" a estrutura de dados foi adequada à mesma. Deste modo, existe uma classe concreta denominada por Category que representa a entidade categoria.

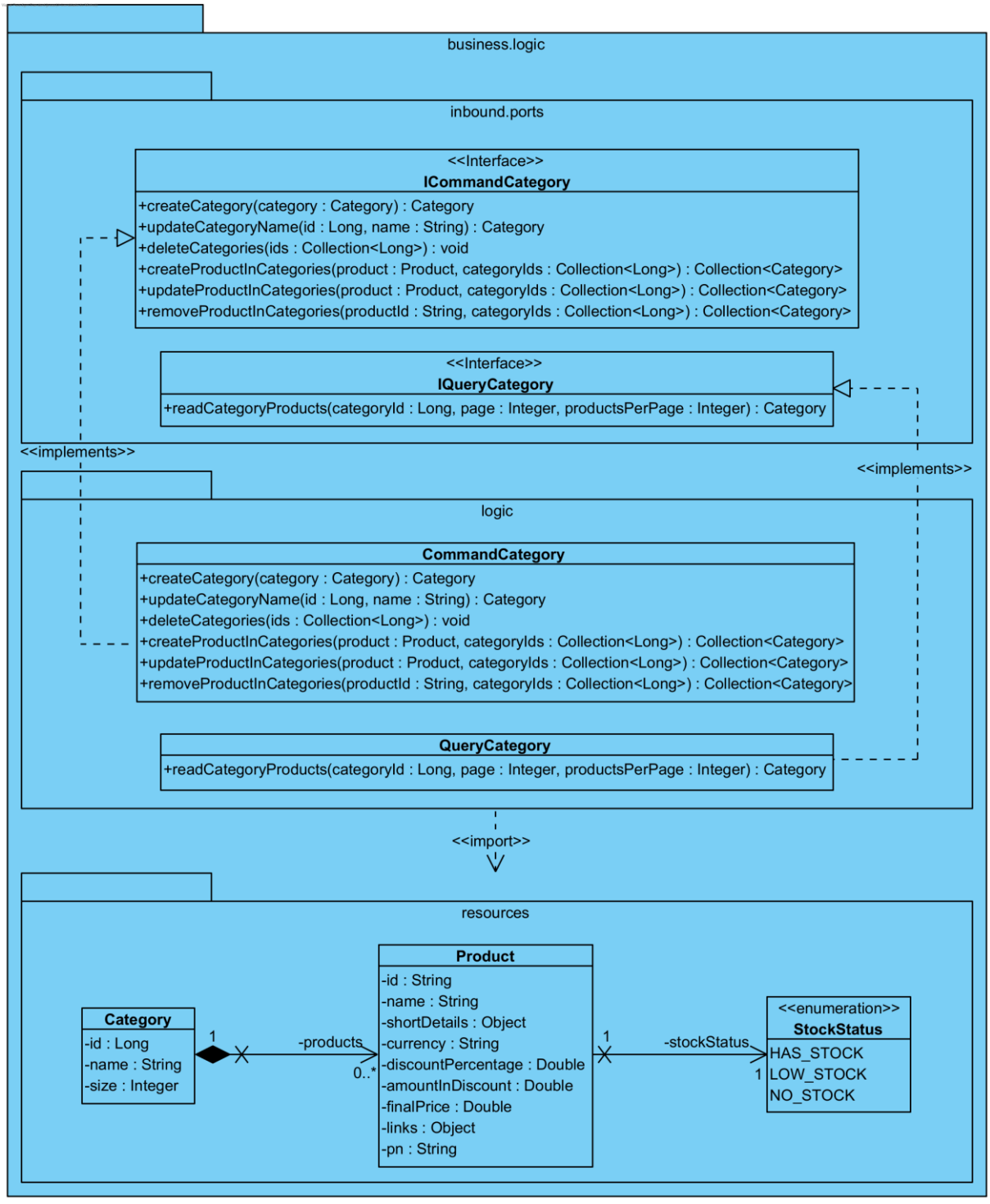


Figura 37: Diagrama de classes da camada da lógica de negócio do microsserviço Query Category Visible Products (QCVP).

Em termos de atributos, por forma a não repetir as mesmas explicações existe um atributo adicional ao que acontece no CommandCategory, denominado por size, que específica a quantidade de produtos associados a esta categoria. Apesar desta informação existir normalmente nas estruturas de dados de coleções, para evitar executar sempre

funcionalidades que obtém esse valor e como se pretende eficiência decidiu-se guardar essa informação na própria categoria e apenas é alterada quando se adiciona ou removem produtos. Esta informação torna-se importante para a aplicação cliente determinar o número de páginas existentes numa categoria para um determinado valor de produtos por página definido pela mesma aplicação. Na verdade, torna esta funcionalidade mais adaptável a diferentes tipos de aplicações clientes, dispositivos e utilizadores.

Por outro lado, existe uma classe concreta denominada por Product que representa a entidade produto, contendo os mesmos atributos que no microsserviço CommandCategory exceto o atributo visibility que como já foi referido acima não faz sentido neste microsserviço, uma vez que, este apenas contém produtos visíveis.

Existe uma relação de composição, denominada por products, entre a classe concreta Category e Product unidirecional de multiplicidade de um para muitos, para representar os produtos de uma determinada categoria. Um ponto importante sobre esta relação consiste na composição da mesma, demonstrando que a categoria tem existência própria neste microsserviço ao contrário do produto que apenas existe para uma determinada categoria. Isto não significa que o mesmo produto não possa existir em várias categorias, no entanto, os mesmos são apenas cópias de dados. Na verdade, os produtos serem apenas cópias de dados não tem qualquer consequência, uma vez que, esta estrutura está desenhada para a responsabilidade deste microsserviço.

Em relação a funcionalidades, existe uma operação de leitura definida na classe concreta QueryCategory que consiste na responsabilidade do microsserviço QCVP, denominada por readCategoryProducts, correspondente aos requisitos número 3 e 6 , que devolve uma subsecção dos produtos visíveis de uma categoria dado o id da mesma, a quantidade de produtos por página e a página da respetiva subsecção que se pretende.

Por outro lado, existe um conjunto de operações de comando sobre a entidade categoria e os produtos destas que permitem a gestão dos dados deste microsserviço que são injetados no mesmo pelo microsserviço CommandCategory, com aplicação do padrão CQRS, que será abordado mais adiante. Das operações comando, definem-se a criação, atualização do nome e remoção de categorias, bem como, a adição, atualização e remoção de produtos em categorias.

Por outro lado, o microsserviço Query Category All Products ao contrário do QCVP tem a responsabilidade das operações de leitura sobre todos os produtos de diferentes categorias,

que estejam definidos como visíveis e não visíveis para o consumidor. Assim, este microsserviço apenas implementa funcionalidades de leitura direcionadas para os gestores e administradores autorizados para consultar qualquer tipo de produto.

A Figura 38, consiste no diagrama de classes da arquitetura do microsserviço Query Category All Products mais propriamente da camada da lógica de negócio.

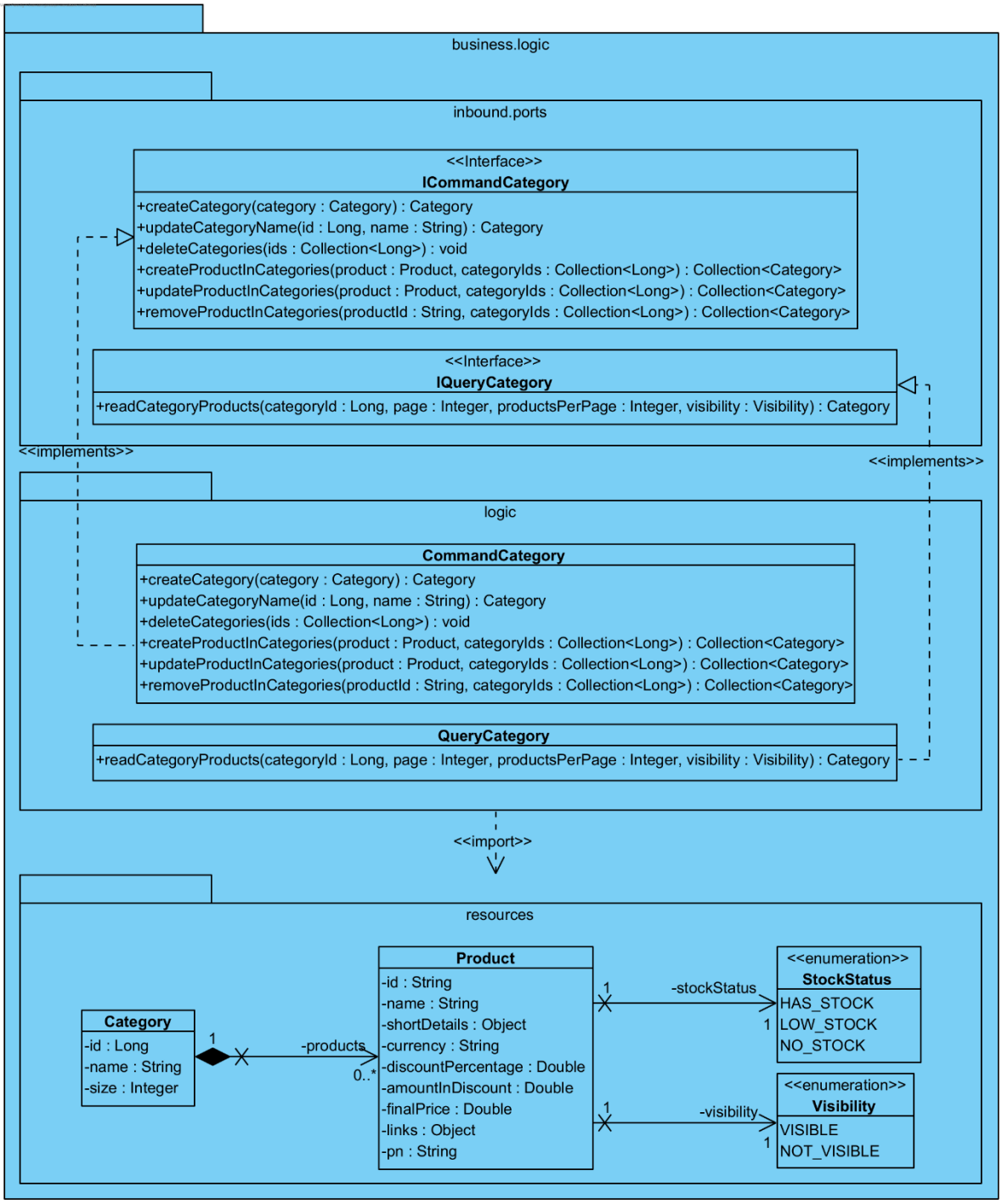


Figura 38: Diagrama de classes da camada da lógica de negócio do microsserviço Query Category All Products (QCAP).

A nível de estruturas de dados a única diferença em relação ao microsserviço QCVP consiste na existência do atributo visibility, uma vez que, faz todo sentido neste microsserviço dadas as suas responsabilidades.

A nível de funcionalidades existe uma operação de leitura na classe concreta QueryCategory, denominada por readCategoryProducts, que corresponde ao requisito número 15 que consulta os produtos de uma determinada categoria dado o id dessa mesma categoria, o número de produtos por página e a página respetiva, bem como o filtro da visibilidade. Ou seja, esta funcionalidade permite "consultar produtos de categorias", sendo que estes podem ser visíveis e/ou não visíveis.

Do mesmo modo que o microsserviço QCVP, existe um conjunto de operações comando definidas na classe concreta CommandCategory que efetuam a gestão dos dados deste microsserviço, injetados pelo microsserviço CommandCategory pela aplicação do padrão CQRS, que será apresentado no seguimento desta secção.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
MongoDB	Dado que o foco são as consultas, para garantir que as mesmas sejam de acesso direto e eficiente, esta tecnologia permite agregar todos os dados num único registo de documento, isto é, os dados da categoria e os produtos, evitando-se operações de junção de dados. Do mesmo modo, esta tecnologia permite facilmente efetuar a paginação de uma coleção de dados, essencial para as funcionalidades destes microsserviços.

Tabela 9 - Tecnologias utilizadas no microsserviço Query Category All Products (QCAP).

5.4.3. Microsserviço Query Category Tree (QCT)

O microsserviço Query Category Tree tem como responsabilidade as operações de leitura sobre a estrutura total das categorias, sem os dados dos produtos associados às mesmas. Na verdade, o objetivo consiste em apresentar uma visão total da estrutura em árvore (tree) das categorias começando pela categoria raiz. Esta visão total torna-se necessária para a gestão das categorias, isto é, antes de efetuar uma operação comando o gestor ou administrador obtém esta estrutura. Mas também existe a possibilidade de usar

este microsserviço para tornar componentes de visualização das aplicações clientes, como listagem de categorias (p.e. sidebars) mais dinâmicas, isto é, atualizadas em tempo real.

A Figura 39, consiste no diagrama de classes da arquitetura do microsserviço Query Category Tree mais propriamente da camada da lógica de negócio.

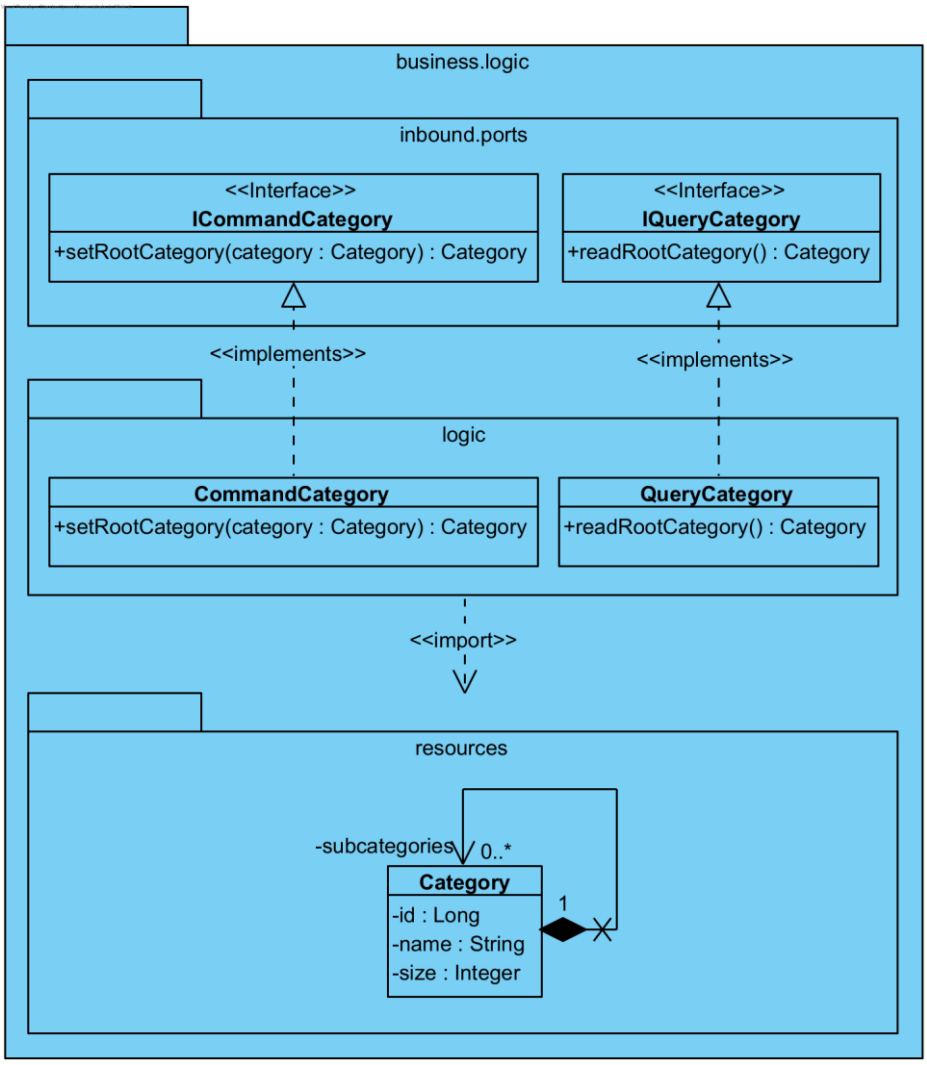


Figura 39: Diagrama de classes da camada da lógica de negócio do microsserviço Query Category Tree (QCT).

A nível de estruturas de dados existe uma classe concreta denominada por Category, que representa a entidade categoria. Em termos de atributos a mesma contém os mesmos que nos microsserviços de leitura anteriores, pelo que, não se repetirá a explicação dos mesmos, dado que, têm o mesmo significado. O mais importante de ressalvar nesta estrutura, consiste na relação recursiva de composição, denominada por subcategories, de multiplicidade de um para muitos, que representa as subcategorias de uma categoria. Esta

relação sendo de composição, tal como já apresentado em microsserviços anteriores, significa que, as subcategorias não têm existência própria sem a categoria com que fazem a relação hierarquicamente superior.

Importante ainda reforçar que, tal como no microsserviço de comando CommandCategory e ao contrário dos outros dois microsserviços de leitura QCVP e QCAP, neste microsserviço existirá apenas um registo da categoria raiz e todas as restantes são subcategorias desta e assim sucessivamente. A motivação desta estrutura e respetivas condições tem como objetivo concentrar num único registo de categoria todos os dados que são necessários para executar a funcionalidade deste microsserviço, tornando a mesma de acesso direto e eficiente.

A nível de funcionalidades existe uma operação de comando definida na classe concreta CommandCategory, denominada por setRootCategory, que como o nome sugere define a categoria raiz. Na verdade, esta funcionalidade define toda a estrutura de categorias, começando pela categoria raiz e devido à relação subcategories cria uma estrutura do tipo árvore com todas as categorias. Importante realçar que, o nome da funcionalidade ser *set,* não é por acaso, serve para indicar que não existe manipulação da estrutura interna, ou seja, trata-se de uma definição completa da estrutura, isto é, sempre que há uma alteração da mesma define-se um novo objeto da categoria raiz.

Por outro lado, existe uma operação de leitura que concretiza a responsabilidade deste microsserviço, definida na classe concreta QueryCategory, denominada por readRootCategory, que corresponde ao requisito número 21 (referente à consulta), que obtém a categoria raiz definida na operação de comando. Na verdade, esta funcionalidade retorna toda a estrutura de categorias, dado que a categoria raiz representa toda a estrutura, isto é, contém subcategorias e estas também sucessivamente, até existir categorias que não tenham subcategorias.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
MongoDB	Dado que o foco são as consultas, para garantir que as mesmas sejam de acesso direto e eficiente, esta tecnologia permite agregar todos os dados num único registo de documento, isto é, a categoria raiz e as suas subcategorias e assim sucessivamente, evitando-se operações de junção de dados. (Nota adicional, seria também interessante, dadas as responsabilidades deste microsserviço a utilização de base de dados em memória, como por exemplo: Redis)

Tabela 10 - Tecnologias utilizadas no microsserviço Query Category Tree (QCT).

5.4.4. Padrão CQRS entre CC, QCVP, QCAP e QCT

Primeiramente, uma das motivações para segregação destes microsserviços consiste na diferença de responsabilidades. Na verdade, existem em primeira instância duas grandes responsabilidades distintas, as operações de comando e leitura.

Nas operações de comando, as mesmas referem-se a gerir as categorias, bem como adicionar e remover produtos de categorias.

Por outro lado, as operações de leitura consistem em "obter a estrutura das categorias" e os "produtos de uma determinada categoria", sendo que, esta última contém duas responsabilidades distintas, uma vez que, se destina a dois diferentes grupos de utilizadores.

Um grupo são os consumidores e o outro os gestores e administradores, deste modo, dada as diferenças entre estes a nível de autorizações consideram-se responsabilidades diferentes, o que originou a segregação no QCVP para consumidores e o QCAP para gestores e administradores. Dada esta segregação, não faria sentido juntar o QCT a nenhum destes microsserviços (QCVP e QCAP), uma vez que, têm responsabilidades diferentes, neste caso, sobre a estrutura total das categorias originando o terceiro microsserviço de leituras.

Outra motivação, deve-se ao facto de tal como foi apresentado anteriormente nas descrições dos microsserviços, a adequação das estruturas de dados às funcionalidades de comando e leitura respetivamente. Na verdade, o microsserviço de comando tira partido da existência de relações bidirecionais, quer as recursivas na categoria, bem como entre esta e o produto. Por outro lado, os microsserviços de leitura que pretendem "consultar a estrutura

total das categorias" ou os "produtos de uma categoria", não têm interesse na relação bidirecional entre categorias, ou entre o produto e as mesmas. Definitivamente, a estrutura de dados dos microsserviços de leitura permite registos unitários, isto é, de acesso direto a todos os dados que se necessitam de devolver numa consulta, por exemplo na funcionalidade de "consulta dos produtos de uma categoria".

Outra motivação consiste na independência da tecnologia, neste caso, de base de dados, uma vez que, por um lado a importância das relações bidirecionais no microsserviço CommandCategory torna-se mais adequado uma tecnologia de base de dados relacional, que neste caso, selecionou-se o PostgreSQL. Por outro lado, os microsserviços de leitura QCVP, QCAP e QCT, obtém vantagem em utilizar a tecnologia não relacional, que neste caso, selecionou-se o MongoDB, uma vez que, permite colocar todos os dados a retornar numa consulta, em um único documento da base de dados, o que torna a funcionalidade de consulta mais simples e eficiente.

Outra motivação consiste na diferença de quantidade de pedidos que as funcionalidades destes microsserviços deste subdomínio recebem. Por um lado, a diferença entre as operações de comando e leitura, que neste caso, as de leitura são mais requeridas, pelo que, a segregação em microsserviços de comando e leitura permite aumentar a disponibilidade destas operações de forma independente com mecanismos de escalabilidade.

Por outro lado, a segregação entre microsserviços de leitura, especificamente o QCVP e QCAP, motivada pelo facto de que o QCVP recebe mais pedidos do que o QCAP, dado que, é focado para os consumidores que são em maior número e requerem mais pedidos de consulta. Deste modo, esta segregação para além de ter reduzido a lógica desnecessária no QCVP, de autorização e filtragem como foi abordado anteriormente, permite aumentar a disponibilidade de forma independente destes microsserviços de leitura com mecanismos de escalabilidade.

A aplicação arquitetural deste padrão permite também que, no futuro possam ser adicionados à mesma outros microsserviços de leitura de forma mais trivial, que de outro modo seria menos consistente e produtivo. Na verdade, isto significa que se no futuro existir um novo microsserviço de leitura, o mesmo tem apenas de subscrever a sua queue ao exchange do CommandCategory com as routing keys associadas às operações de comando que o mesmo pretende receber.

Por outro lado, nem sempre se torna assim tão trivial, uma vez que, dada as estruturas de dados de certos microsserviços de leitura, os mesmos necessitam de receber informação adicional à existente na operação de comando.

Por exemplo, no caso do QCVP e QCAP, quando se remove um produto no CommandCategory não é suficiente informar o id do produto removido como é costume. Na verdade, dadas as estruturas destes microsserviços de leitura, onde existe uma relação unidirecional entre a categoria e o produto, estes microsserviços não conseguem saber apenas pelo id do produto em que categorias o mesmo se encontra. Deste modo, o CommandCategory nesta situação tem de comunicar também os ids das categorias a que este produto está associado antes de ser removido, para que, nos microsserviços de leitura se remova em cada categoria esse mesmo produto, dado que se trata de uma cópia de dados.

Em suma, relativamente a este tema, se não se efetuasse esta segregação e existisse um único microsserviço, seria bastante complexo consentir uma estrutura de dados onde funcionalidades não ficassem prejudicadas.

5.5. Subdomínio das Encomendas

Nesta secção, apresentam-se os microsserviços que tem responsabilidades sobre o subdomínio das encomendas.

Efetivamente, este subdomínio contém três microsserviços com diferentes responsabilidades, mas com um objetivo comum, que consiste na concretização de uma encomenda de produtos. No entanto, necessita-se da contribuição de microsserviços de outros subdomínios, como QueryProduct e Saga para obtenção dos dados dos produtos e orquestração do processo da criação da encomenda.

5.5.1. Microsserviço ShoppingCart (SC)

O microsserviço ShoppingCart, tem como responsabilidade as operações sobre a entidade carrinho de compras. Na verdade, a responsabilidade deste microsserviço consiste em guardar um conjunto de produtos que o consumidor pretende encomendar.

A Figura 40, consiste no diagrama de classes da arquitetura do microsserviço ShoppingCart mais propriamente da camada da lógica de negócio.

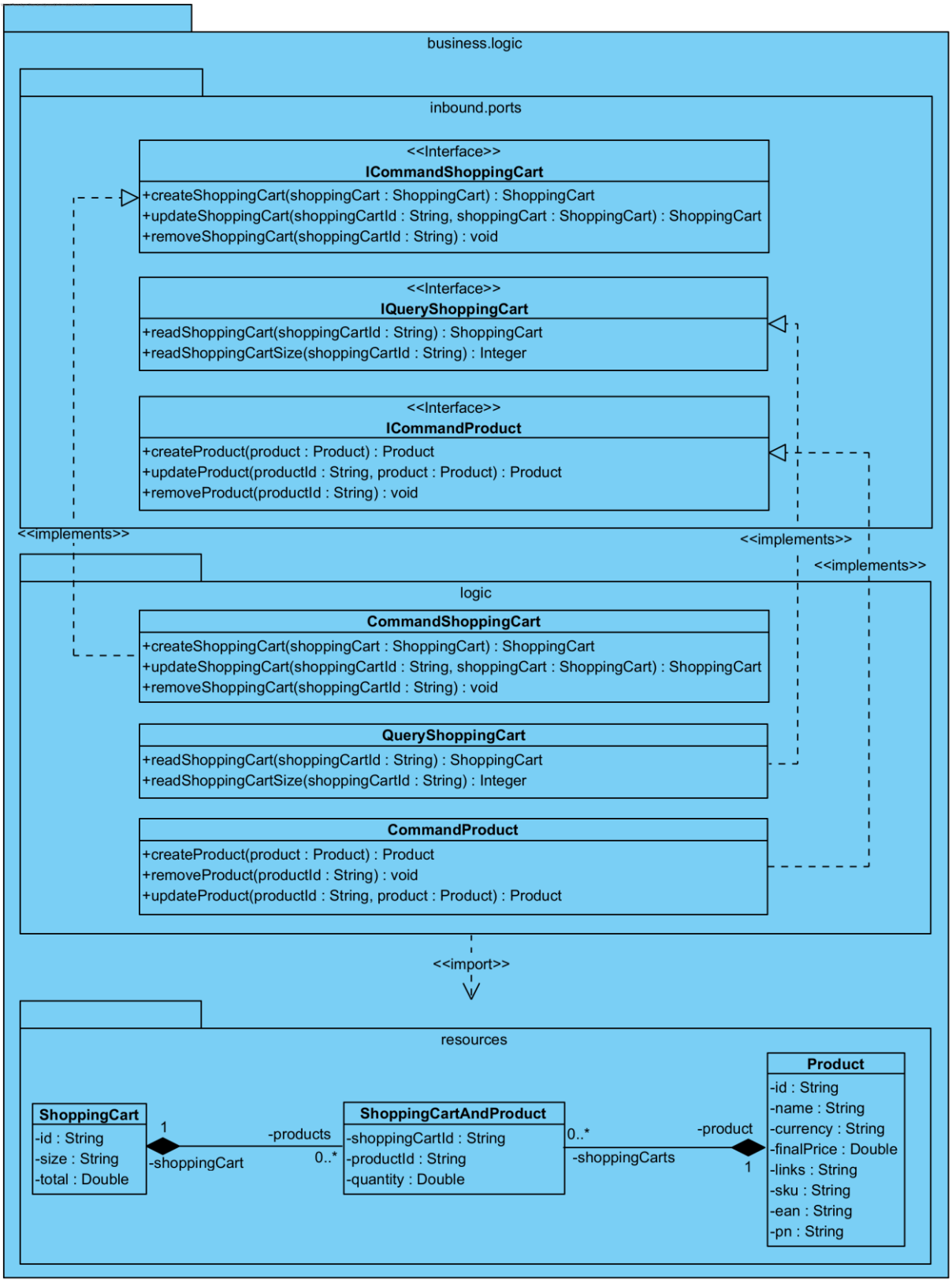


Figura 40: Diagrama de classes da camada da lógica de negócio do microsserviço ShoppingCart.

A nível de estruturas de dados existe uma classe concreta denominada por ShoppingCart que representa a entidade carrinho de compras. Em termos de atributos esta classe concreta contém um id que serve de identificador e será o mesmo que o id do consumidor proprietário deste carrinho de compras. De seguida, contém um atributo size e total que são respetivamente a quantidade de produtos distintos incluídos no carrinho de compras e o preço total do mesmo.

Existe outra classe concreta denominada por Product que representa a entidade produto. A nível de atributos, contém um conjunto que já foram abordados em outros microsserviços anteriores. Tal como referido na secção 5.3, estes dados são injetados neste microsserviço proveniente de outros que tem a responsabilidade sobre os mesmos.

A nível de relações, dado que a relação a nível de negócio entre o carrinho de compras e o produto é de multiplicidade de muitos para muitos e existem atributos necessários nesta relação, tornou-se implícito criar uma classe concreta para representar a mesma, denominada por ShoppingCartAndProduct. Em termos de atributos esta classe concreta contém os identificadores das entidades que relaciona, isto é, o carrinho de compras e o produto, bem como o atributo quantity que representa a quantidade que o consumidor quer comprar de um determinado produto.

Existe uma relação de composição entre as classes ShoppingCart e a ShoppingCartAndProduct, denominada por products, bidirecional com multiplicidade de um para muitos que representa os produtos que estão num carrinho de compras. No entanto, como já foi referido não são diretamente os produtos, mas sim um intermediário da relação com os produtos, que especifica a quantidade. Por outro lado, existe uma relação de composição entre as classes Product e ShoppingCartAndProduct, denominada por shoppingCarts, bidirecional com multiplicidade de um para muitos, que representa os carrinhos de compras onde o produto está incluído em uma determinada quantidade. Ou seja, um objeto da classe ShoppingCartAndProduct associa um produto em uma determinada quantidade a um carrinho de compras.

Em relação a funcionalidades, as mesmas dividem-se em operações de comando e leitura, correspondendo ao requisito número 11. Na verdade, existem operações de comando para gestão da entidade do carrinho de compras, desde criar, atualizar e remover o mesmo. Uma nota adicional, consiste em que, a operação de comando atualizar, implementada na classe concreta CommandShoppingCart, denominada por updateShoppingCart, corresponde

a adicionar e remover produtos do carrinho de compras, bem como a criação deste caso seja a primeira atualização. Por outro lado, existe um conjunto de operações de leitura, desde a consulta de todo o carrinho de compras, bem como apenas o tamanho do mesmo, que pode ser útil em termos de componentes de interface da aplicação cliente.

Importante realçar que, não foram aplicados padrões, como por exemplo Saga, para a criação automática dos carrinhos de compras quando se regista um consumidor, uma vez que, dada a complexidade já existente neste caso de estudo desta dissertação, decidiu-se implementar uma alternativa a esta saga, que num caso normal seria o mais adequado. Deste modo, o que se definiu foi que, caso o consumidor nunca tenha adicionado nada ao seu carrinho o mesmo não vai existir, pelo que, as aplicações clientes devem ser preparadas para lidarem com o erro que vai ser emitido nestas situações. Após a primeira adição de um produto ao carrinho de compras o mesmo será criado para esse utilizador. Por outro lado, o carrinho de compras só será removido caso o administrador o pretenda fazer, por consequência, por exemplo de o consumidor já não ter um registo de conta. No entanto, esta operação de remoção também seria um candidato à implementação de um processo saga, para se garantir a consistência quando se remove um consumidor, uma vez que, não faz mais sentido existir o carrinho de compras do mesmo.

Por fim, ainda em relação a funcionalidades, existe um conjunto de operações de comando sobre a entidade produto, desde criar, atualizar e remover, para permitir a injeção e sincronização destes dados neste microsserviço, que como já foi abordado, são provenientes dos microsserviços responsáveis pelos mesmos.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
PostgreSQL	A existência de muitas relações entre entidades, algumas bidirecionais, levou à escolha de uma base de dados relacional. O PostgreSQL, é uma tecnologia conhecida pelo programador.

Tabela 11 - Tecnologias utilizadas no microsserviço ShoppingCart.

5.5.2. Microsserviço Order

O microsserviço Order, tem como responsabilidade as operações de comando e leitura sobre a entidade encomenda. Na verdade, este microsserviço tem o objetivo de gerir e guardar os registos de encomendas.

A Figura 41, consiste no diagrama de classes da arquitetura do microsserviço Order mais propriamente da camada da lógica de negócio.

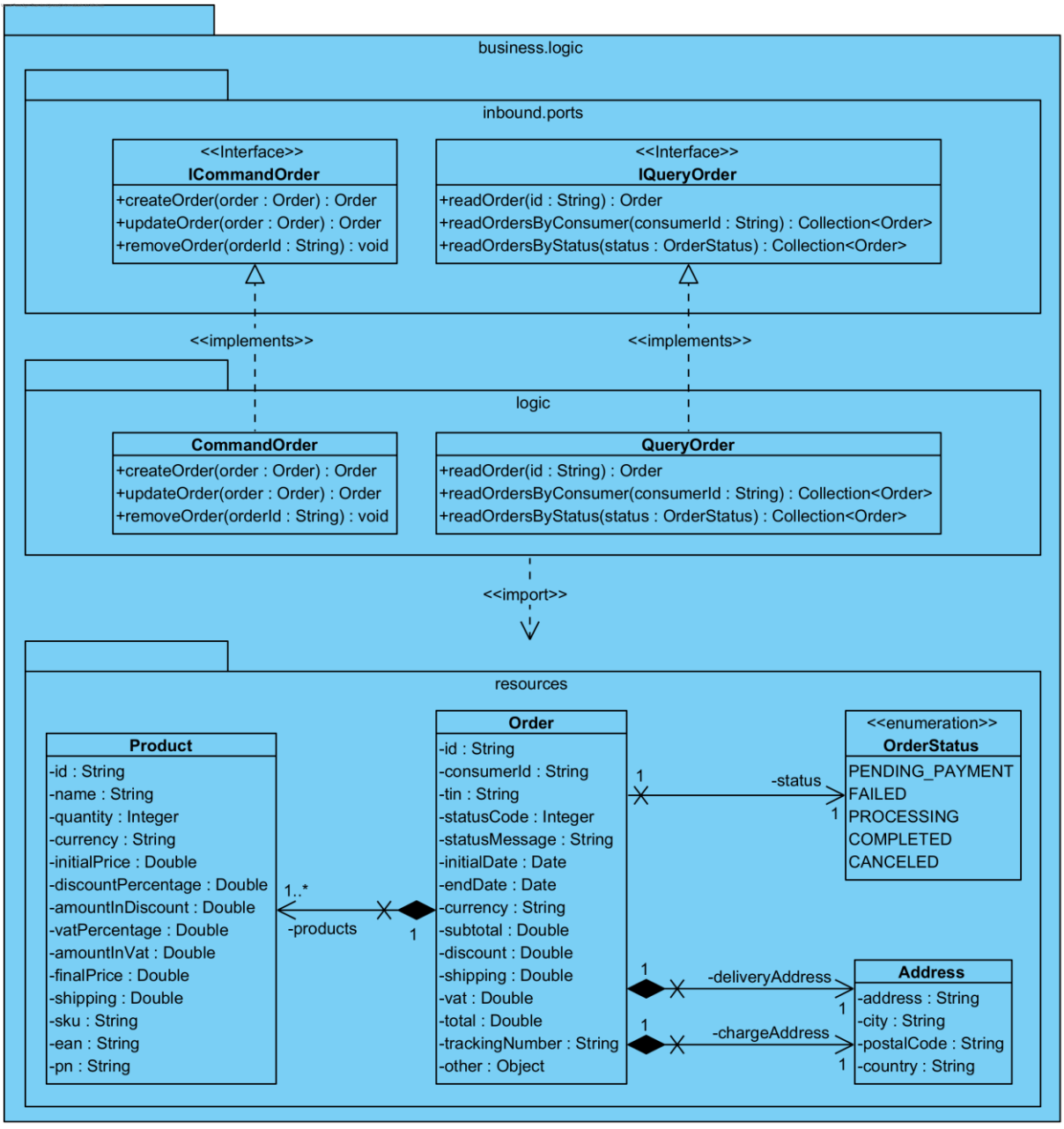


Figura 41: Diagrama de classes da camada da lógica de negócio do microsserviço Order.

Em relação a estruturas de dados existe uma classe concreta, denominada por Order, que representa a entidade encomenda. Em termos de atributos existe um id que serve de identificador da encomenda, um consumerId e tin (p.e. NIF), que identificam o consumidor que efetuou o pedido de encomenda. De seguida, existem um conjunto de atributos de monitorização da encomenda, como por exemplo: status, statusCode, statusMessage, initialDate, endDate e trackingNumber, que permitem analisar o estado atual da encomenda. Ainda, existe um conjunto de valores referentes à contabilidade da encomenda, tais como: currency, subtotal, discount, shipping, vat e total. Todavia, existe um atributo chamado other, para que, possam ser adicionados outros dados que não estão previstos nesta configuração da encomenda e possam ser importantes. Por fim, existem dois endereços, para entrega da encomenda e associação ao pagamento.

Existe outra classe concreta denominada por Product que representa a entidade produto e contém um conjunto de atributos que têm sido abordados ao longo dos microsserviços apresentados, pelo que, já são conhecidos. No entanto, apenas torna-se relevante referir o atributo quantity, que não ocorre nas classes do produto em outros microsserviço e corresponde à mesma quantidade que o consumidor seleciona no carrinho de compras para um determinado produto.

A nível de relações, existe uma relação de composição denominada por products, unidirecional de multiplicidade de um para muitos que representa os produtos de uma determinada encomenda. Um ponto importante a referir nesta relação, consiste em a mesma ser de composição, que implica que, estes produtos apenas existem para uma determinada encomenda. Na verdade, isto significa que, são apenas cópias dos dados correntes dos produtos no momento da realização da encomenda, uma vez que, o que interessa para a encomenda são os dados momentâneos.

A nível de funcionalidades, existem operações de comando implementadas na classe concreta CommandOrder, desde criar, atualizar e remover encomendas. Das operações de comando as duas últimas, atualizar e remover, correspondem ao requisito número 24, enquanto a criação será utilizada durante um processo saga, denominado "criar encomenda", que será abordado na secção 5.6.2, correspondente ao requisito número 11.

Por outro lado, existem operações de leitura implementadas na classe concreta QueryOrder, que correspondem aos requisitos número 13 e 24, respetivamente do consumidor e gestor. A funcionalidade readOrder corresponde à consulta dos dados de uma

encomenda, a funcionalidade readOrdersByConsumer corresponde à consulta de todas as encomendas de um consumidor e a funcionalidade readOrdersByStatus corresponde à consulta de todas as encomendas filtradas pelo estado das mesmas.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
MongoDB	Dado que as operações de comando não necessitam de base de dados relacionais, e para tornar as consultas simples e eficientes, a seleção desta tecnologia permite o acesso direto a todos os dados de uma encomenda, incluindo os produtos, em um único documento, sem necessidade de junção de dados.

5.5.3. Microsserviço Inventory

O microsserviço Inventory, tem a responsabilidade de um conjunto de operações de comando e leitura para gestão do inventário dos produtos.

A Figura 42, consiste no diagrama de classes da arquitetura do microsserviço Inventory mais propriamente da camada da lógica de negócio.

Em relação a estruturas de dados, existe uma classe concreta denominada por Product, que representa a entidade produto. Em relação aos atributos desta classe, para além dos comuns a outros microsserviços, torna-se importante realçar o sku que tem um papel importante neste domínio dos inventários e inclusive poderá ser um índice para procura de produtos, bem como de integração de aplicações terceiras. O atributo stockQuantity e stockUnit representa a quantidade numa determinada unidade de medida existente em inventário de um produto.

Por outro lado, o atributo deleted marca se este produto foi removido ("*soft delete*"), uma vez que, dada a natureza destes dados não se pretende remover de imediato os mesmos, quando se remove o produto, mas sim isso será feito posteriormente pelo administrador.

Por fim, existe um atributo denominado por stockStatus, o qual categoriza a quantidade de stock de um produto. Este atributo tem um papel importante, uma vez que, permite informar os microsserviços interessados sobre o estado do stock de um produto sem necessitar de comunicar as quantidades dos mesmos. Do mesmo modo, ao comunicar o

estado categorizado do stock significa que as alterações da quantidade em stock não vão alterar com tanta frequência esse estado, pelo que, serão necessárias menos comunicações para informar os microsserviços subscritores.

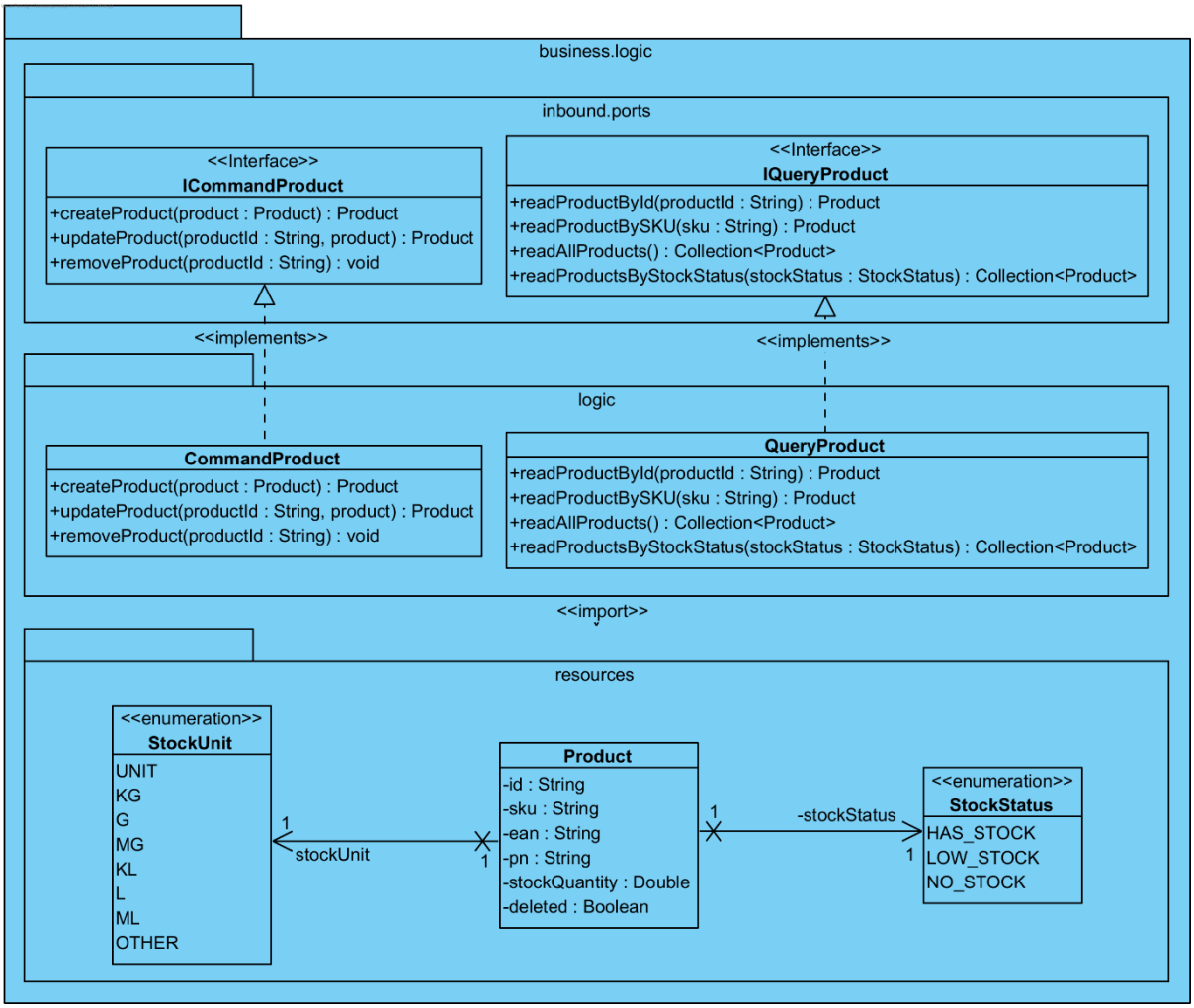


Figura 42: Diagrama de classes da camada da lógica de negócio do microsserviço Inventory.

Em relação a funcionalidades, as mesmas estão segregadas em operações de comando e leitura respetivamente nas classes concretas CommandProduct e QueryProduct, que correspondem ao requisito número 20. Destas funcionalidades destacam-se nas operações de comando a criação, atualização e remoção do produto. Por outro lado, nas operações de leitura, que antecedem as operações de comando, existe a consulta de um produto específico, pelo id ou sku, a consulta de todos os produtos existentes e a consulta de produtos por filtragem do estado de stock dos mesmos.

Tecnologias

Nome	Motivações
Spring Boot	Explicadas na secção 4.2.3
Hibernate	Explicadas na secção 4.2.3
RabbitMQ	Explicadas na secção 4.2.1
PostgreSQL	O PostgreSQL, é uma tecnologia conhecida pelo programador.

Padrão CQRS entre Inventory, CP e CC

A aplicação do padrão CQRS foi efetuada entre os microsserviços Inventory, CommandProduct e CommandCategory. Na verdade, na aplicação deste padrão o microsserviço Inventory assume o papel de publicador de dados, pelo que, faz sentido abordar esta aplicação nesta fase, enquanto os microsserviços CommandProduct e CommandCategory são subscritores desses dados.

Efetivamente, o objetivo da aplicação do padrão CQRS entre estes microsserviços, consiste em evitar pedidos recorrentes dos dados sobre o stock dos produtos ao microsserviço Inventory, tornando os mesmos mais independentes. Na verdade, isso consegue-se, uma vez que, deixa de ser necessário que os microsserviços CommandProduct e CommandCategory façam pedidos sobre os dados do stock ao microsserviço Inventory. De outro modo, esses dados são injetados nesses microsserviços de forma assíncrona, o que os torna independentes bem como as suas funcionalidades. Por forma a tornar este processo mais eficiente, como já foi abordado anteriormente, não são comunicadas as quantidades de stock dos produtos, mas sim o estado das mesmas, uma vez que, tem alterações com menos frequência o que evita comunicações desnecessárias.

Outro ponto positivo a referir, consiste em ter selecionado corretamente os microsserviços a receberem estes dados, uma vez que, tal como já foi abordado, quer o CommandProduct, quer o CommandCategory, também participam em outras aplicações do padrão CQRS, o que permite que, na realidade os dados do estado do stock dos produtos alcancem mais microsserviços. Na verdade, os microsserviços que tem o interesse sobre estes dados do stock são os microsserviços de leitura (QP, QCVP e QCAP). No entanto, deve-se respeitar as responsabilidades, e neste caso, como se trata de uma atualização do estado do stock do produto faz todo sentido comunicar a mesma aos microsserviços de comando. Desta

forma, garante-se que, quando estes dados chegam a estes microsserviços os mesmos comunicarão estes dados aos microsserviços de leitura interessados.

Ainda, importante realçar que, tal como em outros padrões de CQRS implementados, esta arquitetura permite que no futuro sejam adicionados de forma trivial mais microsserviços subscritores interessados nos dados do stock dos produtos, como por exemplo o ShoppingCart, para permitir apresentar os dados do stock aquando da consulta dos produtos do carrinho de compras.

Em suma, com a implementação deste padrão, conseguiu-se tornar um conjunto de microsserviços e as respetivas funcionalidades dependentes apenas do microsserviço onde estão implementadas, sem necessitarem de efetuar internamente pedidos a outros microsserviços, o que lhes concede uma maior independência.

5.6. Fluxo dos Processos Saga

Nesta secção, apresenta-se o fluxo genérico comum a todos os processos saga entre o microsserviço orquestrador e os respetivos microsserviços participantes. Apresenta-se especificamente os pontos mais importantes de cada processo saga que foi implementado neste sistema.

A Figura 43, consiste no diagrama de sequência que representa o fluxo genérico de qualquer processo saga num cenário de sucesso.

O processo inicia-se com um pedido HTTP do utilizador (o método HTTP é variável para cada processo saga e está definido na configuração da mesma) com o identificador da configuração da saga denominado por sagaDefinitionId (p.e. create.product) e os dados necessários para a execução da mesma (passo 1).

No microsserviço Saga executa-se o método initSaga (passo 1.1) que consiste em iniciar o processo saga, criando um registo para o mesmo com um identificador, denominado por sagaId entre outros dados necessários, que foram abordados na secção 5.2 sobre o microsserviço Saga, bem como, observar o exemplo da Listagem 2. Dado que os processos saga são assíncronos, torna-se implícito guardar dados associados aos mesmos, pelo que, o identificador sagaId é reencaminhado entre os orquestradores e os participantes para efetuar a interligação dos registos entre estas entidades.

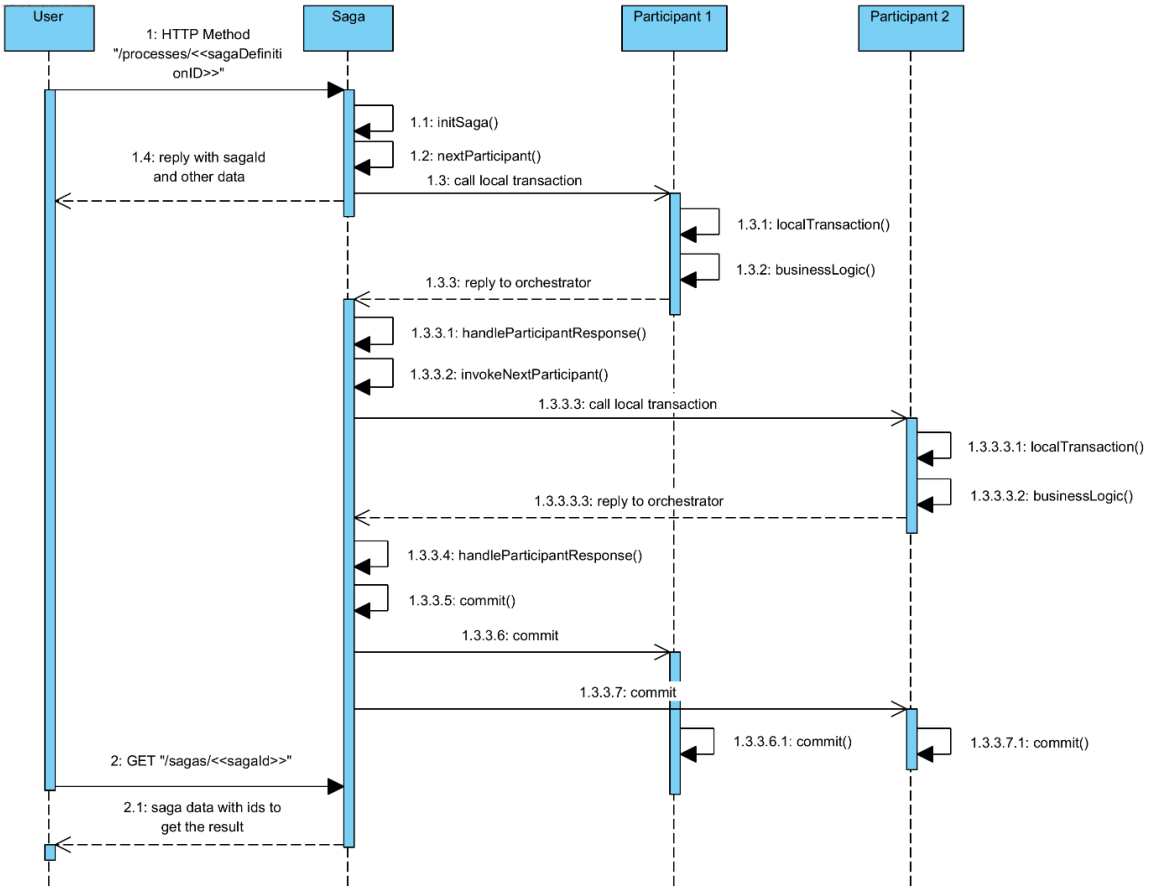


Figura 43: Diagrama de Sequência do fluxo de um processo saga com cenário de sucesso.

SagaId	459a51bc8930
Name	Create.product.sagaDefinition
Start	2022/01/01-00h00m05s
End	--
Status	PENDING
CurrentService	CommandCategory
ErrorService	--
Message	--
Code	--
EntityId	m@bob
Position	1
Size	4
Input	{...}
Output	--

Listagem 2: Registo de um processo Saga denominado Create Product.

De seguida, ocorre a execução do método nextParticipant (passo 1.2) que através da configuração da saga, dado o sagaDefinitionId, determina o primeiro microsserviço participante da sequência a invocar e envia assincronamente uma mensagem do tipo comando para o mesmo (passo 1.3).

Subsequentemente, é retornado para o utilizador os dados do registo que foi criado, semelhante à Listagem 2 , com principal foco no sagaId que permite monitorizar o estado da saga. Este retorno logo após o início do processo ao utilizador, permite que, o mesmo não fique ativamente à espera do resultado do processo da saga que pode ser demorado ou ficar pendente da disponibilidade de algum dos microsserviços participantes. Esta abordagem de tornar o processo saga assíncrono para o utilizador consiste num padrão denominado por Self-contained Service abordado nas secções 2.2.1 e 5.2.

No primeiro participante ao receber a mensagem do tipo comando, executa o método localTransaction (passo 1.3.1), que efetua um conjunto de operações, desde bloqueios e backups de entidades, entre outras necessidades seguido da invocação da lógica de negócio (passo 1.3.2) respetiva para o processo saga. De seguida, responde ao orquestrador de forma assíncrona com uma mensagem do tipo resposta com estado de sucesso. Esta mensagem conterá um conjunto de meta dados, bem como dados que foram recebidos e por isso são reencaminhados, dado que o processo é assíncrono, bem como outros que são adicionados por este participante à mensagem.

O microsserviço Saga recebe esta mensagem e executa o método handleParticipantResponse (passo 1.3.3.1) que analisa a mesma atualizando o registo saga que foi criado no passo 1.1 com o novo estado, como por exemplo o microsserviço atual, a posição, entre outros. De seguida, com os dados do registo atualizados executa o nextParticipant e o processo repete-se para o outro participante.

Após a resposta de sucesso do segundo participante, do mesmo modo, analisa-se a mensagem de resposta recebida. Como se trata do último participante da sequência será executado o método commit (passo 1.3.3.5) que vai atualizar o registo saga com os identificadores ou outros dados que permitam chegar ao resultado da mesma (p.e. identificadores de produtos ou encomendas), bem como atualizar o estado, as mensagens, códigos, entre outros, para os valores de sucesso. De seguida, são enviadas mensagens do tipo comando para todos os participantes com a finalidade de executarem os métodos commit

(passo 1.3.3.6.1 e 1.3.3.7.1) que tem o objetivo principalmente de efetuarem as últimas alterações e desbloquearem as entidades.

Ainda, durante todo o processo, o utilizador pode efetuar a consulta do registo saga para monitorizar o estado da mesma. No entanto, ao fim da execução do método commit (passo 1.3.3.5) no microsserviço Saga, o utilizador já pode consultar o registo com o resultado final do processo saga executado e obter os identificadores que permitem chegar aos dados das entidades resultantes deste processo.

Por fim, em relação ao processo genérico da saga, torna-se necessário apresentar como o orquestrador coordena o processo com a ocorrência de um erro de negócio ou outros nos microsserviços participantes.

A Figura 44, consiste no diagrama de sequência, que representa o fluxo genérico de qualquer processo saga num cenário de erro.

O fluxo do processo é exatamente igual até à ocorrência do erro (retângulo vermelho), que neste caso, ocorre no participante dois (passo 1.3.3.3.2). Na ocorrência de um erro, o microsserviço participante responde ao orquestrador com uma mensagem do tipo resposta, onde o campo status da mensagem vai com valor ERROR. Como em qualquer processo saga, qualquer mensagem recebida no orquestrador é analisada no método handleParticipantResponse (passo 1.3.3.4), que neste caso, irá atualizar o registo da saga com os novos dados, isto é, a ocorrência do erro.

Efetivamente, como se trata de um erro, a grande diferença que ocorre na coordenação consiste em inverter a sequência dos participantes até ao momento executados e efetuar de forma sequencial a execução em cada um dos participantes da compensatingTransaction, que irá inverter as alterações efetuadas pela localTransaction. Após chegar ao primeiro participante executado, do mesmo modo que o processo em caso de sucesso, são enviadas mensagens do tipo comando para todos os microsserviços participantes para efetuarem os commits.

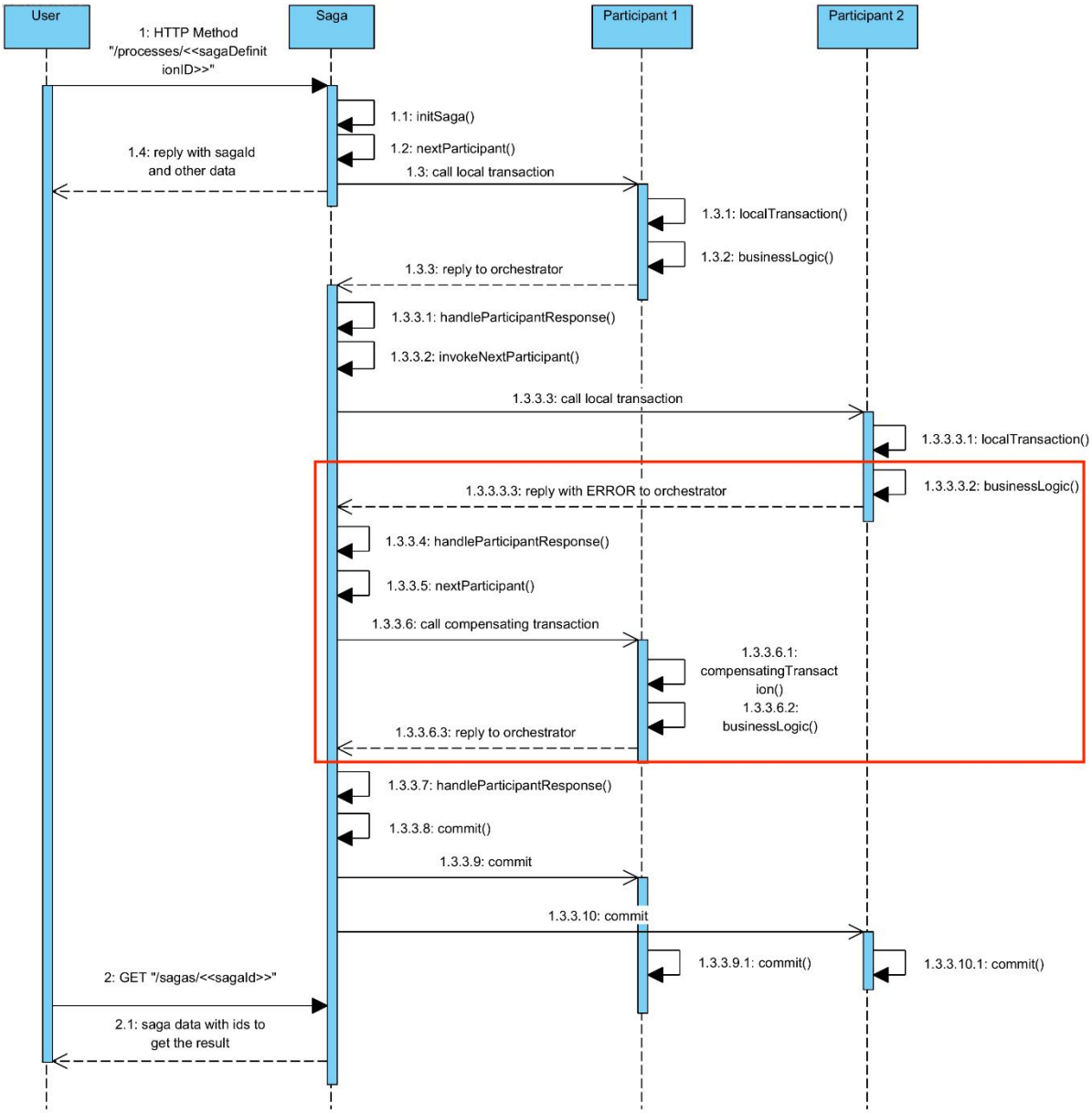


Figura 44: Diagrama de Sequência do fluxo de um processo saga com cenário de erro.

5.6.1. Sagas relativas ao Produto

O objetivo destas sagas, como o nome indica, consiste em criar, atualizar e remover um produto no sistema e-commerce. No entanto, como foi abordado ao longo de várias secções, neste problema de e-commerce o produto é uma entidade central que estabelece relações com quase todas as entidades. Deste modo, necessita-se de garantir as propriedades ACID nestas funcionalidades sobre esta entidade ao longo de múltiplos microsserviços.

A Figura 45, consiste na sequência de execução dos microsserviços participantes nos processos saga denominados por Criar, Atualizar e Remover Produto.

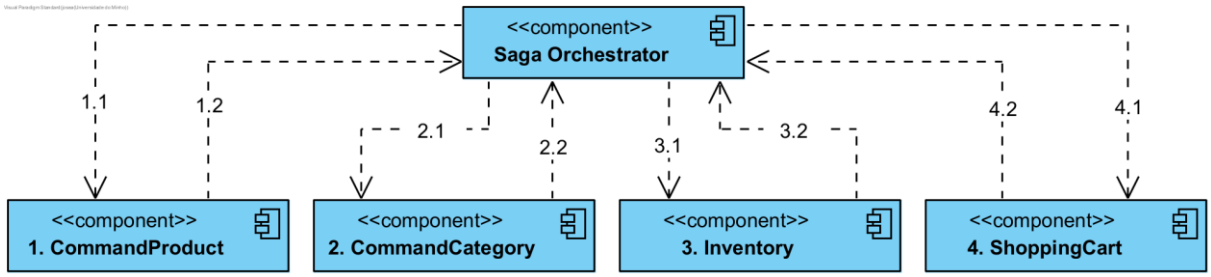


Figura 45: Sequência dos microsserviços participantes dos processos saga denominados por "Criar Produto", "Atualizar Produto" e "Remover Produto".

Efetivamente, dado que estes processos saga incidem sobre a entidade produto decidiu-se que o primeiro microsserviço participante tem de ser o responsável pelo mesmo. Na verdade, pode ser necessário efetuar validações, desde se o produto já existe, se não existe, se os dados recebidos são válidos, gerar identificadores, entre outros procedimentos importantes que são da responsabilidade deste microsserviço e que são subsequentemente comunicados aos microsserviços participantes seguintes.

Por outro lado, as sagas que incidem sobre a mesma entidade, por consequência da forma como o isolamento das sagas foi desenhado e implementado o primeiro participante onde se bloqueia a entidade tem de ser o mesmo.

Na verdade, o motivo deve-se ao facto de que, diferentes processos saga (p.e. atualizar produto e remover produto) que incidam na mesma entidade têm de garantir que o primeiro bloqueio ("lock") da entidade seja feito no mesmo participante. O objetivo consiste em garantir a propriedade de isolamento, mas principalmente para não ocorrer bloqueios de semântica intermináveis, também denominados por *dead locks*.

Por exemplo, imagine-se o caso da Figura 46, em que a saga atualizar produto segue a sequência da Figura 45 e a remover produto troca a ordem do CommandProduct pelo CommandCategory. Ou seja, na saga remover produto, o primeiro participante é o CommandCategory e o segundo é o CommandProduct.

Se forem executadas concorrentemente as sagas sobre o mesmo produto ambas vão bloquear o produto no primeiro microsserviço participante das respetivas sequências. Quando invocarem o segundo microsserviço participante respetivo (CC para a saga atualizar produto e CP para a saga remover produto) o produto vai estar bloqueado em ambos os microsserviços participantes (passo 2.2.1.1 e 2.2.2.1) pela saga concorrente respetivamente.

Deste modo, ambas as sagas ficam em espera indefinidamente, dado que, ocorre um género de *dead lock* que teria de ser resolvido manualmente pelo administrador do sistema.

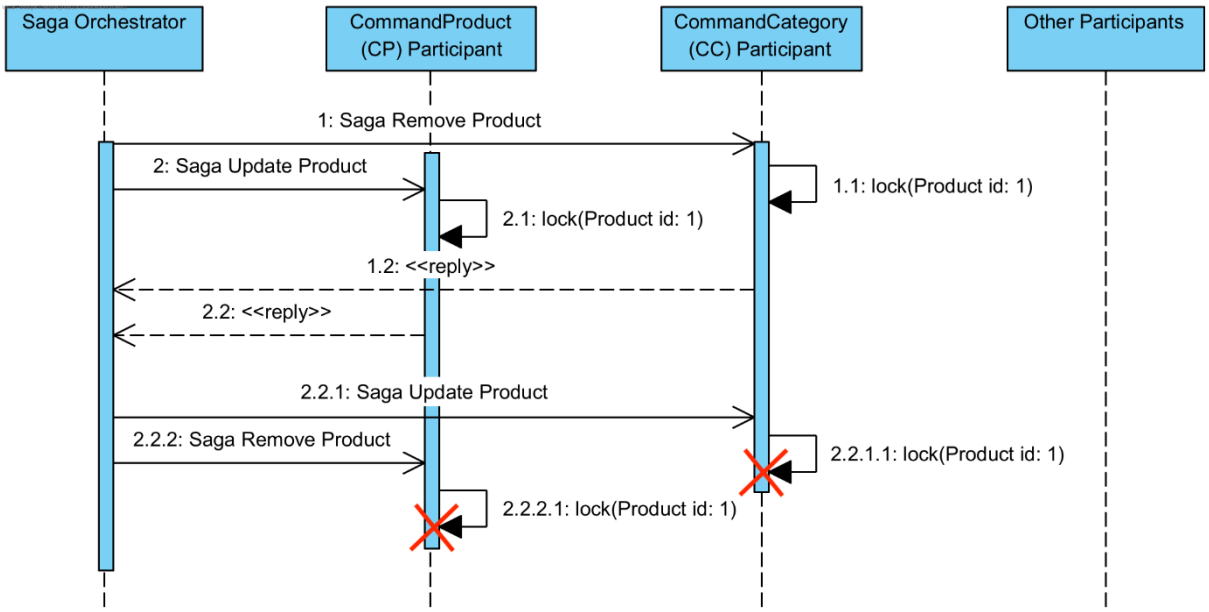


Figura 46: Exemplo do problema de dois processos de sagas diferentes, que atuam sobre a mesma entidade, onde o primeiro participante a bloquear a entidade é diferente.

A Figura 47, representa dois processos de sagas diferentes que atuam sobre a mesma entidade, no entanto, o primeiro participante a bloquear a entidade é o mesmo em ambas as sagas. Deste modo, verifica-se que quando a Remove Product tenta adquirir o bloqueio da entidade, o mesmo não consegue e fica em espera (retângulo vermelho), enquanto a saga Update Product termina o processo.

Após a saga Update Product terminar, isto é, quando são feitos os *commits*, a saga Remove Product consegue adquirir o bloqueio da entidade (retângulo azul) e continuar o processo nos restantes participantes com sucesso.

Efetivamente, torna-se importante respeitar a regra abordada anteriormente, no entanto, a mesma é respeitada se se respeitar as responsabilidades dos microsserviços, uma vez que, normalmente o primeiro microsserviço participante a bloquear a entidade corresponde ao microsserviço responsável pela mesma.

Por exemplo, nas sagas sobre o produto, o primeiro participante a bloquear o mesmo é o CommandProduct, microsserviço que detém a responsabilidade sobre a entidade produto. Deste modo, para maior parte dos casos, ou seja, outros sistemas, esta condição não se torna um problema, principalmente no caso de estudo e-commerce.

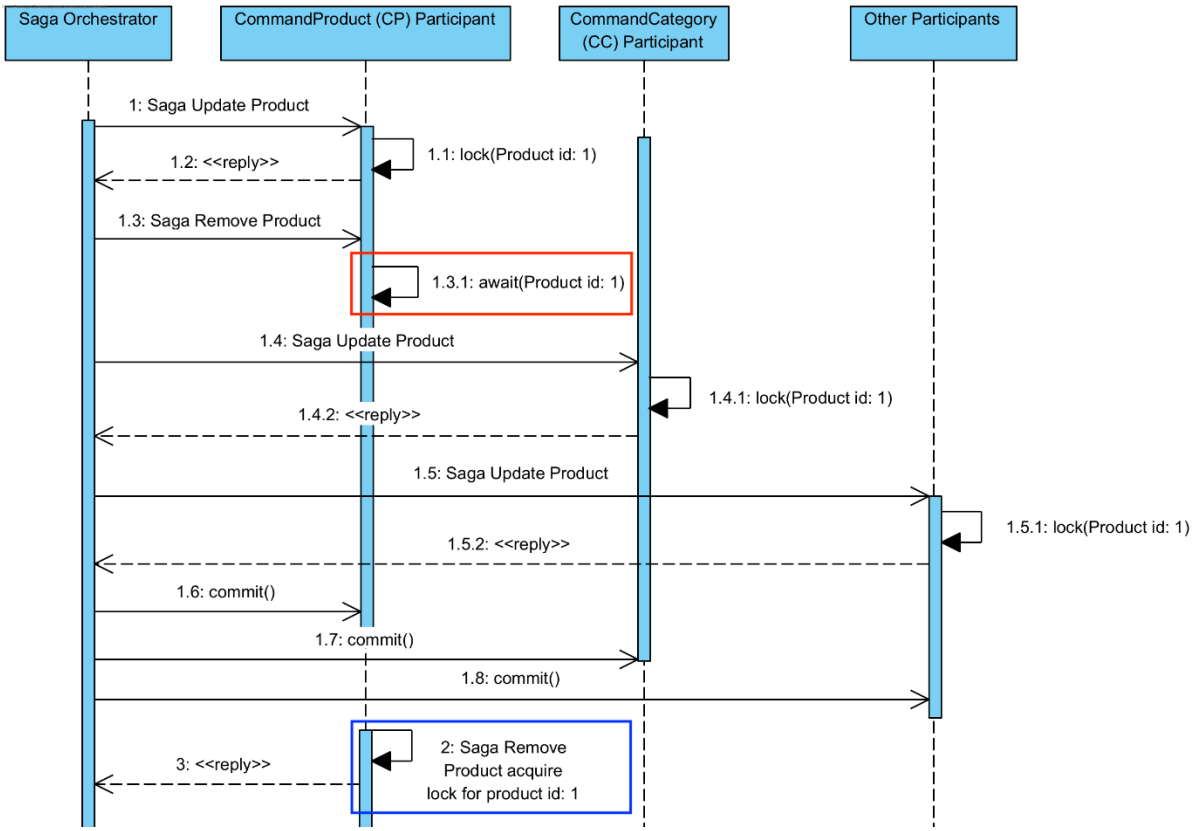


Figura 47: Exemplo de dois processos de sagas diferentes, que atuam sobre a mesma entidade, onde o primeiro participante a bloquear a entidade é o mesmo, logo as sagas são executadas de forma consistente e com sucesso.

5.6.2. Saga Criar Encomenda

O objetivo desta saga consiste em concretizar o resultado esperado deste sistema e-commerce, isto é, a compra de produtos por parte do consumidor.

A Figura 48, consiste na sequência de execução dos microsserviços participantes no processo saga denominado por Criar Encomenda.

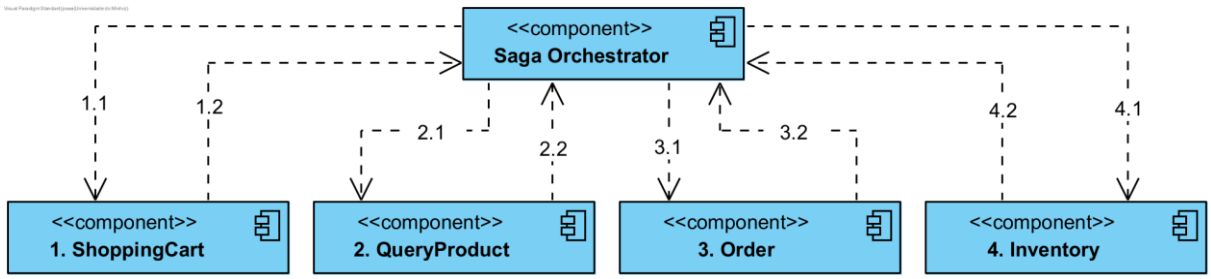


Figura 48: Sequência dos microsserviços participantes do processo saga denominado por Criar Encomenda.

Efetivamente, como a compra de produtos envolve a colaboração e dados de múltiplos microsserviços, que por sua vez efetuam transações locais para esse mesmo efeito, torna-se implícito garantir que a transação global de criar uma encomenda cumpra as propriedades ACID ao longo destes microsserviços.

Uma particularidade deste processo saga em relação aos anteriores, consiste na presença de um microsserviço participante, o QueryProduct, que não efetua uma alteração de dados, isto é, uma transação local propriamente dita, mas sim uma consulta de dados que são encaminhados para os restantes microsserviços participantes. No entanto, independentemente disso o mesmo é tratado como qualquer outro participante, dado que, estes dados são necessários para a criação da encomenda.

Por fim, ainda em relação ao microsserviço QueryProduct, a inclusão do mesmo neste processo saga demonstra a importância de garantir os limites bem definidos das responsabilidades dos microsserviços.

Efetivamente, isso deve-se ao facto de que, por exemplo o microsserviço ShoppingCart tem alguns dos dados referentes à contabilidade do produto necessários para a criação da encomenda, que são obtidos no microsserviço QueryProduct. Uma possível abordagem podia ser adicionar os restantes dados em falta da contabilidade ao microsserviço ShoppingCart e assim evitava-se a inclusão do microsserviço QueryProduct. No entanto, isto corrompe um dos princípios fundamentais das arquiteturas orientadas a microsserviços, que consiste na segregação correta das responsabilidades e definição dos limites das mesmas.

Na verdade, ao adicionar os tais dados ao microsserviço ShoppingCart os mesmos não são necessários para as responsabilidades desse mesmo microsserviço, pelo que, desrespeita-se imediatamente o princípio. Mas também, ao usar o microsserviço ShoppingCart ao invés do QueryProduct no processo saga, para obter esses dados sobre o produto, também vai contra o mesmo princípio pois não faz parte das suas responsabilidades, apesar do mesmo conter parte dos dados necessários.

5.7. Outros Padrões de Arquiteturas orientadas a Microsserviços

Nesta secção, são apresentados os padrões das arquiteturas orientadas a microsserviços que são genéricos.

5.7.1. Service Discovery

Os padrões presentes na categoria de Service Discovery, abordada na secção 2.2.5, foram implementados nesta arquitetura do caso de estudo e-commerce com a utilização da tecnologia de deployment Docker.

Efetivamente, a tecnologia Docker implementa os padrões Service Discovery internamente, na medida em que, permite a comunicação entre serviços utilizando os nomes respetivos dos mesmos. Na verdade, esta tecnologia gere as localizações dos serviços, onde os mesmos são registados com um endereço físico associado a um nome (semelhante ao que acontece com o DNS) na sua rede interna. Assim, permite que os endereços físicos destes serviços, como base de dados, message brokers e a lógica dos microsserviços, possam ser alterados de forma totalmente abstrata para os serviços que os utilizam.

5.7.2. API Gateway e Segurança

Em relação ao padrão API Gateway, abordado na secção 2.2.4, o mesmo foi implementado com recurso à tecnologia Kong.

Efetivamente, a tecnologia Kong tem a responsabilidade de API Gateway expondo a API do backend da arquitetura deste caso de estudo de e-commerce. A seleção desta tecnologia deve-se ao facto de ser bastante recomendada para arquiteturas orientadas a microsserviços [1]. Do mesmo modo, contém um conjunto de plugins interessantes para esta arquitetura, como os padrões de Segurança abordados na secção 2.2.6, que garantem a autenticação e autorização de access tokens dos utilizadores.

Outra característica da tecnologia Kong consiste na capacidade de configuração flexível de regras de roteamento com métodos e caminhos HTTP específicos, que são direcionados para os microsserviços backend.

O Kong sendo um serviço incluído na infraestrutura Docker consegue aproveitar a capacidade de Service Discovery do mesmo, pelo que, no API Gateway no encaminhamento

para os microsserviços configura-se utilizando os nomes dos mesmos, como se pode observar na Listagem 3 no campo denominado por *host* (qp = Query Product).

```
services:
    - name: qp
    host: qp
    path: /qp
    port: 8080
    protocol: http
    // ...
    routes:
    - name: qp_read_product
        methods:
        - GET
        paths:
        - /api/products
        // ...
        plugins:
        - name: acl
          // ...
        - name: jwt
          config:
            // ...
```

Listagem 3: Exemplo de configuração do Kong para microsserviço QueryProduct (QP), com a rota de consulta de produtos.

5.8. Aplicação Cliente

Nesta secção, apresenta-se os motivos, objetivos e o processo de desenvolvimento da aplicação cliente.

Efetivamente, os objetivos e motivações para a implementação desta aplicação cliente deve-se a testar a aplicabilidade da API da arquitetura backend desenvolvida, nas funcionalidades típicas de uma aplicação e-commerce. Deste modo, permite identificar algumas funcionalidades da API backend em falta, que podem ser úteis neste tipo de sistemas e que não foram previamente implementadas.

No desenvolvimento desta aplicação cliente utilizou-se principalmente a framework VueJs[41], bem como as linguagens HTML, CSS e JavaScript. Na verdade, a utilização destas tecnologias e linguagens permitiu também aprofundar os conhecimentos das mesmas.

[41] https://vuejs.org/

Outra particularidade da implementação da aplicação cliente consiste em, como se observou nas arquiteturas de múltiplos microsserviços, existe um atributo denominado por links, que guarda os URIs das imagens dos produtos numa estrutura dinâmica. Por forma a facilitar este processo de obtenção das imagens na aplicação cliente por URIs, utilizou-se um repositório do GitHub[42] como repositório de imagens. No entanto, torna-se importante referir que, caso o caso de estudo não fosse já complexo, seria interessante implementar um microsserviço que tivesse esta responsabilidade.

Por forma a simplificar, apenas será apresentada a visão da homepage do consumidor, no entanto, focada principalmente na aplicabilidade da API da arquitetura backend, isto é, que microsserviços são utilizados nos diversos componentes desta visão. Todas as restantes visões são apresentadas no anexo II.

A Figura 49, consiste na visão da homepage da aplicação cliente quando utilizada pelo consumidor. Nesta visão consegue-se observar a lista de categorias à esquerda, que são dinâmicas nesta aplicação, isto é, se o gestor criar, atualizar ou remover uma categoria, isso terá efeito imediato na visão do consumidor. Os dados referentes às categorias que se veem neste componente desta visão provêm do microsserviço QCT.

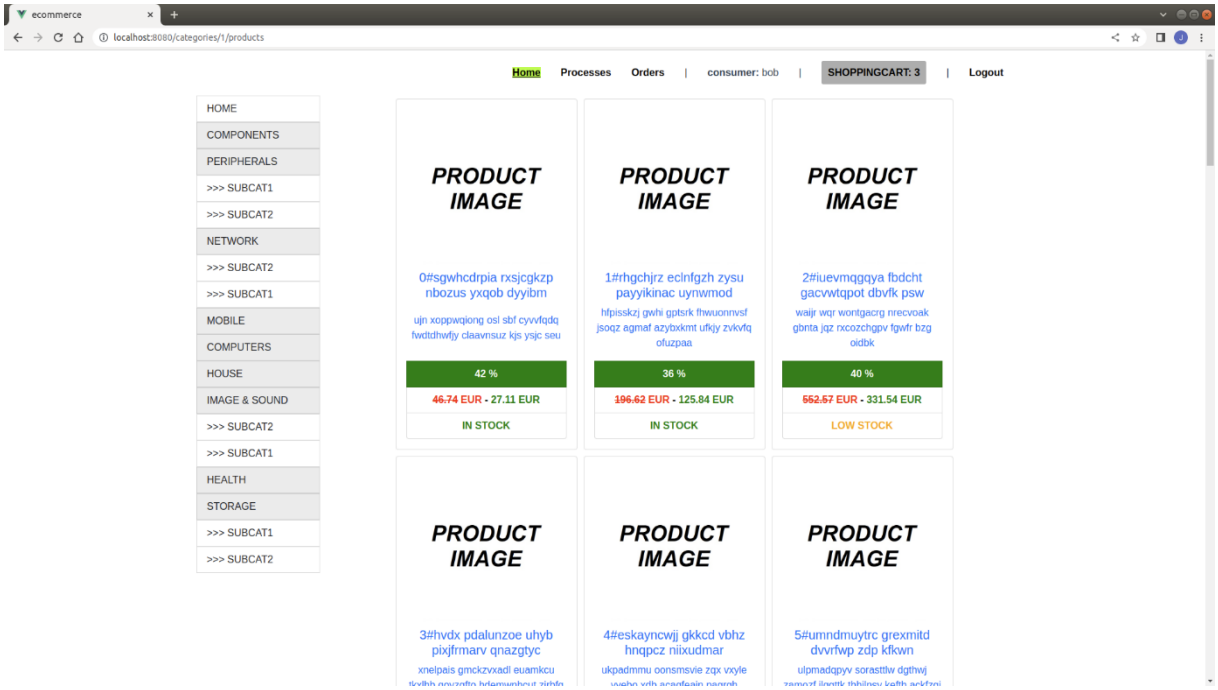


Figura 49: Homepage da aplicação Cliente, com a visão do consumidor autenticado.

[42] https://github.com/

Ainda referente à visão da homepage, tem-se a grelha dos produtos de uma determinada categoria com um conjunto de dados que provêm do microsserviço QCVP, uma vez que, se trata da visão do consumidor. Caso fosse a visão do gestor, os dados apresentados nesta visão da homepage provêm do microsserviço QCAP.

Ainda referente à visão da homepage, existe um botão denominado por shoppingcart, na barra superior à direita, que encaminha para a visão do carrinho de compras. No entanto, apesar de ser um botão encaminhador o mesmo contém um dado referente à quantidade de produtos no carrinho de compras, que provêm do microsserviço ShoppingCart (SC).

Por fim, ainda referente à visão da homepage, existe na barra superior o nome do consumidor e este dado provém do microsserviço Consumer, quando o mesmo autentica-se na aplicação e é devolvido o token.

Em suma, como se pode verificar em apenas uma visão da aplicação cliente, existe a utilização de múltiplos microsserviços para diferentes componentes. Ainda, consegue-se verificar a existência da aplicação dos padrões que permitem que estes componentes dependam apenas de um microsserviço.

Por exemplo, como se pode observar na Figura 49, na grelha dos produtos de uma categoria, os mesmos contém os dados sobre o stock dos produtos, tais como: "IN STOCK", "LOW STOCK", "NO STOCK". Como foi apresentado anteriormente, estes dados não são da responsabilidade do microsserviço QCVP, mas sim do Inventory. No entanto, com a aplicação de diversos padrões, como Saga e CQRS, entre outros, consegue-se injetar estes dados por forma a tornar a funcionalidade utilizada nesta grelha, isto é, a "consulta de produtos de uma categoria pelo consumidor", apenas dependente do microsserviço QCVP.

5.9. Conclusão

Em conclusão, este capítulo apresentou o desenho da arquitetura orientada a microsserviços para o caso de estudo e-commerce. O objetivo consistiu em testar a aplicabilidade dos conhecimentos adquiridos no estudo do estado da arte das arquiteturas orientadas a microsserviços relativamente aos padrões.

A arquitetura desenhada para o sistema e-commerce e respetivas tecnologias, pode ser consultada no diagrama de componentes da Figura 50. Os microsserviços do mesmo subdomínio estão coloridos na mesma cor.

Respetivamente à decomposição dos microsserviços, tal como previsto no estudo do estado da arte não foi uma tarefa algorítmica e fácil. Na verdade, envolveu a aplicação dos dois padrões de decomposição conjugados.

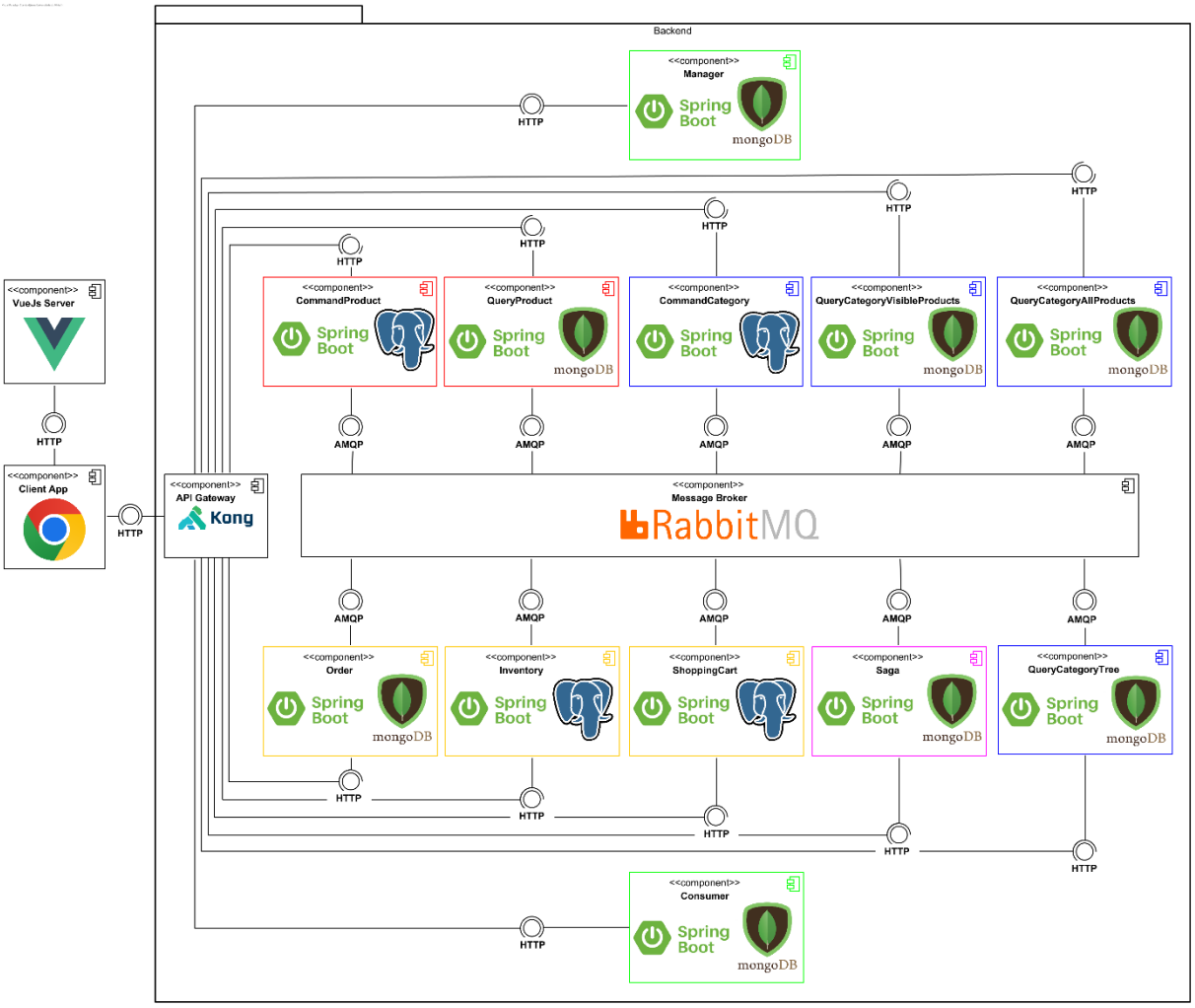


Figura 50: Arquitetura do caso de estudo e-commerce (microsserviços, aplicação cliente, conexões e respetivas tecnologias utilizadas).

Uma dificuldade desta decomposição foi a questão de o produto ser uma entidade central com relações com quase todas as entidades e como se mantinha estas relações numa arquitetura distribuída. Outro ponto refere-se à relação entre o produto e a categoria, onde nos modelos convencionais estão ambos fortemente ligados. Logo, a segregação destas entidades em dois microsserviços distintos foi pensada ao pormenor para determinação se era a decisão arquitetural mais estável.

Relativamente aos padrões implementados na arquitetura, os que tiveram um impacto positivo maior foram o Saga e o CQRS.

Relativamente ao Saga, este padrão foi desenhado inicialmente de forma que as configurações das sagas eram estáticas a nível de código, o que rapidamente se percebeu os problemas ou entraves que esta decisão teria no futuro. Na verdade, ao tornar estas configurações dinâmicas permite agora que este microsserviço seja mais reutilizável neste projeto, bem como em outros projetos. Ainda, permite que o mesmo possa ser escalável a nível de funcionalidade, isto é, pode-se distribuir diferentes sagas por diferentes instâncias e escalar independentemente as mesmas.

Relativamente ao CQRS, este padrão demonstrou ser muito útil para resolver problemas relacionados com a atribuição de responsabilidades, diferenças no número de pedidos de funcionalidades que estariam no mesmo microsserviço, seleção das estruturas e tecnologias de dados mais adequadas, entre outros. No entanto, após a implementação do mesmo, verifica-se que este contribui muito para reduzir o número de microsserviços envolvidos em funcionalidades aumentando a disponibilidade das mesmas.

Exemplo disso, tem-se uma das funcionalidades críticas nestes sistemas e-commerce, a consulta de produtos de uma categoria, que se conseguiu com a aplicação do padrão Saga e CQRS que apenas dependa de um microsserviço. Na verdade, esta funcionalidade retorna dados que são responsabilidade de pelos menos três subdomínios (três microsserviços), os produtos, categorias e inventário. No entanto, com a aplicação dos padrões Saga e CQRS, o microsserviço responsável por esta funcionalidade contém os dados suficientes.

6. Deployment do Caso de Estudo

Neste capítulo, apresenta-se a estrutura e processo de deployment do caso de estudo e-commerce desenhado e implementado. Os ficheiros de deployment e o código dos microsserviços pode ser encontrado no **repositório**[43] desta dissertação.

6.1. Estrutura do deployment

Nesta secção, apresenta-se toda estrutura necessária para efetuar o deployment deste caso de estudo e-commerce.

Efetivamente, a estrutura necessária consiste nos artefactos que são necessários, bem como a infraestrutura para efetuar o deployment do sistema desenvolvido.

Primeiramente, a nível de tecnologia, como já foi abordado anteriormente, utilizou-se o Docker[44]. Na verdade, em ambiente de desenvolvimento foi utilizado o Docker Compose[45], enquanto num ambiente de testes, como para efetuar os testes de carga que são abordados na secção 7.2, utilizou-se o Docker Swarm[46] na Google Cloud Platform[47] (GCP).

O primeiro artefacto necessário são os ficheiros WAR (Web Application Resource) que são gerados para cada lógica de microsserviço desenvolvido neste caso de estudo, os quais agregam todos os recursos necessários que constituem a lógica de um microsserviço.

```
FROM tomcat:9.0.50-jdk11-openjdk
COPY . /usr/local/tomcat/webapps/
CMD ["catalina.sh", "run"]
```

Listagem 4: Formato do ficheiro Dockerfile, para a lógica dos microsserviços deste caso de estudo utilizando o servidor web Tomcat.

De seguida, torna-se necessário criar as imagens para cada microsserviço utilizando um ficheiro denominado por Dockerfile. Tal como se pode observar na Listagem 4, este ficheiro permite importar outras imagens (FROM) como base da imagem do microsserviço (p.e. servidores web), sendo neste caso, a imagem do Tomcat. De seguida especifica-se a cópia (COPY) do ficheiro WAR de um microsserviço, que no caso, está na própria diretoria local ("."

[43] https://github.com/josepereira1/dissertation
[44] https://www.docker.com/
[45] https://docs.docker.com/compose/
[46] https://docs.docker.com/engine/swarm/
[47] https://cloud.google.com/

representa a própria diretoria) para a diretoria destino definida pela tecnologia Tomcat ("/usr/local/tomcat/webapps"). Por fim, especifica-se os comandos a executar (CMD), aquando da execução da imagem, sendo neste caso, a execução do servidor web do Tomcat.

Por conseguinte, configura-se a arquitetura do caso de estudo num ficheiro denominado por *docker-compose.yml*. Na verdade, neste ficheiro especifica-se os múltiplos serviços, sendo que, os serviços correspondem à lógica dos microsserviços ou respetivas bases de dados, bem como message brokers e API Gateways. Nesta especificação, define-se a localização das imagens, portas, networks, volumes e regras dos serviços. No anexo IV, pode ser consultado parte do ficheiro docker-compose.yml utilizado neste caso de estudo e-commerce.

Por fim, para facilitar o processo deployment e sendo uma metodologia recomendada, as imagens da lógica dos microsserviços foram adicionadas a um repositório, neste caso, o Docker Hub[48]. Assim para efetuar o deployment numa determinada infraestrutura, por exemplo qualquer projeto do GCP com Docker Swarm instalado, apenas necessita do ficheiro docker-compose.yml. Na verdade, a tecnologia Docker irá efetuar automaticamente o download das imagens da lógica dos microsserviços, como qualquer outra imagem de terceiros, como por exemplo de tecnologias de base de dados.

Ao publicar as imagens num repositório também facilita a atualização das mesmas e permite aos clientes utilizarem diferentes versões das mesmas.

6.2. Processo de deployment

Nesta secção, apresenta-se os passos para efetuar o deployment do caso de estudo em qualquer plataforma que suporte a tecnologia Docker.

Os artefactos necessários são o ficheiro docker-compose.yml, o ficheiro de configuração do API Gateway e um conjunto de ficheiros script (makefile e bash scripts) que fazem as configurações base do sistema, ao que se chamou de *setup*, tais como a criação dos processos saga, a categoria raiz, utilizadores, entre outros, tal como se pode observar na Listagem *5*.

[48] https://hub.docker.com/

```
deployment: create_volumes_dirs up
up:
    sudo docker-compose up -d --build --force --remove-orphans
create_volumes_dirs:
    sudo mkdir -p ${DOCKER_VOLUMES_DIR}/mongo
    sudo mkdir -p ${DOCKER_VOLUMES_DIR}/postgres
    // ...
setup:
    /bin/sh scripts/setup.sh
```

Listagem 5: Ficheiro script *makefile*, para efetuar o deployment com apenas um comando "make deployment".

6.2.1. Requisitos

1. Docker Compose ou Docker Swarm;
2. Makefile
3. 16+ GB de RAM;
4. Ficheiros de deployment (diretoria src/deployment, que inclui, docker-compose.yml, makefile, configurações do API Gateway, e bash scripts para setup e povoamento), que estão no repositório do GitHub (https://github.com/josepereira1/dissertation);

6.2.2. Processo de deployment

1. git clone https://github.com/josepereira1/dissertation.git
2. cd para diretoria onde se encontra o makefile (denominada src/deployment);
3. make deployment;
4. **aguardar alguns minutos (1 a 3 minutos)**, até que os microsserviços iniciem completamente (ou verificar os logs), esta espera torna-se necessária para **evitar timeouts no setup** e por este motivo, o setup não está incluído no comando make deployment;
5. make setup;

6.3. Conclusão

Em conclusão, desde o desenvolvimento inicial, a infraestrutura foi definida em containers utilizando a tecnologia Docker, o que revelou aumentar a produtividade e ser mais prático. Posteriormente, o sistema foi preparado para o deployment na cloud (GCP) utilizando máquinas em cluster através de Docker Swarm, o que não implicou muitas alterações à estrutura atual, uma vez que, já se utilizava o Docker.

7. Testes ao Caso de Estudo

Neste capítulo, apresenta-se o procedimento, objetivos e conclusões dos testes efetuados ao caso de estudo desenhado e implementado.

7.1. Testes Funcionais

Não foram definidos testes funcionais, como por exemplo utilizando a abordagem TDD (Test-driven Development) para a verificação das funcionalidades do sistema desenvolvido. No caso, apenas foram feitos testes momentâneos, enquanto se efetuava o desenvolvimento do sistema. Ainda, considera-se que a implementação da aplicação cliente tornou-se importante para testar funcionalmente o sistema.

Efetivamente, na aplicação cliente com as funcionalidades de criar, atualizar produto e efetuar a compra do carrinho de compras com sucesso, permitem testar o bom funcionamento da aplicação dos padrões Saga e CQRS, com a verificação da criação de diversos registos ao longo dos múltiplos microsserviços.

De seguida, apresentam-se algumas das funcionalidades, que permitem testar a funcionalidade de alguns dos padrões implementados no sistema.

- **Funcionalidade Criar e Atualizar Produto:**
    - Permite testar o funcionamento das sagas *create-product* e *update-product* em caso de sucesso (sem ocorrência de erros);
    - Permite testar o funcionamento das sagas *create-product* e *update-product* em caso de erro (com ocorrência de erros, e por isso o orquestrador da saga tem de inverter o processo e efetuar transações de compensação);
    - Permite testar o funcionamento dos CQRS dos microsserviços comando (CP e CC) participantes das sagas;
- **Funcionalidade Compra do Carrinho de compras:**
    - Permite testar o funcionamento da saga *create-order* em caso de sucesso (sem ocorrência de erros);

- Permite testar o funcionamento da saga *create-order* em caso de erro (com ocorrência de erros, e por isso o orquestrador da saga tem que inverter o processo e efetuar transações de compensação);

- **Funcionalidade Atualizar Stock:**
  - Permite testar o funcionamento de diversos CQRS;
    - CQRS entre o Inventory e CP e CC;
    - CQRS entre o CP e QP;
    - CQRS entre o CC e QCVP, QCAP;
- **Funcionalidade Login do Gestor e Consumidor:**
  - Permite testar o padrão Access-token nos microsserviços Manager e Consumer, bem como do API Gateway;
- **Funcionalidade de Consulta de Produtos de uma categoria:**
  - Permite verificar que esta funcionalidade depende apenas de um microsserviço, QCVP (no caso do consumidor) e QCAP (no caso do gestor), devido às decisões arquiteturais e implementação principalmente dos padrões Saga e CQRS.

7.2. Testes de Carga

Nesta secção, apresenta-se os testes de carga que tem como objetivo testar a eficácia da aplicabilidade de mecanismos de escalabilidade, em funcionalidades críticas do caso de estudo de e-commerce desenhado e implementado.

Efetivamente, pretende-se demonstrar que, a escalabilidade destas funcionalidades críticas, apenas dependem de um microsserviço, por consequência das decisões arquiteturais abordadas anteriormente, desde tratar-se de uma arquitetura orientada a microsserviços, dos princípios e padrões aplicados, entre outras decisões.

Do mesmo modo, com os testes de carga pretende-se provar a vantagem em ter segregado os microsserviços de leitura QCAP e QCVP.

7.2.1. Preparação e configuração dos testes de carga

Dado os objetivos anteriormente apresentados dos testes de carga, decidiu-se definir uma configuração de testes igual para todas as funcionalidades críticas.

Primeiramente, para se obter os resultados mais confiáveis fez-se deployment da arquitetura do caso de estudo e-commerce para a plataforma da Google Cloud Platform (GCP). Criou-se um projeto no GCP, com cinco máquinas virtuais, cada uma com 2 núcleos de processamento e 8 GigaBytes de memória RAM, sendo que, uma será utilizada para executar os testes de carga e as outras quatro são integradas num cluster Docker Swarm, onde será efetuado o deployment do caso de estudo.

Apesar de existirem quatro máquinas virtuais dedicadas ao caso de estudo, as mesmas não vão estar sempre em funcionamento, isso dependerá da configuração de testes de carga a ser utilizada no momento, tal como se pode verificar na Tabela 12.

Configuração	**Nº de Threads (Utilizadores)**	**Nº Requests por Thread**	**Amostras (#)**	**Nº de Máquinas Virtuais**	**Nº de Réplicas do microsserviço (Lógica e Base de dados)**
1	200	2000	400000	2	2
2	200	2000	400000	4	4
3	400	2000	800000	2	2
4	400	2000	800000	4	4

Tabela 12: Configurações de testes de carga utilizadas para cada funcionalidade e respetivo microsserviço.

Na verdade, deve-se ao facto de que o número de réplicas de um microsserviço na tecnologia Docker Swarm necessita de um número de máquinas virtuais igual ao número máximo de réplicas possíveis de um serviço. Deste modo, seria desvantajoso para as configurações de quatro réplicas que as de duas réplicas usassem quatro máquinas virtuais, pois teriam os serviços, como base de dados, API Gateways, entre outros, mais distribuídos e, por conseguinte, com mais recursos por microsserviço.

Outra nota importante consiste em que, nos testes de carga foram apenas ligados os microsserviços necessários para as funcionalidades a serem testadas para otimizar o financiamento disponível, bem como, não tem interesse ligar os outros microsserviços, dado que, não tem qualquer impacto nestas funcionalidades.

O motivo pelo qual os testes foram executados a partir de uma máquina virtual do próprio projeto GCP, deve-se ao facto de que, cada gibibyte enviado no sentido do GCP para a rede exterior (denominada por *Network Internet Egress*) tem um custo que com os testes programados, tornar-se-ia elevado para o financiamento disponível. Deste modo, a solução mais adequada e que não tem impacto negativo nos objetivos dos testes de carga, foi efetuar os mesmos a partir de uma máquina virtual do próprio projeto GCP.

A nível de tecnologia utilizada para efetuar os testes de carga, selecionou-se a ferramenta JMeter[49], dado que, não requer aprendizagem e gera automaticamente métricas e gráficos em formato de página web (HTML) que permite avaliar e concluir os resultados dos testes de forma mais confiável.

Além de se definir as configurações de testes de carga, decidiu-se que, para cada funcionalidade a ser testada, o respetivo teste deve ser repetido três vezes. Na verdade, o motivo consiste em que, os resultados do primeiro teste são normalmente piores devido ao chamado "aquecimento" dos testes de carga, pelo que, esse primeiro teste não representa resultados confiáveis. Por este motivo, os resultados dos primeiros testes de cada configuração (que corresponderiam nas tabelas à repetição 1), apesar de serem efetuados os mesmos não são apresentados.

Torna-se importante reforçar que, os valores obtidos nos resultados por vezes não são valores que seriam aceites a nível de qualidade de serviço. No entanto, o objetivo com estes testes de carga consiste em avaliar a evolução desses valores com a aplicação de mecanismos de escalabilidade.

Deste modo, para a análise dos testes de carga são utilizadas diversas métricas, uma vez que, obtém-se conclusões mais confiáveis dos resultados. Na verdade, o objetivo nestes testes para a mesma funcionalidade consiste em comparar os resultados entre as configurações 1 e 2 e as configurações 3 e 4.

Ainda, para cada teste apresenta-se o valor de APDEX (*Application Performance Index*) que consiste numa medida padrão calculada para avaliação e qualificação da performance da aplicação em relação aos tempos de resposta. O valor do APDEX foi calculado pelo JMeter e varia de 0 a 1, sendo que, quando mais próximo de 1 significa que à maior performance da aplicação ou microsserviço neste caso.

[49] https://jmeter.apache.org/

Por fim, para efetuar estes testes foi feito previamente um povoamento diversificado de produtos para obter resultados interessantes dos testes de carga. Deste modo, decidiu-se criar 100,000.00 produtos, 18 categorias, às quais foram associadas dos produtos existentes, 360 produtos, que permitem a existência de 15 páginas com 24 produtos por página numa categoria, o que é normal com base na consulta de alguns websites de e-commerce portugueses (p.e. worten, pcdiga, fnac, entre outros).

Com um povoamento diversificado e para efetuar testes de carga com pedidos dinâmicos, na ferramenta JMeter configurou-se as rotas com a utilização da funcionalidade *Random*. Na verdade, o objetivo consiste em variar os atributos das rotas nas diversas funcionalidades testadas, como se pode observar na Listagem 6, da funcionalidade de "consulta dos produtos de uma categoria" do microsserviço QCAP.

```
// ...
      <stringProp name="HTTPSampler.domain">10.128.0.4</stringProp>
      <stringProp name="HTTPSampler.port">8000</stringProp>
      <stringProp name="HTTPSampler.protocol">http</stringProp>
      <stringProp name="HTTPSampler.contentEncoding"></stringProp>
      <stringProp
      name="HTTPSampler.path">/api/categories/${__Random(1,18)}/products?page=
      ${__Random(1,15)}&amp;products_per_page=24&amp;visibility=all
      </stringProp>
// ...
```

Listagem 6: Parte da configuração de testes do JMeter, onde se verifica a utilização da funcionalidade *Random*, para gerar pedidos dinâmicos, na funcionalidade de consulta de produtos de uma categoria.

7.2.2. Testes de carga da consulta dos produtos de uma categoria do microsserviço QCAP

Nesta secção, são apresentados os resultados dos testes de carga executando as quatro configurações à funcionalidade de "consulta dos produtos de uma categoria" que depende apenas do microsserviço QCAP. Deste modo, as réplicas das configurações são aplicadas apenas a este microsserviço.

As quatro tabelas e os gráficos nas figuras presentes nesta secção, representam os resultados dos testes de carga respetivamente para cada configuração.

Efetivamente, em relação aos resultados dos testes de carga desta funcionalidade, conclui-se que, quer entre as configurações 1 e 2 e as configurações e 3 e 4, existem melhorias

a quase todos os níveis. Por exemplo, entre os resultados das configurações 3 e 4 verifica-se um aumento de mais de 40% no valor de referência de performance APDEX.

Em relação aos resultados em formato de gráfico presentes na Figura 51 e Figura 52, respetivamente da configuração 3 e 4, verifica-se da mesma forma que nos valores tabelados uma melhoria nos tempos de reposta.

Na verdade, analisando e comparando os respetivos gráficos verifica-se que houve uma redução do número de amostras que tem tempos acima dos 1500 ms (barra laranja) e as que tem tempos entre os 500 ms e 1500 ms. Deste modo, dos testes de carga da configuração 3 para 4 verifica-se um aumento 190,000.00 amostras qualificadas com tempos abaixo dos 500 ms (barra verde).

Ainda em relação aos gráficos, verifica-se a ocorrência de alguns pedidos com erro (barra vermelha) na configuração 3 em ambas as repetições provocado pela indisponibilidade do microsserviço QCAP (erro apresentado nos resultados: *Non HTTP response code: java.net.SocketException/Non HTTP response message: Connection reset*). Na verdade, estes pedidos com erro deixam de ocorrer na configuração 4, uma vez que, os mecanismos de escalabilidade aplicados eliminam os mesmos.

Repetição do teste	**APDEX**	**Média do tempo de resposta (ms)**	**Tempo de resposta Mínimo (ms)**	**Tempo de resposta Máximo (ms)**	**Mediana do tempo de resposta (ms)**	**Percentil de 90**	**Transações/s**
2	0.608	873.51	4	5478	384.00	1031.00	220.10
3	0.656	751.86	4	5821	342.00	898.00	256.19

Tabela 13: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCAP, para a configuração 1.

Repetição do teste	**APDEX**	**Média do tempo de resposta (ms)**	**Tempo de resposta Mínimo (ms)**	**Tempo de resposta Máximo (ms)**	**Mediana do tempo de resposta (ms)**	**Percentil de 90**	**Transações/s**
2	0.790	563.85	5	9480	36.00	741.00	335.18
3	0.787	564.61	5	8505	32.00	908.00	335.84

Tabela 14: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCAP, para configuração 2.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.516	1773.43	4	63683	409.00	1173.00	217.37
3	0.514	1711.54	5	10876	369.00	1479.00	224.96

Tabela 15: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCAP, para configuração 3.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.741	1376.04	5	19655	68.00	1072.90	274.87
3	0.740	1297.61	5	19998	51.00	1216.00	291.32

Tabela 16: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCAP, para configuração 4.

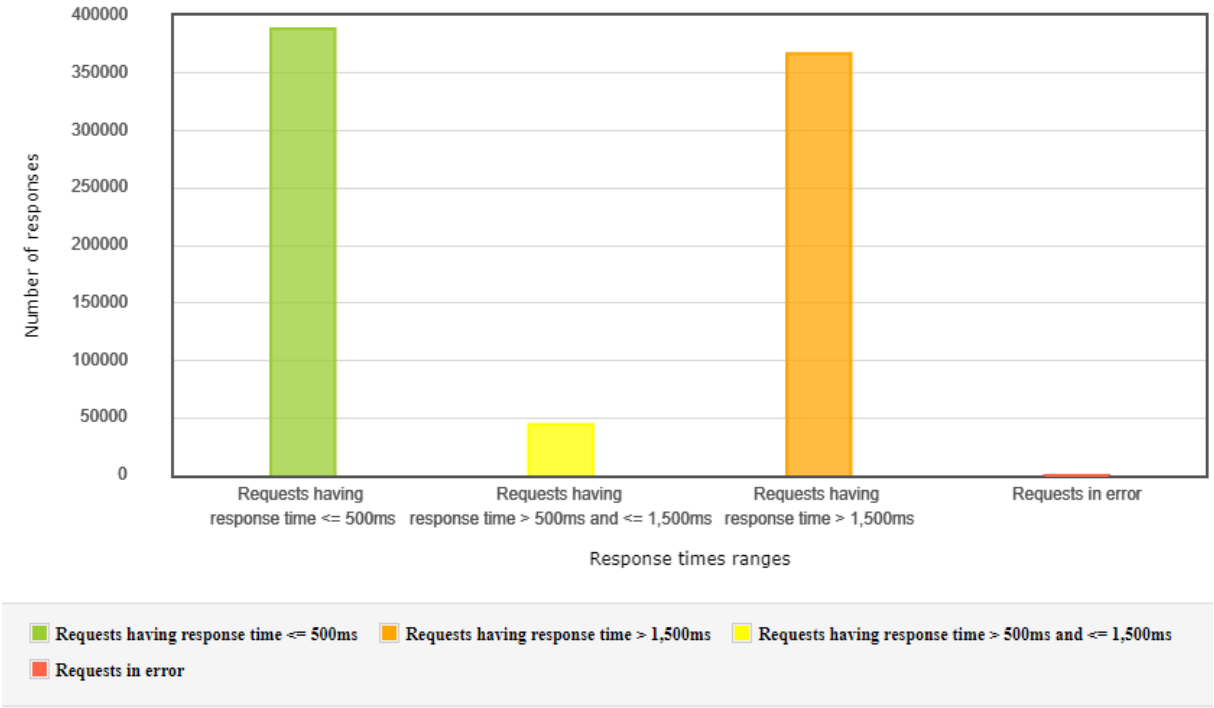


Figura 51: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 3, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCAP.

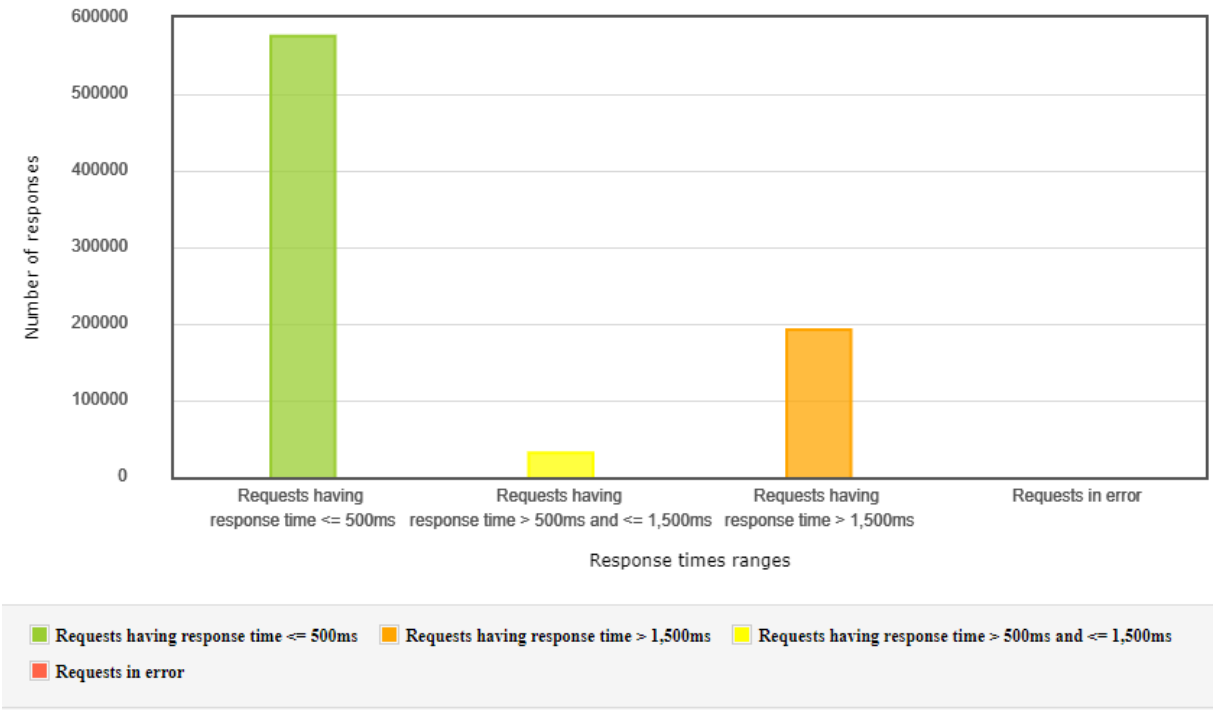


Figura 52: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 4, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCAP.

7.2.3. Testes de carga da consulta dos produtos de uma categoria do microsserviço QCVP

Nesta secção, são apresentados os resultados dos testes de carga executando as quatro configurações à funcionalidade de "consulta dos produtos de uma categoria" que depende apenas do microsserviço QCVP. Deste modo, as réplicas das configurações são aplicadas apenas a este microsserviço.

As quatro tabelas e os gráficos nas figuras presentes nesta secção representam os resultados dos testes de carga respetivamente para cada configuração.

Efetivamente, em relação aos resultados dos testes de carga desta funcionalidade, conclui-se que, quer entre as configurações 1 e 2 e as configurações e 3 e 4 existem melhorias a quase todos os níveis. No entanto, entre as configurações 1 e 2 a nível do valor de APDEX não houve uma melhoria muito notória em comparação aos testes da funcionalidade anterior, devido ao facto de que, a carga exercida não foi suficiente para gerar grandes dificuldades ao microsserviço QCVP.

Em relação aos resultados em formato de gráfico, presentes na Figura 53 e Figura 54, respetivamente da configuração 3 e 4, verifica-se da mesma forma que nos valores tabelados uma melhoria nos tempos de reposta.

Na verdade, analisando e comparando os respetivos gráficos, verifica-se que houve uma redução do número de amostras que têm tempos entre os 500 ms e 1500 ms. Dessa redução, maior parte foi para as amostras abaixo dos 500 ms (barra verde), mais de 175,000.00. Por outro lado, houve um pequeno aumento das amostras acima dos 1500 ms, por consequência, do teste anterior da configuração 3 ter poucas amostras nesta categoria, tornando difícil a obtenção de melhorias.

Repetição do teste	**APDEX**	**Média do tempo de resposta (ms)**	**Tempo de resposta Mínimo (ms)**	**Tempo de resposta Máximo (ms)**	**Mediana do tempo de resposta (ms)**	**Percentil de 90**	**Transações/s**
2	0.916	222.24	3	1497	121.00	329.00	850.93
3	0.913	224.71	3	2037	118.00	348.00	842.88

Tabela 17: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCVP, para configuração 1.

Repetição do teste	**APDEX**	**Média do tempo de resposta (ms)**	**Tempo de resposta Mínimo (ms)**	**Tempo de resposta Máximo (ms)**	**Mediana do tempo de resposta (ms)**	**Percentil de 90**	**Transações/s**
2	0.918	180.90	3	1734	18.00	303.00	1027.64
3	0.947	176.54	4	1415	42.00	299.00	1052.83

Tabela 18: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCVP, para configuração 2.

Repetição do teste	**APDEX**	**Média do tempo de resposta (ms)**	**Tempo de resposta Mínimo (ms)**	**Tempo de resposta Máximo (ms)**	**Mediana do tempo de resposta (ms)**	**Percentil de 90**	**Transações/s**
2	0.764	475.36	3	2711	194.00	536.00	812.53
3	0.762	476.78	4	3064	168.00	537.00	811.33

Tabela 19: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCVP, para configuração 3.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.856	358.47	3	2739	70.00	370.39	1054.61
3	0.859	342.32	3	2734	70.00	449.00	1115.39

Tabela 20: Resultados dos testes de carga, da funcionalidade consulta dos produtos de uma categoria, do microsserviço QCVP, para configuração 4.

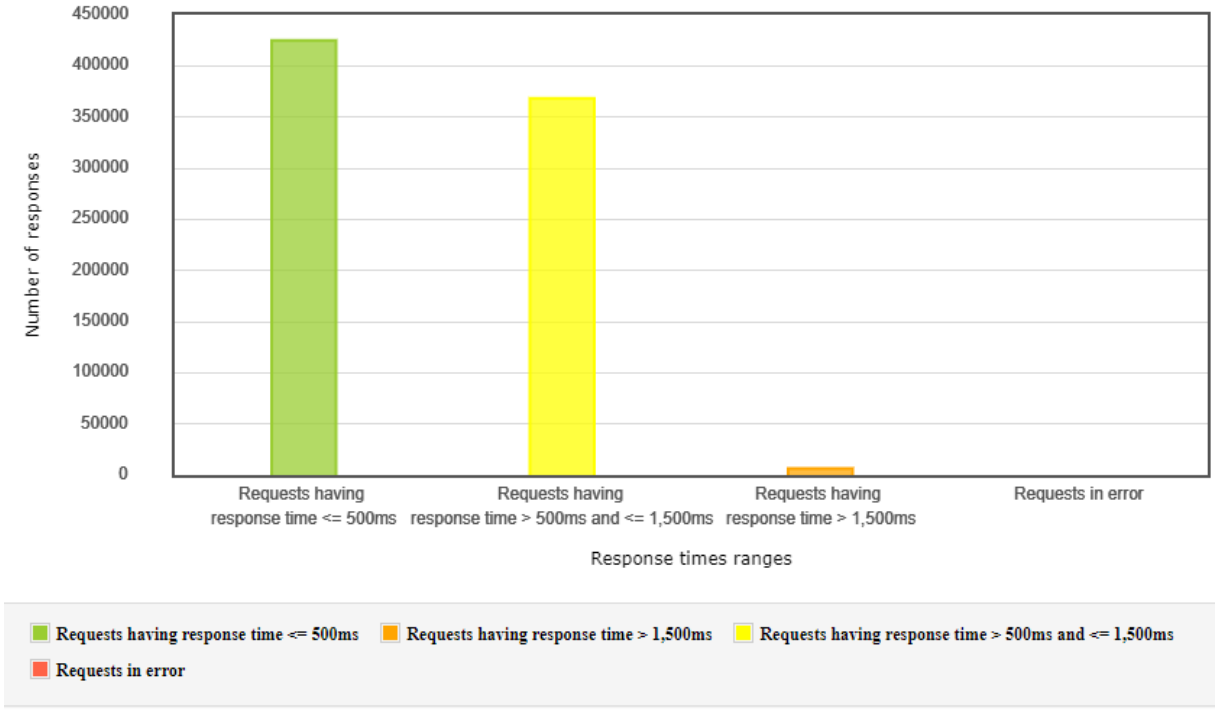


Figura 53: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 3, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCVP.

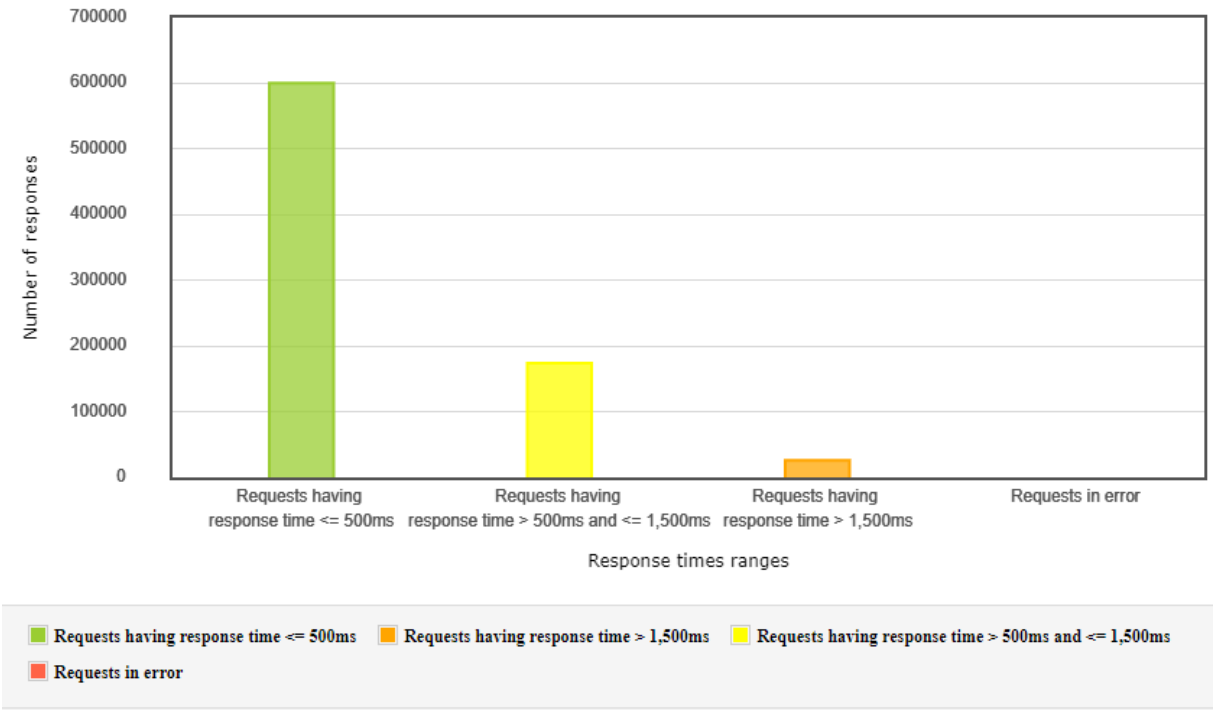


Figura 54: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 4, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCVP.

7.2.4. Justificação da segregação do QCAP e QCVP

Relativamente ao que foi abordado no início da secção 7.2, as comparações dos resultados das funcionalidades de "consulta dos produtos de uma categoria" entre o QCAP e QCVP justificam a segregação destes dois microsserviços. Na verdade, verifica-se que o microsserviço QCVP e respetiva funcionalidade tem resultados muito melhores que a mesma funcionalidade no microsserviço QCAP.

Isto deve-se ao facto de que, tal como abordado na secção 5.4, esta segregação permite ao microsserviço QCVP evitar filtragens e autorizações dos pedidos dos consumidores, uma vez que, este microsserviço só contém os produtos visíveis para estes. Deste modo, comparando os resultados da configuração 4 entre o QCAP e QCVP, isto é, a Tabela 16 e Tabela 20, o valor do APDEX respetivamente de 0.74 para 0.86 representa um aumento de aproximadamente 16%.

Por fim, comparando os gráficos da Figura 55 e Figura 56, que consistem nos percentis dos tempos de resposta, verifica-se que, o microsserviço QCVP obtém sempre tempos de resposta muito mais baixos para os diversos percentis. Por exemplo, enquanto o QCAP para o

percentil de 70 tem um tempo de resposta de 351 ms, por outro lado, o microsserviço QCVP tem um tempo de resposta de 151 ms.

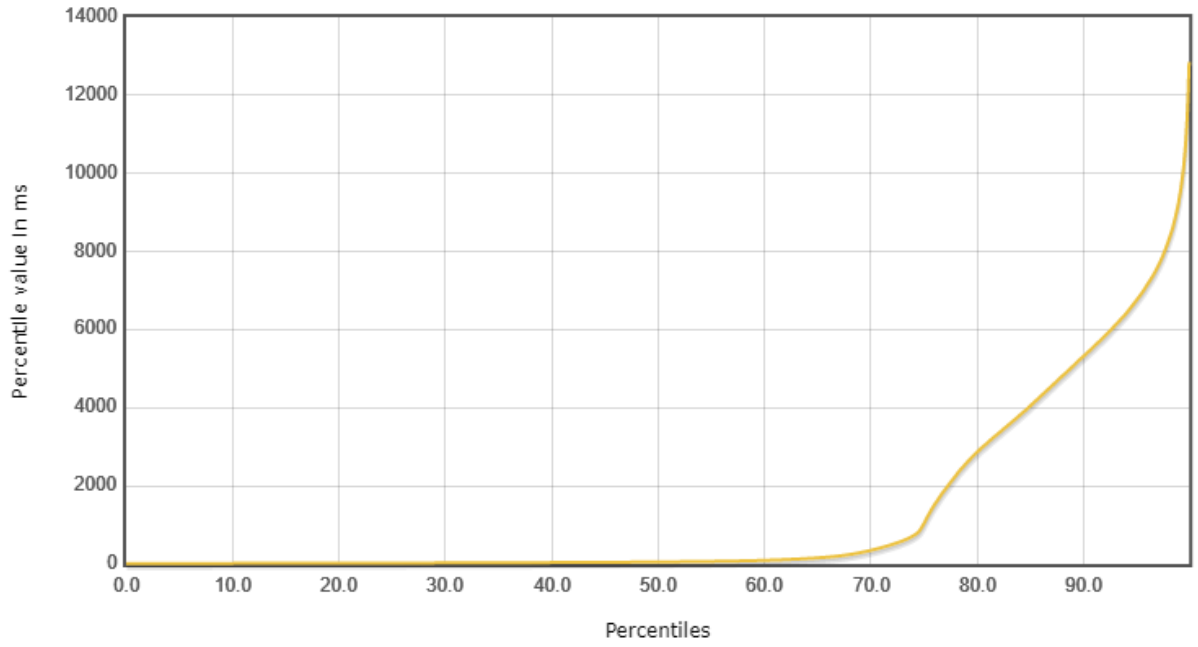


Figura 55: Gráfico dos percentis dos tempos de resposta, da configuração 4, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCAP.

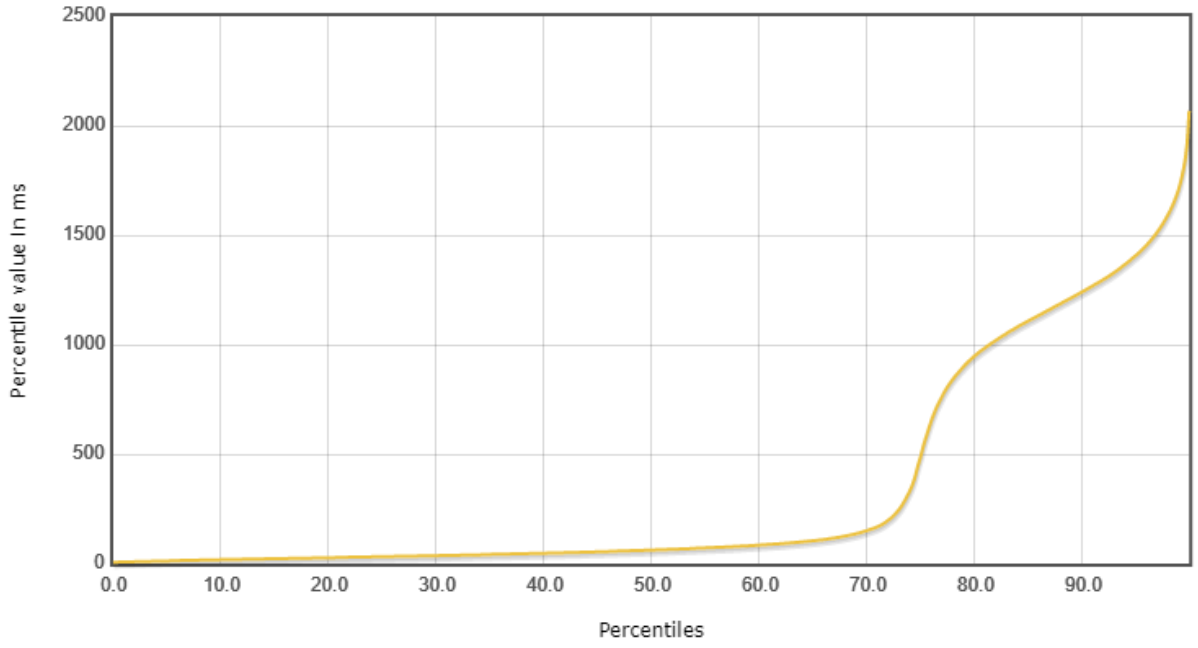


Figura 56: Gráfico dos percentis dos tempos de resposta, da configuração 4, repetição 3, da funcionalidade consulta dos produtos de uma categoria do microsserviço QCVP.

7.2.5. Testes de carga da consulta de um produto do microsserviço QP

Nesta secção, são apresentados os resultados dos testes de carga executando as quatro configurações à funcionalidade de "consulta dos detalhes de um produto", que depende apenas do microsserviço QP. Deste modo, as réplicas das configurações são aplicadas apenas a este microsserviço.

As quatro tabelas e os gráficos nas figuras presentes nesta secção representam os resultados dos testes de carga respetivamente para cada configuração.

Efetivamente, em relação aos resultados dos testes de carga desta funcionalidade, conclui-se que, quer entre as configurações 1 e 2 e as configurações e 3 e 4 existem melhorias a quase todos os níveis. No entanto, entre as configurações 1 e 2 a nível do valor de APDEX não houve uma melhoria muito notória em comparação aos testes de outras funcionalidades, devido ao facto de que, a carga exercida não foi suficiente para gerar grandes dificuldades ao microsserviço QP.

Em relação aos resultados em formato de gráfico presentes na Figura 57 e Figura 58, respetivamente da configuração 3 e 4, verifica-se da mesma forma que nos valores tabelados uma melhoria nos tempos de reposta.

Na verdade, analisando e comparando os respetivos gráficos, verifica-se que houve uma redução do número de amostras que têm tempos entre os 500 ms e 1500 ms. Dessa redução, maior parte foi para as amostras abaixo dos 500 ms (barra verde), mais de 149,000.00. Por outro lado, houve um pequeno aumento das amostras acima dos 1500 ms, por consequência, do teste anterior da configuração 3 ter poucas amostras nesta categoria tornando difícil a obtenção de melhorias.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.979	161.62	3	1212	89.00	265.00	1200.44
3	0.968	173.52	3	1273	100.00	297.00	1118.96

Tabela 21: Resultados dos testes de carga, da funcionalidade consulta dos detalhes de um produto de uma categoria, do microsserviço QP, para configuração 1.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.998	119.27	2	975	51.00	150.00	1615.26
3	0.999	118.14	2	918	54.00	225.00	1649.55

Tabela 22: Resultados dos testes de carga, da funcionalidade consulta dos detalhes de um produto, do microsserviço QP, para configuração 2.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.797	362.52	2	2295	192.00	369.00	1078.26
3	0.816	345.91	2	2542	98.00	459.00	1127.96

Tabela 23: Resultados dos testes de carga, da funcionalidade consulta dos detalhes de um produto, do microsserviço QP, para configuração 3.

Repetição do teste	APDEX	Média do tempo de resposta (ms)	Tempo de resposta Mínimo (ms)	Tempo de resposta Máximo (ms)	Mediana do tempo de resposta (ms)	Percentil de 90	Transações/s
2	0.903	285.15	2	3189	23.00	112.00	1276.77
3	0.905	300.05	2	3817	25.00	150.00	1221.34

Tabela 24: Resultados dos testes de carga, da funcionalidade consulta dos detalhes de um produto, do microsserviço QP, para configuração 4.

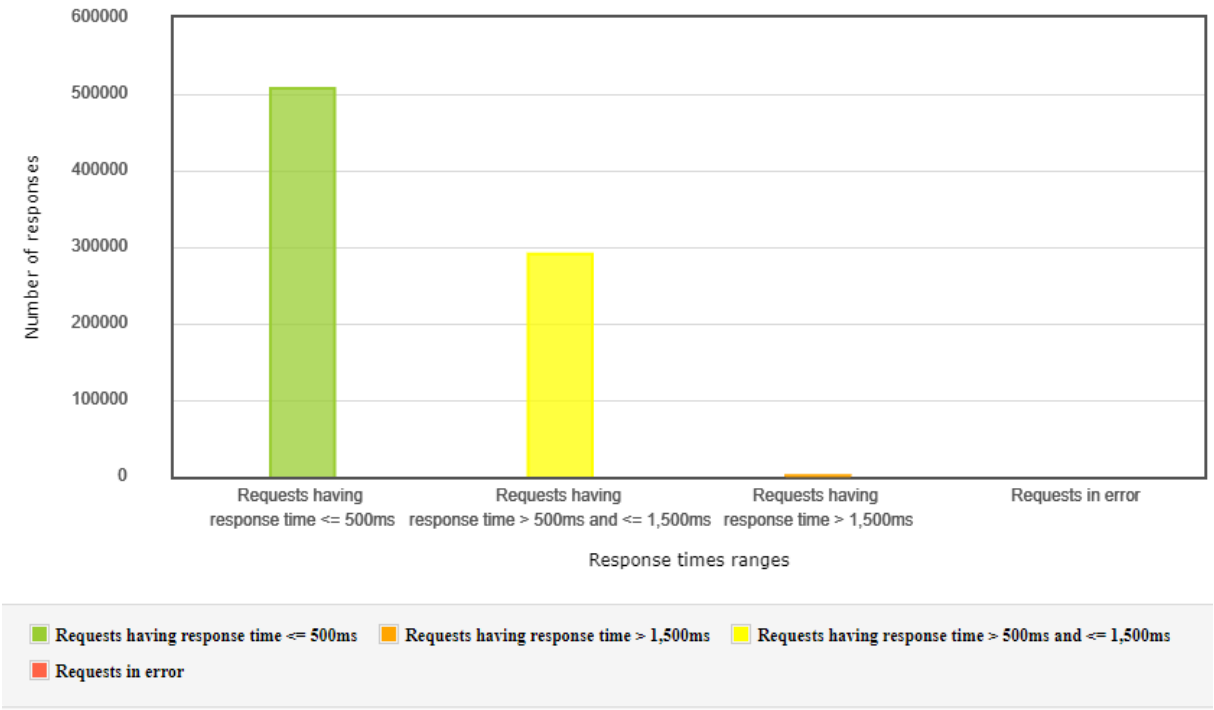


Figura 57: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 3, repetição 3, da funcionalidade consulta dos detalhes de um produto do microsserviço QP.

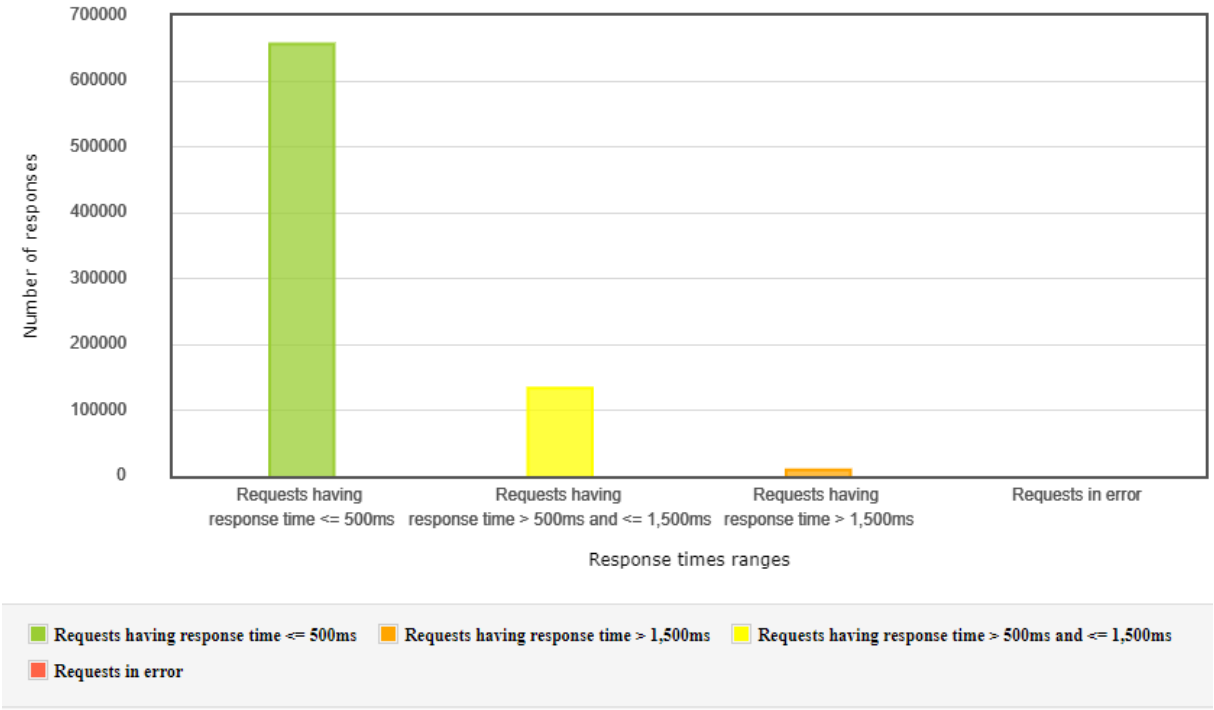


Figura 58: Gráfico com a qualificação dos tempos de resposta das amostras em intervalos de tempo, da configuração 4, repetição 3, da funcionalidade consulta dos detalhes de um produto do microsserviço QP.

7.3. Conclusão

Em suma, com os testes de carga realizados, verifica-se que em primeira instância existem melhorias dos resultados com a aplicação de mecanismos escalabilidade. Do mesmo modo, demonstra-se a importância das decisões arquiteturais, que torna estas funcionalidades consideradas críticas dependentes apenas de um microsserviço.

Relativamente à segregação dos microsserviços de leitura das categorias respetivamente o QCAP e QCVP, demonstrou-se a vantagem da sua segregação com notórias melhorias nos resultados do microsserviço QCVP.

8. Conclusão

Neste capítulo, são apresentadas as conclusões finais sobre o trabalho realizado nesta dissertação face os objetivos definidos na secção 1.2. De seguida, são apresentadas propostas para trabalho futuro relativamente ao resultado desta dissertação.

Efetivamente, ao longo deste documento foram realizadas diversas conclusões dos conteúdos apresentados. No entanto, torna-se importante reunir e centralizar os resultados e conclusões finais de todo o trabalho realizado.

Começando pelo primeiro objetivo, relativamente ao estudo do estado da arte das arquiteturas orientadas a microsserviços, os seus princípios e padrões mais relevantes. Em relação ao estudo do que são estas arquiteturas, esse conhecimento já se dá como adquirido, pelo que, não foram elaboradas grandes pesquisas sobre os mesmos. Por outro lado, não existe uma lista oficial bem definida dos padrões destas arquiteturas, pelo que, inicialmente a pesquisa pelos mesmos foi bastante desafiante.

No entanto, com base na lista proposta por Chris Richardson, no seu livro *Microservices Patterns* [1], foi possível selecionar um conjunto dos mesmos que se achou importante a sua análise. Esta seleção teve em conta aqueles que são necessários e base para maior parte dos sistemas. Abrangendo nestas arquiteturas a decomposição, manutenção dos dados, comunicação e segurança. De seguida, fez-se uma pesquisa comparativa dos mesmos analisados e descritos por outros autores para se perceber a sua relevância.

Relativamente aos padrões estudados, a categoria que foi mais desafiante foi a manutenção de dados, sendo muito importante, uma vez que, um dos pontos essenciais em qualquer sistema são os dados. Na verdade, a sua dificuldade está relacionada com a natureza distribuída de uma arquitetura orientada a microsserviços que aumenta a complexidade para se garantir algumas propriedades dos dados.

Assim relativamente ao estudo do estado da arte, o mesmo foi bastante importante para aumentar o conhecimento das arquiteturas orientadas a microsserviços. Mas, principalmente tornou-se importante para conhecer alguns dos princípios e padrões existentes, bem como a sua necessidade e importância face a resolver alguns dos problemas e desafios que existem na implementação deste tipo de arquiteturas.

O segundo objetivo refere-se à análise do caso de estudo que envolveu a observação do domínio e-commerce, análise e levantamento dos requisitos.

A seleção deste caso de estudo tornou-se desafiante, na medida em que, tem de ter a complexidade suficiente para permitir aplicar maior parte dos princípios e padrões estudados. Deste modo, foram estudados alguns casos de estudo, no entanto, aquele que pareceu mais completo foi um sistema e-commerce, uma vez que, tem complexidade suficiente. Ainda, verifica-se que, este caso de estudo é muitas vezes utilizado como exemplo para explicar e apresentar os princípios e padrões dos microsserviços.

Efetivamente, a conceção do caso de estudo, como referido no capítulo 3, seguiu o método *Unified Process*. Deste modo, realizou-se a observação do domínio e-commerce para se conhecer o mesmo, realizando-se como resultado o modelo de domínio, seguido da análise e levantamento dos requisitos. Em relação à análise do domínio, a mesma foi elaborada com recurso à consulta de aplicações web de e-commerce comuns em Portugal, tais como: PCDiga, PcComponentes, Worten, Fnac, Wook, Amazon, entre outros. Ainda foram analisados alguns recibos de faturação provenientes de algumas das lojas referidas anteriormente. No entanto, o processo de análise passou muito pela visão do consumidor, uma vez que, não existe fácil acesso às funcionalidades dos gestores. Deste modo, a visão do domínio do gestor foi prevista com base nas necessidades do consumidor.

Em suma, o processo de análise torna-se implícito em qualquer sistema, no entanto, verificou-se que, numa arquitetura orientada a microsserviços torna-se crucial, uma vez que, tal como foi abordado nos padrões da categoria de decomposição secção 2.2.1, o conhecimento do domínio é o que permite uma decomposição estável dos microsserviços.

O terceiro objetivo refere-se ao desenho da arquitetura orientada a microsserviços para o caso de estudo e-commerce. Na verdade, o objetivo consiste em testar a aplicabilidade dos conhecimentos adquiridos no estudo do estado da arte das arquiteturas orientadas a microsserviços relativamente aos padrões.

Respetivamente à decomposição dos microsserviços, tal como previsto no estudo do estado da arte, não foi uma tarefa algorítmica e fácil. Na verdade, envolveu a aplicação dos dois padrões de decomposição conjugados.

Uma dificuldade desta decomposição foi a questão de o produto ser uma entidade central com relações com quase todas as entidades e como se mantinha estas relações numa arquitetura distribuída. Outro ponto refere-se à relação entre o produto e a categoria, onde nos modelos convencionais estão ambos fortemente ligados. Logo, a segregação destas

entidades em dois microsserviços distintos foi pensada ao pormenor para determinação se era a decisão arquitetural mais estável.

Relativamente aos padrões implementados na arquitetura, os que tiveram um impacto positivo maior foram o Saga e o CQRS.

Relativamente ao Saga, este padrão foi desenhado inicialmente de forma que as configurações das sagas eram estáticas a nível de código, o que rapidamente se percebeu os problemas ou entraves que esta decisão teria no futuro. Na verdade, ao tornar estas configurações dinâmicas permite agora que este microsserviço seja mais reutilizável neste projeto, bem como em outros projetos. Ainda, permite que o mesmo possa ser escalável a nível de funcionalidade, isto é, pode-se distribuir diferentes sagas por diferentes instâncias e escalar independentemente as mesmas.

Relativamente ao CQRS, este padrão demonstrou ser muito útil para resolver problemas relacionados com a atribuição de responsabilidades, diferenças no número de pedidos de funcionalidades que estariam no mesmo microsserviço, seleção das estruturas e tecnologias de dados mais adequadas, entre outros. No entanto, após a implementação do mesmo, verifica-se que este contribui muito para reduzir o número de microsserviços envolvidos em funcionalidades aumentando a disponibilidade das mesmas.

Exemplo disso, tem-se uma das funcionalidades críticas nestes sistemas e-commerce, a consulta de produtos de uma categoria, que se conseguiu com a aplicação do padrão Saga e CQRS que apenas dependa de um microsserviço. Na verdade, esta funcionalidade retorna dados que são responsabilidade de pelos menos três subdomínios (três microsserviços), os produtos, categorias e inventário. No entanto, com a aplicação dos padrões Saga e CQRS o microsserviço responsável por esta funcionalidade contém os dados suficientes.

Outros resultados e conclusões relativamente a estes padrões e outros serão abordados aquando do objetivo referente à implementação do caso de estudo e-commerce.

Ainda relativamente ao terceiro objetivo, isto é, ao desenho da arquitetura, conclui-se que a aplicação dos padrões arquiteturais Inversion of Control e arquitetura hexagonal foram importantes, uma vez que, contribuem para o aumento de produtividade.

Na verdade, em relação ao Inversion of Control, mais propriamente o Dependecy Injection, o mesmo abstrai do programador a complexidade desnecessária de criação, invocação e integração de serviços internos ou externos.

Em relação à arquitetura hexagonal, a mesma contribui para a produtividade pois estrutura o código de forma que, encontrar uma determinada secção de código seja simples. Do mesmo modo, faz uma abstração da complexidade dos adaptadores em relação à camada da lógica de negócio, o que simplifica a implementação de funcionalidades.

Outro ponto importante relativamente ao desenho da arquitetura consiste na comunicação entre microsserviços. Por um lado, a comunicação entre microsserviços é algo indispensável, tal como se verificou no estudo do estado da arte, bem como dada a natureza distribuída torna-se um desafio. Deste modo, o estudo e desenho da mesma torna-se muito importante. A seleção da comunicação por mensagens dada a sua simplicidade, bem como a definição prévia da estrutura das mensagens foram importantes para o sucesso da comunicação entre microsserviços. Tornou-se num processo mais simplificado com a implementação de uma framework própria para manipular as mensagens de igual forma em todos os microsserviços.

O quarto objetivo, refere-se à implementação da arquitetura orientada a microsserviços desenhada para o caso de estudo e-commerce.

Relativamente a este objetivo, conclui-se que, o mesmo foi muito importante para consolidar o conhecimento dos padrões, uma vez que, o conhecimento na teoria é totalmente diferente do conhecimento na prática. Por exemplo, quando se aponta como desvantagem a complexidade de um padrão, por vezes não se dá a devida importância, mas quando se implementa na prática consegue-se perceber melhor.

Efetivamente, de uma forma geral a implementação na prática realçou alguns dos problemas dos padrões relacionados com a complexidade, a latência da rede, devido a ser uma arquitetura distribuída, entre outros problemas.

Em relação à implementação do padrão Saga na prática, verificou-se que o mesmo é importante para se garantir as propriedades ACID das transações ao longo de múltiplos microsserviços. No entanto, revelou também que se torna importante analisar e balancear entre as vantagens, a complexidade e performance dessas transações. No caso de estudo, maior parte das sagas foram aplicadas à gestão dos produtos, algo que não requer alta performance nestas funcionalidades. Por outro lado, a encomenda de produtos é algo que é altamente requerido, sendo que isso não é um problema para este sistema pois o microsserviço Saga pode ser escalável. A nível de performance, normalmente os

consumidores após efetuarem o pedido de encomenda não ficam à espera do resultado imediato, pelo que, a natureza assíncrona da Saga adequa-se.

Assim, o ponto importante a refletir em relação à implementação da Saga é que nem sempre se torna a solução ideal para qualquer caso de uso, tem de se avaliar os pontos negativos da mesma e em que medida estes não prejudicam a funcionalidade.

Do mesmo modo, a implementação do CQRS, tal como já tinha sido referido no estudo do estado da arte, o mesmo tem alguns problemas relativos à latência da rede que pode provocar inconsistências de dados entre outros problemas. Do mesmo modo, só após a implementação do CQRS se verificou a necessidade do versionamento das mensagens, como foi abordado na secção 4.2.1, para solucionar um problema de inconsistência.

Por outro lado, relativamente à implementação do caso de estudo foi utilizada a framework Spring Boot que aumentou a produtividade do desenvolvimento dos 12 microsserviços, que de outra forma seria moroso. Do mesmo modo, a utilização do *Lombok*, para geração de código das classes (p.e. gets, sets, etc.), bem como de outras frameworks e tecnologias, contribuíram para a produtividade do desenvolvimento do sistema.

O quinto objetivo refere-se à implementação da aplicação cliente, que consiste numa aplicação browser para o caso de estudo e-commerce.

Em suma, tal como já foi referido no secção 5.8, esta aplicação cliente permitiu testar a aplicabilidade do backend desenvolvido. Na verdade, isto significa testar o funcionamento das funcionalidades implementadas, bem como verificar se a lista de requisitos propostos é cumprida. Ainda permitiu aprofundar o conhecimento das tecnologias de frontend utilizadas na sua implementação, que são: VueJs, HTML, CSS e Javascript.

O sexto objetivo refere-se à preparação do sistema implementado para instalação numa infraestrutura. Desde o desenvolvimento inicial, a infraestrutura foi definida em containers utilizando a tecnologia Docker o que revelou aumentar a produtividade e ser mais prático. Posteriormente, o sistema foi preparado para o deployment na cloud utilizando máquinas em cluster através de Docker Swarm, o que não implicou muitas alterações à estrutura atual, uma vez que, já se utilizava o Docker.

O sétimo objetivo refere-se aos testes de carga ao sistema do caso de estudo para testar a aplicação de mecanismos de escalabilidade.

Em suma, relativamente a este objetivo verificou-se de uma forma geral que, em todas as funcionalidades testadas, a aplicação de mecanismos de escalabilidade tem uma melhoria nos resultados, algo que já era de esperar neste tipo de arquiteturas.

Por outro lado, estes testes de carga permitiram demonstrar a vantagem em ter segregado os microsserviços de leitura das categorias QCAP e QCVP.

Os testes de carga permitiram ainda testar o sistema numa infraestrutura cloud para a realização dos mesmos e assim testar a aplicabilidade da mesma.

Em suma, em conclusão final de toda a dissertação, a execução deste projeto permitiu aprofundar os conhecimentos das arquiteturas orientadas a microsserviços, uma área muito interessante, e que se pretende aprofundar muito mais. Outro ponto importante consiste em que, este estudo permitiu perceber que este tipo de arquitetura é uma alternativa aos sistemas atuais, no entanto, nunca assumir a mesma como uma abordagem superior, uma vez que, as arquiteturas orientadas a microsserviços tem as suas vantagens e desvantagens, pelo que, não é adequada a todos os casos de uso [12].

Do mesmo modo, relativamente aos padrões, os mesmos demonstram que o seu conhecimento se torna muito importante. No entanto, também é imprescindível ter espírito critico e saber quando se deve aplicar ou não os mesmos e analisar as diversas alternativas.

Ainda, os princípios das arquiteturas orientadas a microsserviços devem ser respeitados, tais como as regras da segregação das responsabilidades, bem como dos limites bem definidos dos microsserviços.

Os artefactos resultantes desta dissertação a nível de modelação, *deployment* e código podem ser consultados e testados em https://github.com/josepereira1/dissertation.

8.1. Trabalho Futuro

Nesta secção, abordam-se propostas de trabalho futuro para dar continuidade ao trabalho desenvolvido nesta dissertação.

Relativamente ao estudo do estado da arte, existem outras categorias e respetivos padrões a explorar bem como princípios.

Do mesmo modo, em relação ao desenho da arquitetura efetuado e respetiva implementação existem melhorias a fazer.

Primeiramente, implementar em certos pontos da arquitetura uma comunicação por eventos e comparar os resultados de produtividade, compreensão do processo e performance, em relação à comunicação por mensagens.

Do mesmo modo, seria interessante implementar uma das sagas desenvolvidas utilizando uma coordenação em coreografia e comparar os resultados da mesma com a respetiva saga utilizando a coordenação em orquestração.

Ainda relativamente ao microsserviço Saga, seria interessante integrar ou dar a possibilidade de integrar um serviço de notificações para automaticamente enviar uma notificação com o resultado da execução do processo saga.

Podem existir outros subdomínios ou microsserviços candidatos à aplicação do padrão CQRS, como por exemplo Order, ShoppingCart, entre outros.

Em relação a certos microsserviços, pode ser interessante testar outras tecnologias, como por exemplo, nos microsserviços de leitura a utilização de uma *store* em memória para aumentar a performance e reduzir os tempos de resposta.

Outra proposta interessante, seria implementar a arquitetura desenhada utilizando alguns serviços da cloud. Como por exemplo a comunicação por mensagens, API Gateway, base de dados, containers, entre outros. Estes serviços tem a característica de serem mantidos por estes fornecedores de serviços (p.e. AWS, GCP, Azure, entre outros). Deste modo, retira a responsabilidade de tarefas não diferenciadas das empresas concorrentes garantindo disponibilidade, elasticidade, entre outros atributos de qualidade, permitindo ao programador forcar-se nas funcionalidades de negócio.

8.2. Reflexão Pessoal

Em jeito de reflexão, a realização deste projeto de dissertação permitiu obter mais conhecimento numa área de interesse ligada à especialização dos perfis de mestrado, Engenharia de Aplicações e Engenharia de Sistemas de Software. Este estudo dos princípios e padrões das arquiteturas orientadas a microsserviços contribuirá para um maior desempenho e qualidade no desenho de futuras arquiteturas em ambiente profissional.

Por fim, uma analogia muito interessante sobre um problema real da indústria apresentado no livro *Clean Code* [27] de *Robert C. Martin*.

Um dos problemas atuais no desenho e conceção de software consiste em as empresas darem mais importância a implementar as funcionalidades requeridas pelo cliente, o mais rápido possível sem as desenhar com cuidado e disciplina.

Robert C. Martin, dá o exemplo de um cirurgião que antes de cada cirurgia desinfeta as mãos. Imagine-se que o utente, que corresponde ao cliente, pede ao médico para ultrapassar esse passo do processo para ser mais rápido a concluir a cirurgia. No entanto, o médico, como bom profissional que é, não vai ceder à proposta do utente pois tem conhecimento dos riscos que isso pode trazer para a execução com sucesso do seu trabalho. É deste tipo de atitude que o desenho e conceção de software necessita. De transmitir a todos os interessados e principalmente ao cliente sobre a importância do desenho e conceção cuidada e disciplinada auxiliada pelo conhecimento dos princípios, boas práticas e padrões de software e neste caso nas arquiteturas orientadas a microsserviços.

Bibliografia

[1] C. Richardson, Microservices Patterns, New York, Shelter Island: Manning Publications Co, 2019.

[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis e S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software,* vol. 35, nº 3, pp. 24-35, 2018.

[3] M. Fowler e J. Lewis, "Microservices," martinFowler.com, 25 Março 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Acedido em Outubro, Novembro, Dezembro 2020].

[4] J. Gray, "A Conversation with Werner Vogels," *Enclaves in the Clouds,* vol. 4, nº 4, pp. 14-22, 30 Junho 2006.

[5] A. Cockcroft, "The Evolution of Microservices," Association for Computing Machinery (ACM), 2016. [Online]. Available: https://learning.acm.org/binaries/content/assets/leaning-center/webinar-slides/2016/evolutionofmicroservices_webinarslides.pdf. [Acedido em Outubro, Novembro 2020].

[6] I. C. Team, "SOA vs. Microservices: What's the Difference?," IBM, 14 Maio 2021. [Online]. Available: https://www.ibm.com/cloud/blog/soa-vs-microservices. [Acedido em 2022].

[7] C. Richardson e F. Smith, Microservices from design to Deployment, NGINX, Inc, 2016.

[8] J. Thönes, "Microservices," *IEEE Software,* vol. 32, pp. 116-116, 2015.

[9] S. Newman, Building Microservices, 1ª ed., O'Reilly Media, 2015.

[10] M. Fowler, "MicroservicePrerequisites," 28 Agosto 2014. [Online]. Available: https://martinfowler.com/bliki/MicroservicePrerequisites.html. [Acedido em 2020].

[11] S. Newman, Monolith to Microservices, 1ª ed., O'Reilly Media, 2019, pp. 12-16.

[12] M. Fowler, "Microservice Trade-Offs," 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html. [Acedido em 2022].

[13] C. Richardson, "Microservice Architecture," Kong, [Online]. Available: https://microservices.io. [Acedido em Outubro, Novembro, Dezembro 2020].

[14] M. Waseem, P. Liang, G. Márquez, M. Shahin, A. A. Khan e A. Ahmad, "A Decision Model for Selecting Patterns and Strategies to Decompose Applications into Microservices," *Springer,* 2021.

[15] IBM, "ACID properties of transactions," IBM, 2022. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSGMCP_5.4.0/product-overview/acid.html. [Acedido em Dezembro 2020].

[16] S. Krakowiak, Middleware Architecture with Patterns and Frameworks, 2009, pp. 9-15 até 9-23.

[17] C. Richardson, "Managing data consistency in a microservice architecture using Sagas - part 1,2,3,4," 2019. [Online]. Available: https://chrisrichardson.net/post/microservices/2019/07/09/developing-sagas-part-1.html. [Acedido em Novembro 2020].

[18] Microsoft, “Cloud Design Patterns,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/. [Acedido em Novembro, Dezembro 2020].

[19] K. Malyuga, O. Perl, A. Slapoguzov e I. Perl, “Fault Tolerant Central Saga Orchestrator in RESTful Architecture,” em *2020 26th Conference of Open Innovations Association (FRUCT)*, Yaroslavl, Russia, Russia, 2020.

[20] M. Stefanko, O. Chaloupka e B. Rossi, “The Saga Pattern in a Reactive Microservices Environment,” em *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, 2019.

[21] H. Garcia-Molina e K. Salem, “Sagas,” em *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, Princeton, NJ, 1987.

[22] C. K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture,” *IJACSA - International Journal of Advanced Computer Science and Applications,* vol. 9, pp. 18-22, 2018.

[23] T. Janssen, “Distributed Transactions – Don’t use them for Microservices,” Thorben Janssen, [Online]. Available: https://thorben-janssen.com/distributed-transactions-microservices/. [Acedido em Novembro 2020].

[24] G. Hohpe, “Your Coffee Shop Doesn't Use Two-Phase Commit,” *IEEE Software,* vol. 22, nº 2, pp. 64-66, Março-Abril 2005.

[25] Microsoft, “Design patterns for microservices,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns. [Acedido em 2022].

[26] M. Fowler, “CQRS,” martinFowler.com, 14 Julho 2011. [Online]. Available: https://martinfowler.com/bliki/CQRS.html. [Acedido em Novembro, Dezembro 2020].

[27] R. C. Martin, Clean Code - A Handbook of Agile Software Craftsmanship, Upper Saddle River, NJ: Prentice Hall, 2009, pp. 45-46.

[28] B. Meyer, Object-Oriented Software Construction, 2 ed., Santa Barbara, California: Prentice Hall, 1997, pp. 751-754.

[29] Z. Long, “Improvement and Implementation of a High Performance CQRS Architecture,” em *International Conference on Robots & Intelligent System (ICRIS)*, Huai'an, China, 2017.

[30] F. Montesi e J. Weber, “Circuit Breakers, Discovery, and API Gateways in Microservices,” *arxiv.org,* pp. 1-8, 2016.

[31] P. Calçado, “The Back-end for Front-end Pattern (BFF),” Phil Calçado, 18 Setembro 2015. [Online]. Available: https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html. [Acedido em Janeiro 2021].

[32] X. He e X. Yang, “Authentication and Authorization of End User in Microservice Architecture,” *Journal of Physics: Conference Series,* 2017.

[33] R. H. R. J. J. V. Erich Gamma, Design Patterns, United States: Addison-Wesley, 1994, p. 395.

Anexo I - Diagramas de modelação

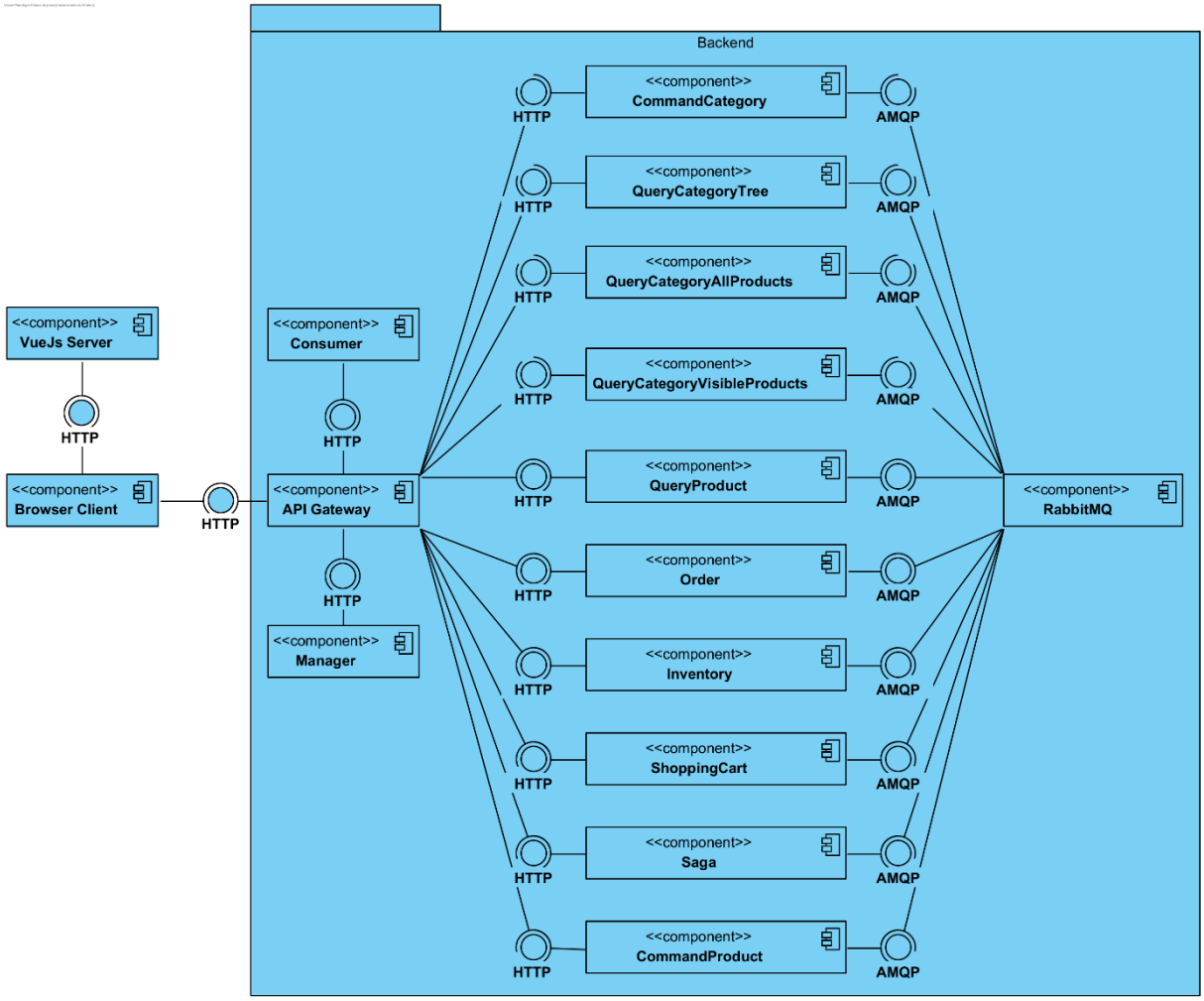


Figura 59: Diagrama de Componentes da arquitetura do caso de estudo e-commerce.

Anexo II - Aspeto da aplicação cliente

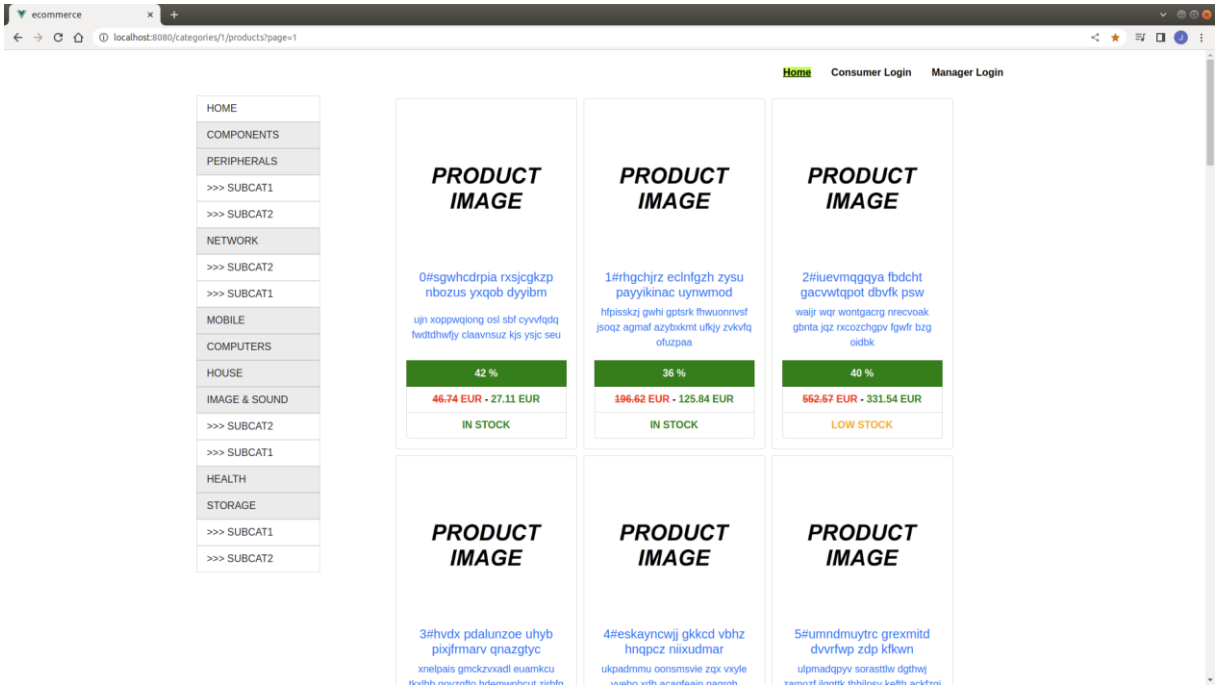


Figura 60: Menu do Consumidor não autenticado, para consulta dos produtos por categoria (página inicial).

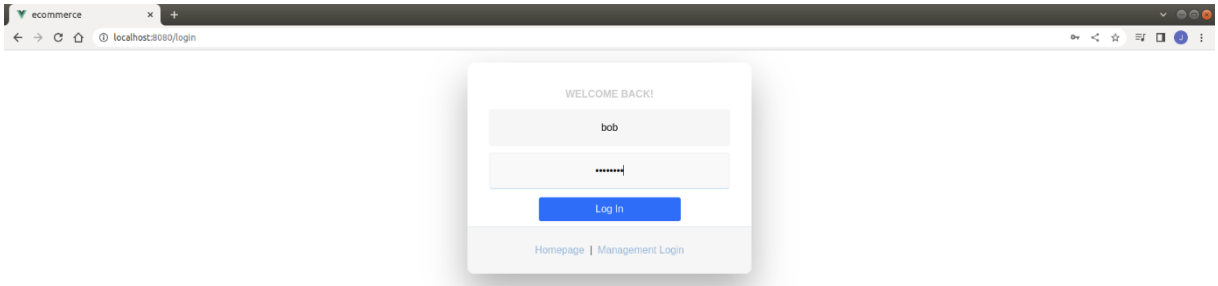


Figura 61: Menu do Consumidor para fazer login.

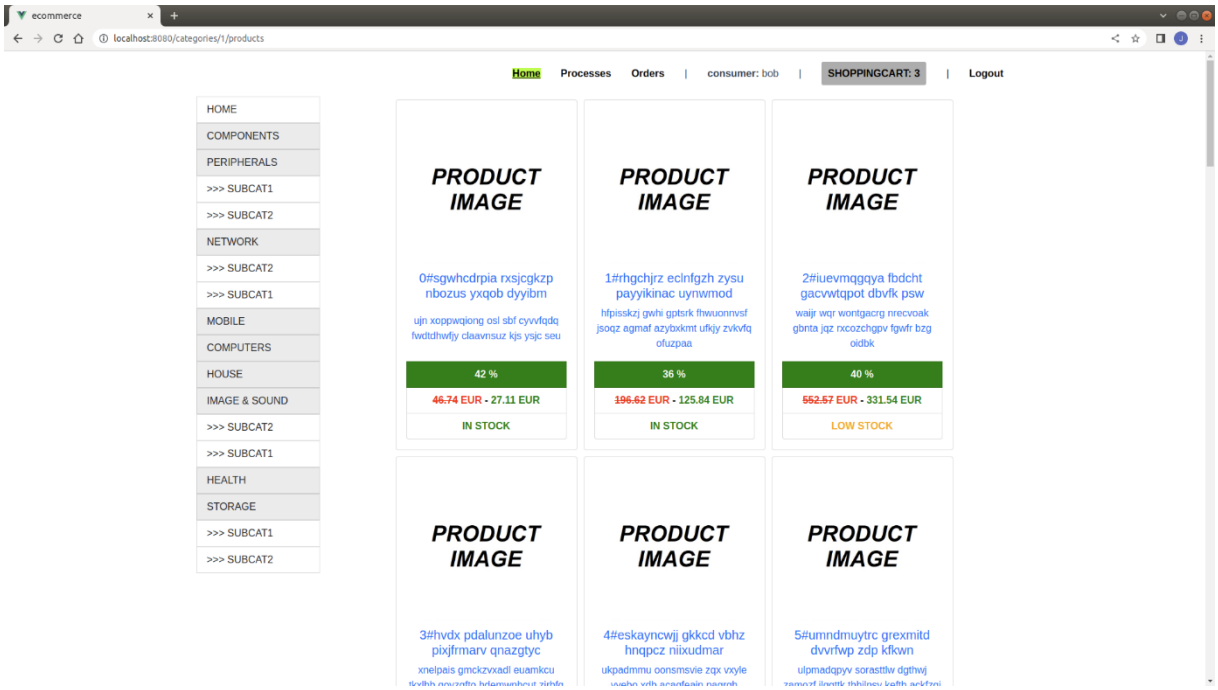


Figura 62: Menu do Consumidor para consulta dos produtos por categoria (página inicial).

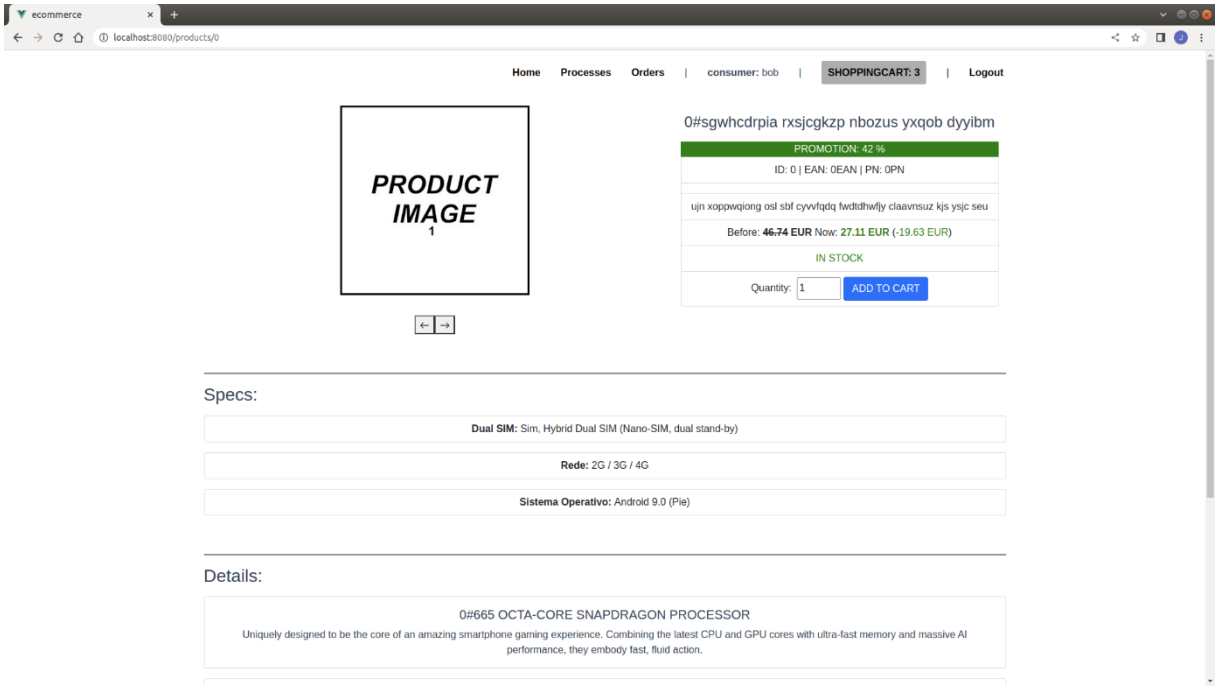


Figura 63: Menu do Consumidor para consulta dos detalhes de um produto.

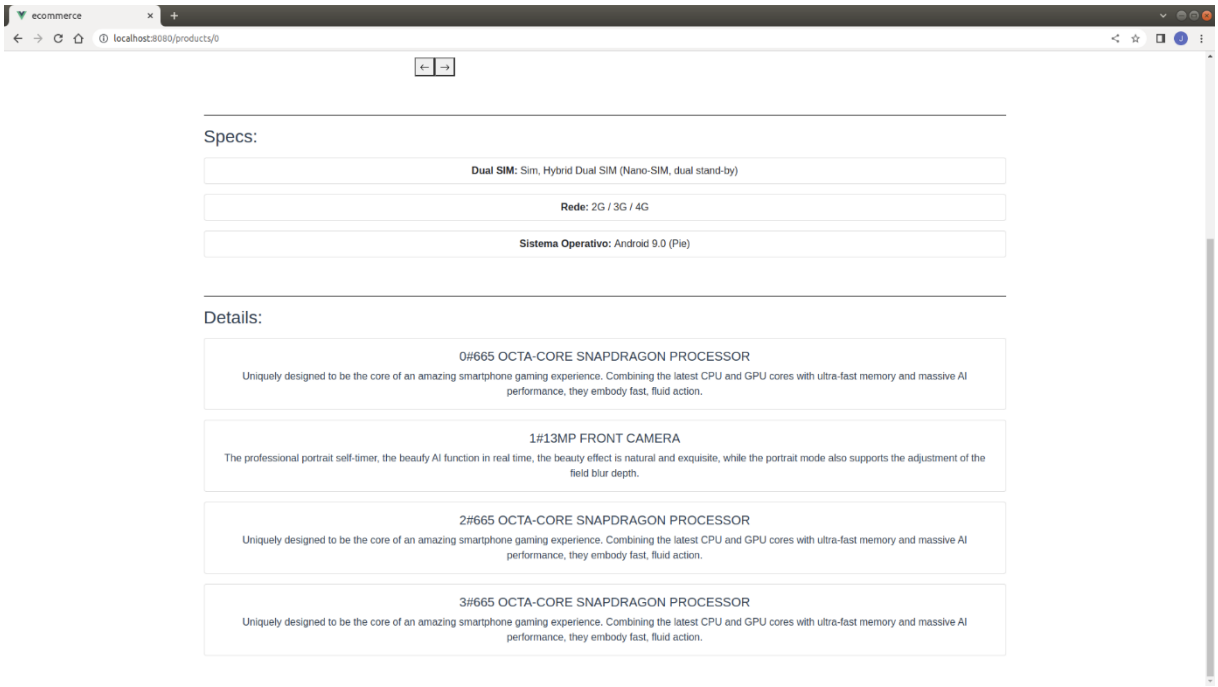


Figura 64: Menu do Consumidor para consulta dos detalhes de um produto, mais concretamente as especificações e informações detalhadas.

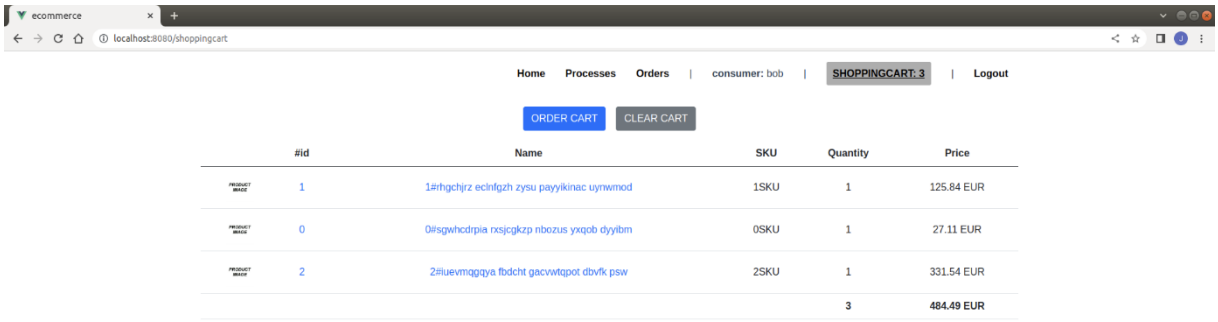


Figura 65: Menu do Consumidor para consulta do carrinho de compras.

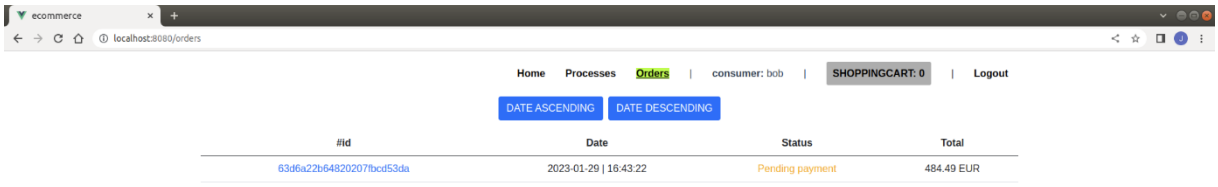


Figura 66: Menu do Consumidor para consulta das suas encomendas.

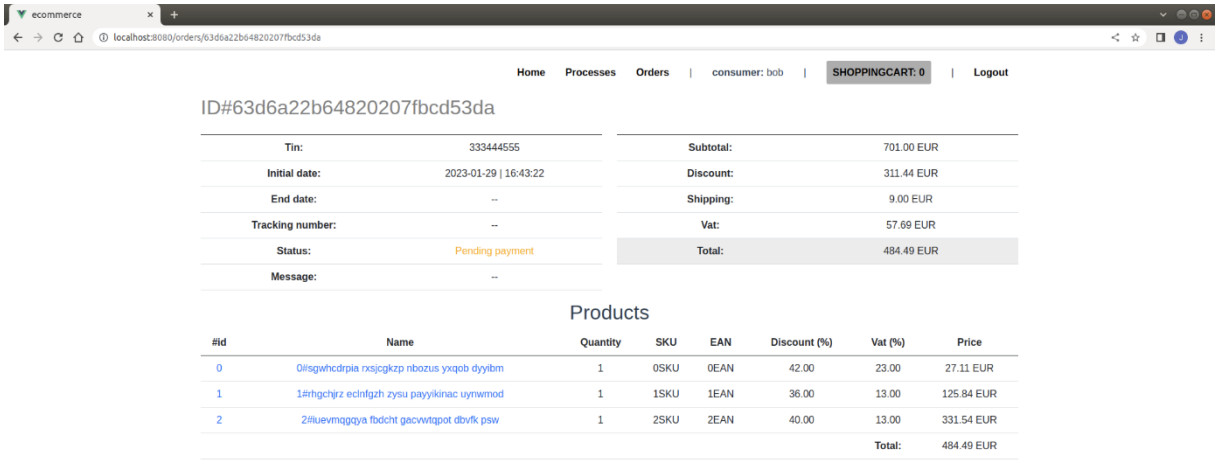


Figura 67: Menu do Consumidor para consulta dos dados de uma encomenda.

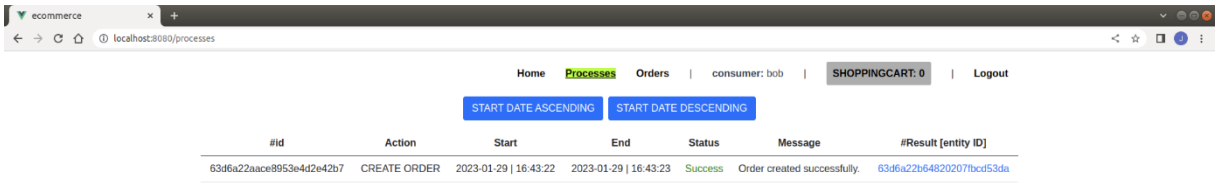


Figura 68: Menu do Consumidor para consulta dos processos assíncronos (sagas) que executou.

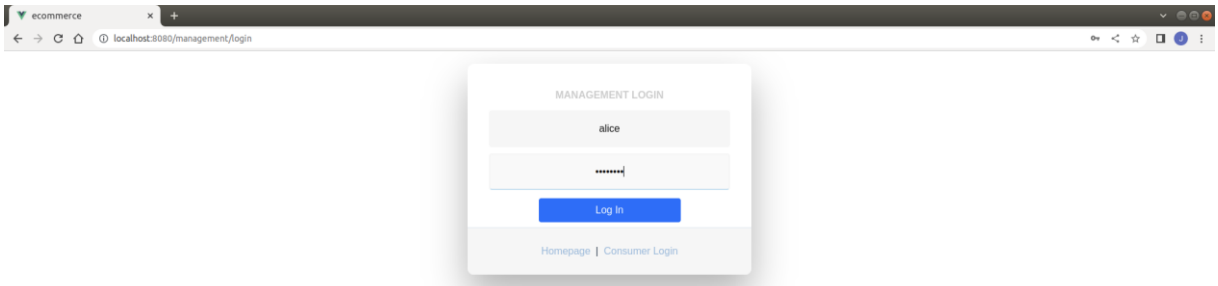


Figura 69: Menu do Gestor para fazer login.

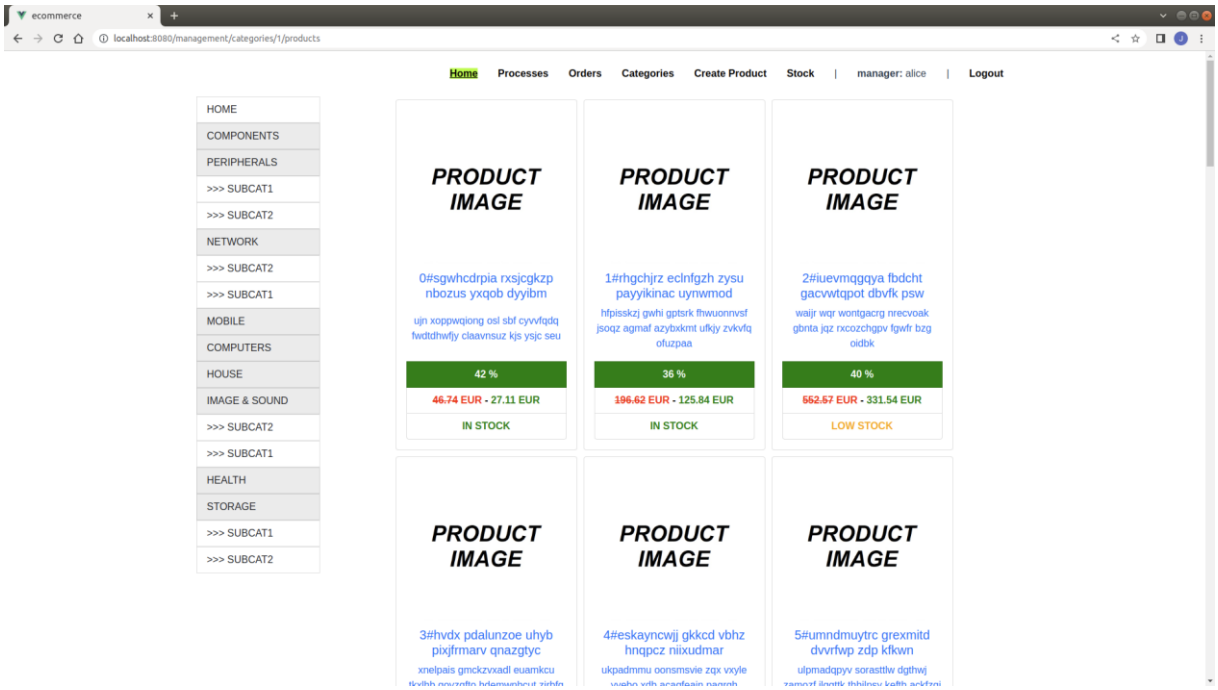


Figura 70: Menu do Gestor para consulta dos produtos por categoria (página inicial).

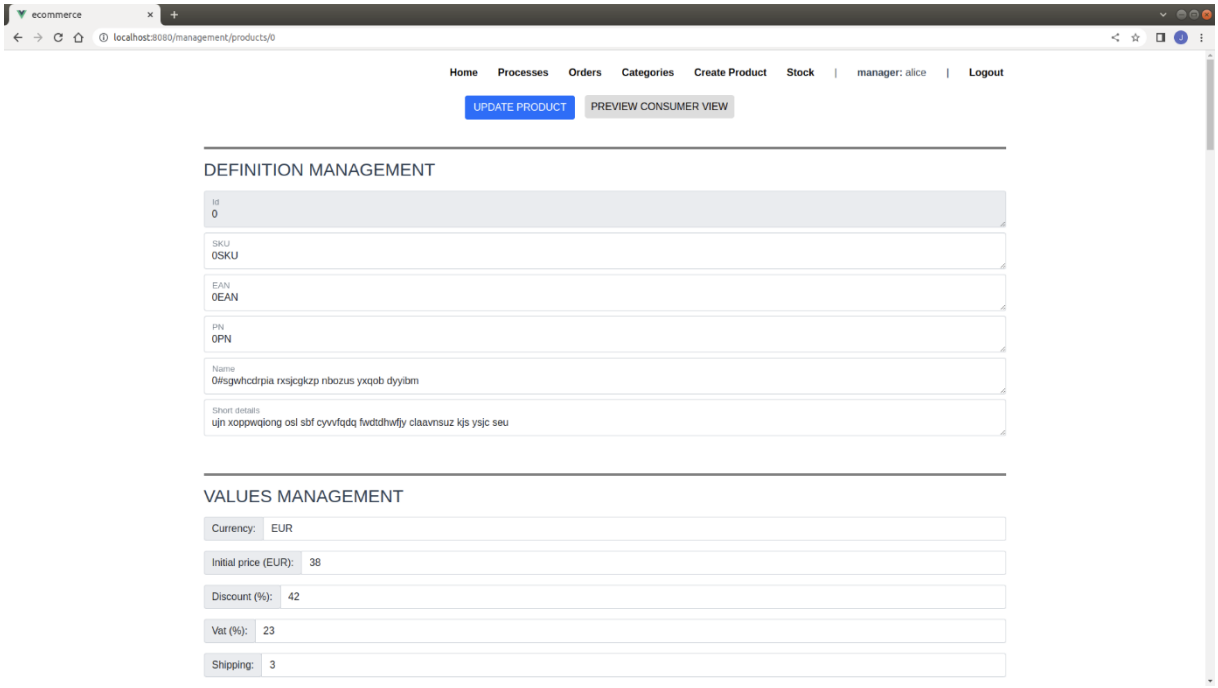


Figura 71: Menu do Gestor para atualização dos dados de um produto, neste caso, nome, pequenos detalhes, preços.

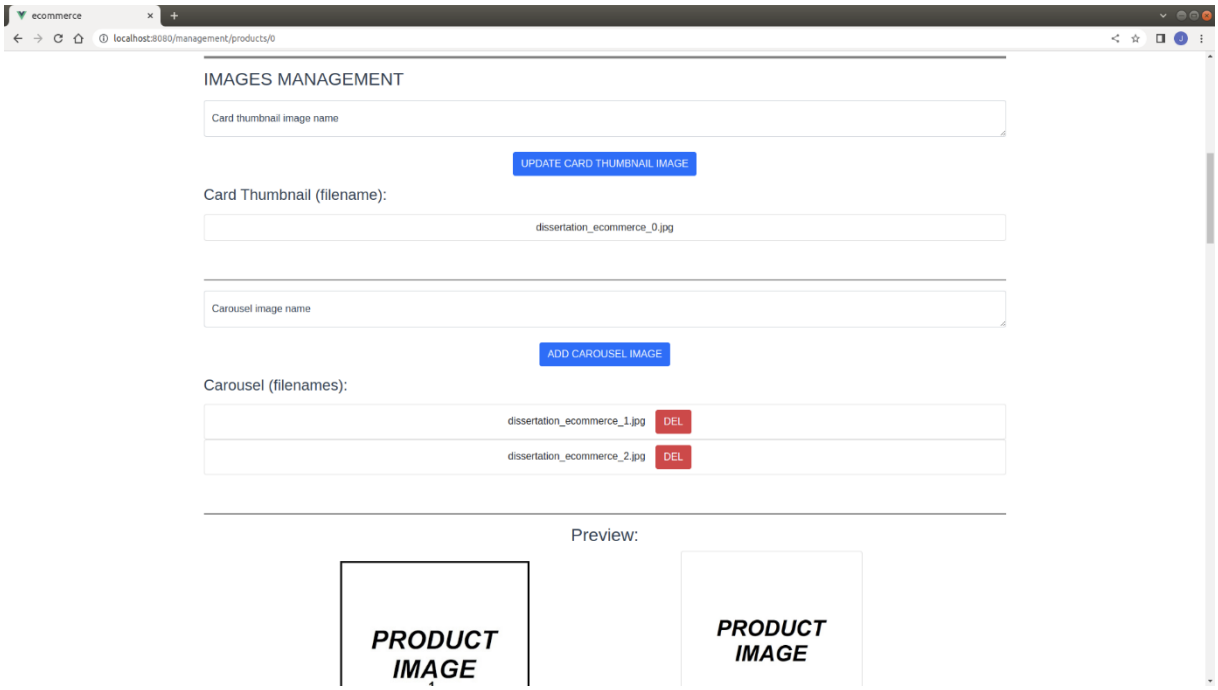


Figura 72: Menu do Gestor para atualizar os dados de um produto, neste caso as imagens dos detalhes principais e para a visão em grelha.

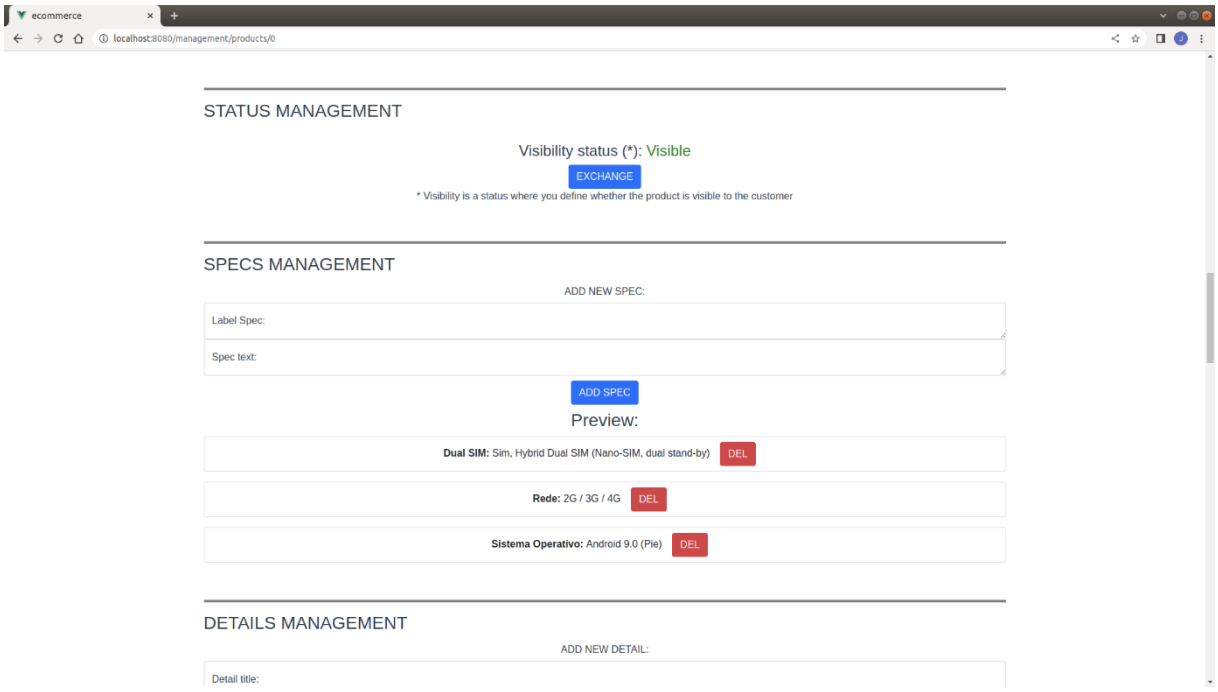


Figura 73: Menu do Gestor para atualizar os dados do produto, neste caso a visibilidade e especificações.

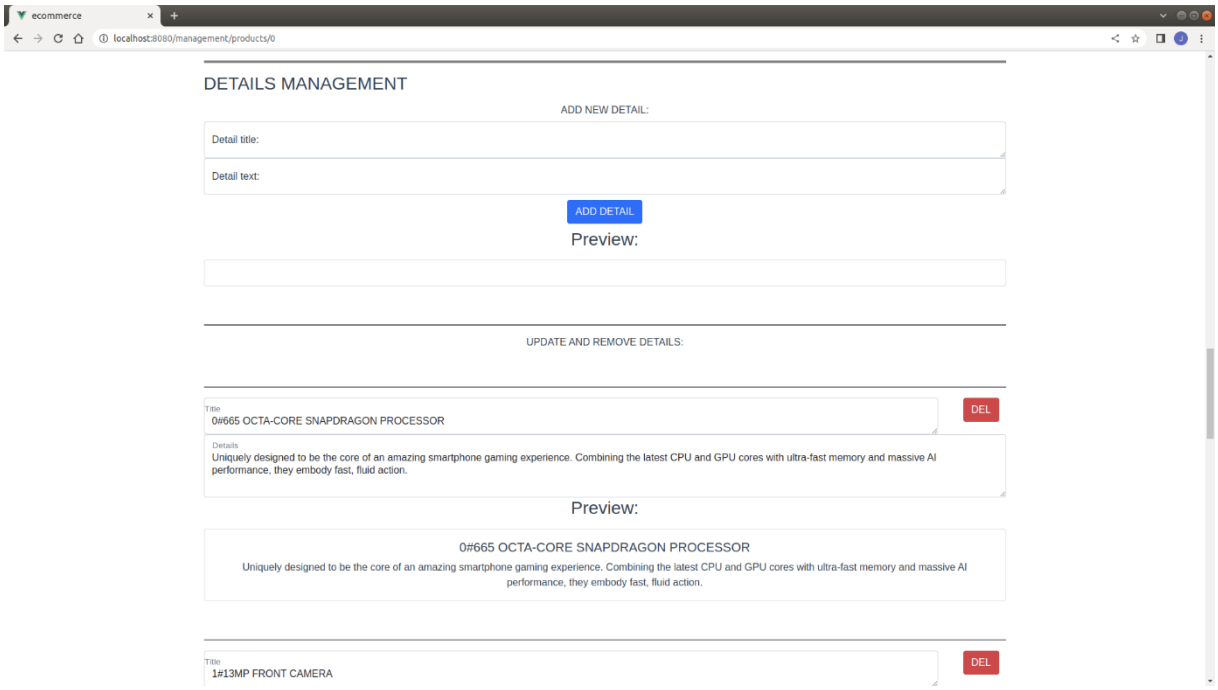


Figura 74: Menu do Gestor para atualização dos dados do produto, neste caso os detalhes.

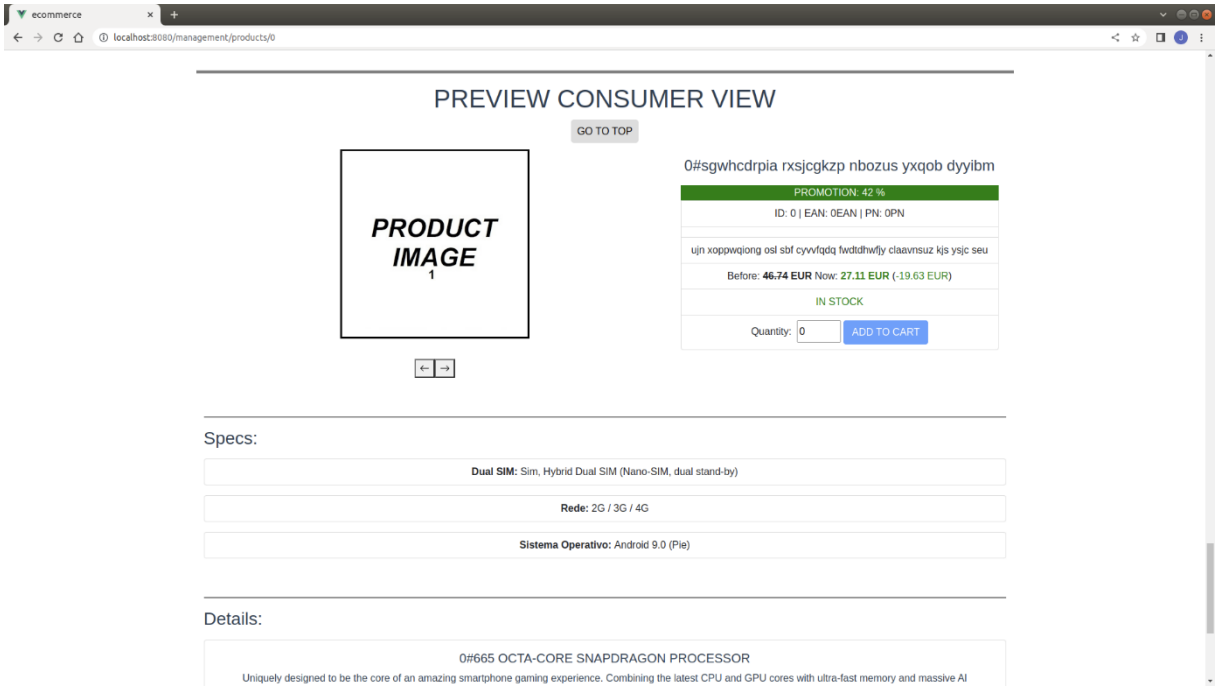


Figura 75: Menu do Gestor para atualização dos dados do produto, neste caso o aspeto final da atualização a ser realizada.

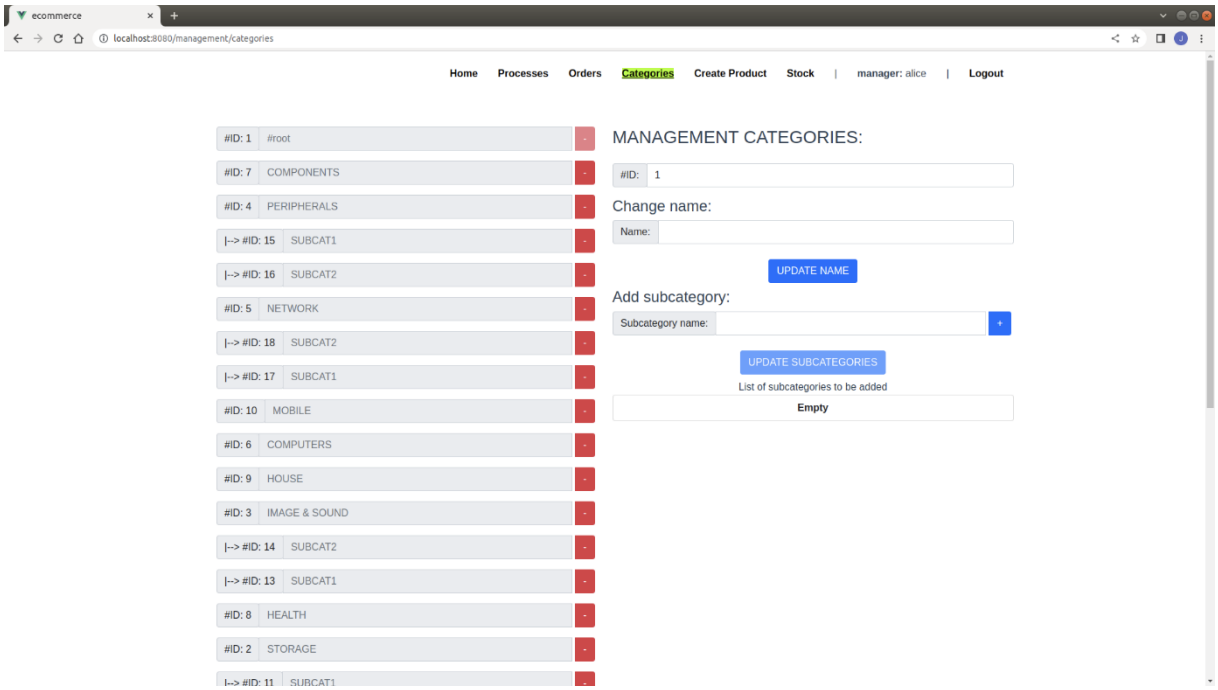


Figura 76: Menu do Gestor para gerir as categorias, mais propriamente, apagar categorias (botão "-"), alterar o nome da categoria e adicionar categorias/subcategorias (região à direita da imagem).

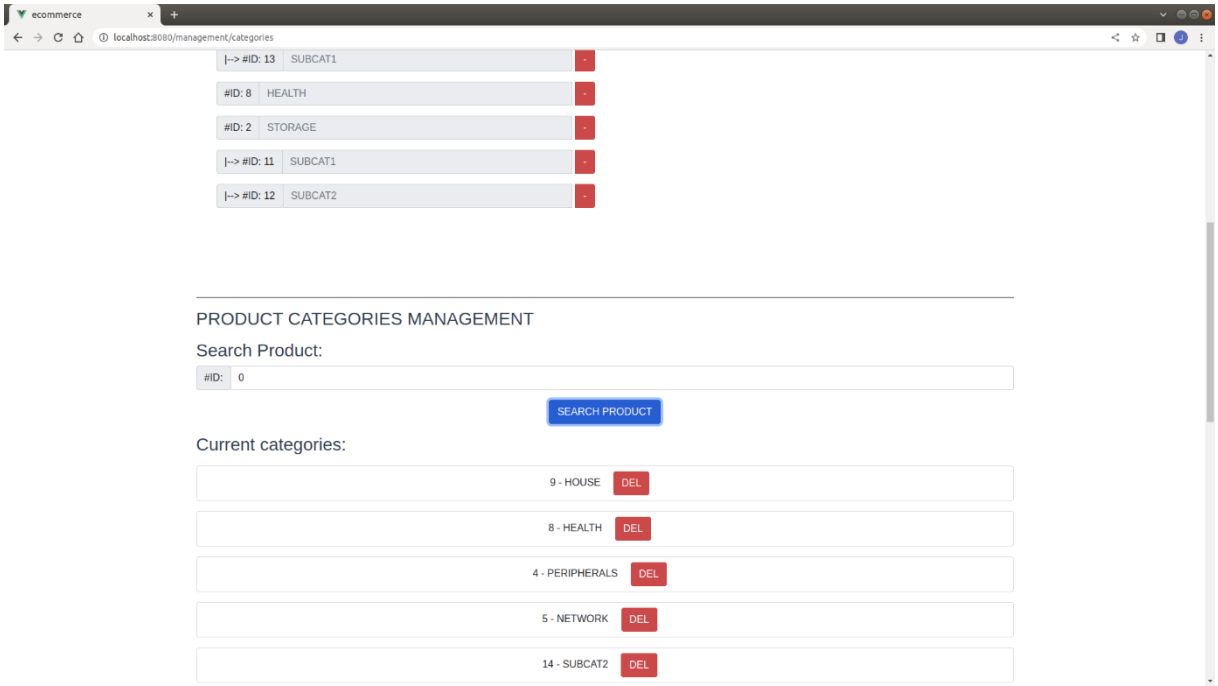


Figura 77: Menu do Gestor para associar e dissociar categorias de produtos.

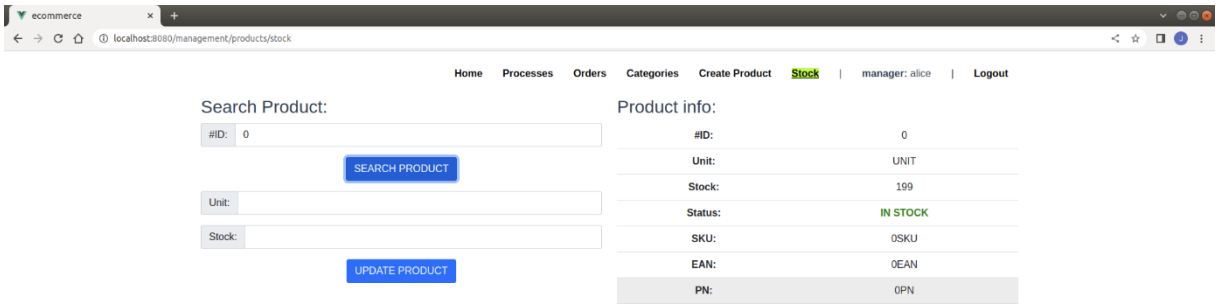


Figura 78: Menu do Gestor para alterar o valor de stock de produtos.

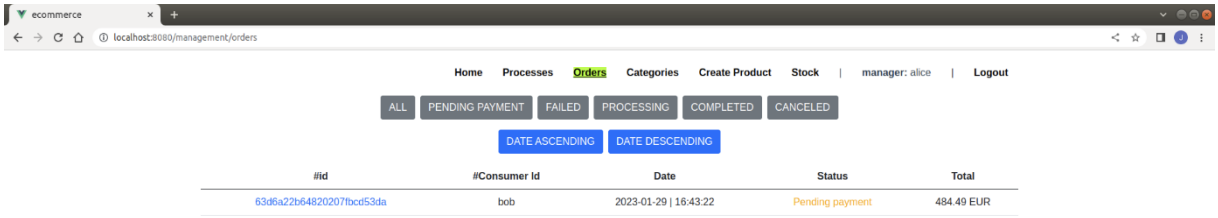


Figura 79: Menu do Gestor para consulta das encomendas dos Consumidores.

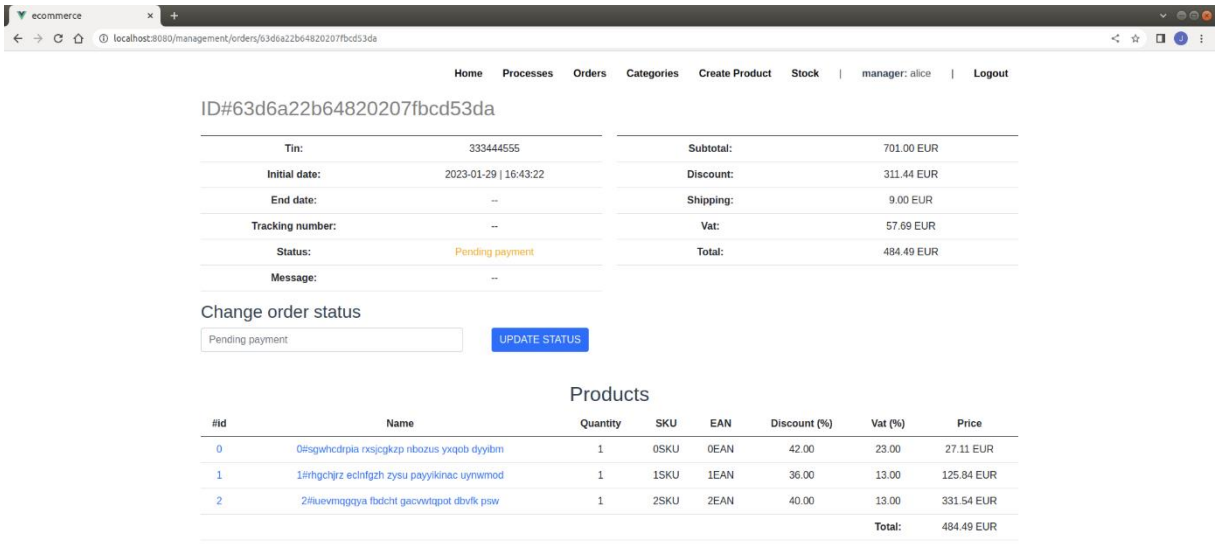


Figura 80: Menu do Gestor para consultar a encomenda de um consumidor e alterar o estado da mesma.

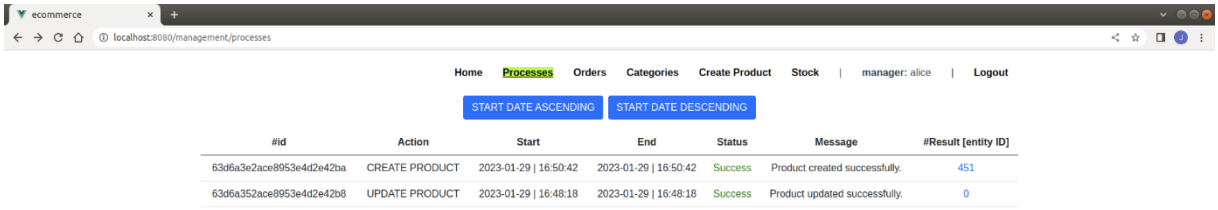


Figura 81: Menu do Gestor para consultar os processos assíncronos (sagas) que executou.

Anexo III - Funcionamento dos Padrões na prática

Assincronismo e não bloqueio das leituras no Saga

Explicação: O objetivo consiste em demonstrar o assincronismo das sagas e o não bloqueio das leituras. Preparou-se um processo saga de "atualização do produto" com id 120, onde vai ser alterado o nome do produto para "120#Smartphone XPTO 23", os detalhes simples para "Smartphone XPTO 23 | Dual SIM | 5 G | Android 12", o preço inicial para 1000 euros (sem IVA) e a percentagem de promoção do produto para 50%. Tal como se pode observar na Figura 82, os dados do produto com id 120 antes da atualização. Antes de se iniciar o processo saga, desligou-se um dos microsserviços participantes, o microsserviço Command Category (CC), tal como se pode observar na Figura 83. De seguida, efetua-se a atualização do produto com id 120, isto é, inicia-se o processo saga, tal como se pode observar na Figura 84. Devido à indisponibilidade do microsserviço Command Category (CC), o processo saga fica num estado pendente, e tal como se pode observar na Figura 85, é possível consultar o produto com id 120, ou seja, não houve bloqueio das leituras, apesar do produto estar bloqueado para atualizações e este mantém os mesmos dados que estavam antes da atualização. De seguida, reinicia-se o microsserviço participante Command Category (CC), tal como se pode observar na Figura 86. Tal como se pode observar na Figura 87, na consulta do produto com id 120, verifica-se a atualização dos dados e na consulta dos processos saga do Gestor, que o mesmo terminou com sucesso.

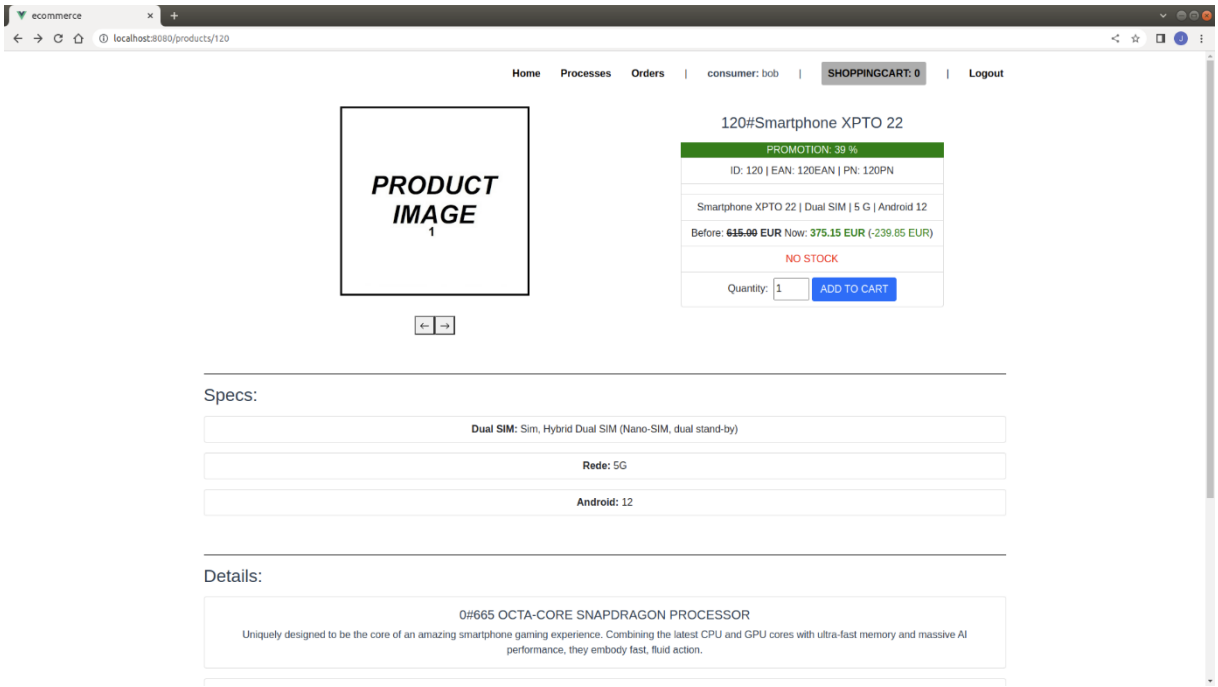


Figura 82: Consulta dos detalhes do produto com id 120, com os dados anteriores à atualização através de um processo saga.

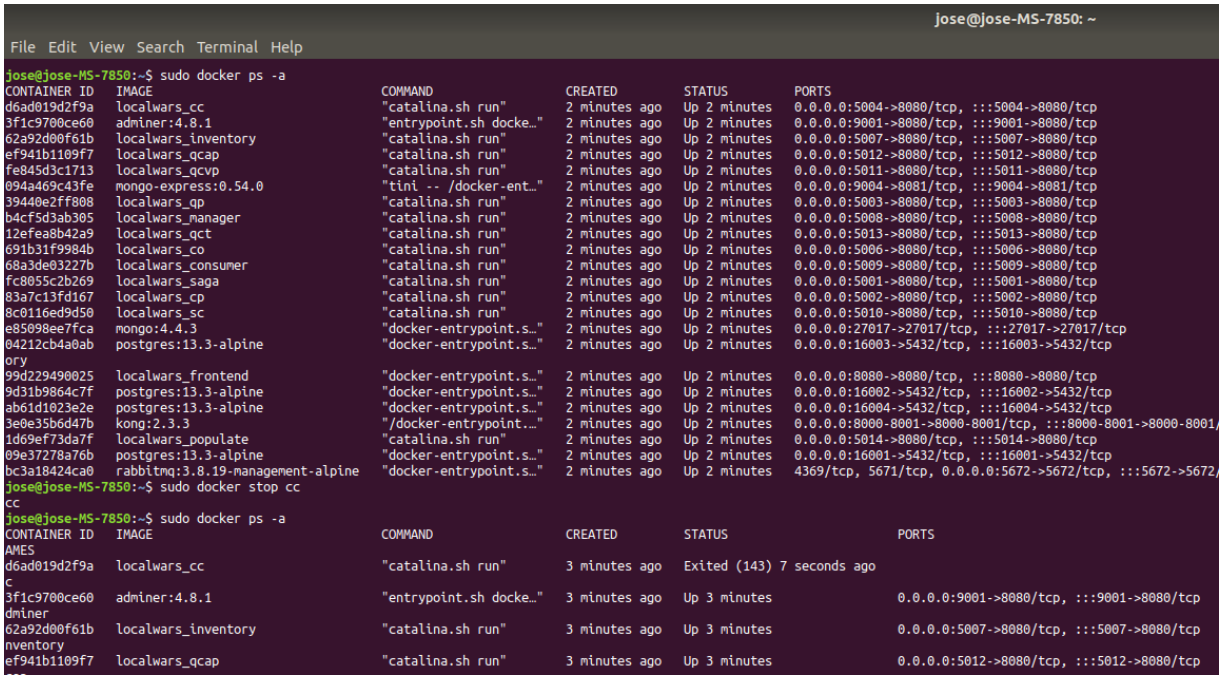


Figura 83: Listagem dos serviços (sudo docker ps -a), seguida da paragem do microsserviço Command Category (CC) (sudo docker stop cc) e listagem dos serviços (sudo docker ps -a), onde se vê que o "status" do serviço CC é "Exited ...".

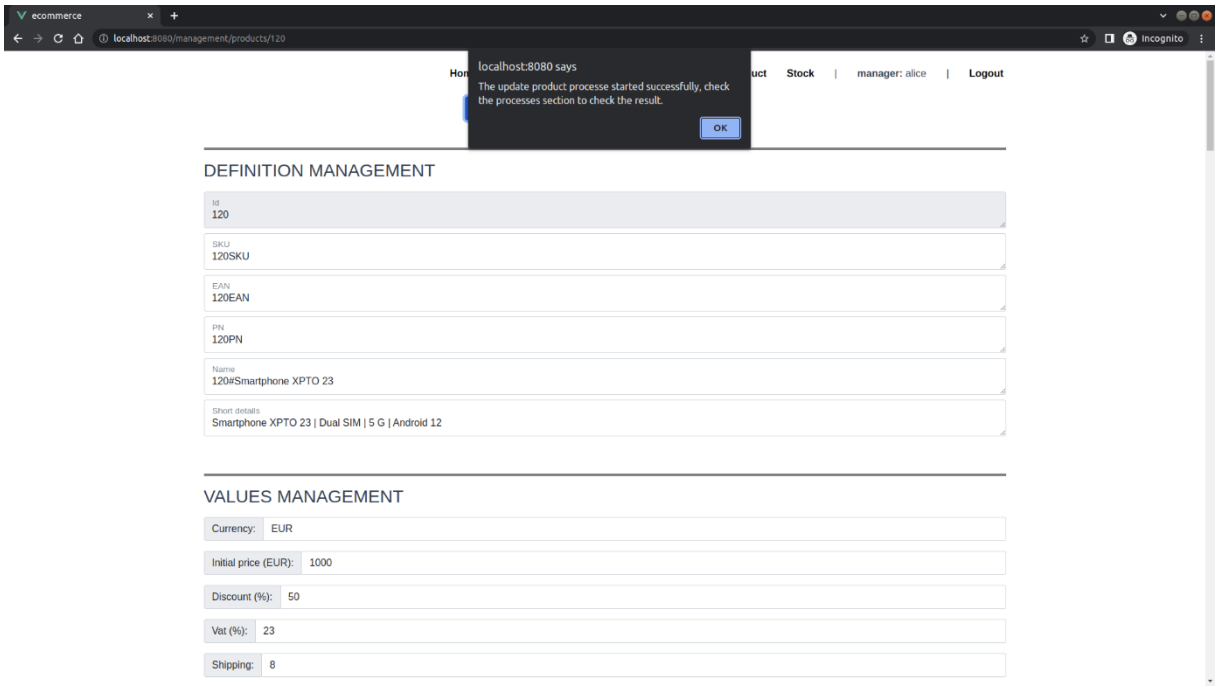


Figura 84: Atualização do produto com id 120 dos novos dados pelo Gestor, onde após clicar no botão "UPDATE PRODUCT" surge uma janela que informa que o processo saga foi iniciado com sucesso (assincronismo).

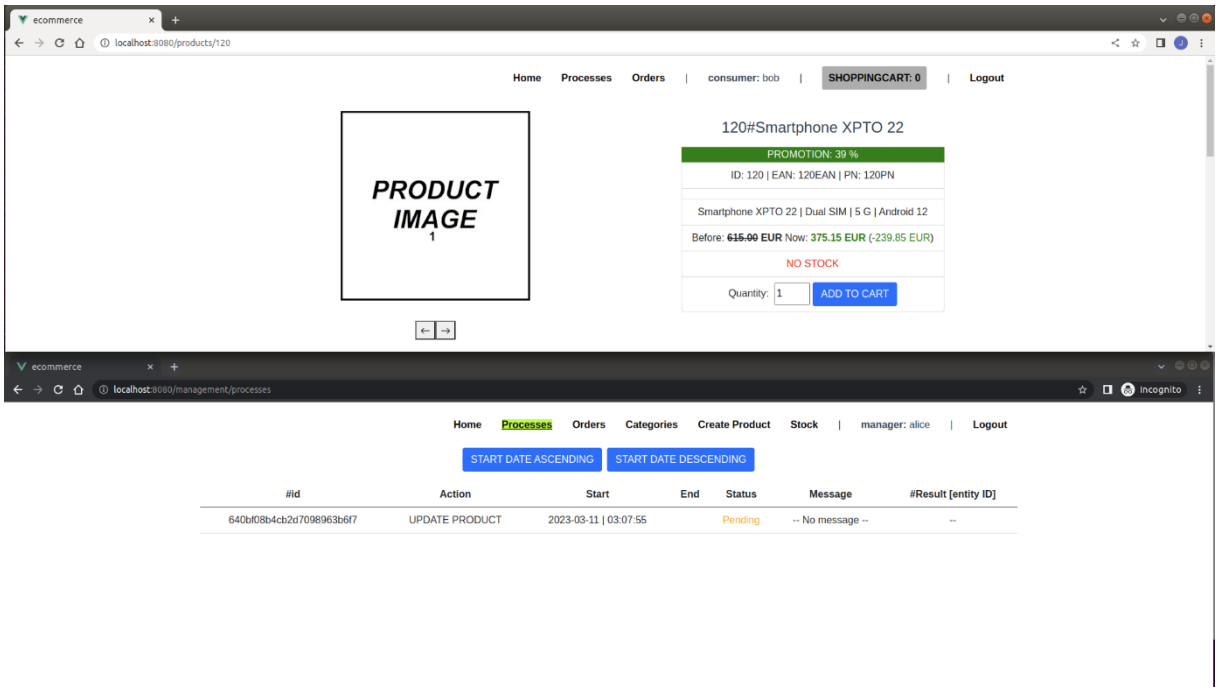


Figura 85: Em cima a consulta do dados do produto com id 120, pelo Consumidor, onde se verifica que mantém os dados anteriores à atualização. Em baixo, têm-se a consulta dos processo saga, onde se verifica que o mesmo está pendente.

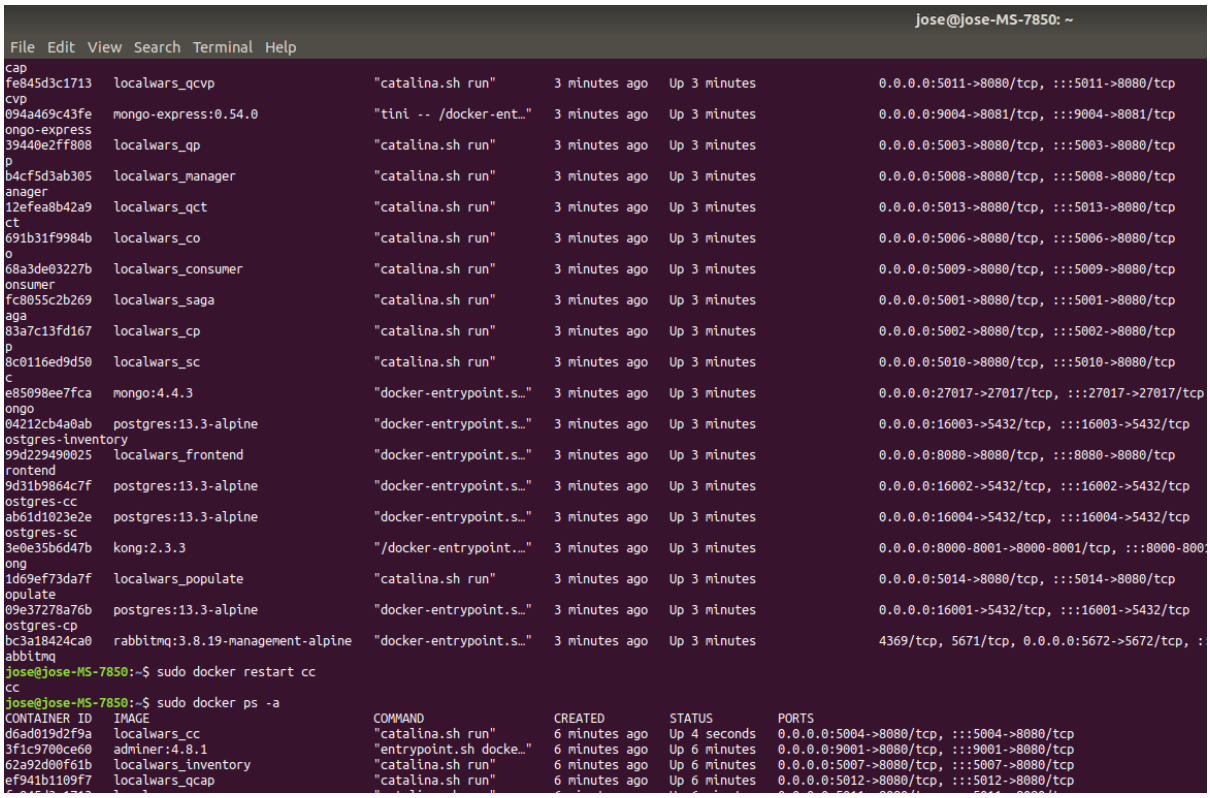


Figura 86: Reinício do microsserviço Command Category (CC) (sudo docker restart cc) e listagem dos serviços (sudo docker ps -a), onde se vê que o "status" do serviço CC é "Up ...".

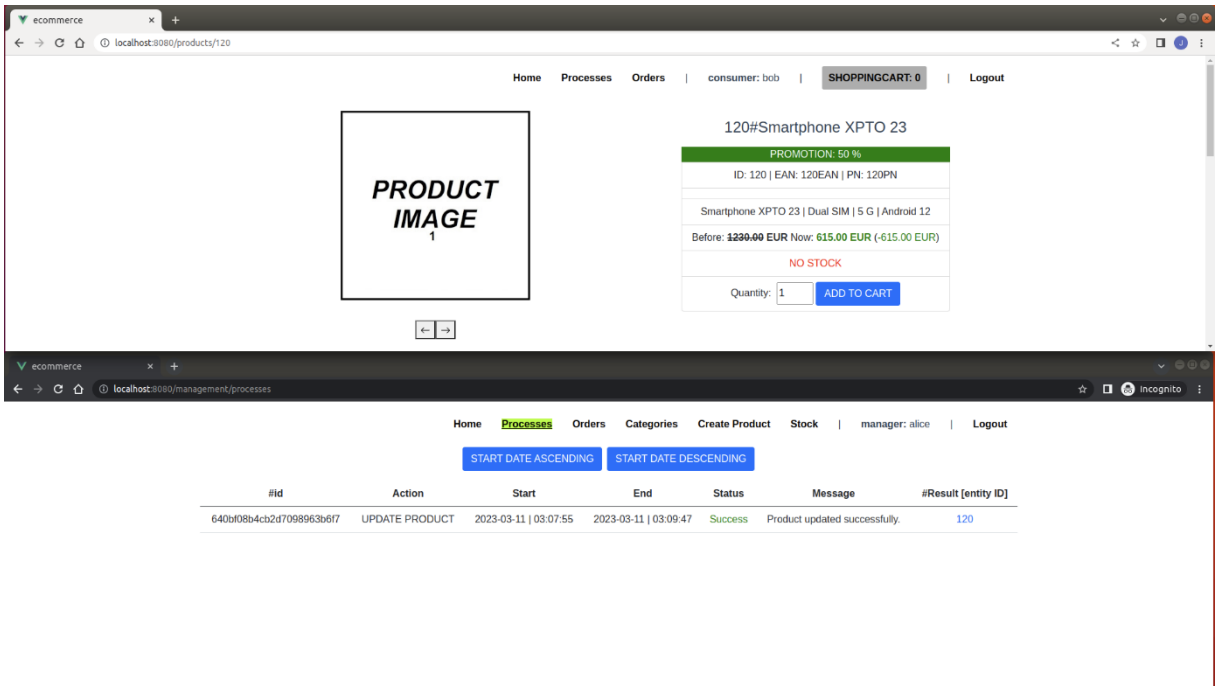


Figura 87: Em cima a consulta do dados do produto com id 120, pelo Consumidor, onde se verifica que os dados foram atualizados. Em baixo, têm-se a consulta dos processo saga, onde se verifica que o mesmo terminou com sucesso.

Anexo IV - Ficheiros de configuração

```
version: '3.8'

volumes:
  kong_data: {}
  portainer_data: {}

networks:
  kong-net:
    external: false
  postgres-cp-net:
    external: false
  postgres-cc-net:
    external: false
  postgres-inventory-net:
    external: false
  postgres-sc-net:
    external: false
  mongo-net:
    external: false
  rabbitmq-net:
    external: false

services:
  frontend:
    image: josepereira1/frontend:0.0.1-beta
    deploy:
      mode: replicated
      replicas: 1
    ports:
      - 8080:8080

  saga:
    image: josepereira1/saga:0.0.1-beta
    deploy:
      mode: replicated
      replicas: 1
    ports:
    - 5001:8080
    depends_on:
      - mongo
      - rabbitmq
    networks:
      - kong-net
```

```
    - mongo-net
    - rabbitmq-net

cp:
  image: josepereira1/cp:0.0.1-beta
  deploy:
    mode: replicated
    replicas: 1
  ports:
  - 5002:8080
  depends_on:
    - postgres
    - rabbitmq
  networks:
    - postgres-net
    - rabbitmq-net

# other microservices (the differences are the ports, depends_on and networks)

postgres-cp:
  container_name: postgres-cp
  image: postgres:13.3-alpine
  environment:
    POSTGRES_USER: root
    POSTGRES_PASSWORD: ecommerce
    POSTGRES_DB: cp
    PGDATA: /data/postgres
  volumes:
      - /home/joseandremp1/cp:/data/postgres
  ports:
    - 16001:5432
  networks:
    - postgres-cp-net
  restart: unless-stopped

# others portgresql databases (database per service)

rabbitmq:
  image: rabbitmq:3.8.19-management-alpine
  deploy:
    mode: replicated
    replicas: 1
  ports:
     - 5672:5672
     - 15672:15672
  volumes:
     - /home/joseandremp1/volumes/rabbitmq/data/:/var/lib/rabbitmq/
```

```
      - /home/joseandremp1/volumes/rabbitmq/log/:/var/log/rabbitmq
    depends_on:
      - postgres
    networks:
      - rabbitmq-net

  # the mongodb database server is shared between microservices (but only in
development)

  mongo:
    image: mongo:4.4.3
    deploy:
      mode: replicated
      replicas: 1
    ports:
      - 27017:27017
    volumes:
      - /home/joseandremp1/volumes/mongo:/data/db
    networks:
      - mongo-net

  kong:
    image: 'kong:2.3.3'
    deploy:
      mode: replicated
      replicas: 1
    ports:
      - '8000:8000'
      - '8443:8443'
      - '8001:8001'
    environment:
      - KONG_DATABASE=off
      - KONG_ADMIN_LISTEN=0.0.0.0:8001
    volumes:
      - /home/joseandremp1/volumes/kong/kong.conf:/etc/kong/kong.conf
      - /home/joseandremp1/volumes/kong/kong.yaml:/usr/local/kong/declarative/kong.yaml
    networks:
      - kong-net
```

Listagem 7: Ficheiro de configuração dos containers de Docker (docker-compose.yml).