P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M

MESTRADO
ENGENHARIA INFORMÁTICA

Conceber arquiteturas baseadas em Micro-serviços escaláveis e fiáveis para o comércio eletrónico

Jorge Daniel Ribeiro Oliveira

10/2022

Conceber arquiteturas baseadas em Micro-serviços escaláveis e fiáveis para o comércio eletrónico
Jorge Daniel Ribeiro Oliveira

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M

MESTRADO
ENGENHARIA INFORMÁTICA

Jorge Daniel Ribeiro Oliveira. Conceber arquiteturas baseadas em Micro-serviços escaláveis e fiáveis para o comércio eletrónico

10/2022

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M

MESTRADO
ENGENHARIA INFORMÁTICA

Conceber arquiteturas baseadas em Micro-serviços escaláveis e fiáveis para o comércio eletrónico

Jorge Daniel Ribeiro Oliveira

Ricardo Jorge da Silva Santos

Agradecimentos

Gostaria de agradecer a todas as pessoas que estiveram diretamente e indiretamente relacionadas com o desenvolvimento deste trabalho.

Agradeço ao professor Ricardo Santos pela apresentação da oportunidade deste tema e à disponibilidade demonstrada e pela sua orientação.

Agradeço à minha família e à minha namorada pelo apoio demonstrado ao longo de todos estes anos que foram fundamentais para chegar até a esta etapa.

Por fim, agradeço a Escola Superior de Tecnologia e Gestão pela oportunidade e por todo o conhecimento que adquiri durante a licenciatura e também durante o mestrado.
Muito obrigado.

Resumo

A utilização da internet vem cada vez mais a crescer um pouco por todo o mundo. A adoção desta tecnologia faz com que os comportamentos e os hábitos dos consumidores, ao nível de compra de bens ou produtos, sofra alterações. A tendência para a utilização de plataformas de comércio eletrónico é naturalmente cada vez maior.

O presente documento visa descrever o estudo da problemática do comércio eletrónico, os desafios que estas plataformas *online* apresentam, e o desenvolvimento de uma plataforma de comércio eletrónico baseada em micro-serviços, de forma que as empresas possam disponibilizar serviços para os seus utilizadores de forma fiável e escalável. Estes tipos de plataformas, naturalmente, passam por picos de utilização que podem deixar com que as empresas fiquem incapazes de disponibilizar os seus serviços, o que pode resultar em perdas monetárias e, consequente, insatisfação dos seus clientes.

Ao longo deste documento são discutidos os conceitos de arquitetura monolítica, arquiteturas micro-serviços, as vantagens e desvantagens que ambas apresentam e as situações nas quais estas são adequadas. Também, são apresentados casos de estudo sobre os desafios e problemas encontrados, nomeadamente em arquiteturas monolíticas. Por outro lado, é relatado as melhorias que a adoção de uma plataforma micro-serviços veio proporcionar a várias empresas mundialmente conhecidas. Por fim é relatado todo desenvolvimento de uma plataforma de comércio eletrónico baseado em micro-serviços.

Palavras-chave: comércio eletrónico, arquitetura monolítica, micro-serviços, fiabilidade, escalabilidade.

Abstract

The use of internet has been increasing at a fast pace everywhere around the world. The adoption of this technology has changed the habits and behaviors of the consumers when it comes to purchases of goods.

This document describes the study and the problems of e-commerce business, the challenges that arise from the use of these online platforms and the development of a microservice based e-commerce platform so that companies can provide services to their costumers in a reliable and scalable way. These kinds of platforms, can naturally experience peeks in usage that may lead to downtime making the company unable to provide their service to their costumers, which may result in loss of income and consequently in costumer dissatisfaction.

Throughout this document it is discussed the concepts of monolithic architectures, microservices architectures, the advantages and disadvantages that both of these approaches present and the situations where the utilization of a type of architecture is the best suitable. Besides, it is presented cases of study about the problems and challenges found when using monolithic architectures. On the other hand, it is also described the benefits that arose for a diverse set of companies, known worldwide, when adopting a microservice architecture. Finally, it is reported the technical development of an e-commerce platform using a microservice architecture approach.

Keywords: e-commerce, monolithic architecture, microservices architecture, reliability, scalability.

Conteúdo

Lista de figuras

Lista de tabelas

Abreviaturas

API – Application Programming Interface

AWS – Amazon Web Services

B2B – Business to Business

CPU - Central Processing Unit

DDD – Domain Drive Design

IoT – Internet of Things

RAM – Random Access Memory

REST - Representational State Transfer

Capítulo 1 – Contextualização

1. Introdução

1.1. Contextualização

Na última década, a adoção e a utilização da internet está presente no dia a dia de praticamente todas as sociedades. Esta tecnologia foi usada regularmente por mais de 3 mil milhões de pessoas em 2020, e cerca de 5.385 mil milhões de pessoas em 2022, o que representa um aumento de 79.5% em apenas dois anos [1]. Isto significa que, a internet, atualmente é utilizada por mais de metade da população mundial (aproximadamente 68%) de forma regular. Consequentemente, com esta rápida adoção, os hábitos dos consumidores naturalmente vêm a sofrer alterações, visto que, desta forma é proporcionado aos mesmos efetuarem a compra de produtos ou de serviços com uma maior facilidade e comodidade e com muito pouco esforço. Em 2018, foi estimado que cerca de 1.8 mil milhões de pessoas utilizaram a internet para adquirir bens ou serviços [2].

Com todo este crescimento, e para dar resposta ao número elevado de procura e compra de bens e produtos pela internet, foram surgindo cada vez mais plataformas de venda ao público, nomeadamente plataformas de comércio eletrónico. Estas podem ser mais pequenas, ou mesmo maiores e mais estabelecidas mundialmente como a *Amazon*, *Aliexpress* ou *Ebay*. Os pequenos negócios para crescerem mais rapidamente, para se tornarem mais competitivos e para alargarem o público-alvo apostam na criação de plataformas de comércio eletrónico. Assim, conseguem encontrar uma forma de se auto sustentarem ou até mesmo aumentar de forma significativa os seus lucros, para expandirem além das lojas físicas que disponibilizam, que simplesmente são mais caras de manter e de ampliar de forma eficiente. Para que seja possível alargar o negócio a novos horizontes, é necessário garantir que estas plataformas disponibilizadas *online* estão preparadas para receber inúmeras visitas em simultâneo, e que estas são capazes de darem resposta à necessidade dos utilizadores mesmo em alturas de maior afluência. Este tipo de pico de utilização é bastante frequente, podendo acontecer mais regularmente nas alturas de saldos, lançamento de produtos ou até mesmo com o crescimento gradual da plataforma que naturalmente ao longo do tempo irá receber cada vez mais visitas.

1.2. Apresentação e Oportunidade do tema

Tradicionalmente, muitas plataformas *online* foram ou ainda são desenvolvidas com recurso a arquiteturas monolíticas, ou seja, todas as funcionalidades implementadas estão

encapsuladas dentro de uma só aplicação. Toda a aplicação é executada exclusivamente numa máquina onde todo o processamento, memória, base de dados e ficheiros são num único processo, isto é, os componentes da aplicação executam de forma acoplada, o que pode comprometer desde logo o desempenho.

Numa fase inicial, as arquiteturas monolíticas permitem que todo o processo de arquitetura e desenvolvimento das aplicações seja concluído mais rapidamente, dado que, é mais fácil desenvolver, testar e realizar o respetivo *deploy* da aplicação num servidor seguindo este tipo de arquitetura. No entanto, estes benefícios tornam-se rapidamente obsoletos quando é necessário desenvolver aplicações com processos de negócio bem definidos e mais complexos, tornando este tipo de arquitetura inadequada e bastante limitada por diversas razões [3]:

- Quaisquer alterações ao código (e.g. novas funcionalidades, correções ou atualizações) necessitam de um novo *deploy* total da aplicação no servidor resultando num *downtime* bastante grande.
- Com o decorrer da introdução de novas funcionalidades, a aplicação fica cada vez mais complexa e difícil de efetuar a sua manutenção.
- Os diferentes componentes tornam-se mais dependentes e acoplados entre si e com responsabilidades cada vez mais indefinidas.
- Para escalar este tipo de aplicações é necessário escalar verticalmente (através da adição de mais recursos a uma máquina como, por exemplo, adição de mais *RAM*, memória ou capacidade de processamento), sendo impossível escalar determinadas funcionalidades individualmente.
- O desempenho fica cada vez mais difícil de gerir devido à partilha do processamento por todos os componentes presentes na aplicação.
- Pouca flexibilidade no tipo de tecnologia utilizada, após o início de um projeto dificilmente é possível utilizar outras linguagens de programação ou *frameworks*, que mais se adequam para resolver um determinado problema no decorrer do desenvolvimento de uma funcionalidade, além da inicialmente escolhida.

Nos últimos anos, o conceito de arquiteturas de micro-serviços vem a ser cada vez mais adotado, falado e utilizado por diferentes tipos de indústrias e empresas no desenvolvimento das suas plataformas. Um micro-serviço é uma pequena aplicação que tem exclusivamente uma única responsabilidade, isto é, realiza apenas um único objetivo final, e que pode ser implementada, escalada e testada independentemente [4]. Um exemplo de uma única responsabilidade poderá ser um requisito funcional ou não funcional [5]. Através desta arquitetura, é possível desenvolver uma aplicação através da construção de diversos micro-

serviços independentes e com o seu próprio processo. Isto permite que a correção de erros ou a adição de novas funcionalidades a um determinado micro-serviço seja mais simples, pois todos os outros micro-serviços são independentes e não são afetados, sendo apenas necessário dar *deploy* ao micro-serviço em questão. Mais, ao adotarmos este tipo de arquitetura é possível escalar individualmente determinados micro-serviços (disponibilizar mais instâncias do micro-serviço), ao contrário da arquitetura monolítica, onde apenas é possível escalar toda a aplicação de uma vez. Também, com a utilização deste tipo de arquitetura, existe bastante liberdade, flexibilidade e heterogeneidade no tipo de tecnologias utilizadas, sendo possível escolher uma linguagem de programação que mais se adequa para resolver um determinado problema cujo um determinado micro-serviço visa resolver. Adicionalmente, este tipo de arquitetura é mais tolerante a falhas, dado que, como os micro-serviços atuam de forma independente, a falha de um micro-serviço geralmente não tem impacto no restante sistema. No entanto, este tipo de arquitetura apresenta alguns desafios [4]:

- Como os micro-serviços são independentes e necessitam de comunicar com outros micro-serviços, para assegurar comunicações entre estes, pode ser necessário recorrer a mecanismos de comunicação assíncronos como o uso de *message brokers* (e.g. *RabbitMQ* ou *Kafka*).
- A arquitetura baseada em micro-serviços adiciona mais complexidade a um projeto, devido a tratar-se essencialmente de um sistema distribuído. De modo a efetuar uma melhor gestão da complexidade introduzida por este tipo de arquitetura, devem ser formadas equipas que assegurem que os micro-serviços estão de acordo com o que foi inicialmente definido, sendo que, este processo deve ser acompanhado desde o desenho inicial da arquitetura e durante a sua implementação.

Atualmente, como cada vez mais empresas vendem os seus produtos pela internet, sendo que, muitas dessas empresas acabam por vender exclusivamente online sem existência de lojas físicas. Assim podem alcançar uma maior audiência com mais facilidade e com menos custos. Para suportar esta forma de negócio, é necessário que as plataformas de comércio eletrónico, disponibilizadas pelas empresas, sejam tolerantes a falhas, fiáveis e facilmente escaláveis para conseguir assegurar o acesso à plataforma de muitos utilizadores em simultâneo. Desse modo passa a ser possível obter uma maior satisfação por parte dos clientes e, consequentemente, melhorar as receitas da empresa que disponibiliza o serviço. A utilização de uma arquitetura monolítica ou de uma arquitetura orientada a micro-serviços deve ser cuidadosamente analisada para cada situação, antes de proceder ao desenvolvimento de uma determinada plataforma, de modo a verificar e perceber qual o tipo

de arquitetura que mais se adequa. Contudo, neste contexto, uma arquitetura de micro-serviços, é uma proposta interessante que visa atender à resolução de problemas resultantes da construção de plataformas de comércio eletrónico principalmente plataformas assentes em arquiteturas monolíticas. Além das vantagens já mencionadas, desta forma é possível garantir que estas mesmas aplicações são robustas e fiáveis em diversos cenários de utilização, podendo, a aplicação, ser facilmente escalada em qualquer momento ou apenas em caso de necessidade como em picos de utilização.

1.3. Objetivos principais

Com a elaboração deste projeto, o principal objetivo passa por estudar e desenvolver uma arquitetura baseada em micro-serviços escalável e fiável direcionada para o comércio eletrónico. Para atingir este objetivo, foram estudados e apresentados padrões de software que permitam implementar uma arquitetura que visa colmatar os problemas anteriormente identificados relacionados com a problemática do comércio eletrónico. Estes problemas, estão principalmente presentes nas plataformas cujas arquiteturas se baseiam nas tradicionais arquiteturas monolíticas. Deste modo torna-se possível que as aplicações sejam facilmente escaláveis, resilientes, com alta disponibilidade e principalmente com custos menores.

Desta forma, este projeto visa apresentar uma solução cujo desempenho, escalabilidade e fiabilidade são um fator determinante para os sistemas e plataformas de comércio eletrónico tendo em conta os vários tipos de sistemas, dentro da área, já existentes e presentes na atualidade de várias empresas. Assim, pretende-se que este projeto consiga ter um impacto real e positivo e que permita às empresas disponibilizarem plataformas de venda de produtos *online* seguindo este tipo de arquitetura micro-serviços. Adicionalmente, pretende-se que, com a elaboração deste projeto e respetiva documentação, seja possível transmitir e demonstrar as vantagens das arquiteturas baseadas em micro-serviços a todos os interessados que desejam explorar esta área, que está cada vez mais presente e que vem a ser adotada no desenvolvimento de projetos altamente complexos e com regras de negócio complexas.

1.4. Organização do documento

A presente dissertação divide-se em cinco diferentes capítulos:

- Capítulo 1 - Contextualização: Neste capítulo é feita uma introdução à temática do comércio eletrónico, relatando o seu âmbito e enquadramento, a problemática e desafios que estas plataformas apresentam e os tipos de arquiteturas normalmente utilizados para o seu desenvolvimento.
- Capítulo 2 – Estado da Arte: Neste capítulo é abordada a fundamentação teórica relativamente às arquiteturas monolíticas, arquiteturas micro-serviços, soluções

existentes em diversas áreas diferentes e que assentam sobre arquiteturas micro-serviços.

- Capítulo 3 - Conceptualização do problema/projeto e arquitetura: Neste capítulo é apresentada uma arquitetura micro-serviços enquadrada na área de uma plataforma de comércio eletrónico, abordagens utilizadas para o desenvolvimento da arquitetura, tecnologias utilizadas e padrões de *software* que visam resolver os desafios apresentados pelas arquiteturas micro-serviços.
- Capítulo 4 - Implementação: Neste capítulo é feita uma abordagem acerca das tecnologias e padrões utilizados no desenvolvimento da aplicação, qual o seu enquadramento e problemas que visam resolver.
- Capítulo 5 - Conclusão: Por último, neste capítulo é feita uma reflexão critica acerca de todo o projeto desenvolvido, desde a fase inicial, o estado da arte, problemática do comércio eletrónico, tecnologias utilizadas, padrões de *software* e vantagens de diferentes tipos de arquiteturas. Na conclusão é feito um pequeno resumo do projeto desenvolvido, vantagens e desvantagens encontradas e propostas de trabalho futuro de modo a melhorar a arquitetura proposta.

Capítulo 2 – Estado da arte

2. Introdução

Ao longo deste capítulo, será apresentado o estado da arte relacionado com arquiteturas micro-serviços, problemática do comércio eletrónico, tecnologias usadas neste âmbito e a investigação apresentada ao longo dos anos sobre plataformas cujas arquiteturas estão assentes em micro-serviços.

2.1. Arquitetura monolítica versus arquitetura micro-serviços

Neste subcapítulo, serão abordados os dois tipos de arquiteturas diferentes, que são atualmente as mais utilizadas na construção de aplicações e plataformas, onde são demonstradas as vantagens e desvantagens de cada uma e principalmente as principais conclusões retiradas da utilização de ambas.

2.1.1. Arquiteturas monolíticas

As empresas, ao optarem por uma abordagem monolítica para a construção das suas aplicações, conseguem atingir um rápido desenvolvimento das suas plataformas. Consequentemente, conseguem disponibilizar para o seu público-alvo, a plataforma muito mais rapidamente de forma a iniciarem os seus serviços e respetiva faturação. Neste tipo de arquitetura, todas as funcionalidades estão agregadas e encapsuladas numa só aplicação [6], os módulos que compõem a aplicação não são executados independentemente, tornando este tipo de arquitetura fortemente acoplada. Este tipo de arquitetura permite que a interface do utilizador possa estar inserida juntamente com o servidor responsável por toda a lógica de negócio e armazenamento de dados, tornando estes componentes dependentes e limitados às tecnologias inicialmente escolhidas. A Figura 1 representa um exemplo das camadas que compõem tipicamente as arquiteturas monolíticas.

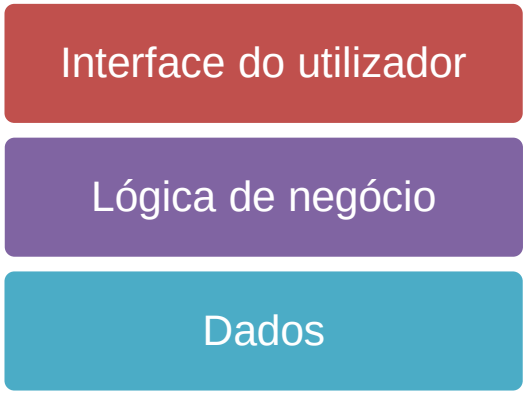


Figura 1 – Exemplo de uma arquitetura monolítica

Uma das maiores vantagens deste tipo de arquitetura, deve-se ao facto de que o desenvolvimento deste tipo de aplicações, numa fase inicial, ser mais simples de pensar, configurar e testar. Estas vantagens permitem uma maior produtividade inicial para as equipas de desenvolvimento, tornando este um ponto principal na escolha que as empresas fazem no momento de decisão e construção das arquiteturas para as suas aplicações. No entanto, ao longo do processo e das etapas de desenvolvimento de *software* e à medida que a aplicação cresce com a adição de dezenas ou centenas de serviços, naturalmente o processo de desenvolvimento desacelera, contrariando uma das principais vantagens inicialmente descritas deste tipo de arquitetura. Com a adição de novas funcionalidades, é necessário um cuidado adicional para perceber o impacto que estas funcionalidades possam ter na aplicação. Devido a este forte acoplamento, todos os componentes existentes necessitam que os testes sejam executados, para perceber o impacto das mudanças introduzidas na aplicação. O tempo de execução dos testes fica cada vez maior e o esforço de adição e correção de testes também é afetado negativamente. Após todo o processo de qualidade feito sobre a aplicação, o processo de *deployment* também sofre de forma significativa, resultando num *downtime* maior da plataforma mesmo que as novas mudanças efetuadas tenham sido muito pequenas. Além do referido, estes tipos de arquitetura requerem um compromisso de longo termo em relação à *stack* tecnológica inicialmente escolhida. A utilização ou necessidade de uma nova tecnologia pode requerer a migração total da plataforma, exigindo um grande esforço e tempo por parte das equipas de desenvolvimento. Adicionalmente, estas plataformas apenas podem escalar verticalmente, através do melhoramento dos recursos presentes na máquina que executam estas aplicações, como o *upgrade* do *CPU*, memória *RAM* disponível e memória física. Este tipo de melhorias ao longo do tempo, fica limitado à inovação tecnológica atual e disponível no mercado, além de que o acesso aos melhores componentes resulta num acréscimo de custos exponencial. Por fim, uma falha ou erro numa determinada funcionalidade da aplicação pode fazer com que toda a aplicação possa ficar inacessível ao público resultando num *downtime* bastante grande e custoso. Até que essa mesma falha seja corrigida, é necessário passar por todo o processo de desenvolvimento, qualidade, testes e de um novo *deploy* total da aplicação até que esteja novamente disponível e acessível ao público, o que representam custos adicionais para a empresa.

Em suma, as arquiteturas monolíticas apresentam várias vantagens e desvantagens na sua utilização. Estas são adequadas para aplicações mais pequenas e menos complexas, permitem um rápido desenvolvimento inicial e disponibilização para o público-alvo. No entanto, estas não são adequadas para aplicações mais complexas e com regras e lógica de negócio muito complexas. Estes tipos de arquiteturas sofrem com a complexidade natural do processo de desenvolvimento que ao longo do tempo, introduzem cada vez mais novas

funcionalidades, tornando o processo de desenvolvimento cada vez mais complexo e lento. Por fim, a escalabilidade fica bastante limitada tendo em conta os custos e a tecnologia disponível dos componentes que permitem o aumento de desempenho.

2.1.2. Arquiteturas micro-serviços

O conceito de arquiteturas micro-serviços devem ser consideravas e devem estar presentes no momento da conceção da arquitetura de uma determinada aplicação. O principal foco deste tipo de arquitetura é centrado na perspetiva da flexibilidade, escalabilidade e confiabilidade. Esta abordagem tem surgido cada vez mais devido à cada vez maior utilização e vantagens da *cloud* para alojar plataformas. Arquitetura micro-serviços pode ser encarada como um novo paradigma para a construção de aplicações através da agregação de serviços muito pequenos, cada um a correr no seu próprio processo e que comunicam através de mecanismos muito leves entre si [6].

A Figura 2 representa um exemplo de uma arquitetura micro-serviços, onde um serviço tem apenas uma única responsabilidade de acordo com o domínio que está inserido e também é responsável pelos seus próprios dados.

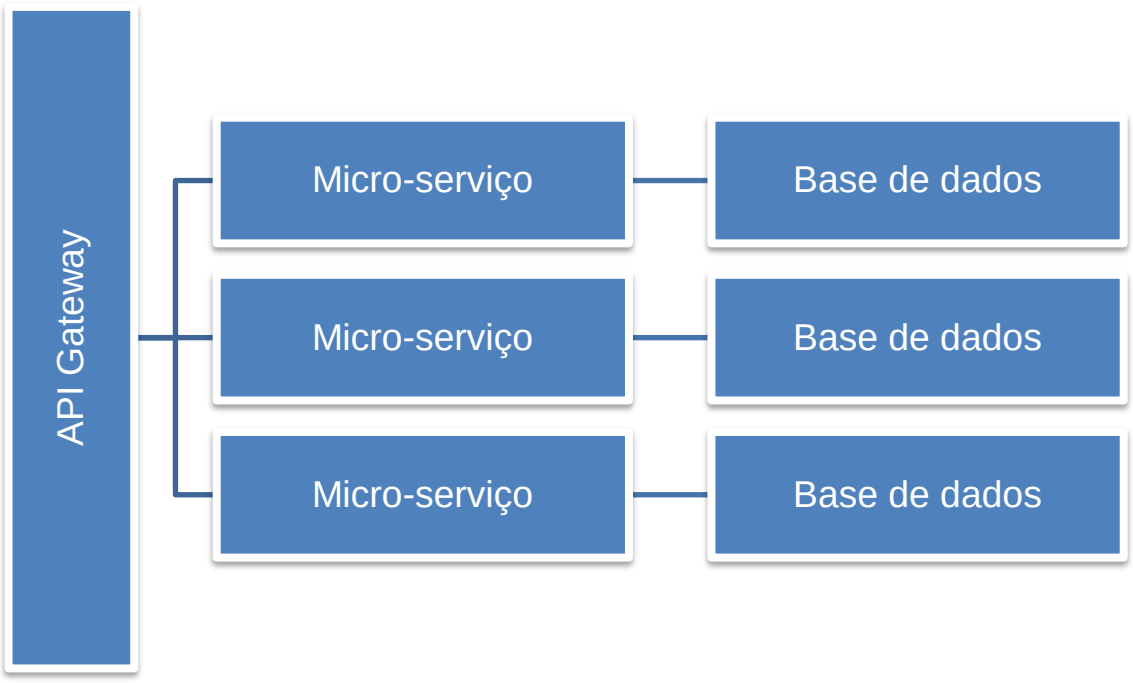


Figura 2 - Exemplo de uma arquitetura micro-serviços

As vantagens deste tipo de arquitetura são diversas e resolvem muitos problemas presentes nas arquiteturas monolíticas. Permitem heterogeneidade na tecnologia ou *framework* no desenvolvimento do microsserviço de forma a solucionar um determinado problema, dando assim um total controlo e liberdade às equipas de escolherem as melhores tecnologias. Como umas das principais características é a decomposição da aplicação em pequenos serviços

totalmente desacoplados, isto permite que a falha num determinado micro-serviço esteja isolado e não é possível comprometer o restante sistema, continuando o resto operacional. A escalabilidade pode ser feita de acordo com a necessidade numa determinada altura, resultante de uma maior afluência por parte dos utilizadores, possibilitando assim às empresas controlarem os custos de alojamento da aplicação. A escalabilidade horizontal permite a adição de novas máquinas designadas para correr novas instâncias do micro-serviço que têm uma maior afluência, não sendo necessário escalar toda a aplicação de uma só vez. Da mesma forma, um micro-serviço pode ser desenvolvido, introduzido novas funcionalidades, correções de erros, testado e efetuado o *deploy* mais rapidamente sem afetar qualquer outro micro-serviço, resultando num *downtime* totalmente localizado e controlado.

Do lado organizacional, as empresas de desenvolvimento, seguindo este tipo de arquitetura, podem organizar as suas equipas mais facilmente através do escalonamento de equipas totalmente focadas para o desenvolvimento de um micro-serviço, garantindo uma boa coordenação, liberdade e foco durante todo o processo de desenvolvimento.

No entanto, estes tipos de arquitetura também acarretam algumas desvantagens que necessitam de atenção. A alta complexidade, quando se atinge um elevado número de micro-serviços, pode ser difícil de gerir, sendo necessário garantir a documentação de todos os serviços desenvolvidos. Alguns micro-serviços podem necessitar de comunicar com outros micro-serviços, para isso é necessário utilizar mecanismos de comunicação assíncrona de modo a evitar o acoplamento de micro-serviços e garantir a sua independência. Um dos problemas resultantes é o aumento da latência e tempo de resposta que pode surgir devido a toda esta comunicação. Para garantir o correto escalonamento dos micro-serviços são necessários mecanismos de descoberta de serviços, à medida que estes são escalados de forma dinâmica e automática.

Em suma, as arquiteturas micro-serviços resolvem muito dos problemas presentes nas arquiteturas monolíticas. Estas devem ser utilizadas em determinados contextos específicos, como áreas de negócio mais complexas, plataformas com elevados número de acessos em simultâneo que necessitam de escalar eficientemente, alta confiabilidade e disponibilidade para os utilizadores. No entanto, são menos adequadas para o desenvolvimento de pequenas aplicações.

2.1.3. Conclusão

Em 2020, *Konrad* e *Wojciech* efetuaram um estudo [7] com o objetivo de comparar em termos de desempenho uma aplicação de comércio eletrónico desenvolvida seguindo as duas

arquiteturas diferentes em estudo: arquitetura micro-serviços e arquitetura monolítica. A comparação, numa primeira fase consistiu em executar 30 mil pedidos em simultâneo e numa segunda fase a execução de 300 mil pedidos em simultâneo. Na primeira fase, a arquitetura monolítica levou a melhor, conseguindo processar cerca de 789 pedidos por segundo em comparação com 600 pedidos por segundo conseguidos pela arquitetura micro-serviços, uma diferença de cerca de 30%. No entanto, na comparação de 300 mil pedidos em simultâneo, a arquitetura micro-serviços apresentou melhores resultados. Esta, conseguiu processar 239 pedidos em simultâneo contra 180 pedidos por segundo efetuados pela arquitetura monolítica, resultando numa diferença de 32% no número de pedidos processados por segundo. Assim, é possível concluir que as arquiteturas micro-serviços são totalmente adequadas em aplicações altamente concorrentes, com altos níveis de acessos por parte de utilizadores e que necessitam de ser escaladas verticalmente.

Estes dois tipos de arquiteturas abordados ao longo deste capítulo apresentam várias vantagens e desvantagens. Para cada contexto e aplicação estas devem ser analisadas para perceber qual o tipo de arquitetura que mais se adequa, o tipo de desenvolvimento necessário, os custos e o tipo de escalabilidade que a aplicação requer.

2.2. Plataformas assentes em arquiteturas micro-serviços

Ao longo deste subcapítulo, serão apresentados vários estudos que incidem principalmente sobre o desenvolvimento e o uso de arquiteturas micro-serviços na construção de aplicações de diferentes ramos, como plataformas de comércio eletrónico, plataformas para cidades inteligentes, plataformas de monitorização de materiais perigosos e algumas plataformas que são mundialmente utilizadas por milhões de utilizadores diariamente como a *Netflix,* o *Spotify* e a *Uber.*

2.2.1. Smart City IoT

Em 2015, Krylovskiy, Jahn e Patti, relatam o design de uma plataforma *Smart City IoT* (*Internet of Things*) [8] através da implementação de uma arquitetura de micro-serviços, cujo principal objetivo é os diferentes *stakeholders* poderem aumentar a eficiência energética de uma cidade. Os autores argumentam que apesar de já terem sido feitos progressos para a padronização na utilização de dispositivos *IoT*, a construção deste tipo de plataformas capazes de se adaptarem a novos padrões e a aplicações continua a ser desafiador. A equipa de desenvolvimento do projeto, contou com pessoas de diferentes áreas (engenheiros de software, engenheiros eletrotécnicos, arquitetos e especialistas em energia), no entanto, a abordagem de uma arquitetura de micro-serviços permitiu que cada um pudesse trabalhar de forma independente e ao mesmo tempo manter compatibilidade total do sistema,

independentemente da tecnologia escolhida por cada um. Em suma, a arquitetura de micro-serviços permitiu uma simplificação na especificação e na implementação dos serviços, porém, existe um aumento na complexidade do sistema distribuído.

2.2.2. Otto.de

Em 2017, os autores Hasselbring e Steinacker, apresentaram um artigo acerca das propriedades das arquiteturas baseadas em micro-serviços que facilitam a escalabilidade, agilidade e confiabilidade presentes numa das maiores plataformas de comércio eletrónico da Europa: *otto.de* [9]. Esta plataforma apresentou lucros na ordem dos 2.5 mil milhões de euros durante o ano fiscal de 2015/2016 e contava com cerca de 1 milhão de visitas por dia. A transição para uma arquitetura baseada em micro-serviços aconteceu em 2011, com o intuito de principalmente melhorar os requisitos não funcionais da plataforma, nomeadamente a escalabilidade, desempenho e tolerância a falhas. Na organização, cada equipa é responsável apenas pelo seu próprio micro-serviço, completamente independente de outros micro-serviços ou equipas. Com a decomposição da plataforma em pequenos micro-serviços, foi implementado o *continuous integration* e *continuous deployment*, o que permitiu aumentar o número de *deployments* por semana (aumento de 40 para 500), e mesmo assim, o número de incidentes reportados continuou bastante baixo (no mesmo nível anterior). Mais, com esta decomposição foi possível a cada equipa monitorizar os micro-serviços pelos quais são responsáveis, permitindo assim perceber, por exemplo, a carga que o micro-serviço está a ser alvo através da monitorização da utilização de *CPU* (*Central Processing Unit*) e o número de pedidos recebidos. Assim, a plataforma *otto.de* reage automaticamente às diferentes cargas de trabalho que possam surgir, ao criar réplicas desses serviços. Em suma, com este exemplo prático, ficou percetível que no caso da *otto.de*, a reimplementação da plataforma numa arquitetura baseada em micro-serviços foi um sucesso ao permitir alcançar escalabilidade, agilidade e confiabilidade do sistema. Também permitiu uma melhor organização das equipas, ao automatizar processos de *deployment* e deteção de falhas durante o desenvolvimento ou até mesmo em ambiente produtivo. No entanto, os autores alertam que manter a consistência, monitorização e tolerância a falhas é muito complexo para um sistema distribuído, pelo que é necessário a existência de uma equipa madura para gerir todos estes serviços.

2.2.3. Plataforma de monitorização de materiais perigosos

Em 2017, Cherradi, Bouziri, Boulmakoul e Zeitouni apresentaram um sistema cujo principal objetivo é a monitorização de materiais perigosos em tempo real, baseado numa arquitetura de micro-serviços [10]. A escolha deste tipo de arquitetura no projeto em questão permitiu organizar o sistema em pequenas peças facilmente identificáveis, uma gestão de dados

descentralizado, heterogeneidade na tecnologia escolhida, ou seja, possibilitou utilizar uma tecnologia que mais se adequa para um determinado problema. Estas características, traduziram-se em escalabilidade e interoperabilidade e também deixam em aberto a possibilidade de um sistema evolucionário onde no futuro seja possível adicionar novas funcionalidades ou capacidades ao sistema sem alterar o comportamento atual.

2.2.4. Plataforma de comércio eletrónico

No ano de 2019, Mohata e Tijare, fizeram um estudo para perceber o impacto que uma arquitetura de micro-serviços tem no desempenho de um *website* de comércio eletrónico [11]. Os autores desenvolveram uma plataforma e-commerce em dois tipos de arquiteturas diferentes: monolítica e micro-serviços, cujo objetivo principal é diminuir ao máximo possível o downtime quando é feito o *deployment* dos serviços através da arquitetura de micro-serviços. No caso da arquitetura monolítica, quando é necessário introduzir novas funcionalidades (que são muito frequentes neste tipo de plataformas onde o crescimento é indispensável e constante), é necessário fazer um novo *deploy* total da aplicação no servidor, o que resulta num *downtime* total da plataforma durante este processo (o que pode levar a perdas monetárias e insatisfação dos clientes). Por outro lado, no contexto da arquitetura de micro-serviços, apenas os serviços afetados é que necessitam de um novo *deploy*, não afetando outros serviços, ou seja, a restante aplicação continuava operacional para os utilizadores. Os autores concluíram que o tempo requerido para fazer um *deploy* de uma aplicação monolítica é maior em comparação com arquitetura de micro-serviços, o que resulta num maior *downtime*.

2.2.5. Plataforma de comércio eletrónico *business to business*

Também no ano de 2019, Wu, Ding e Hou escreveram um artigo em que propõem o desenvolvimento de uma plataforma de e-commerce B2B (*Business to business*) baseada numa arquitetura de micro-serviços após perceberem a escala, o crescimento das transações e do lucro nas plataformas B2B na China durante o ano de 2018 [12]. Com a utilização deste tipo de arquitetura, cada serviço pode ser desenvolvimento de forma independente e cada um tem a sua própria base de dados, o que permite forte isolamento entre serviços, e assim, a possibilidade de alcançar um desenvolvimento e *deployment* ágil. A arquitetura, além dos micro-serviços das unidades de negócio, apresenta algumas características como o registo e descoberta de serviços, *load balancing* (o sistema pode aumentar o número de instâncias de um micro-serviço quando existe um aumento de pedidos, o que permite escalabilidade), *fault tolerance* (utilização da framework *Hystrix* para gerir falhas como por exemplo introdução de *timeouts* ou respostas rápidas aos pedidos) e message driven (utilização do *RabbitMQ* como

message broker para comunicação interna ou externa através de mensagens para transmissão de dados).

2.2.6. Arquitetura micro-serviços e *Domain Driven Design*

Em 2020, Suthendra e Pakereng fizeram uma pesquisa com o objetivo de desenvolver serviços *web* de *e-commerce* utilizando uma arquitetura de micro-serviços [13]. Para o desenvolvimento desta arquitetura, estes abordaram o DDD (*Domain-driven Design*), que foca nos domínios, conceitos, relações entre domínios e processos de negócio existentes, ou seja, usam contextos limitados para identificar serviços numa arquitetura de micro-serviços, e assim, tornar um contexto limitado num micro-serviço. O método de investigação dividiu-se em cinco fases:

1. Análise funcional - identificação e análise dos processos de negócio.
2. Modelação do sistema de acordo com a análise funcional através de UML.
3. Desenho do sistema (desenho dos micro-serviços), descrição da tecnologia utilizada e configuração do sistema (utilização da linguagem de programação *Go*, utilização do padrão de uma base de dados por serviço que mais se adequa e o conceito de containerization utilizando o *Docker* - que permite escalar serviços através da mudança do número de instâncias no *container*).
4. Implementação dos micro-serviços (de acordo com as regras definidas anteriormente).
5. Avaliação do sistema (perceber se o sistema funciona como esperado e se apresenta um bom nível de resiliência).

Os autores concluíram e demonstraram que ao seguirem estes processos, os serviços *web* de *e-commerce* funcionaram como esperado e demonstraram altos níveis de resiliência, ou seja, significa que os serviços tinham baixos níveis de dependência entre serviços e que podem facilmente ser alvos de mudanças. A utilização do *Docker* ajudou no desenvolvimento, implementação e no processo de *deployment*, pois este permite concentrar todas as bibliotecas e dependências num *package*. Por fim, também concluem que as arquiteturas baseadas em micro-serviços permitem flexibilidade e fácil manutenção, ao ser possível utilizar diferentes linguagens de programação e diferentes tipos de base de dados. Mudanças e melhorias nos serviços não afetam outros serviços desde que estes não sejam dependentes entre si, sendo isto essencial, uma vez que, os processos de negócio estão em contínua evolução e estes devem-se adaptar às constantes mudanças.

2.2.7. Netflix

A Netflix é um dos exemplos mais conhecidos na utilização de micro-serviços para a construção da plataforma de *streaming*, sendo uma das empresas pioneiras na utilização e adoção deste tipo de arquitetura para os seus serviços, que são utilizados diariamente por milhões de utilizadores. Em 2008, a falta de apenas um caracter - ";", tornou o website da Netflix totalmente inacessível durante várias horas demonstrando que a arquitetura monolítica permite que a falha num único local afete toda a aplicação tornando a plataforma totalmente indisponível [14]. O acontecimento desta falha, fez com que a Netflix repensasse a arquitetura dos seus serviços, iniciando a migração na direção de uma arquitetura micro-serviços de forma a mitigarem e isolarem potenciais erros que possam acontecer de forma a melhorar a confiabilidade e resiliência da plataforma.

Em 2015, numa conferência, a Netflix revelou a escala do funcionamento dos seus serviços assentes numa arquitetura micro-serviços, como podemos ver na Figura 3.

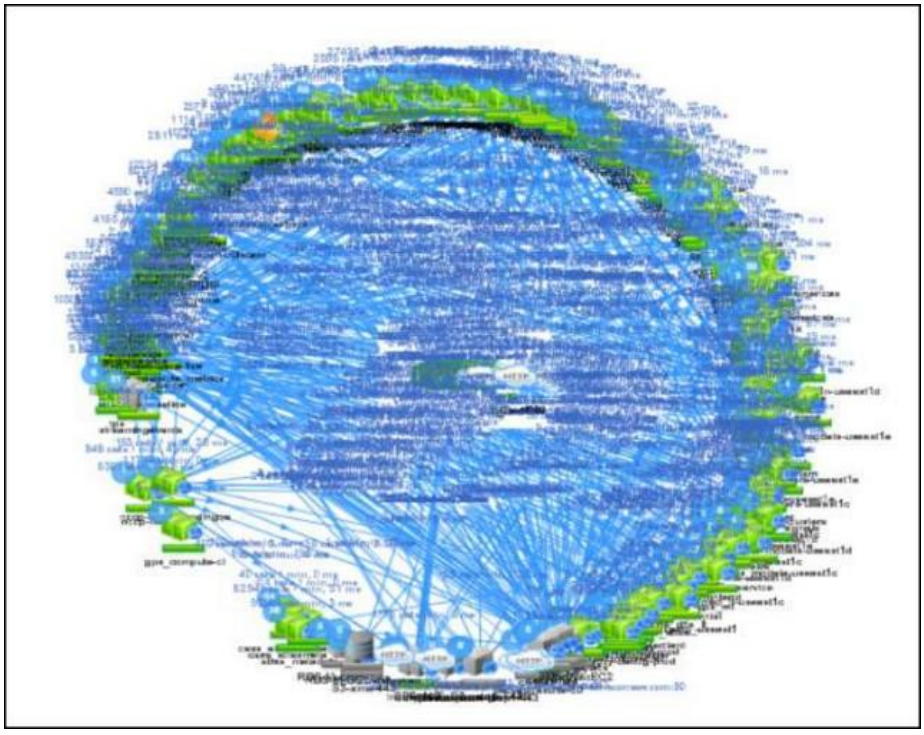


Figura 3 - Arquitetura micro-serviços da *Netflix* [15]

Na altura, contavam com 50 milhões de subscritores, 2 mil milhões de pedidos à *API* por dia, mais de 500 micro-serviços, 30 equipas de engenharia e tudo isto resultava em um terço de todo o tráfego da internet [16].

A utilização de uma arquitetura micro-serviços, permitiu à Netflix muitas vantagens no desenvolvimento da sua plataforma. Desde *deployments* mais rápidos, simples e com possibilidade de reverter mais facilmente, utilização de linguagens ou *frameworks* mais

apropriadas para o domínio em questão, maior resiliência através do isolamento de falhas e uma maior confiabilidade.

2.2.8. Spotify

O *Spotify*, enquadra-se também num dos exemplos mais conhecidos que adota uma arquitetura micro-serviços no desenvolvimento do seu produto. Em 2015 [17], contavam com 75 milhões de utilizadores por mês, operavam em 58 países, adicionavam à sua plataforma cerca de 20 mil músicas novas por dia, apresentavam cerca de 2 mil milhões de *playlists*, como a área de negócio é na área música, enfrentam regras de negócio extremamente complexas e, por fim, atuam num mercado com muita competitividade como é o *streaming* de música.

O desenvolvimento da plataforma contava com 90 equipas distintas, 600 desenvolvedores e cerca de 810 serviços disponíveis que formavam a plataforma. A adoção desta arquitetura permitiu testar, efetuar *deployments* e monitorizar mais facilmente os serviços desenvolvidos e que estão em utilização por parte dos utilizadores. Consequentemente, permitiu que as equipas trabalhassem independentemente do seu serviço com grande autonomia. No caso do *Spotify*, este tipo de arquitetura permitiu a utilização continua de serviços *legacy* à medida que iam sendo substituídos por novos serviços até que os antigos fossem descontinuados.

No entanto, surgiram alguns desafios, como os micro-serviços necessitam de comunicar entre eles, resulta numa maior latência e tempo de resposta para os utilizadores, para colmatar este problema o *Spotify* desenvolveu mecanismos de agregação para diminuir o número de chamadas que o cliente necessita de efetuar. Por outro lado, como as equipas funcionam de forma independente e para combater o risco da existência de trabalho repetido, cada equipa é delegada com requisitos bastantes específicos e claros.

2.2.9. Amazon

A *Amazon* é uma das plataformas de comércio eletrónico mais conhecida e utilizada mundialmente por milhões de utilizadores diariamente. Em 2022, conta com 300 milhões de utilizadores ativos por mês e com cerca de 1.9 milhões de parceiros de venda por todo o mundo. A pandemia da *Covid-19* teve como consequência a adição de 200 mil novos vendedores na plataforma, o que representou um acréscimo de 45% face ao ano anterior [18].

Em 2001 [19], a *Amazon* operava como uma grande arquitetura monolítica, composta por múltiplas camadas e dividida por componentes, no entanto era extremamente acoplada. Este tipo de arquitetura, já com um grande desenvolvimento ao longo dos anos, trouxe várias desvantagens para a *Amazon*. Como existiam centenas de desenvolvedores a trabalhar na

plataforma, as tarefas de resolução de conflitos, *merge* de todo o código, e o processo de transformar uma versão para produtivo tornou o processo extremamente demorado e adicionou muita sobrecarga sobre a *pipeline* de entrega. Toda a *codebase* precisava de passar pelo processo de qualidade, como *build,* casos de teste, e só depois é que poderia passar para produção. Todo este processo fazia com que as funcionalidades só pudessem chegar aos clientes semanas após terem sido implementadas.

Para resolver estes problemas acima mencionados, a *Amazon* decidiu decompor a arquitetura monolítica para uma arquitetura orientada a serviços. Inicialmente, exploraram o código através da análise dos testes unitários funcionais que tinham um único propósito ou funcionalidade. Assim, conseguiram criar funções e serviços que tinham uma única responsabilidade – "*single-purpose functions"*, como por exemplo, criação de uma função cuja única responsabilidade seria de calcular o valor da taxa no momento do pagamento. As "*single-purpose functions"* apenas poderiam comunicar com os restantes serviços através das suas próprias *web APIs*. Desta forma, a imposição desta regra, permitiu à *Amazon* criar um conjunto de serviços completamente desacoplados e que poderiam evoluir à sua medida sem necessidade de coordenação ou depender de outros serviços.

A mudança de uma arquitetura monolítica para uma arquitetura micro-serviços permitiu aumentar a eficiência das equipas da *Amazon*, diminuição da necessidade de processos manuais e redução de tempo desde que uma funcionalidade é implementada até que possa é usada pelos clientes. A Figura 4, apresenta a arquitetura micro-serviços que compõe a plataforma de comércio eletrónico da *Amazon.*

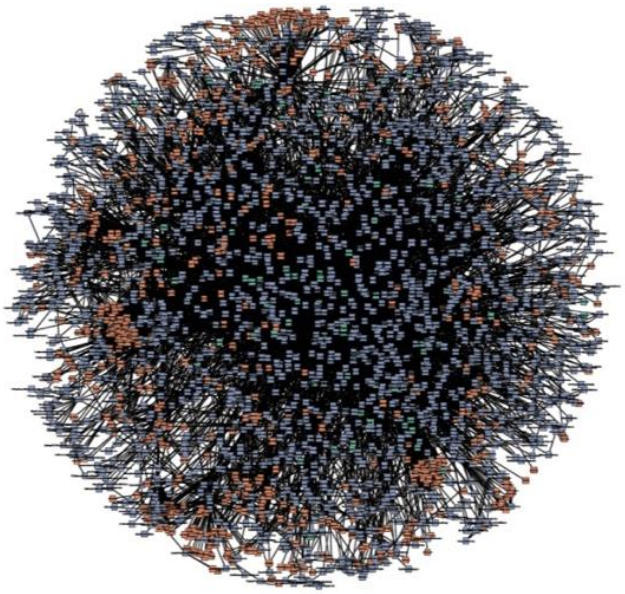


Figura 4 - Estrutura da arquitetura micro-serviços da *Amazon* [15]

2.2.10. Uber

Uber é uma plataforma de prestação de serviços eletrónicos que atua na área do transporte privado. Inicialmente apenas ofereciam os seus serviços e atuavam numa única cidade [20].

Ao longo do tempo e à medida que foram expandindo para outras cidades, começaram a surgir alguns problemas, nomeadamente em termos de escalabilidade. A arquitetura da plataforma era uma arquitetura monolítica onde todas as funcionalidades da aplicação estavam agregadas num único local. Á medida que iam expandindo e adicionavam novas funcionalidades, o processo era cada vez mais complexo devido à necessidade de passar por todo o processo de testes da plataforma e efetuar um novo *deploy* de toda a aplicação. O processo de correção de *bugs* também era bastante complexo devido a toda centralização do código num único repositório.

A transição para uma arquitetura micro-serviços, permitiu simplificar o processo de *deploy* de determinados micro-serviços. Apenas o micro-serviço atualizado é que necessitava de ser *deployed* não afetando todos os outros micro-serviços. A escalabilidade da plataforma foi melhorada, devido à escalabilidade individual de um determinado serviço que está a ser mais utilizado num determinado momento.

2.2.11. Ebay

Ebay é uma das empresas pioneiras que apostou na criação de uma plataforma de comércio eletrónico para a venda de produtos. Inicialmente, todo o processo de compra era a base do sistema de leilões, no entanto, atualmente é utilizado para vender e comprar produtos de preço fixo.

No ano de 2011, iniciaram a transição para uma arquitetura micro-serviços. Na altura contavam com cerca de 97 milhões de utilizadores e cerca de 200 milhões de itens para venda. Ao longo de um dia, a plataforma desempenhava 75 mil milhões de chamadas à base de dados e 250 mil milhões de pesquisas [21].

Esta transição permitiu construir uma plataforma modular e reutilizável o que permitiu reduzir a complexidade em si da plataforma. Assim, foi possível manterem-se competitivos no mercado através do desenvolvimento de novas funcionalidades a um ritmo mais elevado. Consequentemente, foi possível designar equipas próprias para desenvolver determinados serviços, aumentar a produtividade dos desenvolvedores e manter alta disponibilidade da plataforma.

2.2.12. BestBuy

BestBuy é uma empresa de comércio eletrónico de venda de eletrónicos e que atua em vários países como Estados Unidos, Canadá, México e China.

Inicialmente a plataforma *online* de comércio eletrónico foi desenvolvida com recurso a parceiros externos, ou seja, não foi desenvolvida pela própria *BestBuy*. Ao longo dos anos, a plataforma ficou cada vez mais complexa, tornando uma simples mudança num processo extremamente complexo. As *releases* eram planeadas com vários meses de antecedência e muitas vezes eram adiadas. O processo de *deploy* da plataforma necessitava de mais de 60 pessoas e levava mais de 8 horas até ficar concluído. A plataforma era alvo de falhas e ficava impossível de ser usada em alturas de picos de utilização. Para corresponder a estes picos, principalmente em alturas de *black* friday, investiam em infraestrutura para darem resposta e manterem a plataforma estável. No entanto, toda esta infraestrutura era obsoleta nas restantes alturas do ano, tornando o investimento inoperacional durante muito tempo. A mudança para a *cloud* resolveu todo este problema de infraestrutura, podendo assim utilizar recursos consoante o que realmente é necessário. Além disso a plataforma tinha falta de funcionalidades e contava com muitos *bugs* devido a toda a complexidade aqui presente [22].

Em 2010, tomaram a decisão de migrar toda a plataforma seguindo um arquitetura micro-serviços de modo a conseguirem rapidamente adicionar novas funcionalidades à plataforma e responderem às expetativas do mercado. Para esta transição, optaram por o *Strangler Pattern*, ou seja, à medida que os serviços independentes eram criados e estavam prontos para serem usados pelos clientes, o componente correspondente antigo era desligado. A Figura 5 mostra o exemplo do funcionamento deste padrão.

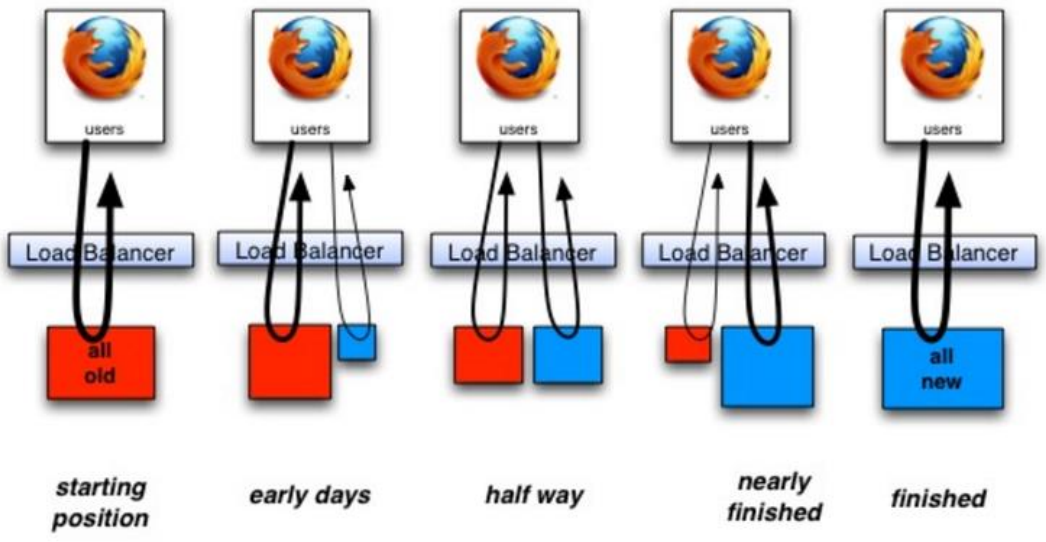


Figura 5 - *Strangler Pattern* [22]

2.2.13. Custos de infraestrutura

O custo da infraestrutura necessária para alojar qualquer tipo de plataforma é um ponto importante no orçamento das empresas. Neste sentido, existem várias opções em que as empresas podem optar por comprar a sua própria infraestrutura e alojarem as plataformas ou podem optar por dar *deploy* na cloud e apenas paga o que utiliza à prestadora do serviço, que aloja a plataforma. O tipo de arquitetura utilizado no desenvolvimento da plataforma pode ter um peso importante no orçamento das empresas.

Em 2016, foi efetuado um estudo acerca do custo da infraestrutura de diferentes três tipos de arquiteturas [23]. Durante esse estudo, foram desenvolvidos três tipos de aplicações. Neste caso de estudo abordamos apenas duas dessas aplicações referidas. A primeira é uma plataforma monolítica com dois serviços *REST* e a segunda é uma plataforma micro-serviços decomposta em dois micro-serviços com serviços *REST*. As aplicações foram ambas *deployed* para a *AWS (Amazon Web Services).* Este estudo contou com três cenários diferentes:

- Cenário 1: 20% dos pedidos incidiam sobre o serviço 1 (S1) e os restantes 80% dos pedidos incidiam sobre o serviço 2 (S2).
- Cenário 2: 50% dos pedidos incidiam sobre o serviço 1 (S1) e 50% dos pedidos incidiam sobre o serviço 2 (S2).
- Cenário 3: 80% dos pedidos incidiam sobre o serviço 1 (S1) e os restantes 20% dos pedidos incidiam sobre o serviço 2 (S2).

Os autores concluíram que durante o período de um mês, a arquitetura micro-serviços obteve melhor desempenho, ou seja, conseguiu processar mais pedidos por minuto ao mesmo tempo com um custo menor. A arquitetura micro-serviços obteve um custo mensal de 390.96 dólares em contraste com 403.20 dólares mensais da arquitetura monolítica. Em termos percentuais, a mesma tecnologia utilizada em dois tipos de arquiteturas diferentes (monolítica e micro-serviços), permitiu uma redução de custos de 9.50% (cenário 1), 13.81% (cenário 2) e 13.42% (cenário 3), a favor do tipo de arquitetura micro-serviços. No entanto, em termos de tempo de resposta, a arquitetura micro-serviços obteve piores resultados no tempo de resposta devido à necessidade de os pedidos necessitarem de passar pelo *gateway* até ao micro-serviço em questão.

2.2.14. Conclusão

Atualmente, muitas empresas como a *Netflix*, *Amazon*, *Spotify, Ebay* e a *Uber,* inicialmente construíram e desenvolveram as suas plataformas seguindo uma arquitetura monolítica. Ao longo dos anos, com o crescimento que estas enfrentaram, naturalmente foram surgindo

problemas devido a esta arquitetura. Nomeadamente, problemas na complexidade das plataformas, processos de qualidade, produtividade dos desenvolvedores, escalabilidade, fiabilidade e os processos de *deploy.*

Para combater todos estes problemas, muitas destas empresas decidiram migrar as suas plataformas para uma arquitetura micro-serviços. O processo de transição necessitou de muitos recursos, no entanto, permitiu resolver imensos problemas que foram identificados ao longo deste capítulo.

As plataformas, agora assentes sobre arquiteturas micro-serviços, permitiram escalar eficientemente os recursos alocados às funcionalidades das plataformas. Desta forma, estas aplicações são alimentadas por diferentes leves e pequenos serviços que comunicam entre si através de diferentes mecanismos transformando este conjunto de serviços numa aplicação. Assim, é possível atingir uma maior eficiência nos recursos utilizados, um maior controlo de custos, maior confiabilidade e resiliência nas suas plataformas [24].

Em suma, o tema das arquiteturas micro-serviços vem cada vez mais a ser falado e a ser utilizada cada vez mais nos últimos anos para o desenvolvimento de plataformas. As empresas cujos serviços estão amplamente solidificados no mercado, como plataformas de comércio eletrónico e *streaming* adotam uma arquitetura micro-serviço. Esta adoção, permitiu resolver imensos problemas que estão relacionados com as arquiteturas micro-serviços. Estes benefícios podem-se traduzir em quatro vertentes diferentes. A produtividade dos desenvolvedores é melhorada durante todo o processo de desenvolvimento até ao processo de *deploy*. Ao mesmo a tempo a empresa consegue disponibilizar uma plataforma robusta, fiável e com alta disponibilidade, assegurando que os seus serviços e respetiva faturação não é afetada. Os clientes têm acesso a plataformas confiáveis e que podem utilizar a qualquer momento. Por último, a utilização de uma arquitetura micro-serviços pode ajudar a diminuir os custos inerentes ao alojamento da plataforma.

Capítulo 3 - Conceptualização do problema/projeto, arquitetura e tecnologias

3. Introdução

Ao longo deste capítulo, será feita uma abordagem sobre a metodologia utilizada para o desenvolvimento da solução, nomeadamente a abordagem *Domain Drive Design (DDD),* contextos limitados e a respetiva arquitetura seguida e desenvolvida para a criação de uma plataforma inserida no domínio do comércio eletrónico.

3.1. *Domain Driven Design*

Domain Driven Design (DDD), é um método orientado a modelos muito utilizado principalmente para obter conhecimento de um determinado domínio que é relevante para a conceção de um determinado *software* [25]. Desta forma, utilizando uma abordagem *DDD*, é possível conceber e implementar sistema através da agilidade e colaboração entre os especialistas do domínio e os desenvolvedores de *software*. O principal objetivo deste modelo é promover a possibilidade de decompor domínios de negócios em contextos concretos, permitindo a associação destes contextos a micro-serviços funcionais com capacidades de negócio distintas.

3.2. Contextos limitados

O conceito de contextos limitados está diretamente relacionado com o DDD, isto é, um contexto limitado é um limite dentro de um determinado domínio de onde esse mesmo domínio é aplicado [26]. Cada contexto tem a sua própria responsabilidade muito bem definida. Na construção de micro-serviços, as utilizações de contextos limitados permitem definir mais facilmente as responsabilidades e funções que cada micro-serviço deve fornecer e resolver.

3.3. Arquitetura

Para a conceção e desenvolvimento da arquitetura micro-serviços direcionada para uma plataforma de comércio eletrónico, foram identificados os domínios e os contextos limitados apresentados na Tabela 1.

Domínio	Contextos Limitados	Descrição
Utilizadores	AccountService Authentication	Gerir e autenticar os utilizadores da plataforma.

Itens	ItemService	Gerir os itens presentes na plataforma.
Compras	OrderService OrderHistoryService CartService	Gerir as ordens de compra, histórico e carrinho de compras.
Pagamento	PaymentProcessorService PaymentValidatorService	Processamento e validação do pagamento de compras.
Avaliações	ReviewService	Avaliações de produtos adquiridos.
Email	EmailService	Notificar utilizadores sobre o estado das compras efetuadas.

Tabela 1 - Domínios e contextos limitados definidos para uma plataforma de comércio eletrónico.

Após a definição dos domínios e contextos limitados acima mencionados, foi possível conceber uma arquitetura com os determinados domínios e micro-serviços definidos, em 12 micro-serviços diferentes, como apresentado na Figura 6.

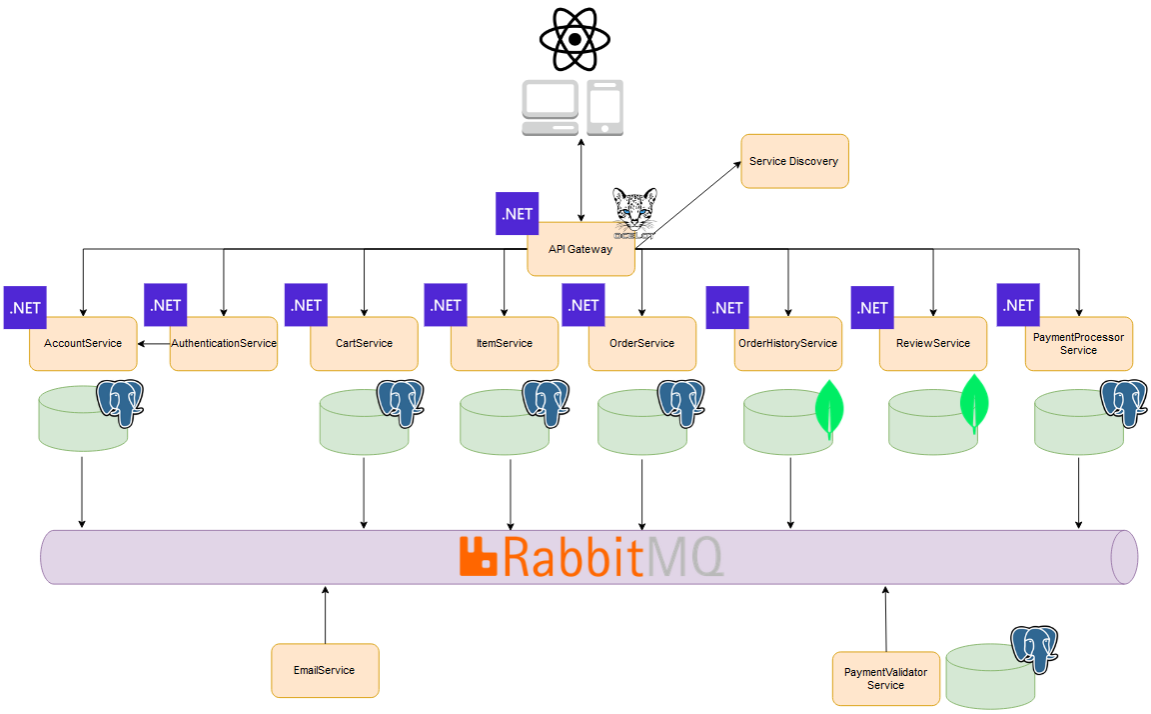


Figura 6 - Arquitetura micro-serviços de uma plataforma comércio eletrónico

O *API Gateway* é o único ponto de entrada de toda a plataforma. Todos os *endpoints* dos micro-serviços que compõem a aplicação estão todos registados aqui. Assim, após cada pedido por parte dos clientes, o *API Gateway* é responsável por redirecionar esses mesmo pedidos para o micro-serviços em questão. À medida que os micro-serviços são escalados, são instanciadas novas instâncias de um micro-serviço. Logo, é necessário um mecanismo para que seja possível aceder a esses novos serviços. A existência do *service discovery* permite resolver este problema, possibilitando os micro-serviços declararem o seu funcionamento e respetiva informação para que possam ser acedidos.

Os micro-serviços *AccountService* e *AuthenticationService*, estão essencialmente responsáveis pela gestão de utilizadores. Os dados são persistidos em base de dados relacional. O serviço de autenticação é responsável por gerar o *token* que serve por autenticar e autorizar ações por parte dos utilizadores.

Os processos de compra são geridos pelos micro-serviços *OrderService*, *OrderHistoryService* e *CartService.* O micro-serviço *CartService* é responsável pela gestão do carrinho de compras de um utilizador (adicionar, remover itens). *OrderService* é responsável pela gestão de todo o processo de compra e persiste todos os dados necessários inerentes a um processo compra. Por último, o micro-serviço *OrderHistoryService* é responsável por gerir o histórico de compras da plataforma, onde cada utilizador pode consultar todas as compras efetuadas.

O micro-serviço *ItemService* é responsável por toda a gestão dos itens disponíveis na plataforma, como a adição de itens, remoção de itens e outras operações básicas como a procura ou filtro por determinados itens.

A gestão de pagamentos é feita pelos micro-serviços *PaymentProcessorService* e *PaymentValidatorService.* O primeiro é responsável por receber o pedido de pagamento e armazenar as informações que deram início ao processo de pagamento e o respetivo estado. O segundo é responsável por verificar se um determinado utilizador tem fundos suficientes para efetuar o pagamento com sucesso.

O micro-serviço *ReviewService* tem como responsabilidade de apenas gerir e armazenar as avaliações que os utilizadores podem efetuar após a compra de um produto.

Por último, o micro-serviço *EmailService* tem como propósito notificar os utilizadores acerca do estado de compra de uma determinada ordem de compra.

Todos os micro-serviços, cujas implementações necessitam de armazenar dados, têm uma base de dados própria, ou seja, cada um apenas gere a sua própria base de dados. A

utilização de um *message broker* permitiu assegurar a comunicação assíncrona entre os diferentes micro-serviços.

A criação de um micro-serviço por cada contexto limitado permitiu uma melhor organização da plataforma. Cada serviço tem uma responsabilidade específica, ou seja, não existe qualquer dependência entre diferentes serviços. Estes podem ser escalados independentemente, qualquer atualização apenas afeta o micro-serviço em questão e a falha num local não afeta o comportamento da plataforma como um todo.

3.4. Tecnologias utilizadas

3.4.1. *.NET e ASP.NET*

.NET é uma *framework open-source* desenvolvida pela *Microsoft*, transversal a muitos sistemas operativos e que permite desenvolver e construir diferentes tipos de aplicações [27]. Adicionalmente, possibilita aos utilizadores escolher diferentes linguagens de programação que mais se adequam para um determinado objetivo específico, como *Visual Basic*, *F#* e *C#*, sendo este último a mais utilizada e conhecida atualmente pelos desenvolvedores. Esta *framework* além de fornecer bibliotecas transversais aos sistemas operativos também fornece bibliotecas mais específicas concebidas para um determinado sistema operativo.

Para o desenvolvido desta plataforma, os micro-serviços foram trabalhados e contruídos utilizando a framework *ASP.NET* [28]. Esta *framework*, estende a *framework* .NET e contém bibliotecas específicas para o desenvolvimento de aplicações *web*, como *APIs REST* que facilitam o desenvolvimento de micro-serviços.

3.4.2. *PostgreSQL*

PostgreSQL é uma base de dados relacional, *open-source* e que usa a linguagem *SQL* para manipular e armazenar dados [29]. Atualmente conta com mais de 30 anos de desenvolvimento, tornando-se um sistema de gestão de base de dados bastante maduro no mundo do *software.*

No contexto deste projeto, foi escolhido para o armazenamento de dados de forma estrutural e relacional, como os dados dos utilizadores, registo de ordens de compra e registo de pagamentos.

	payment_id [PK] integer	order_id integer	user_email text	payment_status text	total_paid double precision	created_date timestamp without time zone	updated_date timestamp without time zone
1	1	1	email@em...	SUCCESS	999.98	2022-09-03 18:50:25.518137	2022-09-03 18:50:27.257216
2	2	2	email@em...	SUCCESS	1259.97	2022-09-03 18:51:09.826282	2022-09-03 18:51:09.919243
3	3	4	email@em...	FAIL	0	2022-09-03 18:52:20.976514	2022-09-03 18:52:21.103962

Figura 7 - Tabela armazenada em *PostgreSQL*

3.4.3. *MongoDB*

MongoDB [30] é uma base de dados não relacional – *NoSQL*, orientada a documentos e que suporta o armazenamento de dados no formato de *JSON (JavaScript Object Notation).* Este tipo de base de dados permite a utilização de um modelo de dados muito flexível podendo evoluir ao longo do tempo sem afetar os documentos anteriormente guardados mesmo após mudanças significativas no modelo original e presente na base de dados.

Adicionalmente, este tipo de base de dados é muito rápido na manipulação de dados, sendo adequada principalmente para a manipulação de grandes quantidades de dados. Mais, o MongoDB, permite facilmente escalar horizontalmente através da replicação de base de dados distribuindo cargas de trabalho por diferentes máquinas. A Figura 8 apresenta um exemplo de uma estrutura guardada em base de dados *MongoDB* no formato *JSON.*

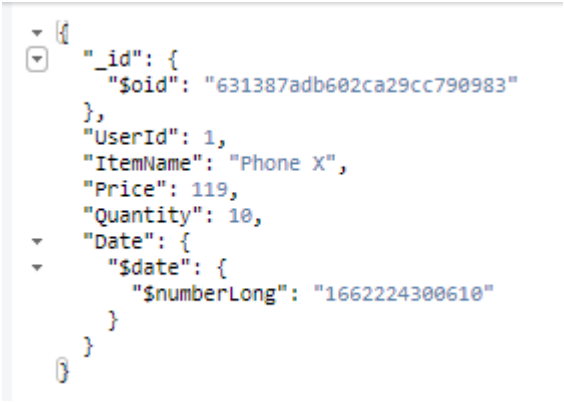


Figura 8 - Documento *JSON* armazenado em *MongoDB*

3.4.4. *RabbitMQ*

RabbitMQ [31], conhecido como um *message broker,* é um *software open-source* que implementa um mecanismo avançado de troca de mensagens - *Advanced Message Queuing Protocol* (*AMQP)*. Este *software* permite a troca de mensagens de forma assíncrona entre micro-serviços, permitindo a eliminação de dependências e acoplamento entre diferentes micro-serviços.

Esta ferramenta, permite aos desenvolvedores implementar um sistema de troca de mensagens de diferentes formas. No caso deste projeto, foi utilizado o tipo *Fanout Exchange* para a comunicação entre micro-serviços, cuja representação visual pode ser vista na Figura 9.

Este mecanismo funciona como *Publish and Subscribe*. O produtor da mensagem (*Producer* ou *Publisher)* publica uma mensagem numa determinada *Exchange* e esta é responsável por transmitir as mensagens para as *queues* que estão vinculadas a esta mesma *Exchange*. Os subscritores (*Subscribers* ou *Consumers*) estão a escuta de mensagens diretamente nas *queues* que estão vinculadas a uma determinada *Exchange*, sendo que esta, pode transmitir mensagens apenas para uma ou várias *queues*.

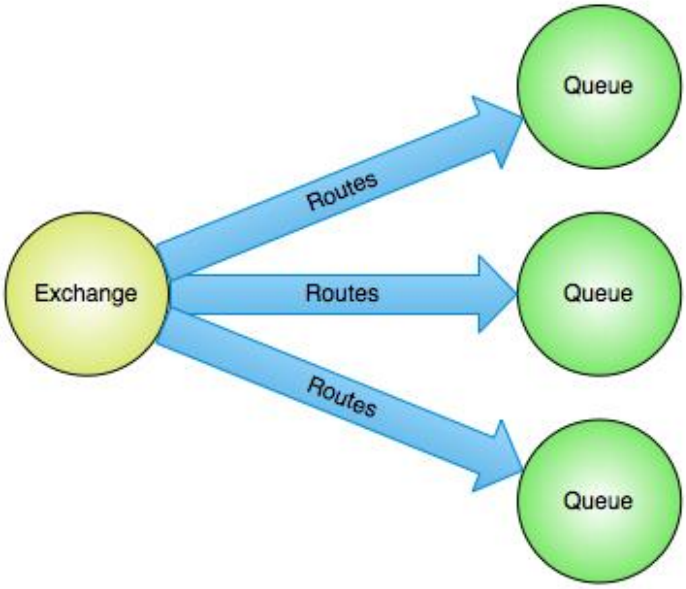


Figura 9 - *Fanout exchange* [32]

3.4.5. Docker

Docker é uma plataforma que permite virtualizar *containers*, ou seja, é uma máquina virtual muito leve em termo de uso de recursos e que permite aos desenvolvedores construir *containers* cujo objetivo é correr as aplicações desenvolvidas.

Umas das principais vantagens, é a possibilidade de aproximar um ambiente produtivo com o ambiente de desenvolvimento, pois todas as dependências necessárias para correr a aplicação, como o código fonte, dependências externas, bibliotecas e o *hardware,* são todas colocadas num *container* permitindo a existência de um ambiente estável e consistente. Igualmente, o processo de testes é mais fiável e estável devido a essencialmente correrem-se testes numa plataforma cujo ambiente é igual ou muito idêntico ao ambiente de produção

onde a aplicação irá estar alojada e utilizada. Um estudo feito pela *IBM* em 2014, descobriu que um *container* é cerca de 26 vezes mais rápido do que uma máquina virtual [33].

No contexto deste projeto, *Docker* foi utilizado para encapsular os diferentes micro-serviços desenvolvidos num *container* de forma a serem agnósticos em termos de ambiente e independentes.

A Figura 10 mostra uma visão geral da arquitetura e funcionamento que constitui o *Docker.*

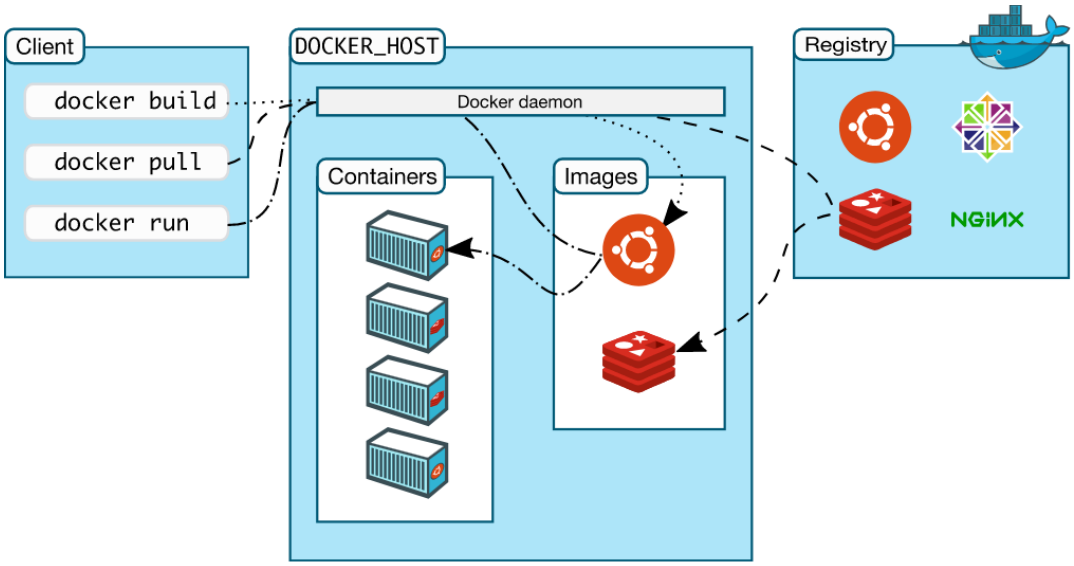


Figura 10 – Arquitetura e funcionamento do *Docker* [34]

3.4.6. *Kubernetes*

Kubernetes é uma ferramenta *open-source* de orquestração de *containers* que automatiza e agiliza o processo de *deploy*, gestão e a escalabilidade de aplicações que estão alojadas em *containers* [35]. Existem alguns conceitos importantes e essenciais para utilização desta ferramenta:

- **Pods**: Um *pod* [36] é um grupo de um ou mais *containers* que partilham essencialmente recursos como armazenamento, recursos de rede ou ficheiros de configuração.
- **Deployment**: É um recurso que permite atualizações declarativas para as aplicações. Permitem replicar *pods*, escalar ou até mesmo reverter para estados anteriores [37].
- **Service**: Permite expor uma aplicação, que corre como um conjunto de *pods*, como um serviço de rede. Assim é possível comunicação entre *pods* dentro de um *cluster* [38].
- **Volume:** Permitem garantir a persistência de dados mesmo quando um *pod* deixa de funcionar ou é destruído[39].

O processo de escalar aplicações pode ser facilmente configurado pelos utilizadores através da definição de algumas métricas, como o uso de memória *RAM*, utilização de CPU ou utilização de memória em disco. Este tipo de métricas permite definir limites de quando um novo *container* é *deployed*, como por exemplo, definir que uma nova instância de um micro-serviço é lançada sempre que a utilização do processador de uma determinada máquina atinge um determinado valor. A Figura 11 mostra os componentes de um *cluster* do *Kubernetes* [40]. Cada *node* representa um *container* onde uma aplicação ou um componente é executado.

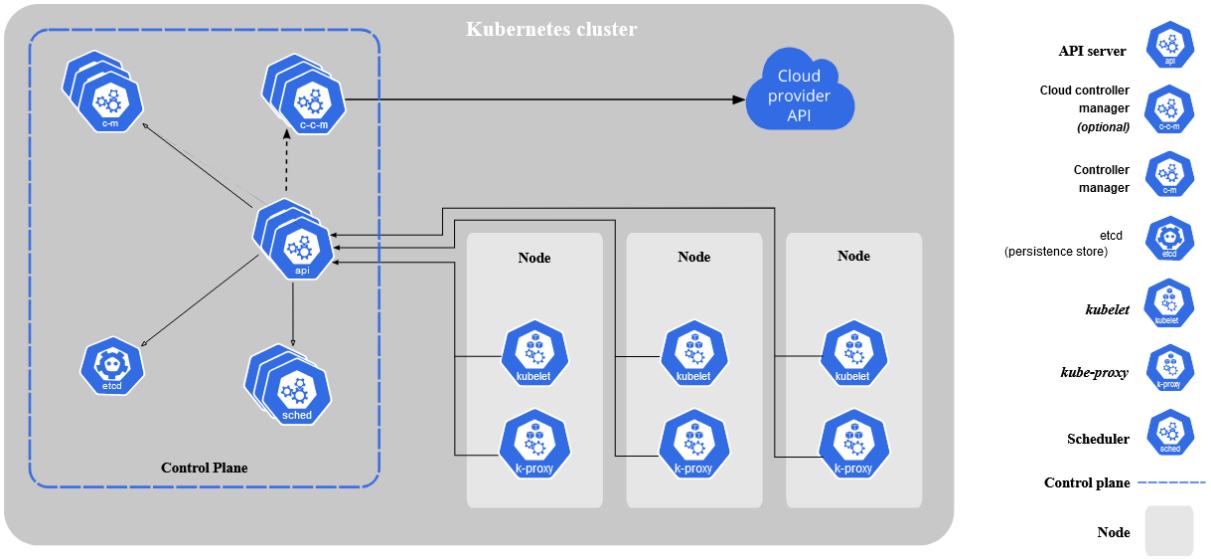


Figura 11 – Exemplo de um cluster do Kubernetes

3.4.7. *React*

React [41] é uma *framework open*-source desenvolvida pelo *Facebook* cujo principal objetivo é o desenvolvimento de interfaces gráficas para a *Web*. Permite o desenvolvimento de interfaces do utilizador com base numa divisão por componentes, tornando o processo de desenvolvimento mais simples e ágil.

No contexto desde projeto, foi a *framework* escolhida para o desenvolvimento do *Frontend.*

3.4.8. *Moq*

A *framework Moq* [42] é normalmente utilizada no desenvolvimento de testes unitários. Esta, permite gerar objetos falsos que se comportam da mesma forma que os objetos reais, de forma a isolar componentes que realmente pretendemos testar. Assim todas as dependências necessárias para testar um determinado componente são lidadas por esta *framework*, tornando o processo de testes mais fácil e rápido.

3.5. Padrões de *software* utilizados no desenvolvimento

3.5.1. *API Gateway*

Um *API Gateway* é um único ponto de entrada para o acesso a várias *APIs*. Este padrão é responsável por fornecer o acesso às *APIs* conforme as necessidades de um determinado utilizador. Por exemplo, diferentes utilizadores com diferentes capacidades e velocidades de rede podem necessitar de acesso a diferentes *APIs* que mais se adequam ao utilizador, como o acesso a recursos mais ou menos intensivos (acesso a imagens ou acesso a apenas texto) [43].

Adicionalmente, a utilização deste padrão possibilita definir, centralizar e registar num único local as diferentes *APIs* que formam uma plataforma decomposta em micro-serviços tornando esta gestão de rotas muito mais simples, pois do lado do cliente apenas é necessário saber o endereço do *API Gateway* para utilizar a aplicação. Também é possível definir a autenticação do utilizador, permitindo autenticar e autorizar os clientes nas diferentes rotas existentes.

A Figura 12 exemplifica o padrão *API Gateway.*

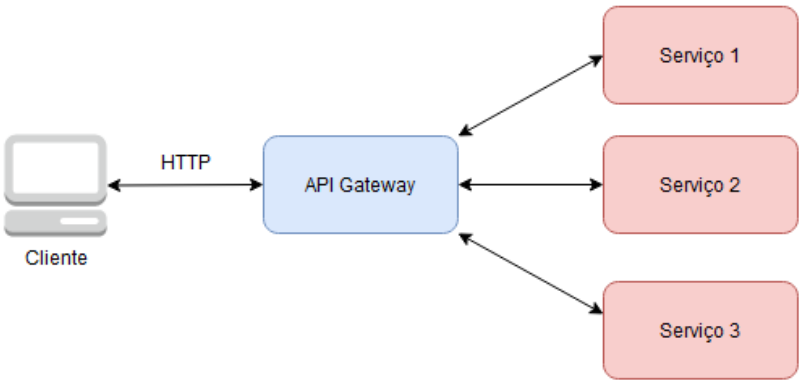


Figura 12 - *API Gateway*

O *API Gateway* é facilmente escalável através da adição de mais instâncias deste serviço. Desta forma, mesmo em situações de elevados pedidos por parte dos utilizadores, não permite que haja um *bottleneck* neste ponto fulcral e único de entrada da aplicação.

No desenvolvimento deste projeto foi utilizado a biblioteca *Ocelot*, que está preparada e concebida para ser utilizada junto com a *framework ASP.NET.*

3.5.2. *Service Discovery*

Como os micro-serviços podem ser replicados e disponibilizados na *cloud*, é necessária a existência de um mecanismo para a descoberta desses mesmos serviços. Para isso é

necessária a existência de um *service registry*, que é um serviço que pode ser usado por componentes para obter informações de ligação a outros componentes. Os micro-serviços publicam as suas localizações enquanto os clientes descobrem os serviços através do acesso ao *service registry* [43]. A Figura 13 demonstra o funcionamento deste padrão.

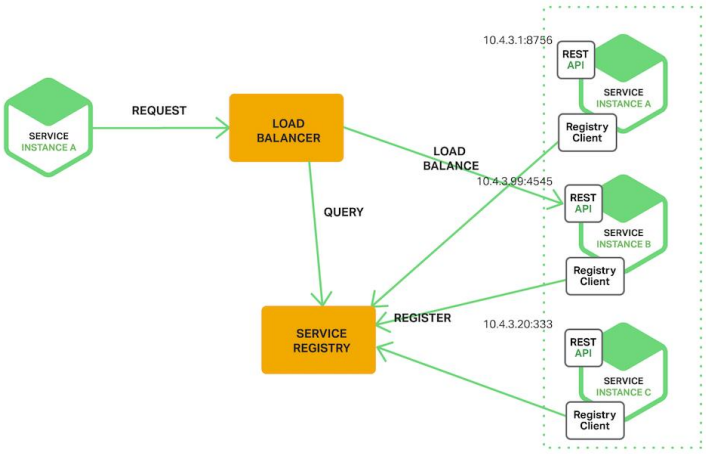


Figura 13 - *Service Discovery* [44]

3.5.3. *Asynchronous Messaging*

O protocolo de comunicação síncrono fornecido pelo mecanismo *REST* é simples e fácil de utilizar, no entanto, em certas situações pode não ser o mais adequado. O padrão de mensagens assíncrono, é útil quando é necessário o processamento de grandes volumes de dados e quando não é esperada uma reposta imediata. Desta forma é possível colocar as mensagens, por exemplo, numa fila para que sejam consumidas quando realmente forem necessárias, ou seja, permite que o fluxo do programa continue sem a existência de um bloqueio pela espera da resposta [45]. A Figura 14, exemplifica este padrão, onde existem os *publishers* que publicam mensagens para uma *queue* e os *subscribers* que estão vinculados a uma determinada *queue* e consomem as mensagens.

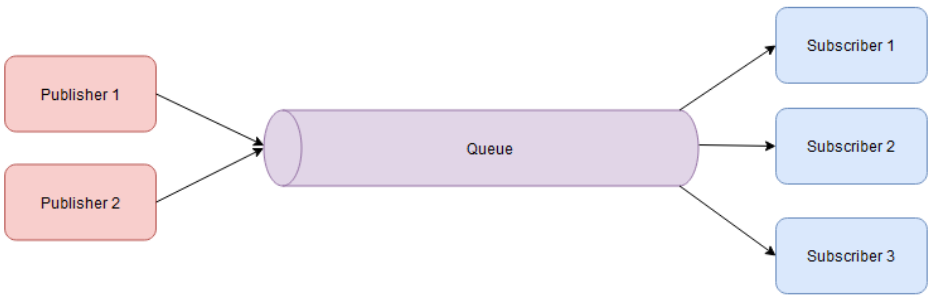


Figura 14 - *Asynchronous Messaging*

3.5.4. *Circuit Breaker*

Circuit Breaker [46] é um padrão de *software* muito utilizado em micro-serviços no contexto de comunicação síncrona. Para garantir a tolerância a falhas e a estabilidade de uma plataforma, é necessário garantir que, quando um micro-serviço falha ou não responde, o comportamento do sistema não é afetado.

Este padrão de *software* permite resolver este tipo de problemas através da introdução de *timeouts* nas tentativas de comunicação. Caso várias tentativas de comunicação falhem um "circuito" é aberto que impede tentativas de comunicação adicionais. Assim, ao reduzir-mos o número de pedidos feito quando um micro-serviço é incapaz de responder, a carga exercida sobre esse micro-serviço é diminuída. Ao fim de um tempo predefinido, o "circuito" volta a fechar e permite novas comunicações. Na eventualidade do micro-serviço continuar sem responder, o circuito volta a abrir e assim sucessivamente. Este funcionamento é repetido até que o micro-serviço esteja disponível e possa responder aos pedidos efetuados. A Figura 15 exemplifica o funcionamento deste padrão.

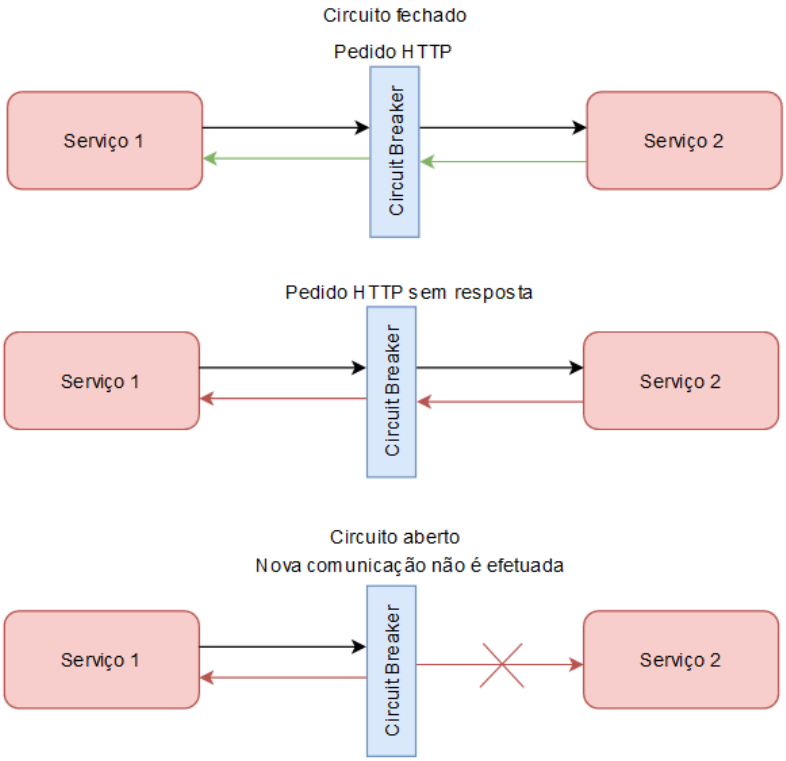


Figura 15 - Padrão *Circuit Breaker*

3.5.5. *Health Monitoring*

O padrão *Health Monitoring* [47] permite monitorizar o estado de funcionamento dos micro-serviços. Permite perceber se um micro-serviço está operacional e é capaz de responder a novos pedidos ou perceber o estado da conexão desse micro-serviço a base de dados.

Na eventualidade de um micro-serviço não estar operacional, o *load balancer* e o *service discovery* têm acesso ao estado de funcionamento desses serviços, de forma a não encaminhar pedidos para esse mesmo micro-serviço ou para poderem lançar novas instâncias desses serviços inoperacionais. A utilização deste padrão permite atingir maior confiabilidade.

3.5.6. Base de dados por serviço

Este princípio pressupõe a utilização de uma base de dados única por serviço [48]. No entanto, este padrão pode ser alargado a três formas distintas de implementação.

- Tabelas privadas por serviço – Cada micro-serviço é reponsável por uma ou mais tabelas, sendo este o único permitido a manipular os dados dessas mesmas tabelas. No entanto, o servidor de base de dados é partilhado por vários serviços.
- *Schema* por serviço – Cada micro-serviço detém um *schema* próprio e que é privado e acessível por esse mesmo serviço. Neste caso, o servidor de base de dados também pode ser partilhado por vários serviços.
- Servidor de base de dados por cada serviço – Esta utilização pressupõe que cada micro-serviço tem ligação a um servidor de base de dados privado e próprio.

Das três abordagens, a primeira tem melhor resultados em termos de consumo de recursos [49]. A Figura 16 exemplifica este padrão.

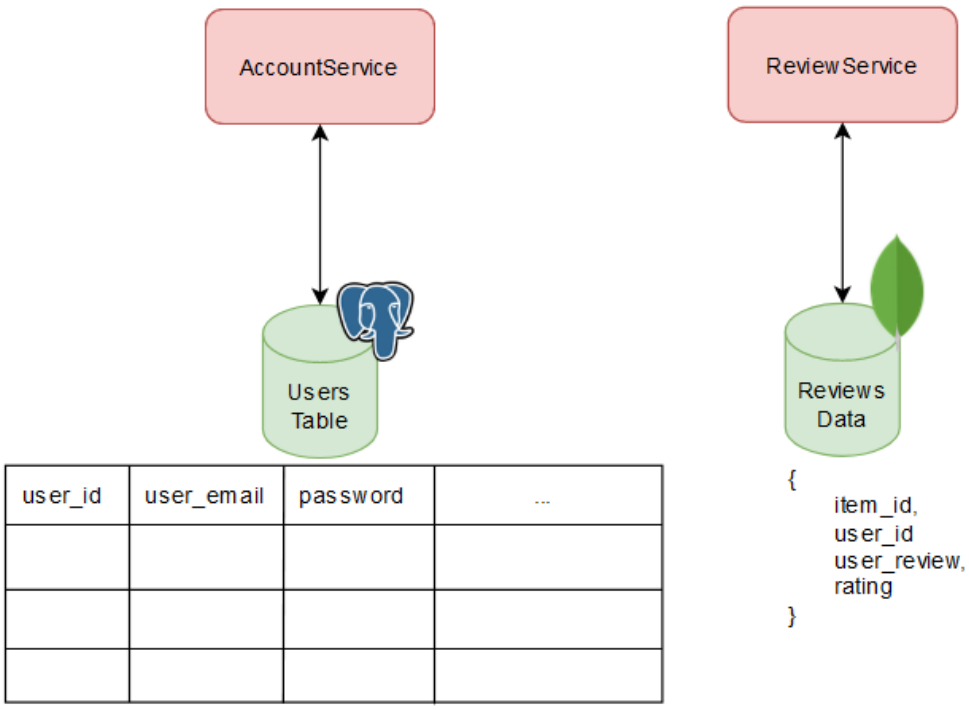


Figura 16 – Padrão base de dados por serviço

Capítulo 4 – Implementação

4. Introdução

O objetivo deste capítulo é apresentar os micro-serviços que constituem a plataforma de comércio eletrónico e respetivos padrões de *software* utilizados no seu desenvolvimento.

4.1. Micro-serviços implementados

Nesta secção serão apresentados de forma detalhada a implementação dos micro-serviços a um nível técnico mais profundo, as funcionalidades acessíveis e os *endpoints* disponíveis e respetiva descrição.

4.1.1. *AccountService*

O micro-serviço é *AccountService* é responsável pelas operações relacionadas com a gestão de utilizadores. As operações são disponibilizadas através de *endpoints REST* e respetivas descrições estão apresentas na Tabela 2.

Método	Caminho	Descrição
GET	/user/{email}	Obter informações sobre um utilizador através do endereço de email.
POST	/user/create	Registo de um utilizador na plataforma.
PUT	/user/update	Atualização da palavra-passe de um utilizador. Contem restrições de pelo menos oito caracteres, um símbolo e um número.
DELETE	/user/delete	Permite apagar o registo de um utilizador da plataforma
POST	/user	Receber e verificar dados de um utilizador (endereço de email e palavra-passe) e verificar se os dados fornecidos correspondem com as credenciais guardadas na base de dados.

Tabela 2 - *Endpoints* do micro-serviço *AccountService*

Os dados dos utilizadores são armazenados numa base de dados relacional, neste caso o sistema de gestão de base de dados (SGBD) *PostgreSQL*. O armazenamento e/ou consulta de dados inerentes a utilizadores da plataforma como, por exemplo, para efeitos de *login*, é exclusivamente executado por este micro-serviço. As credencias são guardadas em base de dados de forma encriptada.

4.1.2. *AuthenticationService*

O micro-serviço *AuthenticationService* tem a única responsabilidade de autenticar um utilizador na plataforma. Este apresenta apenas um único *endpoint* como descrito na Tabela 3.

Método	Caminho	Descrição
POST	/authentication/login	Autenticar um utilizador através do endereço de email e palavra-passe

Tabela 3 - *Endpoints* do micro-serviço *AuthenticationService*

Este micro-serviço comunica com o micro-serviço *AccountService*, descrito no ponto 4.1.1, através de *REST*. Como o micro-serviço *AuthenticationService* não possui acesso à tabela que contém as informações sobre os utilizadores, de forma a verificar se as credencias de acesso introduzidas são válidas ou registadas na plataforma, existiu a necessidade da introdução de comunicação síncrona entre estes dois micro-serviços. Caso as informações introduzidas sejam válidas (email registado na plataforma e palavra-passe correspondente), o micro-serviço *AuthenticationService* é responsável por gerar um *token JWT* cuja função passa por autenticar e autorizar operações de um utilizador na plataforma.

Neste micro-serviço, foi utilizado o padrão *circuit breaker* de modo a assegurar uma maior confiabilidade da plataforma. No caso do micro-serviço *AccountService* estar indisponível ou com uma elevada sobrecarga de pedidos, este padrão permite limitar o número de pedidos *HTTP* durante um determinado tempo de modo a diminuir a sobrecarga já presente. A Figura 17, representa visualmente a comunicação síncrona entre estes dois micro-serviços.

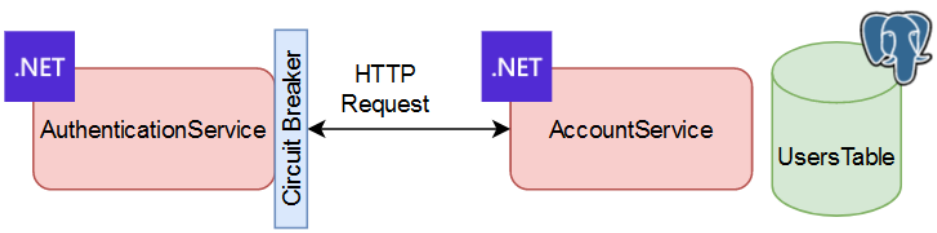


Figura 17 – Comunicação síncrona entre *AuthenticationService* e *AccountService*

4.1.3. *ItemService*

O micro-serviço *ItemService,* tem como responsabilidade a gestão de itens da plataforma. Os *endpoints* e respetiva descrição encontram-se descritos na Tabela 4.

Método	Caminho	Descrição
GET	/item/{itemName}	Obter informações sobre um item.
GET	/item/search	Procurar itens através de uma *string*.
POST	/item/create	Criar item.
DELETE	/item/delete/{itemName}	Apagar item da plataforma através do nome.
GET	/item/all	Obter uma lista de todos os itens da plataforma.
PUT	/item/update	Atualizar *metadata* de um item.
PUT	/item/update/price	Atualizar o preço de um item.

Tabela 4 - *Endpoints* do micro-serviço *ItemService*

As atualizações de preço nas plataformas de comércio eletrónico são muito frequentes, principalmente em alturas de saldos e descontos. As adições de produtos ao carrinho de compras, por parte dos utilizadores, são muito frequentes e permitem mais tarde facilitar a visualização da intenção de compra e tornar mais rápido esse mesmo processo.

A atualização de preço de um determinado item, por parte do administrador da plataforma, aciona o mecanismo de atualização do preço de todos os carrinhos de compras da plataforma. As informações acerca da atualização de preço, como o identificar do item e o novo preço, são publicadas numa *queue* "*PriceUpdate"*, como demonstrado na Figura 18. Desta forma, todos os utilizadores, que tenham um produto no carrinho cujo preço foi alvo de atualização, tem sempre acesso ao preço mais recente praticado pela plataforma.

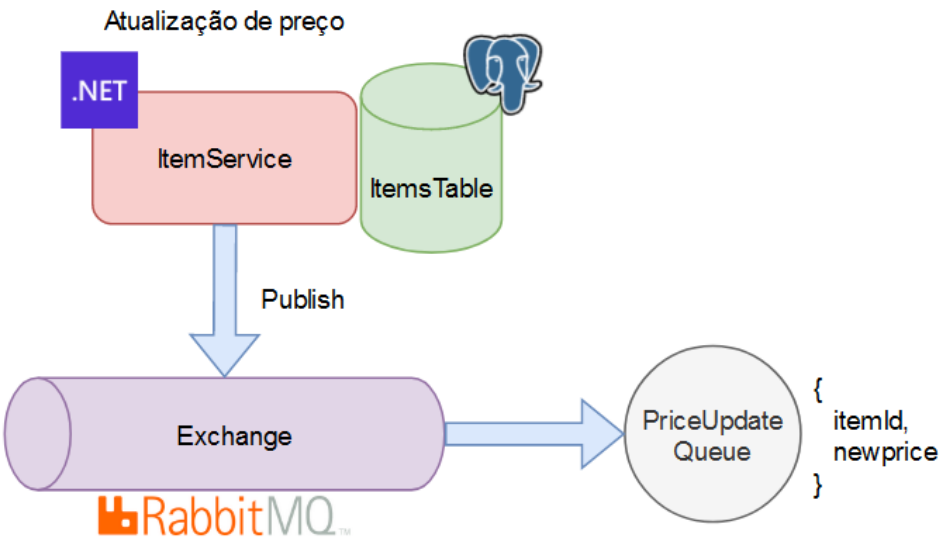


Figura 18 – Atualização assíncrona do preço de um produto

Por outro lado, este micro-serviço, como é responsável pela gestão do inventário da plataforma como por exemplo a quantidade de inventário disponível, necessita de garantir que as quantidades existentes em *stock* estão sempre atualizadas de acordo com as informações mais recentes.

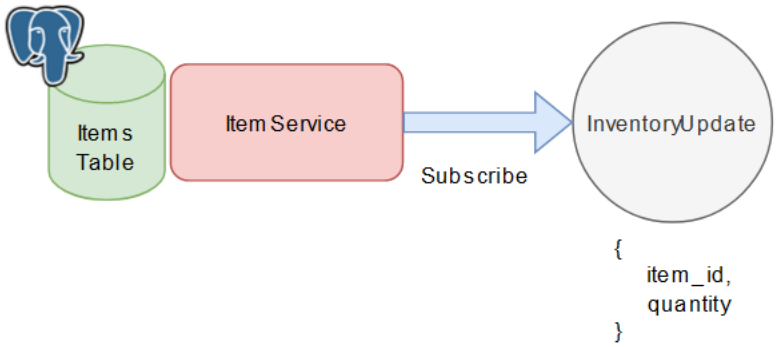


Figura 19 – Atualização da quantidade de inventário

À medida que ordens de compra são efetuadas pelos utilizadores, por consequência é necessário atualizar o inventário disponível relativamente aos produtos adquiridos. Este micro-serviço está subscrito a uma *queue* "*InventoryUpdate",* como mostra a Figura 19. Por cada ordem de compra realizada são publicadas informações relativas ao processo de compra, como a quantidade de produto que foi adquirido. Ao consumir esta mensagem, este micro-serviço atualiza a tabela relativa à quantidade de *stock* disponível desse mesmo produto.

4.1.4. *CartService*

O micro-serviço *CartService* é responsável pela gestão do carrinho de compras de um utilizador. É possível adicionar itens ao carrinho, que são alvo de interesse pelo utilizador, para mais tarde facilitar o processo de compra. Os *endpoints* e respetiva descrição estão descritos na Tabela 5.

Método	Caminho	Descrição
GET	/cart/{userId}	Obter o carrinho de compras de um utilizador.
POST	/cart/create	Criar um carrinho de compras para um determinado utilizador.
DELETE	/cart/{cartId}	Eliminar o carrinho de compras de um utilizador.

Tabela 5 - *Endpoints* do micro-serviço *CartService*

Este micro-serviço está subscrito a uma *queue* "*PriceUpdate*", como mostra a Figura 20, de forma a garantir que o carrinho de compras de um utilizador é sempre o mais recente e está coerente com os preços praticados numa determinada altura pela plataforma (e.g. baixa de preço de um determinado item devido a aplicação de um desconto sobre esse item), como exemplificado no ponto anterior 4.1.4.

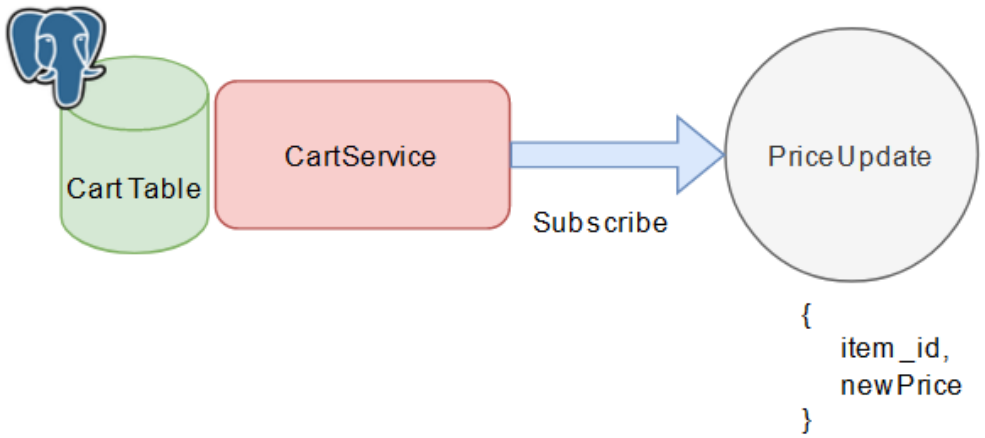


Figura 20 - Subscrição ao evento de atualização de preços

4.1.5. *OrderService*

O micro-serviço OrderService, é responsável por gerir as ordens de compra por parte dos utilizadores. As operações disponibilizadas por este micro-serviços estão descritas na Tabela 6.

Método	Caminho	Descrição
GET	/order/{orderId}	Obter uma ordem de compra efetuada por um utilizador.
POST	/order/create	Criar uma ordem de compra.
GET	/order/pay/{orderId}	Pagar uma ordem de compra.
DELETE	/order/delete/{orderId}	Apagar uma ordem de compra.
GET	/order/all	Obter todas as ordens de compra.

Tabela 6 - *Endpoints* do micro-serviço *OrderService*

Após o início de uma ordem de compra por parte de um utilizador, o evento "*OrderHistory*" é publicado numa *queue*, onde são publicadas as principais informações relativas à ordem de compra efetuada (e.g. itens, descrição, preço, quantidades e data). Adicionalmente, o utilizador após ter feito a ordem de compra, pode efetuar o seu pagamento. O evento "*PaymentProcessing*" é publicado numa *queue* com informações relativas à ordem de compra e ao utilizador. A Figura 21, exemplifica esta comunicação.

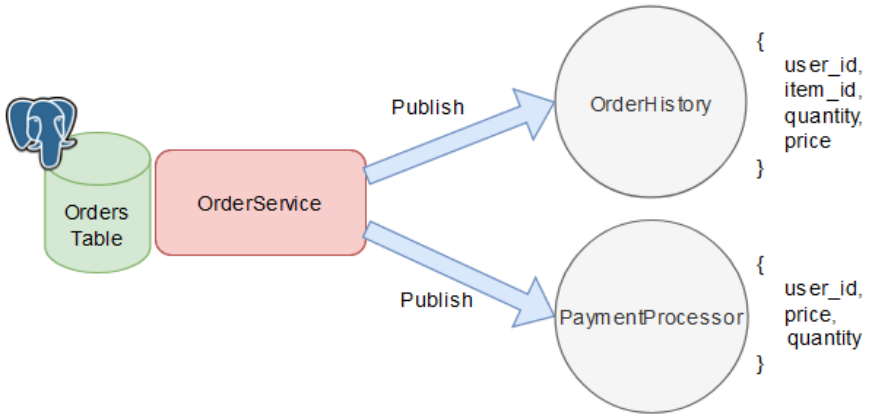


Figura 21 - Publicação de eventos de ordem de compra e ordem de pagamento

4.1.6. *OrderHistoryService*

O micro-serviço *OrderHistoryService* é reponsável por lidar com o histórico de compras da plataforma. As operações e *endpoints* encontram-se descritas na Tabela 7.

Método	Caminho	Descrição
GET	/ordersHistory/all	Obter todo o histórico de compras da plataforma.
GET	/ordersHistory/{userId}	Obter todo o histórico de compras de um utilizador.

Tabela 7 - *Endpoints* do micro-serviço *OrderHistoryService*

Este micro-serviço está subscrito a uma *queue* "*OrderHistory*", que contém informações acerca da ordem de compras por parte dos utilizadores. É responsável por guardar todas as informações relativas às compras efetuadas na plataforma numa base de dados *NoSQL*, que neste caso é o *MongoDB.* Todos os utilizadores têm acesso a todas as compras efetuadas no passado. O administrador tem acesso a todas as compras feitas por todos os utilizadores da plataforma.

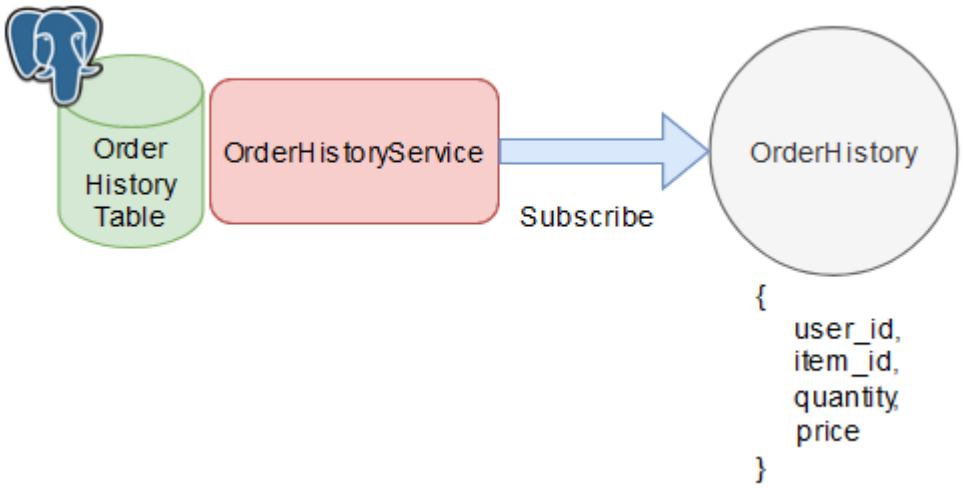


Figura 22 - Subscrição de ordens de compra

4.1.7. PaymentProcessorService

O micro-serviço *PaymentProcessorService* tem como responsabilidade processar o pedido de pagamento de uma ordem de compra efetuada por parte de um utilizador. Este é composto por um único *endpoint*, como está representado na Tabela 8.

Método	Caminho	Descrição
GET	/payment/{orderId}	Estado do pagamento de uma determinada ordem de compra.

Tabela 8 - *Endpoints* do micro-serviço *PaymentProcessorService*

O evento de pagamento de uma ordem de compra, "*PaymentProcessing*", é subscrito por este micro-serviço. Este é responsável por guardar em base de dados a intenção de pagamento por parte do utilizador com informações importantes acerca do pagamento como o preço, data e hora. De seguida, o micro-serviço em questão publica um evento "*PaymentRequest"* e fica à espera do evento "*PaymentResponse*" de modo a conhecer o resultado do pagamento. O estado do pagamento é atualizado adicionado na base de dados de acordo com os seguintes estados:

- PENDING – O pagamento está a ser processado.
- FAIL – O pagamento foi recusado por fundos insuficientes.
- SUCCESS – O pagamento foi efetuado com sucesso.

Após sucesso na validação do pagamento, o evento "*InventoryUpdate*" é publicado numa *queue* de forma a atualizar a quantidade de *stock* disponível relativamente ao produto comprado. O fluxo de funcionamento está demonstrado na Figura 23.

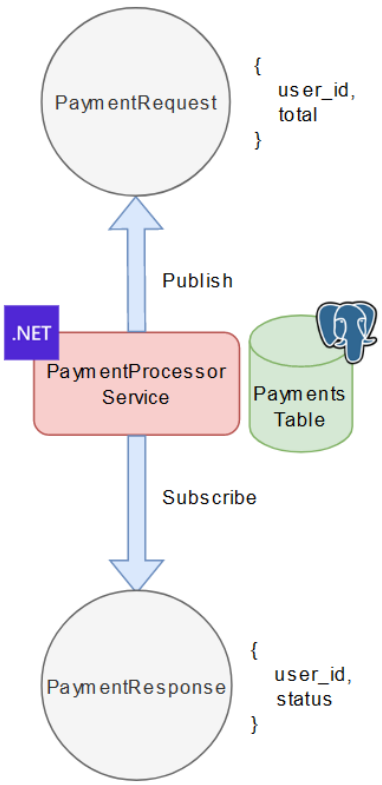


Figura 23 - Publicação e subscrição de eventos de processamento de pagamento

4.1.8. *PaymentValidatorService*

O micro-serviço *PaymentValidatorService* tem como responsabilidade validar os fundos de um utilizador e atualizar a conta de acordo com as compras efetuadas.

Sempre que o evento "*PaymentRequest*" é efetuado com informações relativas à compra (e.g. produtos e preço) e ao utilizador (e.g. identificação do utilizador), este micro-serviço tem como finalidade verificar se o utilizador, que pretende pagar uma determinada compra, possui fundos suficientes para o fazer. Em caso positivo, o micro-serviço é responsável por atualizar a tabela relativa ao saldo do utilizador de acordo com a compra efetuada. Após a validação da compra, é publicado o evento "*PaymentResponse*", responsável por alertar se uma determinada compra foi efetuada com sucesso ou insucesso.

O fluxo de funcionamento é semelhante ao micro-serviço *PaymentProcessorService,* como está representado na Figura 24.

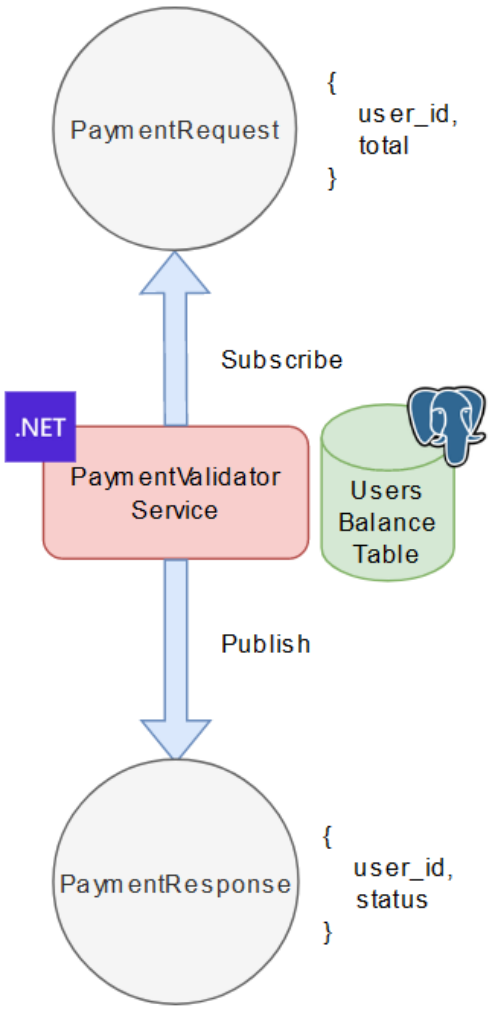


Figura 24 - Publicação e subscrição de eventos de validação de pagamentos

4.1.9. *ReviewService*

O micro-serviço *ReviewService* é responsável pela gestão de avaliações por parte dos utilizadores. Os *endpoints* e operações estão descritas na Tabela 9.

Método	Caminho	Descrição
POST	/review/create	Criar uma avaliação sobre um determinado item.
GET	/review /{id}	Obter uma determinada avaliação.
DELETE	/review/delete/{id}	Apagar uma determinada avaliação da plataforma.
PUT	/review/update	Atualizar uma avaliação presente na plataforma
GET	/review/all	Obter todas as avaliações presentes na plataforma.

Tabela 9 - *Endpoints* do micro-serviço *ReviewService*

As avaliações efetuadas pelos utilizadores são armazenadas numa base de dados *NoSQL*, MongoDB.

4.1.10. *EmailService*

O micro-serviço *EmailService* está subscrito ao evento "*PaymentResponse*", de forma a alertar os utilizadores, por email, o resultado da operação de pagamento.

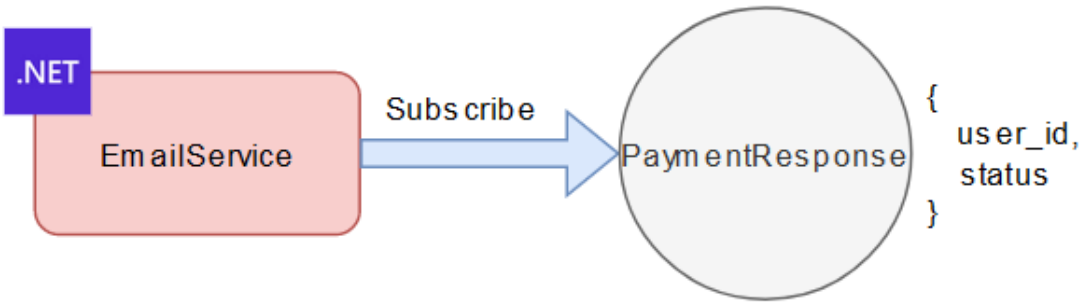


Figura 25 - Subscrição ao evento de validação de pagamento

Desta forma, os utilizadores ao serem notificados por email acerca da validação do pagamento, conseguem saber o estado das suas compras sem necessidade de acesso à plataforma, podendo voltar a efetuar uma nova tentativa de pagamento mais tarde. Este micro-serviço é também de uso exclusivo interno pela plataforma.

4.2. Escalabilidade e Confiabilidade

De modo a garantir a escalabilidade a confiabilidade da plataforma, foi utilizado o *Kubernetes* que disponibiliza mecanismos e ferramentas para automatizar todo o processo e as tarefas de *deployment,* escalabilidade e gestão dos micro-serviços.

A Figura 26 mostra a definição de um *deployment*, em *yaml,* onde é definido o *container* que contém, neste caso, um micro-serviço.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: account-service-deployment
spec:
  selector:
    matchLabels:
      app: account-service
      version: v1
  replicas: 1
  template:
    metadata:
      labels:
        app: account-service
        version: v1
    spec:
      containers:
      - name: account-service
        image: jdro10/account-service
        imagePullPolicy: Never
        ports:
        - containerPort: 80
```

Figura 26 - *Deployment*

Além disso, também podemos configurar as métricas de quando um micro-serviço deve ser escalado, ou seja, quando novas instâncias devem ser lançadas pela plataforma. A Figura 27 mostra um exemplo desta configuração, onde o micro-serviço *OrderService* está configurado para ser escalado quando a utilização do *CPU* da máquina que aloja o serviço atinge os 80%.

Também é possível definir o número mínimo e o número máximo de réplicas que pretendemos lançar num determinado momento.

```yaml
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2
metadata:
  name: account-service
spec:
  maxReplicas: 3
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: account-service
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

Figura 27 - *Horizontal Scaling*

Todos os micro-serviços foram configurados seguindo estes exemplos mostrados na Figura 26 e Figura 27.

4.3. Continuous Integration e testes

Para agilizar o processo de desenvolvimento, e de *deployment* de micro-serviços, foi criada uma *pipeline* em *yaml* que permite detetar não conformidades que possam estar presentes no código. As *pipelines* permitem que as tarefas de *build* e de qualidade, como correr as baterias de teste presente nos diversos micro-serviços, sejam automáticas agilizando o processo de desenvolvimento e a produtividade dos desenvolvedores.

O desenvolvimento deste projeto, contou com o uso de *git* para o controlo de versões. A plataforma utilizada foi o *Github* onde foi criado um *workflow*, como mostra a Figura 28, que dá *build* aos micro-serviços e correm os respetivos testes.

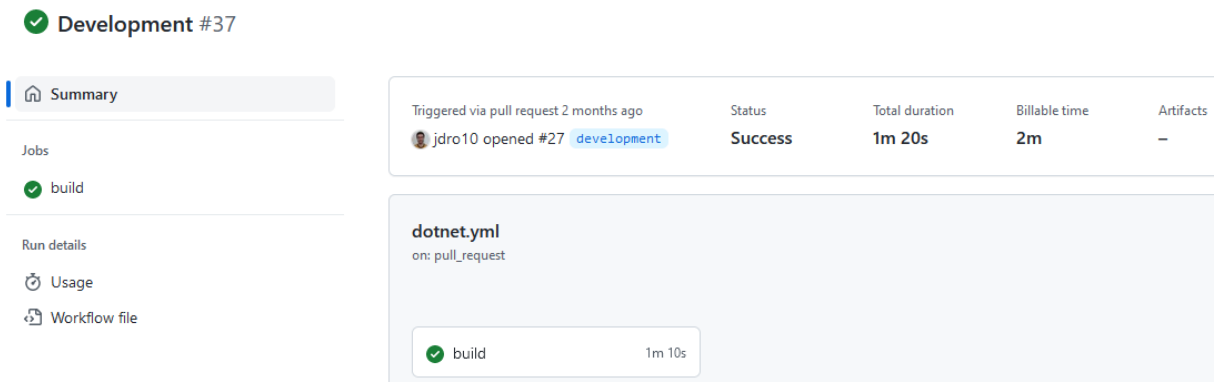


Figura 28 - *Github workflow*

O desenvolvimento de novas funcionalidades para cada micro-serviço foi feito com recurso a uma *branch* de desenvolvimento. Após estas estarem implementadas, ao ser iniciado o processo de *pull request* para o *branch* principal, que contém a plataforma estável, a *pipeline* está configurada para automaticamente dar *build* à plataforma e correr os respetivos testes.

A Figura 29 mostra um excerto *pipeline*, desenvolvida em *yaml*, que dá *build* a um micro-serviço e corre os respetivos testes desenvolvidos.

```yaml
name: .NET

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Setup .NET for ACCOUNT SERVICE
      uses: actions/setup-dotnet@v2
      with:
        dotnet-version: 6.0.x
    - name: Restore dependencies for ACCOUNT SERVICE
      run: dotnet restore
      working-directory: AccountService
    - name: Build ACCOUNT SERVICE
      run: dotnet build --no-restore
      working-directory: AccountService
    - name: Run Unit Tests ACCOUNT SERVICE
      run: dotnet test -l "console;verbosity=normal"
      working-directory: Tests/AccountServiceTests
```

Figura 29 - Pipeline em *yaml*

4.4. *Frontend*

A plataforma desenvolvida, com recurso à *framework React*, foi desenvolvida como uma *Progressive Web App (PWA). PWAs* são agnósticas em termos de sistemas operativos, permitindo desenvolver uma aplicação através de uma única *codebase* diminuindo os recursos necessários para manter as várias aplicações que necessitam de ser desenvolvidas para correrem em diferentes sistemas ou dispositivos.

A plataforma pode ser usada com ou sem autenticação dependendo das ações que o utilizador pretende. O registo pode ser feito através da indicação de alguns dados básicos do utilizador como o endereço de email, palavra-passe, nome e morada. Para o *login* basta apenas indicar o endereço de email e a respetiva palavra-passe. As *views* desenvolvidas estão representadas na Figura 30. Os micro-serviços utilizados nestas duas ações são os seguintes: *AccountService* e *AuthenticationService.*

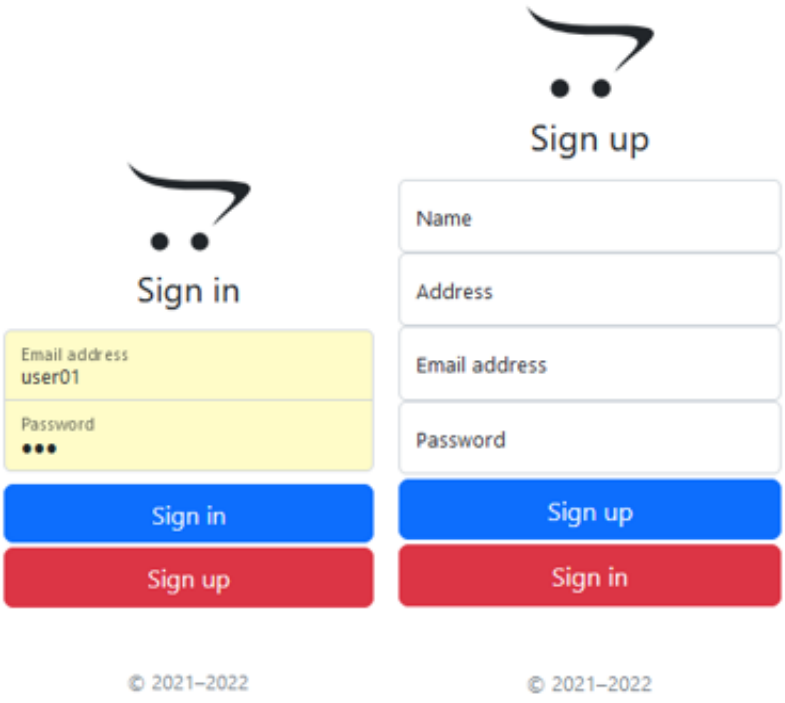


Figura 30 – Autenticação e registo.

A *landing page* é a página principal da aplicação, como podemos ver na Figura 31. Aqui é possível, aos utilizadores, consultarem os itens disponíveis na plataforma. Algumas informações importantes são mostradas como o preço do produto e a disponibilidade em *stock*. Também é possível adicionar produtos ao carrinho, pesquisar determinados produtos através da barra de pesquisa e consultar avaliações de produtos.

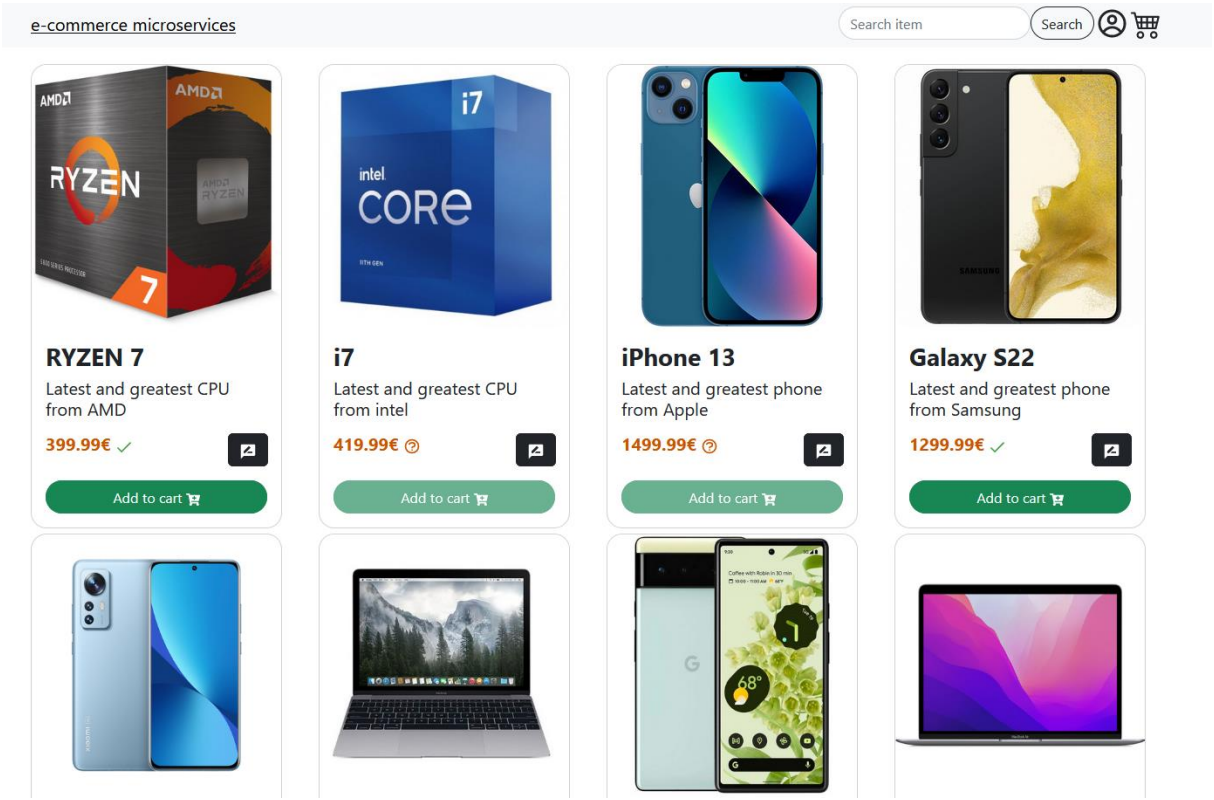


Figura 31 - Página principal

O carrinho de compras, como podemos ver na Figura 32, mostra ao utilizador os produtos que tem adicionados e algumas informações básicas como o preço de cada produto individual e o total. O utilizador pode efetuar a compra neste menu.

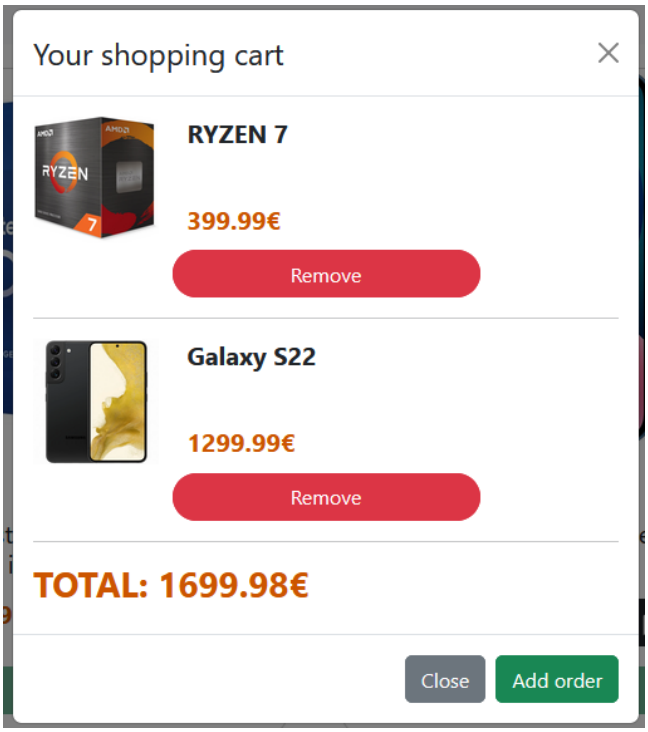


Figura 32 - Carrinho de compras

Adicionalmente, existe uma secção de acesso ao perfil de utilizador, onde utilizador pode ver as informações básicas ou alterar, como o nome, endereço de email e morada. Por último existe uma secção que contém todo o histórico de compras do utilizador onde também pode efetuar o pagamento do produto caso não o tenha feito, como mostrado na Figura 33.

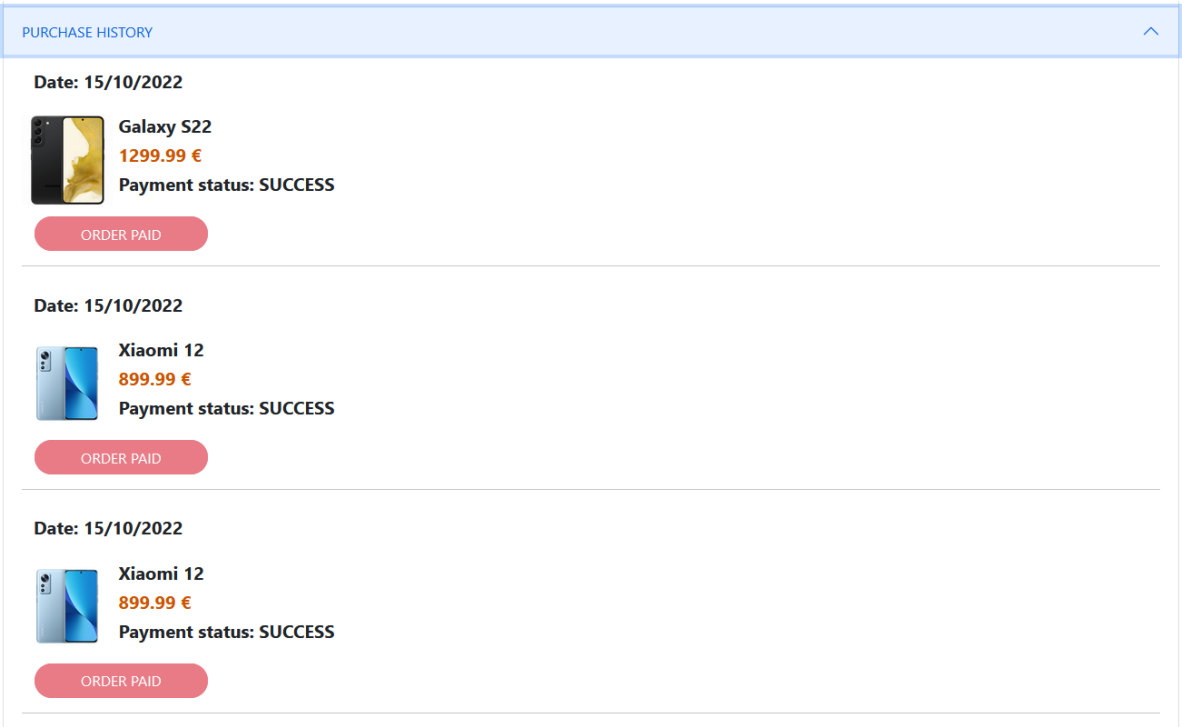


Figura 33 – Histórico de compras

Capítulo 6 – Conclusão e trabalho futuro

6.1. Conclusão

O tópico micro-serviços vem cada vez mais a ser falado e discutido nos últimos anos. Muitas empresas iniciaram o desenvolvimento das suas plataformas baseadas numa arquitetura monolítica. À medida que foram crescendo, naturalmente com o tempo, foram surgindo cada vez mais problemas e muito difíceis de gerir. A complexidade, escalabilidade e a fiabilidade das plataformas foram os principais problemas relatados nestes casos de estudo. Para resolver estes problemas, muitas empresas decidiram migrar as suas plataformas seguindo uma abordagem de arquitetura micro-serviços. Estas empresas rapidamente perceberam os benefícios que esta abordagem possibilitou. A divisão da complexidade das regras negócio em pequenos serviços de fácil gestão, aumentaram essencialmente a produtividade dos desenvolvedores permitindo focarem-se fundamentalmente na resolução de pequenos problemas. Esta divisão permitiu escalar funcionalidades específicas de plataformas de acordo com a utilização real por parte dos utilizadores, o que permitiu, não só, aumentar a fiabilidade e confiabilidade da plataforma, mas também diminuir os custos associados ao alojamento destas plataformas.

As plataformas de comércio eletrónico podem apresentar muitos destes problemas relatados ao longo deste projeto. O pico de utilização que estas podem sofrer é frequente, principalmente em alturas de saldos, lançamentos de novos produtos ou até mesmo de forma imprevisível. O *downtime* que estas plataformas possam eventualmente sofrer traduzem-se em perdas de rendimento e insatisfação dos clientes. As arquiteturas micro-serviços permitem resolver muitos destes problemas e assegurar a confiabilidade, fiabilidade e reduzir os custos, principalmente sobre plataformas de comércio eletrónico.

Naturalmente, as arquiteturas micro-serviços, como são pequenos serviços distribuídos que formam uma plataforma como um conjunto, trazem muitos desafios ao nível da conceção da arquitetura e implementação, que não estão presentes nas arquiteturas monolíticas. A utilização do modelo *Domain Driven Design* e contextos limitados permitem centrar o foco no desenvolvimento de pequenos serviços que visam a resolver um determinado problema muito específico. A utilização de mecanismos de comunicação assíncrona, como *message brokers*, permitem assegurar a comunicação entre micro-serviços mantendo estes totalmente independentes e desacoplados. A utilização de ferramentas, como o *Docker*, permite a colocação de serviços em containers, que contem todos os requisitos necessários para correr um determinado serviço, traduz-se em consistência de ambientes e uma maior simplicidade de *deploy*. A utilização do *Kubernetes* permite alcançar uma automatização total durante todo

o processo de gestão de micro-serviços, escalabilidade e tolerância a falhas, tornando-se uma ferramenta muito importante para os desenvolvedores e empresas, pois deixa de haver necessidade de gestão manual de todos os serviços.

Em síntese, as plataformas monolíticas continuam, e continuarão no futuro, a ser uma abordagem válida na conceção de arquiteturas para o desenvolvimento de plataformas. No entanto, estas devem ser utilizadas em situações em que o rápido desenvolvimento é mais valioso em detrimento da escalabilidade e fiabilidade, ou seja, aplicações que estão pensadas em serem alvo com um número de acesso constante. Assim sendo, uma arquitetura micro-serviços mostra-se muito capaz de assegurar o funcionamento e fiabilidade das plataformas de comércio eletrónico. A prova de conceito desenvolvida, permitiu estudar e aplicar alguns padrões de *software* que visam resolver problemas específicos que advém da utilização deste tipo de arquitetura.

6.2. Trabalho Futuro

O término deste projeto permitiu perceber as principais vantagens e desvantagens das arquiteturas monolíticas e arquiteturas micro-serviços. As arquiteturas micro-serviço mostram-se capazes de resolver muitos problemas relatados e aqui apresentados e que estão particularmente associados a plataformas de comércio eletrónico.

Para trabalho futuro propõe-se o estudo e/ou implementação dos seguintes tópicos:

- *Micro-Frontends* – Da mesma forma que o *backend* de uma plataforma de comércio eletrónico pode tornar altamente complexo o processo de desenvolvimento, explorar os benefícios que os Micro-frontends podem trazer ao nível organizacional de uma empresa.
- *Cloud* – Explorar plataformas de alojamento de aplicações como o *Google Cloud, AWS* e *Azure* e alojar a plataforma desenvolvida no contexto deste projeto.
- Introdução de novos micro-serviços com tecnologias diferentes além do *ASP.NET* utilizado durante o desenvolvimento deste projeto.
- Implementação de novas funcionalidades nesta plataforma, como um sistema de recomendação para os utilizadores tendo em conta os produtos adquiridos ao longo do tempo.
- Realização/implementação de testes de integração entre os diferentes micro-serviços.
- Melhorias visuais do *frontend*, principalmente ao nível da responsividade nos dispositivos móveis.

Referências

[1] “World Internet Users Statistics and 2022 World Population Stats.” https://www.internetworldstats.com/stats.htm

[2] “View of What is a Population in Online Shopping Research? A perspective from Malaysia.” https://turcomat.org/index.php/turkbilmat/article/view/549/354

[3] J. Kazanavicius and D. Mazeika, “Migrating Legacy Software to Microservices Architecture,” *2019 Open Conference of Electrical, Electronic and Information Sciences, eStream 2019 - Proceedings*, Apr. 2019, doi: 10.1109/ESTREAM.2019.8732170.

[4] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, “Microservices,” *IEEE Softw*, vol. 35, no. 3, pp. 96–100, May 2018, doi: 10.1109/MS.2018.2141030.

[5] J. Thönes, “Microservices,” *IEEE Softw*, vol. 32, no. 1, Jan. 2015, doi: 10.1109/MS.2015.11.

[6] O. Al-Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” *18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018 - Proceedings*, pp. 149–154, Nov. 2018, doi: 10.1109/CINTI.2018.8928192.

[7] K. Gos and W. Zabierowski, “The Comparison of Microservice and Monolithic Architecture,” *International Conference on Perspective Technologies and Methods in MEMS Design*, pp. 150–153, 2020, doi: 10.1109/MEMSTECH49584.2020.9109514.

[8] A. Krylovskiy, M. Jahn, and E. Patti, “Designing a Smart City Internet of Things Platform with Microservice Architecture,” *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, pp. 25–30, Oct. 2015, doi: 10.1109/FICLOUD.2015.55.

[9] W. Hasselbring and G. Steinacker, “Microservice architectures for scalability, agility and reliability in e-commerce,” *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pp. 243–246, Jun. 2017, doi: 10.1109/ICSAW.2017.11.

[10] G. Cherradi, A. el Bouziri, A. Boulmakoul, and K. Zeitouni, “Real-Time Microservices Based Environmental Sensors System for Hazmat Transportation Networks Monitoring,” *Transportation Research Procedia*, vol. 27, pp. 873–880, Jan. 2017, doi: 10.1016/J.TRPRO.2017.12.087.

[11] P. Mohata and P. Tijare, “Implementing Microservice Architecture for improving E-commerce websites performance,” *IOSR Journal of Engineering (IOSRJEN) www.iosrjen.org ISSN*, vol. 09, pp. 38–45, 2019, Accessed: Jan. 15, 2022. [Online]. Available: www.iosrjen.org

[12] M. Wu, X. Ding, and R. Hou, “Design and implementation of B2B E-commerce platform based on microservices architecture,” *ACM International Conference Proceeding Series*, pp. 30–34, May 2019, doi: 10.1145/3339363.3339369.

[13] J. A. Suthendra, M. Ariance, and I. Pakereng, “Implementation of Microservices Architecture on E-Commerce Web Service,” *ComTech: Computer, Mathematics and Engineering Applications*, vol. 11, no. 2, pp. 89–95, Dec. 2020, doi: 10.21512/COMTECH.V11I2.6453.

[14] “Why Netflix Moved to a Microservices Architecture | ProgrammableWeb.” https://www.programmableweb.com/news/why-netflix-moved-to-microservices-architecture/elsewhere-web/2016/04/02

[15] “10 companies that paved the way for developing microservices | Divante.” https://www.divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others

[16] “Scalable Microservices at Netflix. Challenges and Tools of the Trade.” https://www.infoq.com/presentations/netflix-ipc/?utm_source=infoq&utm_medium=slideshare&utm_campaign=slidesharesf

[17] K. Goldsmith, “Microservices @ Spotify”.

[18] “Amazon Stats: Growth, sales, and more - Tips, info, and stories about selling in Amazon stores. Whether you’re just getting started selling online or building an ecommerce empire, the Amazon Selling Partner blog features articles to help you get there.” https://sell.amazon.com/blog/grow-your-business/amazon-stats-growth-and-sales

[19] “What Led Amazon to its Own Microservices Architecture - The New Stack.” https://thenewstack.io/led-amazon-microservices-architecture/

[20] “Microservice Architecture — Learn, Build, and Deploy Applications - DZone Microservices.” https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a

[21] “eBay Architecture.” https://www.slideshare.net/tcng3716/ebay-architecture?next_slideshow=1

[22] “Monolith to Microservices: Transforming a web-scale, real-world e-commerce platform using the Strangler Pattern — Runscope Blog.” https://web.archive.org/web/20220430061931/https://blog.runscope.com/posts/monolith-microservices-transforming-real-world-ecommerce-platform-using-strangler-pattern

[23] M. Villamizar *et al.*, “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures,” *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, pp. 179–182, Jul. 2016, doi: 10.1109/CCGRID.2016.37.

[24] M. Villamizar *et al.*, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” *2015 10th Colombian Computing Conference, 10CCC 2015*, pp. 583–590, Nov. 2015, doi: 10.1109/COLUMBIANCC.2015.7333476.

[25] F. Rademacher, J. Sorgalla, and S. Sachweh, “Challenges of domain-driven microservice design: A model-driven perspective,” *IEEE Softw*, vol. 35, no. 3, pp. 36–43, May 2018, doi: 10.1109/MS.2018.2141028.

[26] “Domain analysis for microservices - Azure Architecture Center | Microsoft Learn.” https://learn.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis

[27] “What is .NET? An open-source developer platform.” https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

[28] “ASP.NET | Open-source web framework for .NET.” https://dotnet.microsoft.com/en-us/apps/aspnet

[29] “PostgreSQL: The world’s most advanced open source database.” https://www.postgresql.org/

[30] “MongoDB: The Developer Data Platform | MongoDB | MongoDB.” https://www.mongodb.com/

[31] “Messaging that just works — RabbitMQ.” https://www.rabbitmq.com/

[32] “AMQP 0-9-1 Model Explained — RabbitMQ.” https://www.rabbitmq.com/tutorials/amqp-concepts.html

[33] C. Anderson, “Docker,” *IEEE Softw*, vol. 32, no. 3, pp. 102–105, May 2015, doi: 10.1109/MS.2015.62.

[34] “Docker overview | Docker Documentation.” https://docs.docker.com/get-started/overview/

[35] “Kubernetes.” https://kubernetes.io/

[36] “Pods | Kubernetes.” https://kubernetes.io/docs/concepts/workloads/pods/

[37] “Deployments | Kubernetes.” https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[38] “Service | Kubernetes.” https://kubernetes.io/docs/concepts/services-networking/service/

[39] “Volumes | Kubernetes.” https://kubernetes.io/docs/concepts/storage/volumes/

[40] “Kubernetes Components | Kubernetes.” https://kubernetes.io/docs/concepts/overview/components/

[41] “React – A JavaScript library for building user interfaces.” https://reactjs.org/

[42] “moq/moq4: Repo for managing Moq 4.x.” https://github.com/moq/moq4

[43] F. Montesi and J. Weber, “Circuit Breakers, Discovery, and API Gateways in Microservices,” Sep. 2016, doi: 10.48550/arxiv.1609.05830.

[44] “Service Discovery in a Microservices Architecture - NGINX.” https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

[45] A. Akbulut and H. G. Perros, “Performance Analysis of Microservice Design Patterns,” *IEEE Internet Comput*, vol. 23, no. 6, pp. 19–27, Nov. 2019, doi: 10.1109/MIC.2019.2951094.

[46] “Circuit Breaker.” https://microservices.io/patterns/reliability/circuit-breaker.html

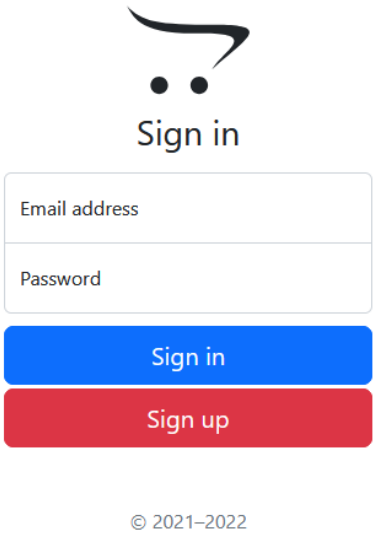
[47] “Health Check.” https://microservices.io/patterns/observability/health-check-api.html

[48] “Database per service.” https://microservices.io/patterns/data/database-per-service.html

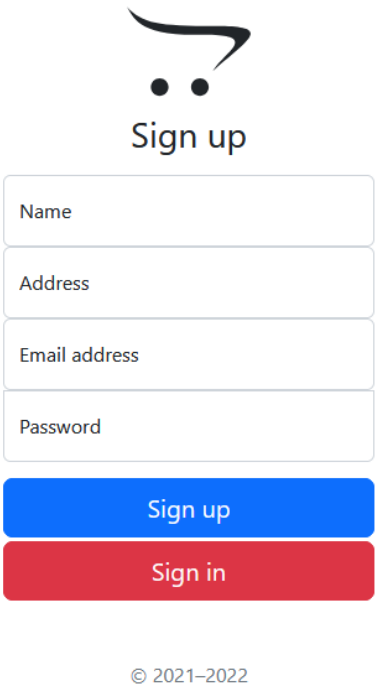
[49] K. Munonye and P. Martinek, “Evaluation of Data Storage Patterns in Microservices Archicture,” *SOSE 2020 - IEEE 15th International Conference of System of Systems Engineering, Proceedings*, pp. 373–380, Jun. 2020, doi: 10.1109/SOSE50414.2020.9130516.

ANEXOS

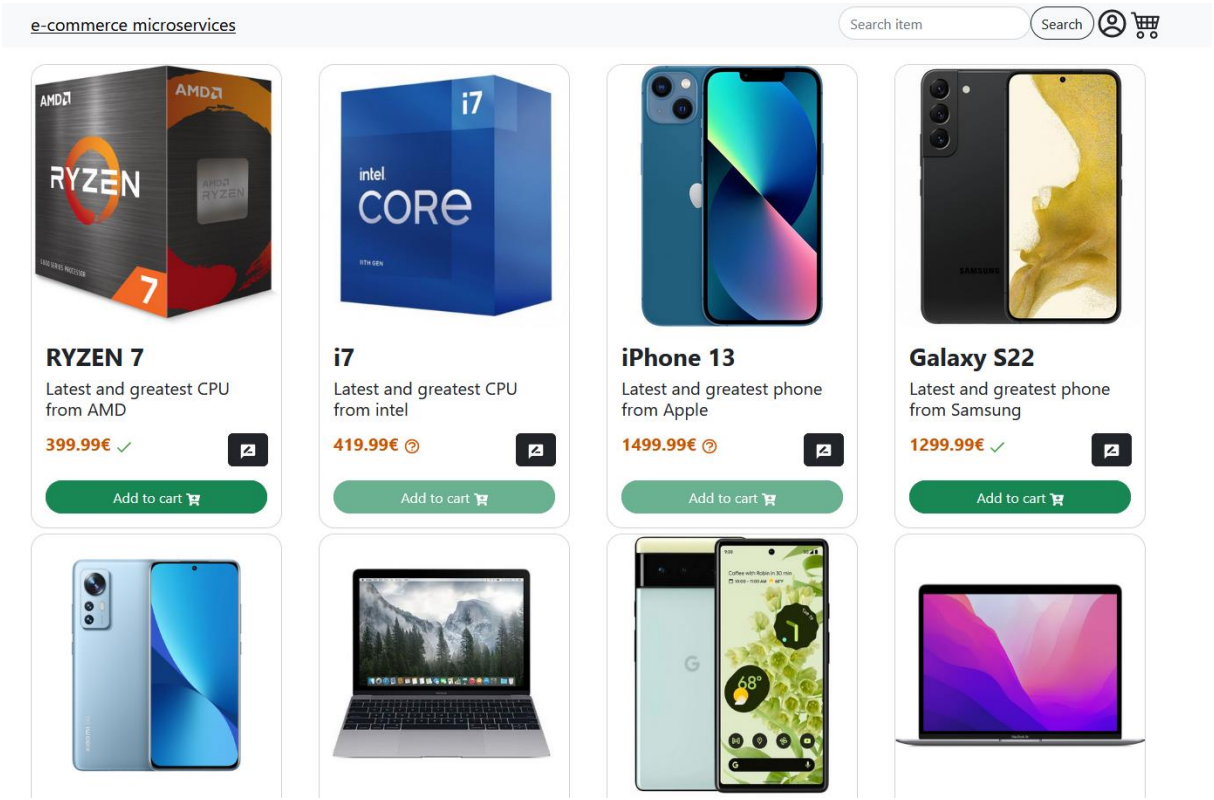
Página de login:



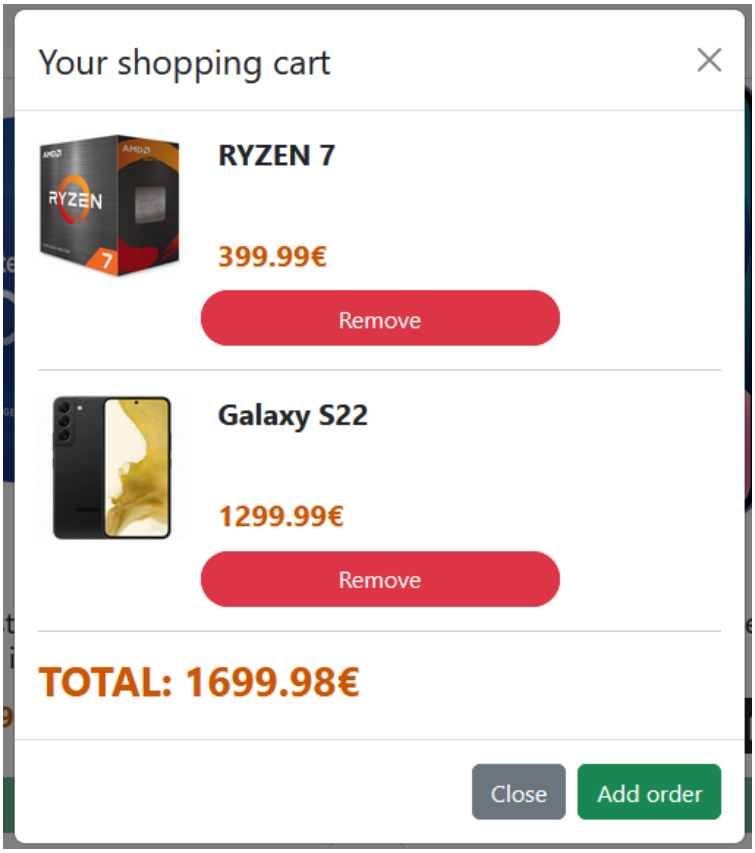
Página de registo:



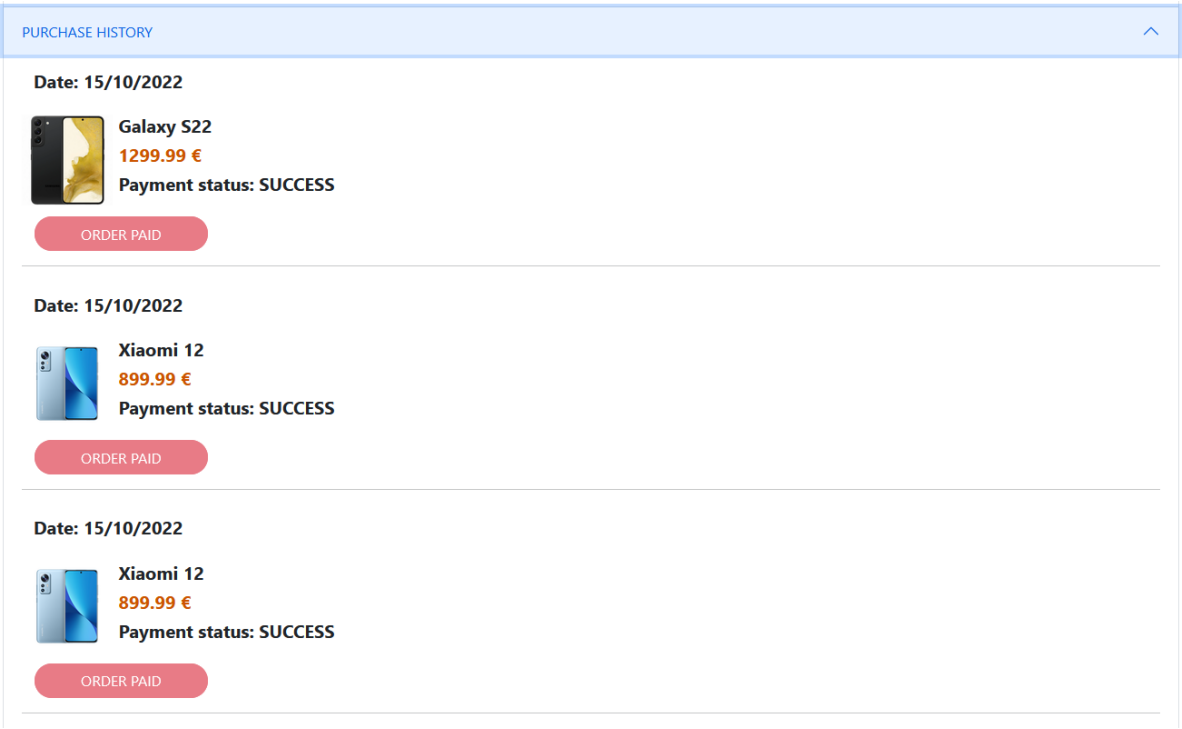
Página principal:



Carrinho de compras:



Histórico de compras de um utilizador:



Avaliações de produtos: